SCAPI Pseudocode Specification

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.

Cryptography and Computer Security Research Group

Department of Computer Science

Bar-Ilan University

11/16/2010

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

1 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., P_1,P_2 or P_1,…,P_n, 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:
    a. SEND:
    b. WAIT:
    c. COMPUTE:
    d. SAMPLE:
    e. RUN SUBPROTOCOL:
    f. OUTPUT:
    g. 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.

2 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.

2.1 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 $2^t < q$
Parties' Inputs:	• Common input statement: h • P's private input: a value $w \in Z_q$ such that $h=g^w$
Parties' Outputs:	• P: nothing • V: ACC or REJ

SIGMA_DLOG Protocol Specification	
Prover message 1 (a):	SAMPLE a random $r \in Z_q$ and COMPUTE $a = g^r$
Verifier challenge (e):	SAMPLE a random challenge $e \in \{0,1\}^t$
Prover message 2 (z):	COMPUTE $z = r + ew \bmod q$
Verifier check:	ACC IFF VALID_PARAMS(G,q,g)=TRUE AND $h \in G$ AND $g^z = ah^e$

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 $e \in \{0,1\}^t$
Computation:	SAMPLE a random $z \in Z_q$ COMPUTE $a = g^z \cdot h^{-e}$ (where $-e$ here means $-e \bmod q$) 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.

2.2 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 $2^t < q$
Parties' Inputs:	• Common input: (h,u,v) and a parameter t such that $2^t < q$ • P's private input: a value $w \in Z_q$ such that $u=g^w$ and $v=h^w$
Parties' Outputs:	• P: nothing • V: ACC or REJ

SIGMA_DH Protocol Specification	
Prover message 1 (a,b):	SAMPLE a random $r \in Z_q$ and COMPUTE $a = g^r$ and $b = h^r$
Verifier challenge (e):	SAMPLE a random challenge $e \in \{0,1\}^t$
Prover message 2 (z):	COMPUTE $z = r + ew \bmod q$
Verifier check:	ACC IFF VALID_PARAMS(G,q,g)=TRUE AND $h \in G$ AND $g^z = au^e$ AND $h^z = bv^e$

SIGMA_DH Simulator (*M*) Specification	
Input:	Parameters (G,q,g) and t, input (h,u,v) and a challenge $e \in \{0,1\}^t$
Computation:	SAMPLE a random $z \in Z_q$ COMPUTE $a = g^z \cdot u^{-e}$ and $b = h^z \cdot v^{-e}$ (where $-e$ here means $-e \bmod q$) OUTPUT $((a,b),e,z)$

2.3 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 $(g_1,\ldots,g_m,h_1,\ldots,h_m)$ is an ***extended*** Diffie-Hellman tuple, meaning that there exists a single $w \in Z_q$ such that $h_i=g_i^w$ 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 $2^t < q$
Parties' Inputs:	• Common input: $(g_1,\ldots,g_m,h_1,\ldots,h_m)$and t such that $2^t < q$ • P's private input: a value $w \in Z_q$ such that $h_i=g_i^w$ for all i
Parties' Outputs:	• P: nothing • V: ACC or REJ

SIGMA_EXTEND_DH Protocol Specification	
Prover message 1 (a):	SAMPLE a random $r \in Z_q$ and COMPUTE $a_i = g_i^r$ for all i SET $a=(a_1,\ldots,a_m)$.
Verifier challenge (e):	SAMPLE a random challenge $e \in \{0,1\}^t$
Prover message 2 (z):	COMPUTE $z = r + ew \bmod q$
Verifier check:	ACC IFF VALID_PARAMS(G,q,g)=TRUE AND all $g_1,\ldots,g_m \in G$ AND for all $i=1,\ldots,m$ it holds that $g_i^z = a_i \cdot h_i^e$

SIGMA_EXTEND_DH Simulator (*M*) Specification	
Input:	Parameters (G,q,g) and t, input (h,u,v) and a challenge $e \in \{0,1\}^t$
Computation:	SAMPLE a random $z \in Z_q$ For every $i=1,\ldots,m$, COMPUTE $a_i = g_i^z \cdot h_i^{-e}$ (where $-e$ here means $-e \bmod q$) OUTPUT $((a_1,\ldots,a_m),e,z)$

2.4 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 4.1 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 $2^t < q$
Parties' Inputs:	• Common input: **(*h,c*)** • P's private input: values $x,r \in Z_q$ such that $c = g^r \cdot h^x$
Parties' Outputs:	• P: nothing • V: ACC or REJ

SIGMA_PEDERSEN Protocol Specification	
Prover message 1 (a):	SAMPLE random values $\alpha \in Z_q$ and $\beta \in Z_q$ and COMPUTE $a = h^{\alpha} \cdot g^{\beta}$
Verifier challenge (e):	SAMPLE a random challenge $e \in \{0,1\}^t$
Prover message 2 (z):	COMPUTE $u = \alpha + ex \bmod q$ and $v = \beta + er \bmod q$
Verifier check:	ACC IFF VALID_PARAMS(***G,q,g***)=TRUE AND $h \in G$ AND $h^u \cdot g^v = a \cdot c^e$

SIGMA_PEDERSEN Simulator (*M*) Specification	
Input:	Parameters (***G,q,g***) and ***t***, input **(*h,c*)** and a challenge $e \in \{0,1\}^t$
Computation:	SAMPLE random values $u \in Z_q$ and $v \in Z_q$ COMPUTE $a = h^u \cdot g^v \cdot c^{-e}$ (where $-e$ here means $-e \bmod q$) OUTPUT **(*a,e,*(*u,v*))**

2.5 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 $c = g^r \cdot h^x$, it suffices to prove knowledge of r s.t. $g^r = c \cdot h^{-x}$. This is just a DLOG Sigma protocol. See Section 4.1 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 $2^t < q$
Parties' Inputs:	• Common input: **(*h, c*)** and ***x*** • P's private input: the value $r \in Z_q$ such that $c = g^r \cdot h^x$
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: $h' = c \cdot h^{-x}$ • P's private input: a value $r \in Z_q$ such that $h' = g^r$

Sigma-Protocol Simulator (see Section 2.15): same as SIGMA_DLOG with inputs appropriately defined.

2.6 Sigma Protocol for El Gamal Commitment Knowledge (SIGMA_ELGAMAL_COMMIT)

Note that an ElGamal commitment to x is a tuple (h,c_1,c_2) where h is a public key, and (c_1,c_2) 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,c_1, c_2)
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 $2^t < q$
Parties' Inputs:	• Common input statement: (h,c_1,c_2) • P's private input: a value $w \in Z_q$ such that $h = g^w$ (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 $w \in Z_q$ such that $h = g^w$

Sigma-Protocol Simulator (see Section 2.15): same as SIGMA_DLOG with inputs appropriately defined.

2.7 Sigma Protocol that an ElGamal-Committed Value is x (SIGMA_COMMITTED_VALUE_ELGAMAL)

Note that an ElGamal commitment to x is a tuple (h,c_1,c_2) where h is a public key, and (c_1,c_2) 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,c_1, c_2) 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 $2^t < q$
Parties' Inputs:	• Common input statement: (h,c_1,c_2) and x • P's private input: a value $r \in Z_q$ such that $c_1=g^r$ and $c_2 =h^r \cdot x$
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: $(g,h,u,v) = (g,h,c_1,c_2/x)$ • P's private input: a value $r \in Z_q$ such that $c_1=g^r$ and $c_2/x =h^r$

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 2.15): same as SIGMA_DH with inputs appropriately defined.

2.8 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 $2^t < q$
Parties' Inputs:	• Common input statement: an ElGamal public-key ***h*** • P's private input: a value $w \in Z_q$ such that $h=g^w$
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 $w \in Z_q$ such that $h=g^w$

Sigma-Protocol Simulator (see Section 2.15): same as SIGMA_DLOG with inputs appropriately defined.

2.9 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 (c_1, c_2) with public-key h is x
References:	None: this is just a DH Sigma Protocol

2.9.1 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 $2^t < q$
Parties' Inputs:	• Common input statement: (c_1,c_2), h and x • P's private input: a value $w \in Z_q$ such that $h=g^w$ and $c_2/c_1^w = x$ (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: $(g,h,u,v) = (g,c_1,h,c_2/x)$ • P's private input: a value $w \in Z_q$ such that $h=g^w$ and $c_2/x = c_1^w$

Sigma-Protocol Simulator (see Section 2.15): same as SIGMA_DH with inputs appropriately defined.

2.9.2 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 $2^t < q$
Parties' Inputs:	• Common input statement: (c_1,c_2), ***h*** and ***x*** • P's private input: a value $r \in Z_q$ such that $c_1=g^r$ and $c_2/x = h^r$ (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: $(g,h,u,v) = (g,h,c_1,c_2/x)$ • P's private input: a value $r \in Z_q$ such that $c_1=g^r$ and $c_2/x =h^r$

Sigma-Protocol Simulator (see Section 2.15): same as SIGMA_DH with inputs appropriately defined.

2.10 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. Currently only the case that the randomness is known is specified. [Not sure about the other one right now.]

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 (u_1,u_2,e,v) with public-key g_1,g_2,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 $2^t < q$
Parties' Inputs:	• Common input statement: (u_1,u_2,e,v), g_1,g_2,c,d,h and x • P's private input: a value $r \in Z_q$ such that $u_1=g_1^r$, $u_2=g_2^r$, $e=h^r \cdot x$ and $v = (cd^{H(u1,u2,e)})^r$
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: $(g_1,g_2,g_3,g_4,h_1,h_2,h_3,h_4) = (g_1,g_2,h,cd^{H(u1,u2,e)},u_1,u_2,e/x,v)$ • P's private input: a value $r \in Z_q$ such that $h_i=g_i^r$ for all i

Sigma-Protocol Simulator (see Section 2.15): same as SIGMA_EXTEND_DH with inputs appropriately defined.

2.11 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 $r \in Z^*_{N'}$ such that $c=r^N \bmod N'$ • Note $N=n^s$ and $N'=N^{s+1}$
Parties' Outputs:	• P: nothing • V: ACC or REJ

SIGMA_ ZERO_DAMGARD_JURIK Protocol Specification	
Prover message 1 (a):	SAMPLE a random value $s \in Z^*_n$ and COMPUTE $a = s^N \bmod N'$
Verifier challenge (e):	SAMPLE a random challenge $e \in \{0, 1\}^t$
Prover message 2 (z):	COMPUTE $z = s \cdot r^e \bmod n$
Verifier check:	ACC IFF c,a,z are relatively prime to n AND AND $z^N = a \cdot c^e \bmod N'$

SIGMA_ ZERO_DAMGARD_JURIK Simulator (*M*) Specification	
Input:	Parameter t, input (n,c) and a challenge $e \in \{0, 1\}^t$
Computation:	SAMPLE a random value $z \in Z^*_n$ COMPUTE $a = z^N/c^e \bmod N'$ 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 $m=n^{-1} \bmod \varphi(n)=n^{-1} \bmod (p-1)(q-1)$. Then, it can recover the randomness r from c by computing $c^m \bmod n$ (this equals $r^{n/n} \bmod n = r$). Once given r, the prover can proceed with the above protocol.

2.12 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 $r \in Z^*_{N'}$ such that $c=(1+n)^x r^N \bmod N'$ • Note $N=n^s$ and $N'=N^{s+1}$
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 $c'=c\cdot(1+n)^{-x}$ • P's private input: a value $r \in Z_q$ such that $c'=r^N \bmod N'$

Sigma-Protocol Simulator: same as SIGMA_ZERO_DAMGARD_JURIK with inputs appropriately defined.

2.13 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 c_1,c_2,c_3 are encryptions of values x_1,x_2,x_3 s.t. $x_1 \cdot x_2 = x_3 \bmod N$
References:	Damgard-Jurik

SIGMA_ZERO_DAMGARD_JURIK Protocol Parameters	
Parties' Identities:	Prover (P) and Verifier (V)
Common Parameters:	Length parameter s and soundness parameter $t < \lvert n \rvert / 3$
Parties' Inputs:	• Common input statement: public key n and ciphertexts c_1,c_2,c_3 • P's private input: value $r_1,r_2,r_3 \in Z^*_{N'}$ s.t. $c_i=(1+n)^{x_i} \cdot r_i^N \bmod N'$ • Note $N=n^s$ and $N'=N^{s+1}$
Parties' Outputs:	• P: nothing • V: ACC or REJ

SIGMA_ ZERO_DAMGARD_JURIK Protocol Specification	
Prover message 1 (a):	SAMPLE random values $d \in Z_N$, $r_d \in Z^*_n$, $r_{db} \in Z^*_n$, COMPUTE $a_1=(1+n)^d r_d^N \bmod N'$ and $a_2=(1+n)^{d \cdot x2} r_{db}^N \bmod N'$ and SET $a = (a_1,a_2)$
Verifier challenge (e):	SAMPLE a random challenge $e \in \{0,1\}^t$
Prover message 2 (z):	COMPUTE $z_1=e \cdot x_1+d \bmod N$, $z_2 = r_1^e \cdot r_d \bmod n$, $z_3=r_2^{z1}/(r_{db} \cdot r_3^e) \bmod n$, and SET $z=(z_1,z_2,z_3)$
Verifier check:	ACC IFF $c_1,c_2,c_3,a_1,a_2,z_1,z_2,z_3$ are relatively prime to n AND $c_1^e \cdot a_1 = (1+n)^{z1} z_2^N \bmod N'$ AND $c_2^{z1}/(a_2 \cdot c_3^e) = z_3^N \bmod N'$

SIGMA_ ZERO_DAMGARD_JURIK Simulator (*M*) Specification	
Input:	Parameter s, input (n,c_1,c_2,c_3) and a challenge $e \in \{0,1\}^t$
Computation:	SAMPLE random values $z_1 \in Z_N$, $z_2 \in Z^*_n$, $z_3 \in Z^*_n$ COMPUTE $a_1 = (1+n)^{z1} z_2^N / c_1^e \bmod N'$ AND $a_2 = c_2^{z1}/(z_3^N \cdot c_3^e) \bmod N'$ OUTPUT (a,e,z) where $a = (a_1,a_2)$ AND $z=(z_1,z_2,z_3)$

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 $m=n^{-1} \bmod \varnothing(n)=n^{-1} \bmod (p-1)(q-1)$. Then, it can recover the randomness r from c by computing $c^m \bmod n$ (this equals $r^{n/n} \bmod n = r$). Once given r, the prover can proceed with the above protocol.

2.14 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 $2^t < q$
Parties' Inputs:	• Common input: a series of m statements $\{x_i\}$ • P's private input: a series of witnesses $\{w_i\}$ such that for every i $(x_i, w_i) \in R_i$
Parties' Outputs:	• P: nothing • V: ACC or REJ

AND_SIGMA Protocol Specification	
Prover message 1 $(a_1,\ldots,a_m)$:	COMPUTE all first prover messages $a_1,\ldots,a_m$
Verifier challenge (e):	SAMPLE a single random challenge $e \in \{0, 1\}^t$
Prover message 2 $(z_1,\ldots,z_m)$:	COMPUTE all second prover messages $z_1,\ldots,z_m$
Verifier check:	ACC IFF all verifier checks are ACC

AND_SIGMA Simulator (*M*) Specification	
Input:	A series of m statements $\{x_i\}$ and a challenge $e \in \{0, 1\}^t$
Computation:	RUN each Sigma protocol simulator with e

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

2.15 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 $2^t < q$
Parties' Inputs:	• Common input: pair (x_0, x_1) • P's private input: w such that $(x_b, w) \in R$ for some bit b
Parties' Outputs:	• P: nothing • V: ACC or REJ

OR_2_SIGMA Protocol Specification	
Let (a_i, e_i, z_i) denote the steps of a Σ protocol π_i for proving that $x_i \in L_{Ri}$ (i=0,1)	
Prover message 1 (a_1,a_2):	COMPUTE the first message a_b in π_b, using (x_b, w) as input SAMPLE a random challenge $e_{1-b} \in \{0, 1\}^t$ RUN the simulator M for π_i on input (x_{1-b}, e_{1-b}) to obtain $(a_{1-b}, e_{1-b}, z_{1-b})$ The message is (a_1, a_2); e_{1-b}, z_{1-b} are stored for later
Verifier challenge (e):	SAMPLE a single random challenge $e \in \{0, 1\}^t$
Prover message 2 (e_0,z_0,e_1,z_1):	SET $e_b = e \oplus e_{1-b}$ COMPUTE the response z_b to (a_b, e_b) in π_b using input (x_b, w) The message is e_0, z_0, e_1, z_1
Verifier check:	ACC IFF all verifier checks are ACC

OR_2_SIGMA Simulator (*M*) Specification	
Input:	A pair of statements x_0, x_1 and a challenge $e \in \{0, 1\}^t$
Computation:	SAMPLE a random e_0, COMPUTE $e_1 = e \oplus e_0$ and then run the Sigma protocol simulator for each protocol with the resulting e_0, e_1 values.

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

2.16 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 $2^t < q$ A field **GF[2^t]** (let **1,…,n** be well defined elements of the field)
Parties' Inputs:	• Common input: $(x_1,\ldots, x_n)$ • P's private input: a set of ***k*** witnesses w_i (w_i is a witness that can be used to prove x_i)
Parties' Outputs:	• P: nothing • V: ACC or REJ

OR_MANY_SIGMA Protocol Specification	
Let (a_i,e_i,z_i) denote the steps of a Σ protocol π_i for proving that $x_i \in L_{Ri}$ Let ***I*** denote the set of indices for which P has witnesses	
Prover message 1 (*a*):	For every $j \notin I$, SAMPLE a random element $e_j \in GF[2^t]$ For every $j \notin I$, RUN the simulator on statement x_j and challenge e_j to get transcript (a_j,e_j,z_j) For every $i \in I$, RUN the prover P on statement x_i to get first message a_i SET $a=(a_1,\ldots,a_n)$
Verifier challenge (e):	WAIT for messages $a_1,\ldots,a_n$ SAMPLE a single random challenge $e \in GF[2^t]$
Prover msg 2 ($Q,e_1,z_1,\ldots,e_n,z_n$):	INTERPOLATE the points **(0,e)** and **{(j,ej)}** for every $j \notin I$ to obtain a degree $n-k$ polynomial **Q** (s.t. **Q(0)=e** and **Q(j)=e_j** for every $j \notin I$) For every $i \in I$, SET $e_i = Q(i)$ For every $i \in I$, COMPUTE the response z_i to (a_i, e_i) in π_i using input (x_i,w_i) The message is $Q,e_1,z_1,\ldots,e_n,z_n$ (where by ***Q*** we mean its coefficients)
Verifier check:	ACC IFF ***Q*** is of degree $n-k$ AND **Q(i)=e_i** for all **i=1,…,n** AND **Q(0)=e**, and the verifier output on (a_i,e_i,z_i) for all **i=1,…,n** is ACC

OR_MANY_SIGMA Simulator (*M*) Specification	
Input:	A series of n statements $\{x_i\}$ and a challenge $e \in GF[2^t]$
Computation:	SAMPLE random points $e_1,\ldots,e_{n-k} \in GF[2^t]$. COMPUTE the polynomial Q and values $e_{n-k},\ldots,e_n$ like in the protocol. RUN the simulator on each statement/challenge pair (x_i,e_i) for all $i=1,\ldots,n$ to obtain (a_i,e_i,z_i). OUTPUT $(a_1,e_1,z_1),\ldots,(a_n,e_n,z_n)$

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

3 Zero-knowledge

3.1 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 $(x,w) \in R$.
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 $e \in \{0,1\}^t$
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 $\boldsymbol{e} \in \{0,1\}^t$
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.

3.2 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*)** $\in$ ***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*** $\in$ $\{0,1\}^t$
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 $e \in \{0,1\}^t$
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.

3.3 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)*** $\in$ ***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

3.4 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 $(x,w) \in R$.
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 a_i in π, using (x,w) as input (use fresh randomness each time) COMPUTE the third message z_i^0 in π, using (x,w) as input, a_i as the first message, and $e = 0^t$ as the challenge COMPUTE the third message z_i^1 in π, using (x,w) as input, a_i 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 a_i, and with V as the receiver RUN UC_COMMIT.commit with P as the committer with input z_i^0, and with V as the receiver RUN UC_COMMIT.commit with P as the committer with input z_i^1, and with V as the receiver
Step 2 (V):	SAMPLE a random challenge $E \in \{0,1\}^T$ SEND E to P
Step 3 (both):	Let $E = E_1,\ldots,E_T$ FOR $i = 1$ to T RUN UC_COMMIT.decommit with P as the committer and with V as the receiver in order to reveal z_i^{Ei}

Step 5 (V):	IF • All decommit calls are valid, AND • FOR $i = 1$ to T, transcript (a_i, E_i, z_i^{Ei}) 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 a_i in π, using (x,w) as input (use fresh randomness each time) COMPUTE the third message z_i^0 in π, using (x,w) as input, a_i as the first message, and $e = 0^t$ as the challenge COMPUTE the third message z_i^1 in π, using (x,w) as input, a_i 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 a_i, and with V as the receiver RUN UC_COMMIT.commit as the committer with input z_i^0, and with V as the receiver RUN UC_COMMIT.commit as the committer with input z_i^1, and with V as the receiver
Step 3:	WAIT for a message E from V
Step 4:	Let $E = E_1,\ldots,E_T$ FOR $i = 1$ to T RUN UC_COMMIT.decommit as the committer and with V as the receiver in order to reveal z_i^{Ei}

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 a_i RUN UC_COMMIT.commit as the receiver, with P as the committer, to obtain a commitment to z_i^0 RUN UC_COMMIT.commit as the receiver, with P as the committer, to obtain a commitment to z_i^1
Step 2:	SAMPLE a random challenge $E \in \{0, 1\}^T$ 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 z_i^{Ei}
Step 4:	IF • All decommit calls are valid, AND • FOR $i = 1$ to T, transcript (a_i, E_i, z_i^{Ei}) is accepting in π on input x OUTPUT ACC ELSE OUTPUT REJ

4 Commitment schemes

4.1 Pedersen Commitment (COMMIT_PEDERSEN)

Protocol Name:	Pedersen commitment
Protocol Reference:	COMMIT_PEDERSEN
Protocol Type:	Perfectly-Hiding Commitment
Protocol Description:	Pedersen commitment
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 $x \in Z_q$
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 $x \in Z_q$

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 $a \in Z_q$ COMPUTE $h = g^a$ SEND ***h*** to C
Step 2 (Both):	C: IF **NOT** $h \in G$ REPORT ERROR and HALT SAMPLE a random value $r \in Z_q$ COMPUTE $c = g^r \cdot h^x$ SEND ***c*** R: OUTPUT nothing and STORE values ***(h,c)***
Decommit phase	

Step 1 (C):	SEND **(*r, x*)** to R
Step 2 (R):	IF $c = g^r \cdot h^x$ AND $x \in Z_q$ 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** $h \in G$ REPORT ERROR and HALT SAMPLE a random value $r \in Z_q$ COMPUTE $c = g^r \cdot h^x$ 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 $a \in Z_q$ COMPUTE $h = g^a$ SEND ***h*** to C
Step 2:	WAIT for message ***c*** from C OUTPUT nothing and STORE values (***h,c***)
Decommit phase	
Step 1:	WAIT for **(*r, x*)** from C
Step 2:	IF $c = g^r \cdot h^x$ AND $x \in Z_q$ OUTPUT ACC and value ***x*** ELSE OUTPUT REJ

The sampling of ***h*** and sending it to the committer can be carried out once for many commitments.

4.2 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***. 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*)**.

4.3 Pedersen Commitment with Trapdoor for Equivocation (COMMIT_WITH_TRAPDOOR_PEDERSEN)

Protocol Name:	Pedersen commitment
Protocol Reference:	COMMIT_WITH_TRAPDOOR_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. This trapdoor is output by the receiver and can be used by a higher-level application (e.g., by the ZK transformation of a sigma protocol to a zero-knowledge proof of knowledge)
References:	Protocol 6.5.3, page 164 of Hazay-Lindell

COMMIT_WITH_TRAPDOOR_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 $x \in Z_q$
Parties' Outputs:	• C: nothing • R's output from the COMMIT phase: ○ A trapdoor **trap** • R's output from the DECOMMIT phase: ○ ACC or REJ, and if ACC then a value $x \in Z_q$

This is identical to COMMIT_PEDERSEN except that the receiver outputs the value ***a*** used in the first (preprocessing) step of COMMIT_PEDERSEN. This value ***a*** is called the trapdoor.

4.4 Hash Pedersen Commitment with Trapdoor for Equivocation (COMMIT_WITH_TRAPDOOR_HASH_PEDERSEN)

As in COMMIT_WITH_TRAPDOOR_PEDERSEN, this is exactly the same as COMMIT_HASH_PEDERSEN except that the receiver outputs the value ***a*** used in the first (preprocessing) step.

4.5 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 $x \in G$ (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 $x \in G$

COMMIT_ELGAMAL Protocol Specification	
Commit phase	
Step 1 (C):	IF **NOT** VALID_PARAMS(***G,q,g***) REPORT ERROR and HALT SAMPLE random values $a,r \in Z_q$ COMPUTE $h = g^a$ COMPUTE $u = g^r$ and $v = h^r \cdot x$ 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

	• $h \in G$, AND • $u=g^r$, AND • $v = h^r \cdot x$, AND • $x \in G$ 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 $a,r \in Z_q$ COMPUTE $h = g^a$ COMPUTE $u = g^r$ and $v = h^r \cdot x$ 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 • $h \in G$, AND • $u=g^r$, AND • $v = h^r \cdot x$, AND • $x \in G$ 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.

4.6 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*)**.

4.7 Hash-Based Commitment (Basic) (COMMIT_HASH_BASIC COMMIT_ROM_BASIC)

Protocol Name:	Hash-based commitment (heuristic)
Protocol Reference:	COMMIT_HASH_BASIC, COMMIT_ROM_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. For COMMIT_ROM_BASIC, we view the hash as a *random oracle*, the scheme is actually a UC secure commitment for static adversaries.
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 $\boldsymbol{x} \in \{0,1\}^t$
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 $\boldsymbol{x} \in \{0,1\}^t$

COMMIT_HASH_BASIC Protocol Specification	
Commit phase	
Step 1 (C):	SAMPLE a random value $\boldsymbol{r} \in \{0,1\}^n$ COMPUTE $\boldsymbol{c} = \boldsymbol{H(r,x)}$ (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** • $\boldsymbol{c} = \boldsymbol{H(r,x)}$, AND • $\boldsymbol{x} \in \{0,1\}^t$ OUTPUT REJ

	ELSE OUTPUT ACC and value ***x***

COMMIT_HASH_BASIC Committer (C) Specification	
Commit phase	
Step 1:	SAMPLE a random value $r \in \{0, 1\}^n$ 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 • $x \in \{0, 1\}^t$ OUTPUT REJ ELSE OUTPUT ACC and value ***x***

4.8 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 $x \in \{0,1\}^n$
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 $x \in \{0,1\}^n$

COMMIT_HASH Protocol Specification	
Commit phase	
Step 1 (C):	SAMPLE a random value $r \in \{0,1\}^{3n}$ 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 • $x \in \{0,1\}^n$ OUTPUT REJ ELSE OUTPUT ACC and value x

COMMIT_HASH Committer (C) Specification	
Commit phase	
Step 1:	SAMPLE a random value $r \in \{0, 1\}^{3n}$ 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 • $x \in \{0, 1\}^n$ OUTPUT REJ ELSE OUTPUT ACC and value x

4.9 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 $x \in \{0, 1\}^t$
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 $x \in \{0, 1\}^t$

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

4.10 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 $x \in \{0, 1\}^t$
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 $x \in \{0, 1\}^t$

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):

3. Use COMMIT_PEDERSEN and SIGMA_ PEDERSEN
4. 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.

4.11 Fully Trapdoor Commitments

4.11.1 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 $x \in \{0, 1\}^t$
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 $x \in \{0, 1\}^t$

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:

5. Use COMMIT_PEDERSEN, SIGMA_PEDERSEN and SIGMA_COMMITTED_VALUE_PEDERSEN
6. 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.)

4.11.2 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

4.12 Additive Homomorphic Operation on Pedersen Commitments (PEDERSEN_ADD)

The Pedersen commitment scheme is additively homomorphic in Z_q, as follows. Given $c_1 = g^r h^x$ and $c_2 = g^s h^y$, observe that $c_1 \cdot c_2$ (with multiplication in the group) equals $g^{r+s} h^{x+y}$. In order to rerandomize, multiply again by g^u 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 $x+y \bmod q$, 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 $c_1, c_2 \in G$
Party's Output:	A single commitment c such that if c_1 is a commitment to x and c_2 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 $u \in Z_q$ COMPUTE $c = g^u \cdot c_1 \cdot c_2$ OUTPUT c

4.13 Multiplicative Homomorphic Operation on ElGamal Commitments (ELGAMAL_MULT)

The ElGamal commitment scheme is mulitplicatively homomorphic in G, as follows. Given $c_1 = (g^r, h^r \cdot x)$ and $c_2 = (g^s, h^s \cdot y)$, observe that $c_1 \cdot c_2$ (with multiplication of each element separately) equals $(g^{r+s}, h^{r+s} \cdot x \cdot y)$. In order to rerandomize, multiply again by (g^w, h^w) 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 $x \cdot y$ (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 $c_1 = (h,u_1,v_1)$, $c_2 = (h,u_2,v_2)$ OBSERVE: the same h value must appear in both
Party's Output:	A single commitment c such that if c_1 is a commitment to x and c_2 is a commitment to y, then c is a random commitment to $x \cdot y$

ELGAMAL_MULT Protocol Specification	
Step 1:	IF **NOT** (VALID_PARAMS(G,q,g) AND the same h value appears in c_1,c_2 AND u_1,v_1,u_2,v_2 are all elements of G), REPORT ERROR and HALT SAMPLE a random value $w \in Z_q$ COMPUTE $u = g^w \cdot u_1 \cdot u_2$ COMPUTE $v = h^w \cdot v_1 \cdot v_2$ OUTPUT $c = (u,v)$

4.14 UC-Secure Commitments

4.14.1 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: Universally Composable Commitments Using Random Oracles. TCC 2004: 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 $\boldsymbol{pid_C}$ and $\boldsymbol{pid_R}$, respectively. Any non-interactive string commitment protocol (e.g., COMMIT_BASIC_HASH).
Parties' Inputs:	C's private input: a value $\boldsymbol{x} \in \{0,1\}^t$
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 $\boldsymbol{x} \in \{0,1\}^t$

COMMIT_ UC_HASH Protocol Specification	
Commit phase	
Step 1 (C):	SAMPLE two random values $r_1, r_2 \in \{0,1\}^n$ COMPUTE $c_1 = COMMIT(H(sid,ssid,pid_C,pid_R,x,r_1),r_2)$ COMPUTE $c_2 = COMMIT(H(sid,r_2))$ STORE r_1, r_2 SEND $c=(sid,ssid,c_1,c_2)$ to R
Step 2 (R):	WAIT for a value $c=(sid,ssid,c_1,c_2)$ STORE ***c*** (ABORT if ***sid,ssid*** are incorrect)
Decommit phase	
Step 1 (C):	SEND (x,r_1,r_2) to R
Step 2 (R):	IF **NOT** • $c_1 = COMMIT(H(sid,ssid,pid_C,pid_R,x,r_1),r_2)$ AND • $c_2 = COMMIT(H(sid,r_2))$ AND • $x \in \{0,1\}^t$ OUTPUT REJ

	ELSE OUTPUT ACC and value ***x***

COMMIT_ UC_HASH Committer (C) Specification	
Commit phase	
Step 1:	SAMPLE two random values $r_1,r_2 \in \{0,1\}^n$ COMPUTE $c_1 = COMMIT(H(sid,ssid,pid_C,pid_R,x,r_1),r_2)$ COMPUTE $c_2 = COMMIT(H(sid,r_2))$ STORE r_1,r_2 SEND $c=(c_1,c_2)$ to R
Decommit phase	
Step 1:	SEND (x,r_1,r_2) to R
Step 2:	OUTPUT nothing

COMMIT_ UC_HASH Receiver (R) Specification	
Commit phase	
Step 1:	WAIT for a value $c=(sid,ssid,c_1,c_2)$ STORE ***c*** (ABORT if ***sid,ssid*** are incorrect)
Decommit phase	
Step 1:	WAIT for (x,r_1,r_2) from C
Step 4:	IF **NOT** • $c_1 = COMMIT(H(sid,ssid,pid_C,pid_R,x,r_1),r_2)$ AND • $c_2 = COMMIT(H(sid,r_2))$ AND • $x \in \{0,1\}^t$ OUTPUT REJ ELSE OUTPUT ACC and value ***x***

4.14.2 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 $\boldsymbol{pid_C}$ and $\boldsymbol{pid_R}$, respectively. A **common reference string** with random group elements $\boldsymbol{(g_1,g_2,c,d,h,h_1,h_2)}$ A statistical security parameter ***t'*** (for Sigma protocol security; error is $\boldsymbol{2^{-t'}}$)
Parties' Inputs:	C's private input: a value $\boldsymbol{x \in \{0,1\}^t}$
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 $\boldsymbol{x \in \{0,1\}^t}$

COMMIT_ UC_DDH Protocol Specification	
Commit phase	
Step 1 (C):	MAP the string $\boldsymbol{(x,sid,ssid,pid_C,pid_R)}$ to the group ***G***; call the result ***m*** SAMPLE a random value $\boldsymbol{r \in Z_q}$ and encrypt m using Cramer-Shoup with public key $\boldsymbol{(g_1,g_2,c,d,h)}$. Call the result $\boldsymbol{(u_1,u_2,e,v)}$. SEND $\boldsymbol{c=(sid,ssid,c_1,c_2)}$ 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 $\boldsymbol{(x,sid,ssid,pid_C,pid_R)}$ to the group ***G***; call the result ***m*** SAMPLE a random $\boldsymbol{e \in \{0,1\}^{t'}}$ and MAP ***e*** to an element $\boldsymbol{e' \in G}$ SAMPLE random $\boldsymbol{R,S \in Z_q}$, and COMPUTE $\boldsymbol{c'=(g_1^R \cdot g_2^S,\ h_1^R \cdot h_2^S \cdot e')}$. 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 $c'=(g_1^R \cdot g_2^S, h_1^R \cdot h_2^S \cdot e')$.
Step 4 (R):	IF **NOT** • $c_1, c2 \in G$ • $x \in \{0,1\}^t$ • 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 $(x,sid,ssid,pid_C,pid_R)$ to the group ***G***; call the result ***m*** SAMPLE a random value $r \in Z_q$ and encrypt m using Cramer-Shoup with public key (g_1,g_2,c,d,h). Call the result (u_1,u_2,e,v). SEND $c=(sid,ssid,c_1,c_2)$ 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 $c'=(g_1^R \cdot g_2^S, h_1^R \cdot h_2^S \cdot e')$. 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 $(x,sid,ssid,pid_C,pid_R)$ to the group ***G***; call the result ***m*** SAMPLE a random $e \in \{0,1\}^{t'}$ and MAP ***e*** to an element $e' \in G$ SAMPLE random $R,S \in Z_q$, and COMPUTE $c'=(g_1^R \cdot g_2^S, h_1^R \cdot h_2^S \cdot e')$. 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** • $c_1, c2 \in G$ • $x \in \{0,1\}^t$ • Sigma protocol output was ACC OUTPUT REJ ELSE OUTPUT ACC and value ***x***

4.15 Non-Malleable Commitments

4.15.1 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

4.15.2 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 $x \in \{0,1\}^t$

COMMIT_NON_MALLEABLE_CRS Protocol Specification	
Commit phase	
Step 1 (C):	SAMPLE a random value $r \in \{0,1\}^n$ COMPUTE $c = E_{pk}(x;r)$ (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** $c = E_{pk}(x;r)$ OUTPUT REJ ELSE OUTPUT ACC and value x

COMMIT_NON_MALLEABLE_CRS Committer (C) Specification	
Commit phase	
Step 1:	SAMPLE a random value $r \in \{0, 1\}^n$ COMPUTE $c = E_{pk}(x;r)$ (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** $c = E_{pk}(x;r)$ OUTPUT REJ ELSE OUTPUT ACC and value x

5 Oblivious Transfer

5.1 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_SEMIHONEST 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 $\sigma \in \{0, 1\}$
Parties' Outputs:	• S: nothing • R: x_σ

0T_DDH_SEMIHONEST Protocol Specification	
Step1 (R):	SAMPLE random values $\alpha \in Z_q$ and $h \in G$ COMPUTE h_0,h_1 as follows: 1. If $\sigma = 0$ then $h_0 = g^\alpha$ and $h_1=h$ 2. If $\sigma = 1$ then $h_0=h$ and $h_1 = g^\alpha$ SEND (h_0,h_1) to S
Step 2 (S):	SAMPLE a random value $r \in \{0, \ldots, q-1\}$ COMPUTE: • $u = g^r$ • $k_0 = h_0^r$ • $v_0 = x_0$ XOR $KDF(\lvert x_0\rvert, k_0)$ • $k_1 = h_1^r$ • $v_1 = x_1$ XOR $KDF(\lvert x_1\rvert, k_1)$ SEND (u,v_0,v_1) to R OUTPUT nothing
Step 3 (R):	COMPUTE $k_\sigma = (u)^\alpha$ OUTPUT $x_\sigma = v_\sigma$ XOR $KDF(\lvert v_\sigma\rvert, k_\sigma)$

0T_DDH_SEMIHONEST Sender (S) Specification	
Step 1:	WAIT for message (h_0,h_1) from R
Step 2:	SAMPLE a random value $r \in \{0, \ldots, q-1\}$ COMPUTE: • $u = g^r$ • $k_0 = h_0^r$ • $v_0 = x_0$ XOR $KDF(\lvert x_0 \rvert, k_0)$ • $k_1 = h_1^r$ • $v_1 = x_1$ XOR $KDF(\lvert x_1 \rvert, k_1)$ 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_SEMIHONEST Receiver (R) Specification	
Step 1:	SAMPLE random values $\alpha \in Z_q$ and $h \in G$ COMPUTE h_0,h_1 as follows: 1. If $\sigma = 0$ then $h_0 = g^\alpha$ and $h_1 = h$ 2. If $\sigma = 1$ then $h_0 = h$ and $h_1 = g^\alpha$ SEND (h_0,h_1) to S
Step 2:	WAIT for the message (u, v_0,v_1) from S
Step 3:	COMPUTE $k_\sigma = (u)^\alpha$ OUTPUT $x_\sigma = v_\sigma$ XOR $KDF(\lvert v_\sigma \rvert, k_\sigma)$

5.2 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 $(x_1^0, x_1^1), \ldots, (x_m^0, x_m^1)$; 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 $\sigma_1,\ldots,\sigma_m \in \{0,1\}$
Parties' Outputs:	• S: nothing • R: $x_1^{\sigma_1}, \ldots, x_m^{\sigma_m}$

0T_ SEMIHONEST_EXTENSION Protocol Specification	
Step 1 (Both):	S: SAMPLE a random string $s \in \{0,1\}^n$ of length ***n***; denote it $s_1,\ldots,s_n$ R: SAMPLE ***n*** random strings $T_1,\ldots,T_n \in \{0,1\}^m$ 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): $(T_i, T_i \text{ XOR } \sigma)$, where $\sigma=\sigma_1,\ldots,\sigma_m$ • S (playing receiver): s_i
Step 3 (S):	• Denote the output of S in the OT executions by $Q_1,\ldots,Q_n$ • Let **Q** be the matrix with ***m*** rows and ***n*** columns: $[Q_1 \mid Q_2 \mid \ldots \mid Q_n]$ 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: $[T_1 \mid T_2 \mid \ldots \mid T_n]$ and denote by ***T[i]*** the i^{th} row of ***Q*** (of length ***n***)
The i^{th} Transfer (with inputs (x_i^0, x_i^1) and σ_i)	
Step 1 (S):	SEND $y_i^0 = x_i^0 \oplus H(i, Q[i])$ and $y_i^1 = x_i^1 \oplus H(i, Q[i] \oplus s)$
Step 2 (R):	WAIT for (y_i^0, y_i^1) from S OUTPUT $z_i = y_i^{\sigma_i} \oplus H(i, T[i])$

0T_ SEMIHONEST_EXTENSION Sender (S) Specification	
Step 1:	SAMPLE a random string $s \in \{0,1\}^n$ of length n; denote it $s_1,\ldots,s_n$
Step 2:	For $i = 1$ **to** n, RUN OT_SEMIHONEST as the receiver with input s_i
Step 3:	• Denote the output of S in the OT executions by $Q_1,\ldots,Q_n$ • Let **Q** be the matrix with m rows and n columns: $[Q_1 \mid Q_2 \mid \ldots \mid Q_n]$ and denote by $Q[i]$ the i^{th} row of Q (of length n)
The i^{th} Transfer (with inputs (x_i^0, x_i^1) and σ_i)	
Step 1:	SEND $y_i^0 = x_i^0 \oplus H(i, Q[i])$ and $y_i^1 = x_i^1 \oplus H(i, Q[i] \oplus s)$ OUTPUT nothing

0T_ SEMIHONEST_EXTENSION Receiver (R) Specification	
Step 1:	SAMPLE n random strings $T_1,\ldots,T_n \in \{0,1\}^m$ each of length m
Step 2:	For $i = 1$ **to** n, RUN OT_SEMIHONEST as sender, with input $(T_i, T_i \text{ XOR } \sigma)$
Step 3:	• Let **T** be the matrix with m rows and n columns: $[T_1 \mid T_2 \mid \ldots \mid T_n]$ and denote by $T[i]$ the i^{th} row of Q (of length n)
The i^{th} Transfer (with inputs (x_i^0, x_i^1) and σ_i)	
Step 2:	WAIT for (y_i^0, y_i^1) from S OUTPUT $z_i = y_i^{\sigma_i} \oplus H(i, T[i])$

5.3 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 $\sigma \in \{0, 1\}$
Parties' Outputs:	• S: nothing • R: x_σ

0T_DDH_PRIVATE Protocol Specification	
Step 1 (Both):	IF **NOT** VALID_PARAMS(G,q,g) REPORT ERROR and HALT
Step 2 (R):	SAMPLE random values $\alpha,\beta,\gamma \in \{0, \ldots, q-1\}$ COMPUTE a as follows: 3. If $\sigma = 0$ then $a = (g^\alpha, g^\beta, g^{\alpha\beta}, g^\gamma)$ 4. If $\sigma = 1$ then $a = (g^\alpha, g^\beta, g^\gamma, g^{\alpha\beta})$ SEND a to S
Step 3 (S):	DENOTE the tuple a received by S by (x, y, z_0, z_1) IF **NOT** • $z_0 = z_1$ • $x, y, z_0, z_1 \in G$ REPORT ERROR (cheat attempt) SAMPLE random values $u_0,u_1,v_0,v_1 \in \{0, \ldots, q-1\}$ COMPUTE: • $w_0 = x^{u0} \cdot g^{v0}$ • $k_0 = (z_0)^{u0} \cdot y^{v0}$ • $w_1 = x^{u1} \cdot g^{v1}$ • $k_1 = (z_1)^{u1} \cdot y^{v1}$ • $c_0 = x_0$ **XOR** $KDF(\lvert x_0\rvert, k_0)$ • $c_1 = x_1$ **XOR** $KDF(\lvert x_1\rvert, k_1)$ SEND (w_0, c_0) and (w_1, c_1) to R OUTPUT nothing
Step 4 (R):	IF NOT • $w_0, w_1 \in G$, AND

	• c_0, c_1 are binary strings of the same length REPORT ERROR COMPUTE $k_\sigma = (w_\sigma)^\beta$ OUTPUT $x_\sigma = c_\sigma$ **XOR** $KDF(\|c_\sigma\|,k_\sigma)$

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 S by (x, y, z_0, z_1) IF **NOT** • $z_0 = z_1$ • $x, y, z_0, z_1 \in G$ REPORT ERROR (cheat attempt) SAMPLE random values $u_0,u_1,v_0,v_1 \in \{0, \ldots, q-1\}$ COMPUTE: • $w_0 = x^{u0} \cdot g^{v0}$ • $k_0 = (z_0)^{u0} \cdot y^{v0}$ • $w_1 = x^{u1} \cdot g^{v1}$ • $k_1 = (z_1)^{u1} \cdot y^{v1}$ • $c_0 = x_0$ **XOR** $KDF(\|x_0\|,k_0)$ • $c_1 = x_1$ **XOR** $KDF(\|x_1\|,k_1)$ 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 SAMPLE random values $\alpha,\beta,\gamma \in \{0, \ldots, q-1\}$ COMPUTE ***a*** as follows: 1. If $\sigma = 0$ then $a = (g^\alpha, g^\beta, g^{\alpha\beta}, g^\gamma)$ 2. If $\sigma = 1$ then $a = (g^\alpha, g^\beta, g^\gamma, g^{\alpha\beta})$ 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 • $w_0, w_1 \in G$, AND • c_0, c_1 are binary strings of the same length REPORT ERROR COMPUTE $k_\sigma = (w_\sigma)^\beta$ OUTPUT $x_\sigma = c_\sigma$ **XOR** $KDF(\|c_\sigma\|,k_\sigma)$

5.4 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 $\sigma \in \{0, 1\}$
Parties' Outputs:	• S: nothing • R: x_σ

0T_DDH_ONESIDEDSIM Protocol Specification	
Step 1 (Both):	IF **NOT** VALID_PARAMS(G,q,g) REPORT ERROR and HALT
Step 2 (R):	SAMPLE random values $\alpha,\beta,\gamma \in \{0, \ldots, q-1\}$ COMPUTE a as follows: 1. If $\sigma = 0$ then $a = (g^\alpha, g^\beta, g^{\alpha\beta}, g^\gamma)$ 2. If $\sigma = 1$ then $a = (g^\alpha, g^\beta, g^\gamma, g^{\alpha\beta})$ SEND a to S
Step 3 (Both):	DENOTE the tuple a received by S by (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** • $z_0 = z_1$ • $x, y, z_0, z_1 \in G$ REPORT ERROR (cheat attempt) SAMPLE random values $u_0,u_1,v_0,v_1 \in \{0, \ldots, q-1\}$ COMPUTE: • $w_0 = x^{u0} \cdot g^{v0}$ • $k_0 = (z_0)^{u0} \cdot y^{v0}$ • $w_1 = x^{u1} \cdot g^{v1}$ • $k_1 = (z_1)^{u1} \cdot y^{v1}$

	• $c_0 = x_0$ **XOR** $KDF(|x_0|,k_0)$ • $c_1 = x_1$ **XOR** $KDF(|x_1|,k_1)$ SEND (w_0, c_0) and (w_1, c_1) to R OUTPUT nothing
Step 5 (R):	IF NOT • $w_0, w_1 \in G$, AND • c_0, c_1 are binary strings of the same length REPORT ERROR COMPUTE $k_\sigma = (w_\sigma)^\beta$ OUTPUT $x_\sigma = c_\sigma$ **XOR** $KDF(|c_\sigma|,k_\sigma)$

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 received by (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** • $z_0 = z_1$ • $x, y, z_0, z_1 \in G$ REPORT ERROR (cheat attempt) SAMPLE random values $u_0,u_1,v_0,v_1 \in \{0, \ldots, q-1\}$ COMPUTE: • $w_0 = x^{u0} \cdot g^{v0}$ • $k_0 = (z_0)^{u0} \cdot y^{v0}$ • $w_1 = x^{u1} \cdot g^{v1}$ • $k_1 = (z_1)^{u1} \cdot y^{v1}$ • $c_0 = x_0$ **XOR** $KDF(|x_0|,k_0)$ • $c_1 = x_1$ **XOR** $KDF(|x_1|,k_1)$ 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 SAMPLE random values $\alpha,\beta,\gamma \in \{0, \ldots, q-1\}$ COMPUTE a as follows: 1. If $\sigma = 0$ then $a = (g^\alpha, g^\beta, g^{\alpha\beta}, g^\gamma)$ 2. If $\sigma = 1$ then $a = (g^\alpha, g^\beta, g^\gamma, g^{\alpha\beta})$ 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 • $w_0, w_1 \in G$, AND • c_0, c_1 are binary strings of the same length REPORT ERROR COMPUTE $k_\sigma = (w_\sigma)^\beta$ OUTPUT $x_\sigma = c_\sigma$ **XOR** $KDF(\lvert c_\sigma \rvert, k_\sigma)$

5.5 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,g_0) where (G,q,g_0) 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 $\sigma \in \{0, 1\}$
Parties' Outputs:	• S: nothing • R: x_σ

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

1. SAMPLE random values $s,t \in \{0, \ldots, q-1\}$
2. COMPUTE $u = w^s \cdot y^t$
3. COMPUTE $v = x^s \cdot z^t$
4. OUTPUT (u,v)

0T_DDH_FULLSIM Protocol Specification	
Step 1 (Both):	IF **NOT** VALID_PARAMS(G,q,g_0) REPORT ERROR and HALT
Step 2 (R):	SAMPLE random values $y, \alpha_0, r \in \{0, \ldots, q-1\}$ SET $\alpha_1 = \alpha_0 + 1$ COMPUTE 1. $g_1 = (g_0)^y$ 2. $h_0 = (g_0)^{\alpha 0}$ 3. $h_1 = (g_1)^{\alpha 1}$ 4. $g = (g_\sigma)^r$ 5. $h = (h_\sigma)^r$ 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 $(g_0,g_1,h_0,h_1/g_1)$ and private input for the prover α_0. If verifier-output is REJ, REPORT ERROR (cheat attempt) and HALT
Step 4 (S):	COMPUTE $(u_0,v_0) = \text{RAND}(g_0,g,h_0,h)$ COMPUTE $(u_1,v_1) = \text{RAND}(g_1,g,h_1,h)$

	COMPUTE $c_0 = x_0$ **XOR** $KDF(|x_0|,v_0)$ COMPUTE $c_1 = x_1$ **XOR** $KDF(|x_1|,v_1)$ SEND (u_0,c_0) and (u_1,c_1) to R OUTPUT nothing
Step 5 (R):	WAIT for (u_0,c_0) and (u_1,c_1) from S IF NOT • $u_0, u_1 \in G$, AND • c_0, c_1 are binary strings of the same length REPORT ERROR OUTPUT $x_\sigma = c_\sigma$ **XOR** $KDF(|c_\sigma|,(u_\sigma)^r)$

0T_DDH_ FULLSIM Sender (S) Specification	
Step 1:	IF **NOT** VALID_PARAMS(G,q,g_0) REPORT ERROR and HALT WAIT for message from R
Step 2:	DENOTE the values received by (g_1,h_0,h_1) and (g,h) Run the verifier in ZKPOK_FROM_SIGMA with Sigma protocol SIGMA_DH. Use common input $(g_0,g_1,h_0,h_1/g_1)$. If output is REJ, REPORT ERROR (cheat attempt) and HALT
Step 3:	COMPUTE (u_0,v_0) = **RAND**(g_0,g,h_0,h) COMPUTE (u_1,v_1) = **RAND**(g_1,g,h_1,h) COMPUTE $c_0 = x_0$ **XOR** $KDF(|x_0|,v_0)$ COMPUTE $c_1 = x_1$ **XOR** $KDF(|x_1|,v_1)$ SEND (u_0,c_0) and (u_1,c_1) to R
Step 4:	OUTPUT nothing

0T_DDH_ FULLSIM Receiver (R) Specification	
Step 1:	IF **NOT** VALID_PARAMS(G,q,g_0) REPORT ERROR and HALT SAMPLE random values $y,\alpha_0,r \in \{0, \ldots, q-1\}$ SET $\alpha_1 = \alpha_0 + 1$ COMPUTE 1. $g_1 = (g_0)^y$ 2. $h_0 = (g_0)^{\alpha 0}$ 3. $h_1 = (g_1)^{\alpha 1}$ 4. $g = (g_\sigma)^r$ 5. $h = (h_\sigma)^r$ 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 $(g_0,g_1,h_0,h_1/g_1)$ and private input α_0.
Step 3:	WAIT for messages (u_0,c_0) and (u_1,c_1) from S
Step 4:	IF NOT • $u_0, u_1 \in G$, AND • c_0, c_1 are binary strings of the same length REPORT ERROR OUTPUT $x_\sigma = c_\sigma$ **XOR** $KDF(|c_\sigma|,(u_\sigma)^r)$

5.6 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,g_0) where (G,q,g_0) 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 $\sigma \in \{0, 1\}$
Parties' Outputs:	• S: nothing • R: x_σ

0T_DDH_FULLSIM_ROM Protocol Specification	
Step 1 (Both):	IF **NOT** VALID_PARAMS(G,q,g_0) REPORT ERROR and HALT
Step 2 (R):	SAMPLE random values $y,\alpha_0,r \in \{0, \ldots, q-1\}$ SET $\alpha_1 = \alpha_0 + 1$ COMPUTE 1. $g_1 = (g_0)^y$ 2. $h_0 = (g_0)^{\alpha 0}$ 3. $h_1 = (g_1)^{\alpha 1}$ 4. $g = (g_\sigma)^r$ 5. $h = (h_\sigma)^r$ Run ZKPOK_FS_SIGMA with Sigma protocol SIGMA_DH using common input $(g_0,g_1,h_0,h_1/g_1)$ and private input α_0. Let t_P denote the resulting proof transcript. SEND (g_1,h_0,h_1), (g,h) and t_P to S
Step 3 (S):	Verify t_P using common input $(g_0,g_1,h_0,h_1/g_1)$. If verifier-output is REJ, REPORT ERROR (cheat attempt) and HALT COMPUTE (u_0,v_0) = **RAND**(g_0,g,h_0,h) COMPUTE (u_1,v_1) = **RAND**(g_1,g,h_1,h) COMPUTE $c_0 = x_0$ **XOR** $KDF(\lvert x_0\rvert,v_0)$ COMPUTE $c_1 = x_1$ **XOR** $KDF(\lvert x_1\rvert,v_1)$ SEND (u_0,c_0) and (u_1,c_1) to R OUTPUT nothing

Step 4 (R):	WAIT for (u_0,c_0) and (u_1,c_1) from S IF NOT • $u_0, u_1 \in G$, AND • c_0, c_1 are binary strings of the same length REPORT ERROR OUTPUT $x_\sigma = c_\sigma$ **XOR** $KDF(\lvert c_\sigma \rvert, (u_\sigma)^r)$

0T_DDH_ FULLSIM_ROM Sender (S) Specification	
Step 1:	IF **NOT** VALID_PARAMS(G,q,g_0) REPORT ERROR and HALT WAIT for message from R
Step 2:	DENOTE the values received by (g_1,h_0,h_1), (g,h) and t_P . Verify t_P using common input $(g_0,g_1,h_0,h_1/g_1)$. If output is REJ, REPORT ERROR (cheat attempt) and HALT COMPUTE (u_0,v_0) = **RAND**(g_0,g,h_0,h) COMPUTE (u_1,v_1) = **RAND**(g_1,g,h_1,h) COMPUTE $c_0 = x_0$ **XOR** $KDF(\lvert x_0 \rvert, v_0)$ COMPUTE $c_1 = x_1$ **XOR** $KDF(\lvert x_1 \rvert, v_1)$ SEND (u_0,c_0) and (u_1,c_1) to R
Step 3:	OUTPUT nothing

0T_DDH_ FULLSIM_ROM Receiver (R) Specification	
Step 1:	IF **NOT** VALID_PARAMS(G,q,g_0) REPORT ERROR and HALT SAMPLE random values $y,\alpha_0,r \in \{0, \ldots, q-1\}$ SET $\alpha_1 = \alpha_0 + 1$ COMPUTE 6. $g_1 = (g_0)^y$ 7. $h_0 = (g_0)^{\alpha 0}$ 8. $h_1 = (g_1)^{\alpha 1}$ 9. $g = (g_\sigma)^r$ 10. $h = (h_\sigma)^r$ Run ZKPOK_FS_SIGMA with Sigma protocol SIGMA_DH using common input $(g_0,g_1,h_0,h_1/g_1)$ and private input α_0. Let t_P denote the resulting proof transcript. SEND (g_1,h_0,h_1), (g,h) and t_P to S
Step 2:	WAIT for messages (u_0,c_0) and (u_1,c_1) from S
Step 3:	IF NOT • $u_0, u_1 \in G$, AND • c_0, c_1 are binary strings of the same length REPORT ERROR OUTPUT $x_\sigma = c_\sigma$ **XOR** $KDF(\lvert c_\sigma \rvert, (u_\sigma)^r)$

5.7 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,g_0) and (g_0,g_1,h_0,h_1) 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 $\sigma \in \{0, 1\}$
Parties' Outputs:	• S: nothing • R: x_σ

0T_DDH_UC_PVW Protocol Specification	
Step 1 (R):	SAMPLE a random value $r \in \{0, \ldots, q-1\}$ COMPUTE $g = (g_\sigma)^r$ and $h = (h_\sigma)^r$ SEND (g,h) to S
Step 2 (S):	COMPUTE $(u_0,v_0) = \text{RAND}(g_0,g,h_0,h)$ COMPUTE $(u_1,v_1) = \text{RAND}(g_1,g,h_1,h)$ COMPUTE $c_0 = x_0 \text{ XOR } KDF(\lvert x_0\rvert, v_0)$ COMPUTE $c_1 = x_1 \text{ XOR } KDF(\lvert x_1\rvert, v_1)$ SEND (u_0,c_0) and (u_1,c_1) to R OUTPUT nothing
Step 3 (R):	WAIT for (u_0,c_0) and (u_1,c_1) from S IF NOT • $u_0, u_1 \in G$, AND • c_0, c_1 are binary strings of the same length REPORT ERROR OUTPUT $x_\sigma = c_\sigma \text{ XOR } KDF(\lvert c_\sigma\rvert, (u_\sigma)^r)$

0T_DDH_ UC_PVW Sender (S) Specification	
Step 1:	WAIT for message **(*g*,*h*)** from R
Step 2:	COMPUTE **(u_0,v_0) = RAND(g_0,*g*,h_0,*h*)** COMPUTE **(u_1,v_1) = RAND(g_1,*g*,h_1,*h*)** COMPUTE c_0 = x_0 **XOR** ***KDF***(|x_0|,v_0) COMPUTE c_1 = x_1 **XOR** ***KDF***(|x_1|,v_1) SEND **(u_0,c_0)** and **(u_1,c_1)** to R OUTPUT nothing
Step 3:	OUTPUT nothing

0T_DDH_ UC_PVW Receiver (R) Specification	
Step 1:	SAMPLE a random value $r \in \{0, \ldots, q-1\}$ COMPUTE $g = (g_\sigma)^r$ and $h = (h_\sigma)^r$ SEND **(*g*,*h*)** to S
Step 2:	WAIT for messages **(u_0,c_0)** and **(u_1,c_1)** from S
Step 3:	IF NOT • $u_0, u_1 \in G$, AND • c_0, c_1 are binary strings of the same length REPORT ERROR OUTPUT $x_\sigma = c_\sigma$ **XOR** ***KDF***(|c_σ|,$(u_\sigma)^r$)

6 Batch Oblivious Transfer

6.1 Semi-Honest Batch OT (OT_DDH_SEMIHONEST_BATCH)

Protocol Name:	Semi-Honest Batch OT assuming DDH
Protocol Reference:	OT_DDH_SEMIHONEST_BATCH
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:	None. Optimization of semi-honest protocol for single execution.

0T_DDH_SEMIHONEST_BATCH Protocol Parameters	
Parties' Identities:	Sender (S) and Receiver (R)
Parties' Inputs:	• S's private input: m pairs $(x_1^0, x_1^1), \ldots, (x_m^0, x_m^1)$; 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 $\sigma_1, \ldots, \sigma_m \in \{0, 1\}$
Parties' Outputs:	• S: nothing • R: $x_1^{\sigma_1}, \ldots, x_m^{\sigma_m}$

0T_DDH_ SEMIHONEST_BATCH Protocol Specification	
Step1 (R):	For every $i=1,\ldots,m$, SAMPLE random values $\alpha_i \in Z_q$ and $h_i \in G$ For every $i=1,\ldots,m$, COMPUTE h_i^0, h_i^1 as follows: 3. If $\sigma_i = 0$ then $h_i^0 = g^{\alpha i}$ and $h_i^1 = h_i$ 4. If $\sigma_i = 1$ then $h_i^0 = h_i$ and $h_i^1 = g^{\alpha i}$ For every $i=1,\ldots,m$, SEND (h_i^0, h_i^1) to S
Step 2 (S):	SAMPLE a single random value $r \in \{0, \ldots, q-1\}$ and COMPUTE $u=g^r$ For every $i=1,\ldots,m$, COMPUTE: • $k_i^0 = (h_i^0)^r$ • $v_i^0 = x_i^0$ XOR $KDF(\lvert x_i^0 \rvert, k_i^0)$ • $k_i^1 = (h_i^1)^r$ • $v_i^1 = x_1$ XOR $KDF(\lvert x_1 \rvert, k_i^1)$ For every $i=1,\ldots,m$, SEND (u, v_i^0, v_i^1) to R OUTPUT nothing
Step 3 (R):	For every $i=1,\ldots,m$, COMPUTE $k_i^\sigma = (u)^{\alpha i}$ For every $i=1,\ldots,m$, OUTPUT $x_i^\sigma = v_i^\sigma$ XOR $KDF(\lvert v_i^\sigma \rvert, k_i^\sigma)$

Note: all of the exponentiations of R are **fixed-base**.

0T_DDH_ SEMIHONEST_BATCH Sender (S) Specification	
Step 1:	For every $i=1,\ldots,m$, WAIT for message (h_i^0,h_i^1) from R
Step 2:	SAMPLE a single random value $r \in \{0, \ldots, q-1\}$ and COMPUTE $u=g^r$ For every $i=1,\ldots,m$, COMPUTE: • $k_i^0 = (h_i^0)^r$ • $v_i^0 = x_i^0$ XOR $KDF(\lvert x_i^0 \rvert, k_i^0)$ • $k_i^1 = (h_i^1)^r$ • $v_i^1 = x_1$ XOR $KDF(\lvert x_1 \rvert, k_i^1)$ For every $i=1,\ldots,m$, SEND (u, v_i^0, v_i^1) to R
Step 3:	OUTPUT nothing

0T_DDH_ SEMIHONEST_BATCH Receiver (R) Specification	
Step 1:	For every $i=1,\ldots m$, SAMPLE random values $\alpha_i \in Z_q$ and $h_i \in G$ For every $i=1,\ldots,m$, COMPUTE h_i^0, h_i^1 as follows: 1. If $\sigma_i = 0$ then $h_i^0 = g^{\alpha i}$ and $h_i^1 = h_i$ 2. If $\sigma_i = 1$ then $h_i^0 = h_i$ and $h_i^1 = g^{\alpha i}$ For every $i=1,\ldots,m$, SEND (h_i^0, h_i^1) to S
Step 2:	WAIT for the message u from S For every $i=1,\ldots,m$, WAIT for the message (v_i^0, v_i^1) from S
Step 3:	For every $i=1,\ldots,m$, COMPUTE $k_i^\sigma = (u)^{\alpha i}$ For every $i=1,\ldots,m$, OUTPUT $x_i^\sigma = v_i^\sigma$ XOR $KDF(\lvert v_i^\sigma \rvert, k_i^\sigma)$

6.2 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,g_0) where (G,q,g_0) 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 $\sigma \in \{0, 1\}$
Parties' Outputs:	• S: nothing • R: x_σ

0T_DDH_FULLSIM_BATCH Protocol Specification	
Initialization Phase	
Step 1 (Both):	IF **NOT** VALID_PARAMS(G,q,g_0) REPORT ERROR and HALT
Step 2 (R):	SAMPLE random values $y,\alpha_0 \in \{0, \ldots, q-1\}$ SET $\alpha_1 = \alpha_0 + 1$ COMPUTE 1. $g_1 = (g_0)^y$ 2. $h_0 = (g_0)^{\alpha 0}$ 3. $h_1 = (g_1)^{\alpha 1}$ 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 $(g_0,g_1,h_0,h_1/g_1)$ and private input for the prover α_0. If verifier-output is REJ, REPORT ERROR (cheat attempt) and HALT
Transfer Phase (with inputs x_0,x_1 and σ)	
Step 1 (R):	SAMPLE a random value $r \in \{0, \ldots, q-1\}$ COMPUTE $g = (g_\sigma)^r$ and $h = (h_\sigma)^r$ SEND (g,h) to S
Step 2 (S):	COMPUTE (u_0,v_0) = **RAND**(g_0,g,h_0,h) COMPUTE (u_1,v_1) = **RAND**(g_1,g,h_1,h) COMPUTE $c_0 = x_0$ **XOR** $KDF(\lvert x_0\rvert,v_0)$ COMPUTE $c_1 = x_1$ **XOR** $KDF(\lvert x_1\rvert,v_1)$ SEND (u_0,c_0) and (u_1,c_1) to R OUTPUT nothing
Step 3 (R):	WAIT for (u_0,c_0) and (u_1,c_1) from S IF NOT • $u_0, u_1 \in G$, AND • c_0, c_1 are binary strings of the same length REPORT ERROR OUTPUT $x_\sigma = c_\sigma$ **XOR** $KDF(\lvert c_\sigma\rvert,(u_\sigma)^r)$

0T_DDH_ FULLSIM_BATCH Sender (S) Specification	
Initialization Phase	
Step 1:	IF **NOT** VALID_PARAMS(G,q,g_0) REPORT ERROR and HALT WAIT for message from R
Step 2:	DENOTE the values received by (g_1,h_0,h_1) Run the verifier in ZKPOK_FROM_SIGMA with Sigma protocol SIGMA_DH. Use common input $(g_0,g_1,h_0,h_1/g_1)$. 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 **(u_0,v_0) = RAND(g_0,*g*,h_0,*h*)** COMPUTE **(u_1,v_1) = RAND(g_1,*g*,h_1,*h*)** COMPUTE c_0 = x_0 **XOR** ***KDF***$(|x_0|,v_0)$ COMPUTE c_1 = x_1 **XOR** ***KDF***$(|x_1|,v_1)$ SEND **(u_0,c_0)** and **(u_1,c_1)** to R
Step 3:	OUTPUT nothing

0T_DDH_ FULLSIM_BATCH Receiver (R) Specification	
Initialization Phase	
Step 1:	SAMPLE random values $y, \alpha_0 \in \{0, \ldots, q-1\}$ SET $\alpha_1 = \alpha_0 + 1$ COMPUTE 1. $g_1 = (g_0)^y$ 2. $h_0 = (g_0)^{\alpha 0}$ 3. $h_1 = (g_1)^{\alpha 1}$ 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 **(g_0,g_1,h_0,h_1/g_1)** and private input α_0.
Transfer Phase	
Step 1:	SAMPLE a random value $r \in \{0, \ldots, q-1\}$ COMPUTE $g = (g_\sigma)^r$ and $h = (h_\sigma)^r$ SEND **(*g*,*h*)** to S
Step 2:	WAIT for messages **(u_0,c_0)** and **(u_1,c_1)** from S
Step 3:	IF NOT • $u_0, u_1 \in G$, AND • c_0, c_1 are binary strings of the same length REPORT ERROR OUTPUT $x_\sigma = c_\sigma$ **XOR** ***KDF***$(|c_\sigma|,(u_\sigma)^r)$

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.

6.3 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 $(x_1^0, x_1^1), \ldots, (x_m^0, x_m^1)$; 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 $\sigma_1,\ldots, \sigma_m \in \{0, 1\}$
Parties' Outputs:	• S: nothing • R: $x_1^{\sigma_1}, \ldots, x_m^{\sigma_m}$

0T_ SEMIHONEST_EXTENSION Protocol Specification	
Initialization Phase	
Step 1 (Both):	S: SAMPLE a random string $s \in \{0,1\}^n$ of length ***n***; denote it $s_1,\ldots,s_n$ R: SAMPLE ***n*** random strings $T_1,\ldots,T_n \in \{0,1\}^m$ each of length ***m*** and SAMPLE a random choice string $\tau=\tau_1,\ldots, \tau_m$
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): $(T_i, T_i \text{ XOR } \tau)$ • S (playing receiver): s_i

Step 3 (S):	• Denote the output of S in the OT executions by $\mathbf{Q_1,\ldots,Q_n}$ • Let **Q** be the matrix with ***m*** rows and ***n*** columns: $[Q_1 \mid Q_2 \mid \ldots \mid Q_n]$ 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: $[T_1 \mid T_2 \mid \ldots \mid T_n]$ and denote by ***T[i]*** the i^{th} row of ***Q*** (of length ***n***)
Transfer Phase *i* (with inputs (x_i^0, x_i^1) and σ_i)	
Step 1 (R):	If $\sigma_i = \tau_i$ then send ***b=0***; else send ***b=1***
Step 2 (S):	WAIT to receive a bit ***b*** IF ***b=0*** then SEND $y_i^0 = x_i^0 \oplus H(i, Q[i])$ and $y_i^1 = x_i^1 \oplus H(i, Q[i] \oplus s)$ IF ***b=1*** then SEND $y_i^0 = x_i^1 \oplus H(i, Q[i])$ and $y_i^1 = x_i^0 \oplus H(i, Q[i] \oplus s)$
Step 3 (R):	WAIT for (y_i^0, y_i^1) from S OUTPUT $z_i = y_i^{\tau_i} \oplus H(i, T[i])$

0T_ SEMIHONEST_EXTENSION Sender (S) Specification	
Initialization Phase	
Step 1:	SAMPLE a random string $s \in \{0,1\}^n$ of length ***n***; denote it $s_1,\ldots,s_n$
Step 2:	For ***i* = 1 to *n***, RUN OT_SEMIHONEST as the receiver with input s_i
Step 3:	• Denote the output from the OT executions by $\mathbf{Q_1,\ldots,Q_n}$ • Let **Q** be the matrix with ***m*** rows and ***n*** columns: $[Q_1 \mid Q_2 \mid \ldots \mid Q_n]$ and denote by ***Q[i]*** the i^{th} row of ***Q*** (of length ***n***)
Transfer Phase *i* (with inputs (x_i^0, x_i^1) and σ_i)	
Step 1:	WAIT to receive a bit ***b*** IF ***b=0*** then SEND $y_i^0 = x_i^0 \oplus H(i, Q[i])$ and $y_i^1 = x_i^1 \oplus H(i, Q[i] \oplus s)$ IF ***b=1*** then SEND $y_i^0 = x_i^1 \oplus H(i, Q[i])$ and $y_i^1 = x_i^0 \oplus H(i, Q[i] \oplus s)$ OUTPUT nothing

0T_ SEMIHONEST_EXTENSION Receiver (R) Specification	
Initialization Phase	
Step 1:	SAMPLE ***n*** random strings $T_1,\ldots,T_n \in \{0,1\}^m$ each of length ***m*** and SAMPLE a random choice string $\tau = \tau_1,\ldots, \tau_m$
Step 2:	For ***i* = 1 to *n***, RUN OT_SEMIHONEST as the sender, with input ***(T_i,T_i XOR τ)***
Step 3:	Let **T** be the matrix with ***m*** rows and ***n*** columns: $[T_1 \mid T_2 \mid \ldots \mid T_n]$ and denote by ***T[i]*** the i^{th} row of ***Q*** (of length ***n***)
Transfer Phase *i* (with inputs (x_i^0, x_i^1) and σ_i)	
Step 1:	If $\sigma_i = \tau_i$ then send ***b=0***; else send ***b=1***
Step 2:	WAIT for (y_i^0, y_i^1) from S OUTPUT $z_i = y_i^{\tau_i} \oplus H(i, T[i])$

7 Coin Tossing

7.1 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 P_1 and Party P_2
Parties' Inputs:	None
Parties' Outputs:	The same bit b

COIN_TOSSING_BLUM Protocol Specification	
Step 1 (both):	P_1: SAMPLE a random bit $b_1 \in \{0,1\}$ P_2: SAMPLE a random bit $b_2 \in \{0,1\}$
Step 2 (P_1):	RUN subprotocol COMMIT.commit on b_1
Step 3 (P_2):	SEND b_2 to P_1
Step 4 (P_1):	RUN subprotocol COMMIT.decommit to reveal b_1 IF COMMIT.decommit returns INVALID REPORT ERROR (cheat attempt)
Step 5 (Both)	OUTPUT b_1 XOR b_2

COIN_TOSSING_BLUM Party P_1 Specification	
Step 1:	SAMPLE a random bit $b_1 \in \{0,1\}$
Step 2:	RUN subprotocol COMMIT.commit on b_1
Step 3:	WAIT for a bit b_2 from P_2
Step 4:	RUN subprotocol COMMIT.decommit to reveal b_1
Step 5	OUTPUT b_1 XOR b_2

COIN_TOSSING_BLUM Party P_2 Specification	
Step 1:	SAMPLE a random bit $b_2 \in \{0,1\}$
Step 2:	WAIT for COMMIT.commit on b_1
Step 3:	SEND b_2 to P_1
Step 4:	RUN subprotocol COMMIT.decommit to receive b_1
Step 5	IF COMMIT.decommit returns INVALID REPORT ERROR (cheat attempt) ELSE OUTPUT b_1 XOR b_2

7.2 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 P_1 and Party P_2
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, s_1 and s_2 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):	P_1: SAMPLE a random ***L***-bit string $s_1 \in \{0,1\}^L$ P_2: SAMPLE a random ***L***-bit string $s_2 \in \{0,1\}^L$
Step 2 (P_1):	RUN subprotocol COMMIT.commit on s_1
Step 3 (both):	RUN ZKPOK_FROM_SIGMA applied to a SIGMA protocol that P_1 knows the committed value s_1.

	If the verifier output is REJ, then P_2 HALTS and REPORTS ERROR.
Step 4 (P_2):	SEND s_2 to P_1
Step 5 (P_1):	P_1 sends s_1 to P_2 (but does not run decommit)
Step 6 (both):	RUN ZK_FROM_SIGMA applied to a SIGMA protocol that the committed value was s_1. If the verifier output is REJ, then P_2 HALTS and REPORTS ERROR.
Step 7 (both):	OUTPUT s_1 XOR s_2

COIN_TOSSING_STRING Party P_1 Specification	
Step 1:	SAMPLE a random L-bit string $s_1 \in \{0,1\}^L$
Step 2:	RUN subprotocol COMMIT.commit on s_1
Step 3:	RUN the prover in a ZKPOK_FROM_SIGMA applied to a SIGMA protocol that P_1 knows the committed value s_1
Step 4:	WAIT for an L-bit string s_2 from P_2
Step 5:	SEND b_1 to P_2
Step 6:	RUN the prover in ZK_FROM_SIGMA applied to a SIGMA protocol that the committed value was s_1
Step 7:	OUTPUT s_1 XOR s_2

COIN_TOSSING_STRING Party P_2 Specification	
Step 1:	SAMPLE a random L-bit string $s_2 \in \{0,1\}^L$
Step 2:	WAIT to receive a COMMIT.commit from P_1
Step 3:	RUN the verifier in a ZKPOK_FROM_SIGMA applied to a SIGMA protocol that P_1 knows the committed value. If the verifier output is REJ, then HALT and REPORT ERROR.
Step 4:	SEND s_2 to P_1
Step 5:	WAIT for an L-bit string s_1 from P_1
Step 6:	RUN the verifier in ZK_FROM_SIGMA applied to a SIGMA protocol that the committed value was s_1. If the verifier output is REJ, then HALT and REPORT ERROR.
Step 7:	OUTPUT s_1 XOR s_2

7.3 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 P_1 is corrupted and fulfills a definition of "pseudorandomness" when P_2 is corrupted.
References:	Implicit in [Goldreich-Kahan]

COIN_TOSSING_SEMI Protocol Parameters	
Parties' Identities:	Party P_1 and Party P_2
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):	P_1: SAMPLE a random ***L***-bit string $s_1 \in \{0,1\}^L$ P_2: SAMPLE a random ***L***-bit string $s_2 \in \{0,1\}^L$
Step 2 (both):	RUN subprotocol COMMIT_PERFECT_HIDING.commit on s_1 with P_1 as the committer
Step 3 (both):	RUN subprotocol COMMIT_PERFECT_BINDING.commit on s_2 with P_2 as the committer
Step 4 (both):	RUN subprotocol COMMIT_PERFECT_HIDING.decommit to reveal s_1
Step 5 (both):	RUN subprotocol COMMIT_PERFECT_BINDING.decommit to reveal s_2
Step 6 (both):	OUTPUT s_1 XOR s_2

COIN_TOSSING_STRING Party P_1 Specification	
Step 1:	SAMPLE a random ***L***-bit string $s_1 \in \{0,1\}^L$
Step 2:	RUN the committer in subprotocol COMMIT_PERFECT_HIDING.commit on s_1

Step 3:	RUN the receiver in subprotocol COMMIT_PERFECT_BINDING.commit
Step 4:	RUN the committer in subprotocol COMMIT_PERFECT_HIDING.decommit to reveal s_1
Step 5:	RUN the receiver in subprotocol COMMIT_PERFECT_BINDING.decommit to receive s_2
Step 6:	OUTPUT s_1 XOR s_2

COIN_TOSSING_STRING Party P_2 Specification	
Step 1:	SAMPLE a random L-bit string $s_2 \in \{0,1\}^L$
Step 2:	RUN the receiver in subprotocol COMMIT_PERFECT_HIDING.commit
Step 3:	RUN the committer in subprotocol COMMIT_PERFECT_BINDING.commit on s_2
Step 4:	RUN the receiver in subprotocol COMMIT_PERFECT_HIDING.decommit to receive s_1
Step 5:	RUN the committer in subprotocol COMMIT_PERFECT_BINDING.decommit to reveal s_2
Step 6:	OUTPUT s_1 XOR s_2

8 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 P_1 and Party P_2
Parties' Inputs:	• Common input: (***G,q,g***) where (***G,q, g***) is a DLOG description and a parameter ***m*** determining the input length • P_1 's private input: $k = (g^{a_0}, a_1, \ldots, a_m)$ where $a_0, a_1, \ldots, a_m \in Z_q$ are random • P_2's private input: $x = x_1, \ldots, x_m$ of length *m*
Parties' Outputs:	• P_1: nothing • P_2: $y = g^{a_0 \cdot \prod_{i=1}^{m} a_i^{x_i}}$

Protocol Specification	
Step 1 (Both):	BOTH: If NOT VALID_PARAMS(***G,q,g***) then REPORT ERROR P_1: SAMPLE random values $r_1, \ldots, r_m \in Z_q$
Step 2 (Both):	RUN ***m*** OT executions for any OT (for efficiency this should be a BATCH_OT). The i^{th} execution for $i=1,\ldots,m$ is run as follows: P_1 runs the sender (S) with input pair $(r_i, r_i \cdot a_i)$ (with multiplication in Z^*_q) P_2 runs the Receiver (R) input bit x_i
Step 3 (Both):	IF the output of any of the oblivious transfers is $\perp$ REPORT ERROR
Step 4 (P_2):	LET $y^1_{x1}, \ldots, y^m_{xm}$ be the outputs of the OT executions. If there exists an i such that $y^i_{xi} \notin Z^*_q$ then SET $y^i_{xi} = 1$
Step 5 (P_1):	COMPUTE $\tilde{g} = g^{a_0 \cdot \prod_{i=1}^{m} 1/r_i}$ SEND $\tilde{g}$ to P_2
Step 6 (Both):	P_1: OUTPUT nothing P_2: IF $\tilde{g}$ is not of order ***q*** (for a prime order group, this is equivalent to saying that $\tilde{g} \in G$ and $\tilde{g} \neq 1$), then REPORT ERROR COMPUTE $y = \tilde{g}^{\prod_{i=1}^{m} y^i_{x_i}}$ OUTPUT y

Party P_1 Specification	
Step 1:	If NOT VALID_PARAMS(***G,q,g***) then REPORT ERROR P_1: SAMPLE random values $r_1,\ldots, r_m \in Z_q$
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 x_i
Step 3:	IF the output of any of the oblivious transfers is ⊥ REPORT ERROR
Step 4:	COMPUTE $\tilde{g} = g^{a_0 \cdot \prod_{i=1}^{m} 1/r_i}$ SEND $\tilde{g}$ to P_2
Step 5:	OUTPUT nothing

Party P_2 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 $(r_i, r_i \cdot a_i)$ (with multiplication in Z^*_q)
Step 3:	IF the output of any of the oblivious transfers is ⊥ REPORT ERROR
Step 4:	LET $y^1_{x1},\ldots, y^m_{xm}$ be the outputs of the OT executions. If there exists an ***i*** such that $y^i_{xi} \notin Z^*_q$ then SET $y^i_{xi} = 1$
Step 5:	WAIT for message $\tilde{g}$ from P_1 IF $\tilde{g}$ is not of order ***q*** (for a prime order group, this is equivalent to saying that $\tilde{g} \in G$ and $\tilde{g} \neq 1$), then REPORT ERROR
Step 6:	COMPUTE $y = \tilde{g}^{\prod_{i=1}^{m} y^i_{x_i}}$ OUTPUT y

9 Key Exchange Protocols

9.1 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 P_1 and P_2
Common parameters:	None
Parties' Inputs:	Both parties have a pairs of symmetric keys K_1,K_2, where K_1 is for encryption and K_2 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 $K_{ENC} = K_1$ and $K_{MAC} = K_2$

9.2 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 P_1 and P_2
Common parameters:	A DLOG group description ($\boldsymbol{G,q,g}$) and integers $\boldsymbol{L_{ENC}}$ and $\boldsymbol{L_{MAC}}$ which are the respective lengths of the encryption and MAC keys to be generated.
Parties' Inputs:	None
Parties' Outputs:	• An encryption key K_{ENC} • A MAC key K_{MAC}

KEY_EXCHANGE_DH Protocol Specification	
Step 1 (P_1):	SAMPLE a random $\boldsymbol{a \leftarrow Z_q}$ COMPUTE $\boldsymbol{h_1 = g^a}$ SEND $\boldsymbol{h_1}$ to P_2
Step 2 (P_2):	SAMPLE a random $\boldsymbol{b \leftarrow Z_q}$ COMPUTE $\boldsymbol{h_2 = g^b}$ SEND $\boldsymbol{h_2}$ to P_1
Step 3 (P_1):	WAIT for $\boldsymbol{h_2}$ from P_2 COMPUTE $\boldsymbol{(K1\ ,K2) = KDF(L_{ENC}+L_{MAC}, (h_2)^a)}$ OUTPUT $\boldsymbol{K_{ENC} = K_1}$ and $\boldsymbol{K_{MAC} = K_2}$
Step 4 (P_2):	WAIT for $\boldsymbol{h_1}$ from P_1 COMPUTE $\boldsymbol{(K_1\ ,K_2) = KDF(L_{ENC}+L_{MAC}, (h_1)^b)}$ OUTPUT $\boldsymbol{K_{ENC} = K_1}$ and $\boldsymbol{K_{MAC} = K_2}$

KEY_EXCHANGE_DH Party P_1 Specification	
Step 1:	SAMPLE a random $\boldsymbol{a \leftarrow Z_q}$ COMPUTE $\boldsymbol{h_1 = g^a}$ SEND $\boldsymbol{h_1}$ to P_2
Step 2:	WAIT for $\boldsymbol{h_2}$ from P_2 COMPUTE $\boldsymbol{(K_1\ ,K_2) = KDF(L_{ENC}+L_{MAC}, (h_2)^a)}$ OUTPUT $\boldsymbol{K_{ENC} = K_1}$ and $\boldsymbol{K_{MAC} = K_2}$

KEY_EXCHANGE_DH Party P_2 Specification	
Step 1:	SAMPLE a random $\boldsymbol{b \leftarrow Z_q}$ COMPUTE $\boldsymbol{h_2 = g^b}$ SEND $\boldsymbol{h_2}$ to P_1
Step 2:	WAIT for $\boldsymbol{h_1}$ from P_1 COMPUTE $\boldsymbol{(K_1, K_2) = KDF(L_{ENC}+L_{MAC}, (h_1)^b)}$ OUTPUT $\boldsymbol{K_{ENC} = K_1}$ and $\boldsymbol{K_{MAC} = K_2}$

9.3 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 P_1 and P_2
Common parameters:	• Public keys (pk_1,pk_2) • 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:	• P_1 has private key sk_1 (associated with pk_1) • P_2 has private key sk_2 (associated with pk_2)
Parties' Outputs:	• An encryption key K_{ENC} • A MAC key K_{MAC} • The session identifier ***sid***

KEY_EXCHANGE_UCDH Protocol Specification	
Step 1 (P_1):	SAMPLE a random $a \leftarrow Z_q$ COMPUTE $h_1 = g^a$ SEND (P_1,sid,h_1) to P_2
Step 2 (P_2):	WAIT for (P_1,sid,h_1) from P_1 SAMPLE a random $b \leftarrow Z_q$ COMPUTE $h_2 = g^b$ COMPUTE $s = Sign(sk_2,(P_2,sid,h_2,h_1,P_1))$ COMPUTE $h_3 = h_1^b$ ERASE b SEND (P_2,sid,h_2,s) to P_1
Step 3 (P_1):	WAIT for (P_2,sid,h_2,s) from P_2 IF $Verify(pk_2,(P_2,sid,h_2,h_1,P_1),s) = false$ [Verify(key,message,signature)] OUTPUT FAIL (cheat attempt) COMPUTE $s' = Sign(sk_1,(P_1,sid,h_1,h_2,P_2))$ COMPUTE $h_3 = h_2^a$ ERASE a COMPUTE $(K_1,K_2) = KDF(L_{ENC}+L_{MAC}, h_3)$ SEND (P_1,sid,s') OUTPUT sid, $K_{ENC} = K_1$ and $K_{MAC} = K_2$

Step 4 (P_2):	WAIT for (P_1,sid,s') from P_1 IF $\mathbf{Verify}(pk_1,(P_1,sid,h_1,h_1,P_2),s') = \mathbf{false}$ OUTPUT FAIL (cheat attempt) COMPUTE $(K_1,K_2) = \mathbf{KDF}(L_{ENC}+L_{MAC}, h_3)$ OUTPUT sid, $K_{ENC} = K_1$ and $K_{MAC} = K_2$	[Verify(key,message,signature)]

KEY_EXCHANGE_UCDH Party P_1 Specification		
Step 1:	SAMPLE a random $a \leftarrow Z_q$ COMPUTE $h_1 = g^a$ SEND (P_1,sid,h_1) to P_2	
Step 2:	WAIT for (P_2,sid,h_2,s) from P_2 IF $\mathbf{Verify}(pk_2,(P_2,sid,h_2,h_1,P_1),s) = \mathbf{false}$ OUTPUT FAIL (cheat attempt) COMPUTE $s' = \mathbf{Sign}(sk_1,(P_1,sid,h_1,h_2,P_2))$ COMPUTE $h_3 = h_2^a$ ERASE a COMPUTE $(K_1,K_2) = \mathbf{KDF}(L_{ENC}+L_{MAC}, h_3)$ SEND (P_1,sid,s') OUTPUT sid, $K_{ENC} = K_1$ and $K_{MAC} = K_2$	[Verify(key,message,signature)]

KEY_EXCHANGE_UCDH Party P_2 Specification		
Step 1:	WAIT for (P_1,sid,h_1) from P_1 SAMPLE a random $b \leftarrow Z_q$ COMPUTE $h_2 = g^b$ COMPUTE $s = \mathbf{Sign}(sk_2,(P_2,sid,h_2,h_1,P_1))$ COMPUTE $h_3 = h_1^b$ ERASE b SEND (P_2,sid,h_2,s) to P_1	
Step 2:	WAIT for (P_1,sid,s') from P_1 IF $\mathbf{Verify}(pk_1,(P_1,sid,h_1,h_1,P_2),s') = \mathbf{false}$ OUTPUT FAIL (cheat attempt) COMPUTE $(K_1,K_2) = \mathbf{KDF}(L_{ENC}+L_{MAC}, h_3)$ OUTPUT sid, $K_{ENC} = K_1$ and $K_{MAC} = K_2$	[Verify(key,message,signature)]

Note: In the above Pseudocode specification, values P_1 and P_2 are used. These are identifiers of the parties (the IP address suffices).

10 Secure Channel

10.1 Basic Secure Channel

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

10.2 Adaptive Secure Channel (with Erasures)

Release 2

11 Special Encryption Schemes

11.1 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 P_1 (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 P_1 (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 $N=n^s$, and $N' = n^{s+1}$. • Plaintext $x \in Z_N$
Outputs:	Ciphertext $c \in Z_{N'}$

DJ_ENCRYPTION Protocol Specification	
Step 1:	CHOOSE a random r in $Z_{N'}^*$ (This can be done by choosing a random value between 1 and $N'-1$, which is with overwhelming probability in $Z_{N'}^*$.) COMPUTE $c = (1+n)^x r^N \bmod N'$. 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 P_1 (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 $N=n^s$, and $N' = n^{s+1}$. • Private key ***t*** • Ciphertext $c \in Z_{N'}$
Outputs:	Plaintext $x \in Z_N$

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 $c^d \bmod N'$. COMPUTE x as the discrete logarithm of c^d to the base ***(1+n)*** modulo ***N'***. This is done by the following computation $a=c^d$ $x=0$ *for j = 1 to s do* *begin* $t1= ((a \bmod n^{j+1}) - 1) / n$ $t2 = x$ *for k = 2 to j do* *begin* $x = x - 1$ $t2 = t2 * x \bmod n^j$ $t1 = (t1 - (t2 * n^{k-1}) / factorial(k)) \bmod n^j$ *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 P_1 (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 $N=n^s$, and $N' = n^{s+1}$. • Ciphertexts $c1,c2 \in Z_{N'}$
Outputs:	Ciphertext $c \in Z_{N'}$ (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 $c = c1*c2 \bmod N'$ CHOOSE a random r in $Z_{N'}^*$ (This can be done by choosing a random value between 1 and $N'-1$, which is with overwhelming probability in $Z_{N'}^*$.) COMPUTE $c = c * r^N \bmod N'$ 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 P_1 (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 $N=n^s$, and $N' = n^{s+1}$. • Ciphertexts $c1,c2 \in Z_{N'}$
Outputs:	Ciphertext $c \in Z_{N'}$ (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 $c = c1*c2 \bmod N'$ /* 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 P_1 (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 $N=n^s$, and $N' = n^{s+1}$. • Ciphertext $c1 \in Z_{N'}$ • $a \in Z_N$
Outputs:	Ciphertext $c \in Z_{N'}$ (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 $c = (c1)^a \bmod N'$ CHOOSE a random r in $Z_{N'}^*$ (This can be done by choosing a random value between 1 and $N'-1$, which is with overwhelming probability in $Z_{N'}^*$.) COMPUTE $c = c * r^N \bmod N'$ 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 P_1 (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 $N=n^s$, and $N' = n^{s+1}$. • Ciphertext $c1 \in Z_{N'}$ • $a \in Z_N$
Outputs:	Ciphertext $c \in Z_{N'}$ (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 $c = (c1)^a \bmod N'$ /* the only difference from DJ_ADD is that two lines of code were removed here */ OUTPUT ***c***

11.2 Attribute-Based Encryption (Release 2)

11.3 Identity-Based Encryption (Release 2)

12 Non-Interactive Primitives

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

12.1 Random Oracle

Protocol Name:	Random Oracle
Protocol Reference:	RANDOM_ORACLE
Protocol Type:	A method achieving a "random oracle" type behavior
Protocol Description:	HMAC with a fixed key
References:	The soundness of this approach is based on the paper "To Hash or Not to Hash Again? (In)differentiability Results for H^2 and HMAC" by Dodis et al. that appeared at CRYPTO 2012.

RANDOM_ORACLE Protocol Parameters	
Parties' Identities:	Party P_1 (this is a non-interactive function)
Common parameters:	• A parameter ***L*** determining how many bits to obtain
Parties' Inputs:	• An input ***x*** of arbitrary length
Parties' Outputs:	• A string ***y*** of length ***L***

RANDOM_ORACLE Protocol Specification	
Step 1:	• Call HKDF with input string ***SKM=x***, optional string ***CTXinfo="RandomOracle"***, and with output length parameter ***L***

12.2 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 P_1 (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 $\mathbf{R_1}$ CALL random generator of operating system; denote the output $\mathbf{R_2}$ Examples from Windows operating systems: • In Windows XP, use CAPI and the CryptGenRandom() function • In Windows Vista and above, use CNG and BCryptGenRandom() OUTPUT $\mathbf{R_1} \oplus \mathbf{R_2}$

12.3 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 P_1 (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 $L \cdot m > outlen$, where ***L*** is the output length of ***F***. FOR $i = 1$ to m COMPUTE $Y_i = F(k,(x,outlen,i))$ [key=k, data=(x,outlen,i)] OUTPUT the first ***outlen*** bits of $Y_1,\ldots,Y_m$

12.4 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 P_1 (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 $\mathbf{x_{1,0},x_{1,1},\ldots,x_{L,0},x_{L,1}}$ • For $\boldsymbol{i=1,\ldots,L}$, compute $\mathbf{y_{i,0}=H(x_{i,0})}$ and $\mathbf{y_{i,1}=H(x_{i,1})}$
Output (keys):	• Public-key **pk**: $\mathbf{y_{1,0},y_{1,1},\ldots,y_{L,0},y_{L,1}}$ • Private-key **sk**: $\mathbf{x_{1,0},x_{1,1},\ldots,x_{L,0},x_{L,1}}$

HASH_ONE-TIME_SIG Sign	
Parties' Identities:	Party P_1 (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 $\boldsymbol{sk = x_{1,0},x_{1,1},\ldots,x_{L,0},x_{L,1}}$ • An input message ***m*** (of any length)
Signing alg:	• Compute **z=H(m)**; let $\mathbf{z_1,\ldots,z_L}$ be the bits of **z** • Output the signature $\mathbf{x_{1,z1},\ldots,x_{L,zL}}$

HASH_ONE-TIME_SIG Verify	
Parties' Identities:	Party P_1 (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 $\boldsymbol{pk = y_{1,0},y_{1,1},\ldots,y_{L,0},y_{L,1}}$ • An input message ***m*** (of any length) • A signature $\mathbf{x_{1,z1},\ldots,x_{L,zL}}$
Signing alg:	• Compute **z=H(m)**; let $\mathbf{z_1,\ldots,z_L}$ be the bits of **z** • For $\boldsymbol{i=1,\ldots,L}$, verify that $\mathbf{y_{i,zi}=H(x_{i,zi})}$ • Output **ACCEPT** if all are equal; else output **REJECT**

12.5 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 P_1 (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 L_0 be the first $\|x\|/2$ bits of x Let R_0 be the second $\|x\|/2$ bits of x FOR **i = 1 to 4** SET $L_i = R_{i-1}$ COMPUTE $R_i = L_{i-1} \oplus$ **PRF_VARY_INOUT**$(k,(R_{i-1},i),L)$ [key=k, data=(R_{i-1},i), outlen = L] OUTPUT (L_4,R_4)

12.6 Naor-Reingold Pseudorandom Function

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

12.7 Universal One-Way Hashing

12.8 Universal Hash Functions

Protocol Name:	Universal hash function
Protocol Reference:	Evaluation Hash Function
Protocol Type:	ε -AXU family over GF[2^8] and $t \leq 2^{24}$. These parameters give a value of ε of $t/2^n$ (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 P_1 (this is a non-interactive function)
Common parameters:	• A finite field **GF(2^{64})** 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(2^{64})***
Parties' Outputs:	• The hash result of ***M(α)·α ∈ GF(2^{64})***

Universal hash function	
Step 1:	• The field **GF(2^{64})** is represented as ***GF(2)[x]/f(x)***, with ***f(x) = $x^{64} + x^4 + x^3 + x + 1$***. ***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(2^{64})*** as follows. Every 64 bits of ***m*** are viewed as an element in ***GF(2^{64})***. 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(2^{64})*** [key=***α***, data= ***M(x)***] as follows Evaluate the polynomial ***M(x)*** on ***α*** to get ***M(α) ∈ GF(2^{64})***

Multiply by $\alpha \in GF(2^{64})$ to get $M(\alpha)\cdot\alpha \in GF(2^{64})$

12.9 Information-Theoretic MAC

12.10 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 P_1 (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*)**