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

11/6/2013

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.

1 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.

1.1 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* mod *q***) OUTPUT ***(a,e,z)***

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

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

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

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

1.3 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 $((a_1,\ldots,a_m),e,z)$

1.4 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 3.1 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 <u>IFF</u> 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 **($a,e,(u,v)$)**

1.5 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 3.1 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 1.15): same as SIGMA_DLOG with inputs appropriately defined.

1.6 Sigma Protocol for El Gamal Commitment Knowledge (SIGMA_ELGAMAL_COMMIT)

Note that an ElGamal commitment to $\boldsymbol{x}$ is a tuple $(\boldsymbol{h},\boldsymbol{c_1},\boldsymbol{c_2})$ where $\boldsymbol{h}$ is a public key, and $(\boldsymbol{c_1},\boldsymbol{c_2})$ are an encryption of $\boldsymbol{x}$.

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 $(\boldsymbol{h},\boldsymbol{c_1}, \boldsymbol{c_2})$
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 $(\boldsymbol{G},\boldsymbol{q},\boldsymbol{g})$ and a soundness parameter $\boldsymbol{t}$ such that $\boldsymbol{2^t < q}$
Parties' Inputs:	• Common input statement: $(\boldsymbol{h},\boldsymbol{c_1},\boldsymbol{c_2})$ • P's private input: a value $\boldsymbol{w} \in \boldsymbol{Z_q}$ such that $\boldsymbol{h = g^w}$ (given $\boldsymbol{w}$ 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 $(\boldsymbol{G},\boldsymbol{q},\boldsymbol{g})$ and **t** • Common input: $\boldsymbol{h}$ (1st element of commitment) • P's private input: a value $\boldsymbol{w} \in \boldsymbol{Z_q}$ such that $\boldsymbol{h = g^w}$

Sigma-Protocol Simulator (see Section 1.15): same as SIGMA_DLOG with inputs appropriately defined.

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

Note that an ElGamal commitment to $\boldsymbol{x}$ is a tuple $\boldsymbol{(h,c_1,c_2)}$ where $\boldsymbol{h}$ is a public key, and $\boldsymbol{(c_1,c_2)}$ are an encryption of $\boldsymbol{x}$.

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 $\boldsymbol{(h,c_1, c_2)}$ is $\boldsymbol{x}$
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 $\boldsymbol{(G,q,g)}$ and a soundness parameter $\boldsymbol{t}$ such that $\boldsymbol{2^t < q}$
Parties' Inputs:	• Common input statement: $\boldsymbol{(h,c_1,c_2)}$ and $\boldsymbol{x}$ • P's private input: a value $\boldsymbol{r} \in \boldsymbol{Z_q}$ such that $\boldsymbol{c_1=g^r}$ and $\boldsymbol{c_2 = h^r \cdot x}$
Parties' Outputs:	• P: nothing • V: ACC or REJ

SIGMA_COMMITTED_VALUE_ELGAMAL Protocol Specification	
Specification:	RUN SIGMA_DH with: • Common parameters $\boldsymbol{(G,q,g)}$ and **t** • Common input: $\boldsymbol{(g,h,u,v) = (g,h,c_1,c_2/x)}$ • P's private input: a value $\boldsymbol{r} \in \boldsymbol{Z_q}$ such that $\boldsymbol{c_1=g^r}$ and $\boldsymbol{c_2/x = h^r}$

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

Sigma-Protocol Simulator (see Section 1.15): same as SIGMA_DH with inputs appropriately defined.

1.8 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 ($\boldsymbol{G,q,g}$) and a soundness parameter $\boldsymbol{t}$ such that $\boldsymbol{2^t < q}$
Parties' Inputs:	• Common input statement: an ElGamal public-key $\boldsymbol{h}$ • P's private input: a value $\boldsymbol{w} \in \boldsymbol{Z_q}$ such that $\boldsymbol{h=g^w}$
Parties' Outputs:	• P: nothing • V: ACC or REJ

SIGMA_SK_ELGAMAL Protocol Specification	
Specification:	RUN SIGMA_DLOG with: • Common parameters **(**$\boldsymbol{G,q,g}$**)** • Common input: $\boldsymbol{h}$ (the public key) • P's private input: a value $\boldsymbol{w} \in \boldsymbol{Z_q}$ such that $\boldsymbol{h=g^w}$

Sigma-Protocol Simulator (see Section 1.15): same as SIGMA_DLOG with inputs appropriately defined.

1.9 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

1.9.1 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 1.15): same as SIGMA_DH with inputs appropriately defined.

1.9.2 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 1.15): same as SIGMA_DH with inputs appropriately defined.

1.10 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 1.15): same as SIGMA_EXTEND_DH with inputs appropriately defined.

1.11 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 $m=n^{-1} \bmod \varnothing(n)=n^{-1} \bmod (p-1)(q-1)$. 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.

1.12 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.

1.13 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 $t < \|n\|/3$
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. $c_i=(1+n)^{xi} \cdot r_i^N \bmod N'$ • 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 <u>IFF</u> $c_1,c_2,c_3,a_1,a_2,z_1,z_2,z_3$ 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 $z=(z_1,z_2,z_3)$

<u>NOTE:</u> 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.

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

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

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

<u>NOTE</u>: The result of this transformation on Sigma protocols is a Sigma protocol.

1.15 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.

1.16 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 $\mathbf{GF[2^t]}$ (let $\mathbf{1,\ldots,n}$ 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 $\mathbf{j \notin I}$, SAMPLE a random element $\mathbf{e_j \in GF[2^t]}$ For every $\mathbf{j \notin I}$, RUN the simulator on statement $\mathbf{x_j}$ and challenge $\mathbf{e_j}$ to get transcript $\mathbf{(a_j,e_j,z_j)}$ For every $\mathbf{i \in I}$, RUN the prover P on statement $\mathbf{x_i}$ to get first message $\mathbf{a_i}$ SET $\mathbf{a=(a_1,\ldots,a_n)}$
Verifier challenge (e):	WAIT for messages $\mathbf{a_1,\ldots,a_n}$ SAMPLE a single random challenge $e \in \mathbf{GF[2^t]}$
Prover msg 2 $(Q,e_1,z_1,\ldots,e_n,z_n)$:	INTERPOLATE the points $\mathbf{(0,e)}$ and $\mathbf{\{(j,ej)\}}$ for every $\mathbf{j \notin I}$ to obtain a degree $n\text{-}k$ polynomial $\mathbf{Q}$ (s.t. $\mathbf{Q(0)=e}$ and $\mathbf{Q(j)=e_j}$ for every $\mathbf{j \notin I}$) For every $\mathbf{i \in I}$, SET $e_i = Q(i)$ For every $\mathbf{i \in I}$, 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 $n\text{-}k$ AND $\mathbf{Q(i)=e_i}$ for all $\mathbf{i=1,\ldots,n}$ AND $\mathbf{Q(0)=e}$, and the verifier output on $\mathbf{(a_i,e_i,z_i)}$ for all $\mathbf{i=1,\ldots,n}$ is ACC

OR_MANY_SIGMA Simulator (*M*) Specification	
Input:	A series of ***n*** statements {x_i} and a challenge ***e*** ∈ **GF[2^t]**
Computation:	SAMPLE random points $e_1,\ldots,e_{n-k}$ ∈ **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,…,n** to obtain **(a_i,e_i,z_i)**. OUTPUT **(a_1,e_1,z_1),…, (a_n,e_n,z_n)**

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

2 Zero-knowledge

2.1 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 $\pi$, 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 $\pi$
                    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 $\pi$ on input $x$
                    OUTPUT ACC
                ELSE
                    OUTPUT REJ
```

ZK_FROM_SIGMA Prover (P) Specification	
Step 1:	RUN the receiver in COMMIT.commit with V as the committer
Step 2:	COMPUTE the first message ***a*** in π, using **(*x*,*w*)** as input SEND ***a*** to V
Step 3:	RUN the receiver in COMMIT.decommit with V as the committer

Step 4:	IF COMMIT.decommit returns some ***e***
	    COMPUTE the response ***z*** to **(*a*,*e*)** according to π
	    SEND ***z*** to V
	    OUTPUT nothing
	ELSE (IF COMMIT.decommit returns INVALID)
	    OUTPUT ERROR (CHEAT_ATTEMPT_BY_V)

ZK_FROM_SIGMA Verifier (V) Specification	
Step 1:	SAMPLE a random challenge $e \in \{0,1\}^t$
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.

2.2 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_WITH_TRAPDOOR_PEDERSEN and COMMIT_ WITH_TRAPDOOR _HASH_PEDERSEN schemes.

Currently the implementation does not allow the user choose the commitment scheme he wants, but always use COMMIT_WITH_TRAPDOOR_PEDERSEN.

2.3 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 Commitment schemes

3.1 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.

3.2 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*)**.

3.3 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 ($\boldsymbol{G,q,g}$)
Parties' Inputs:	C's private input: a value $\boldsymbol{x} \in \boldsymbol{Z_q}$
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 $\boldsymbol{x} \in \boldsymbol{Z_q}$

This is identical to COMMIT_PEDERSEN except that the receiver outputs the value $\boldsymbol{a}$ used in the first (preprocessing) step of COMMIT_PEDERSEN. This value $\boldsymbol{a}$ is called the trapdoor.

3.4 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 $x \in G$ (<u>Important</u>: 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.

3.5 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*)**.

3.6 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 $\boldsymbol{n}$
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 $\boldsymbol{c} = \boldsymbol{H}(\boldsymbol{r},\boldsymbol{x})$ (c concatenated with r) SEND $\boldsymbol{c}$ to R
Step 2 (R):	WAIT for a value $\boldsymbol{c}$ STORE $\boldsymbol{c}$
Decommit phase	
Step 1 (C):	SEND $(\boldsymbol{r}, \boldsymbol{x})$ to R

Step 2 (R):	IF **NOT**
	  • $\boldsymbol{c} = \boldsymbol{H}(\boldsymbol{r},\boldsymbol{x})$, 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***

3.7 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 $\boldsymbol{x} \in \{0, 1\}^t$
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_EQUIVOCAL Protocol Specification	
Commit phase	
Step 1 (Both):	RUN any COMMIT protocol for C to commit to $\boldsymbol{x}$
Decommit phase, using ZK protocol π of decommitment value	
Step 1 (C):	SEND $\boldsymbol{x}$ to R

Step 2 (Both):	Run π with C as the prover and R as the verifier, that $\boldsymbol{x}$ is the correct decommitment value
	R: IF verifier-output of π is ACC
	      OUTPUT ACC and $\boldsymbol{x}$
	   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 Oblivious Transfer

4.1 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: $(\boldsymbol{G,q,g})$ where $(\boldsymbol{G,q,g})$ is a DLOG description • S's private input: $\boldsymbol{x_0, x_1}$ 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 $\boldsymbol{\sigma} \in \{\boldsymbol{0, 1}\}$
Parties' Outputs:	• S: nothing • R: $\boldsymbol{x_\sigma}$

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 $r \in \{0, . . . , q\text{-}1\}$ COMPUTE: • $u = g^r$ • $k_0 = h_0^r$ • $v_0 = x_0$ XOR $KDF(\|x_0\|,k_0)$ • $k_1 = h_1^r$ • $v_1 = x_1$ XOR $KDF(\|x_1\|,k_1)$ SEND (u,v_0,v_1) to R OUTPUT nothing
Step 3 (R):	COMPUTE $k_\sigma = (u)^\alpha$ OUTPUT $x_\sigma = v_\sigma$ XOR $KDF(\|v_\sigma\|,k_\sigma)$

0T_DDH_SEMIHONEST Sender (S) Specification	
Step 1:	WAIT for message ***(h_0,h_1)*** from R
Step 2:	SAMPLE a random value ***r*** ∈ **{0, . . . , *q*-1}** COMPUTE: • $u = g^r$ • $k_0 = h_0^r$ • $v_0 = x_0$ XOR $KDF(\|x_0\|,k_0)$ • $k_1 = h_1^r$ • $v_1 = x_1$ XOR $KDF(\|x_1\|,k_1)$ SEND **(u,v_0,v_1)** to R
Step 3:	OUTPUT nothing

<u>Note</u>: 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 $KDF(\|v_\sigma\|,k_\sigma)$

4.2 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 $\alpha,\beta,\gamma \in \{0, . . . , q-1\}$
	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 $u_0,u_1,v_0,v_1 \in \{0, . . . , q-1\}$
	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 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 $u_0,u_1,v_0,v_1 \in \{0, . . . , q-1\}$
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 $\alpha,\beta,\gamma \in \{0, . . . , q-1\}$
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 $KDF(|c_\sigma|, k_\sigma)$
```

4.3 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 $\alpha,\beta,\gamma \in \{0, . . . , q-1\}$ 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 $u_0,u_1,v_0,v_1 \in \{0, . . . , q-1\}$ 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 $u_0,u_1,v_0,v_1 \in \{0, . . . , q\text{-}1\}$
	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 $\alpha,\beta,\gamma \in \{0, . . . , q\text{-}1\}$
	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 $KDF(|c_\sigma|, k_\sigma)$
```

4.4 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 $s,t \in \{0, . . . , q-1\}$
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	
Initialization Phase	

Step 1 (Both):	IF **NOT** VALID_PARAMS(G,q,g_0)
	    REPORT ERROR and HALT

Step 2 (R):	SAMPLE random values $y,\alpha_0 \in \{0, . . . , q-1\}$
	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 $r \in \{0, . . . , q\text{-}1\}$
	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 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 $KDF(\lvert x_0 \rvert,v_0)$
	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 Receiver (R) Specification	
Initialization Phase	
Step 1:	SAMPLE random values $y,\alpha_0 \in \{0, . . . , q\text{-}1\}$
	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 $(\boldsymbol{g_1},\boldsymbol{h_0},\boldsymbol{h_1})$ to S
Step 2:	Run the prover in ZKPOK_FROM_SIGMA with Sigma protocol SIGMA_DH. Use common input $(\boldsymbol{g_0},\boldsymbol{g_1},\boldsymbol{h_0},\boldsymbol{h_1}/\boldsymbol{g_1})$ and private input α_0.
	Transfer Phase
Step 1:	SAMPLE a random value $\boldsymbol{r} \in \{0, . . . , q\text{-}1\}$ COMPUTE $\boldsymbol{g} = (\boldsymbol{g_\sigma})^r$ and $\boldsymbol{h} = (\boldsymbol{h_\sigma})^r$ SEND $(\boldsymbol{g},\boldsymbol{h})$ to S
Step 2:	WAIT for messages $(\boldsymbol{u_0},\boldsymbol{c_0})$ and $(\boldsymbol{u_1},\boldsymbol{c_1})$ from S
Step 3:	IF NOT • $\boldsymbol{u_0}, \boldsymbol{u_1} \in \boldsymbol{G}$, AND • $\boldsymbol{c_0}, \boldsymbol{c_1}$ are binary strings of the same length REPORT ERROR OUTPUT $\boldsymbol{x_\sigma} = \boldsymbol{c_\sigma}$ **XOR** $KDF(\lvert c_\sigma\rvert,(u_\sigma)^r)$

4.5 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: $(\boldsymbol{G,q,g_0})$ where $(\boldsymbol{G,q,g_0})$ is a DLOG description • S's private input: $\boldsymbol{x_0, x_1}$ 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 $\boldsymbol{\sigma \in \{0, 1\}}$
Parties' Outputs:	• S: nothing • R: $\boldsymbol{x_\sigma}$

0T_DDH_FULLSIM_ROM Protocol Specification	
Initialization Phase	

Step 1 (Both):	IF **NOT** VALID_PARAMS($\boldsymbol{G,q,g_0}$)
	    REPORT ERROR and HALT

Step 2 (R):	SAMPLE random values $\boldsymbol{y,\alpha_0 \in \{0, . . . , q\text{-}1\}}$
	SET $\boldsymbol{\alpha_1 = \alpha_0 + 1}$
	COMPUTE
	  1. $\boldsymbol{g_1 = (g_0)^y}$
	  2. $\boldsymbol{h_0 = (g_0)^{\alpha 0}}$
	  3. $\boldsymbol{h_1 = (g_1)^{\alpha 1}}$
	SEND $\boldsymbol{(g_1,h_0,h_1)}$ to S

Step 3 (Both):	Run ZKPOK_FS_SIGMA with Sigma protocol SIGMA_DH using common input $\boldsymbol{(g_0,g_1,h_0,h_1/g_1)}$ and private input $\boldsymbol{\alpha_0}$. 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 $\boldsymbol{r \in \{0, . . . , q\text{-}1\}}$ COMPUTE $\boldsymbol{g = (g_\sigma)^r}$ and $\boldsymbol{h = (h_\sigma)^r}$ SEND $\boldsymbol{(g,h)}$ to S
Step 2 (S):	COMPUTE $\boldsymbol{(u_0,v_0)}$ = **RAND**$\boldsymbol{(g_0,g,h_0,h)}$ COMPUTE $\boldsymbol{(u_1,v_1)}$ = **RAND**$\boldsymbol{(g_1,g,h_1,h)}$ COMPUTE $\boldsymbol{c_0 = x_0}$ **XOR** $\boldsymbol{KDF(\|x_0\|,v_0)}$ COMPUTE $\boldsymbol{c_1 = x_1}$ **XOR** $\boldsymbol{KDF(\|x_1\|,v_1)}$

	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_ROM 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_FS_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 (with inputs x_0,x_1)	
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 $KDF(\lvert x_0 \rvert,v_0)$** 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	
Initialization Phase	
Step 1:	IF **NOT** VALID_PARAMS(**G,q,g_0**) REPORT ERROR and HALT SAMPLE random values **y,$\alpha_0 \in \{0, . . . , q\text{-}1\}$** 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 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**.
Transfer Phase (with inputs x_0,x_1 and σ)	

Step 1:	SAMPLE a random value $r \in \{0, . . . , q-1\}$ 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** $KDF(\|c_\sigma\|,(u_\sigma)^r)$

This is identical to OT_DDH_FULLSIM except that the ZKPOK scheme is ZKPOK_FS_SIGMA instead of ZKPOK_FROM_SIGMA.

4.6 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 $r \in \{0, . . . , q\text{-}1\}$ 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 $KDF(\|x_0\|,v_0)$ 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 (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_ 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 **$r \in \{0, . . . , q\text{-}1\}$** 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 $KDF(\|c_\sigma\|,(u_\sigma)^r)$**

This is identical to OT_DDH_FULLSIM without the preprocess phase.

5 Batch Oblivious Transfer

5.1 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**.

OT_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

OT_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 Coin Tossing

6.1 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 $\boldsymbol{b_2} \in \{\boldsymbol{0,1}\}$
Step 2:	WAIT for COMMIT.commit on $\boldsymbol{b_1}$
Step 3:	SEND $\boldsymbol{b_2}$ to P_1
Step 4:	RUN subprotocol COMMIT.decommit to receive $\boldsymbol{b_1}$

Step 5	IF COMMIT.decommit returns INVALID
	    REPORT ERROR (cheat attempt)
	ELSE
	    OUTPUT $\boldsymbol{b_1}$ XOR $\boldsymbol{b_2}$

6.2 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 $\boldsymbol{s_2}$ to P_1
Step 5 (P_1):	P_1 sends $\boldsymbol{s_1}$ 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 $\boldsymbol{s_1}$. If the verifier output is REJ, then P_2 HALTS and REPORTS ERROR.
Step 7 (both):	OUTPUT $\boldsymbol{s_1}$ XOR $\boldsymbol{s_2}$

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

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

6.3 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