Description of Algorithms used in Cashlib

August 12, 2010

Contents

1 Security Parameters

In our system, we will use a unified security parameter *sec*, which corresponds to the conjecture that any attack that breaks the system must perform roughly 2^{sec} steps. We also define the following security parameters for ease of notation:

RSALength defines the bit-length of an RSA modulus.

stat defines the statistical security parameter. That is to say, two statistically indistinguishable distributions will be 2^{-stat} close to each other.

hashLength defines the length of the output of a hash function used for collision resistance or as a random oracle.

orderLength defines the bit-length of the order of the prime-order group we will use. *primeLength* defines the bit-length of the modulus of $Z^*_{primeModulus}$. The prime-order group will be a subgroup of $Z^*_{primeModulus}$ of order *primeOrder*.

To get $sec = 80$, which is considered the bare minimum requirement for security, we need to have $RSALength = 1024$, $stat = sec = 80$, $hashLength = 2 * sec = 160$, $orderLength = 2 * sec = 160$, $primeLength = 1024$. We will use these numbers for performance evaluation. The reasoning for hash functions and prime order is related to the birthday bound, and states the best known algorithms break them in $2^{hashLength/2}$ and $2^{orderLength/2}$ time respectively (DON'T WE NEED A CITATION FOR THIS?). We will use SHA-1 as our hash function.

To get a longer-lasting security, we should use $sec = 128$, which corresponds to $RSALength = 2048$, $stat = sec = 128$, $hashLength = 2 * sec = 256$, $orderLength = 2 * sec = 256$, $primeLength = 1024$. For our BitTorrent system, we should use AES-128 as the block cipher and SHA-256 as the hash function whenever necessary (note that these numbers correspond to $sec = 112$ [**?**, **?**]).

Finally, if we are feeling paranoid, we can easily set $RSALength = 4096$, $primeLength = 2048$, and use SHA-512 and AES-256. The only issue is that we will sacrifice some amount of efficiency.

2 Assumptions

The suggested security parameters above are set to satisfy the following assumptions (by conjecture).

Strong RSA Assumption: For $N = pq$ with p, q prime, it is difficult for any computationally bounded adversary $\mathcal{A}$ to compute x on input N, e, and $y = x^e \bmod N$ for some e such that $\gcd(e, \phi(N)) = 1$, even if $\mathcal{A}$ is allowed to pick e itself (note that this last condition is what makes this assumption the "strong" verison of the standard RSA assumption).

Discrete Logarithm Assumption: For a cyclic group G of order q with generator g, it is difficult for any computationally bounded adversary $\mathcal{A}$ to compute x on input g^x, for $x \in \mathbb{Z}_q$.

3 Setup

In our system, we will use three different types of groups: special RSA groups, prime-order groups, and Paillier groups (i.e., groups of the form $\mathbb{Z}^*_{N^2}$). In this section we focus on the former two group types, as the Paillier groups are used only in a superficial way for verifiable encryption and are not considered parameters of the system. Our groups are usually generated by a trusted third party, but we also describe methods that can be used if for some reason the setup is untrusted.

In what follows, let QR_n denote the group of quadratic residues modulo n.

Algorithm 1: Setup for generating a special RSA group.

Input: security parameter *RSALength*, number of generators m
Pre-conditions: *RSALength* must be at least 1024. m must be at least 1.
Output: modulus n, primes p, q, p', q', generators $g_1, \ldots, g_m, h$, exponents $a_1, \ldots, a_m$
Post-conditions: The length of n must be *RSALength*. $n = pq$, $p = 2p' + 1$, $q = 2q' + 1$. p, q, p', q' are primes. $p = 3 \mod 4$, $q = 3 \mod 4$. $|p| = |q| = RSALength/2$

`Special RSA Group Setup`

1 Choose primes p, q of length $RSALength/2$ each such that $p = 2p' + 1$ and $q = 2q' + 1$ where p', q' are primes (so p, q are safe primes, p', q' are Sophie Germain primes, and n is a special RSA modulus). Note that we will need $p = 3 \mod 4$ and $q = 3 \mod 4$ in order to have n be a Blum integer, and that this will indeed be the case for primes chosen as above.
2 Compute $n = pq$.
3 Choose $h \leftarrow QR_n$. To do this, pick a random residue, square it, and check that $h^{(p-1)/2} = 1 \mod p$ and $h^{(q-1)/2} = 1 \mod q$. Alternatively, pick $h_p \leftarrow Z_p^*$ and $h_q \leftarrow Z_q^*$, and set $h = h_p^{q-1} * h_q^{p-1} \mod n$.
4 **for** i : 1..m **do**
5 Choose $a_i \leftarrow \{0,1\}^{RSALength+stat}$ and set $g_i = h^{a_i} \mod n$
6 Output modulus n, primes p, q, p', q', generators $g_1, \ldots, g_m, h$, and exponents $a_1, \ldots, a_m$.

Algorithm 2: Setup for generating a prime-order group.

Input: security parameters *primeLength*, *orderLength*, number of generators m
Pre-conditions: *primeLength* must be at least 512. *orderLength* must be at least 160 or $2 * stat$, whichever is larger.[a] m must be at least 1.
Output: modulus *primeModulus*, order *primeOrder*, generators $g_1, \ldots, g_m, h$, exponents $a_1, \ldots, a_m$
Post-conditions: The length of *primeModulus* must be *primeLength*. The length of *primeOrder* must be *orderLength*.

`Prime Order Group Setup`

1 Pick a prime order *primeOrder* of length *orderLength* and a prime modulus *primeModulus* of length *primeLength* where *primeModulus* $- 1$ is divisible by *primeOrder*. This results in the order *primeOrder* subgroup of $Z^*_{primeModulus}$.
2 Pick generator h for the subgroup with order *primeOrder*. To do this, pick $h' \leftarrow Z^*_{primeModulus}$ and set $h = h'^{(primeModulus-1)/primeOrder} \mod primeModulus$.
3 **for** i : 1..m **do**
4 Pick generators g_i each with order *primeOrder* (same method as above)
5 Output modulus *primeModulus*, order *primeOrder*, generators $g_1, \ldots, g_m, h$, exponents $a_1, \ldots, a_m$

[a] The best algorithms to solve Discrete Logarithm problem in prime-order groups have running time that depends on the size of either the order of the subgroup or the modulus of the group [**?**, 5]

Definitions of Groups:
Throughout this paper, the "definition of the RSA group" means the modulus n and possibly the

bases $g_1, \ldots, g_m, h$ if they are not explicitly defined.

Similarly, the "definition of the prime-order group" means the modulus *primeModulus* and the order *primeOrder*, and possibly the bases $g_1, \ldots, g_m, h$ if they are not explicitly defined.

Bases will generally be explicitly defined in the algorithms.

Group Operations:

A group operation is an operation on a group element. We will use multiplicative notation when denoting the group operation and inverse of an element. All group operations must be done modulo the modulus of the group. Therefore, all group operations will be done modulo n for the RSA group, and modulo *primeModulus* for the prime-order group. A simple example of this is $g * h \mod n$.

Operations on Exponents:

An operation on exponents means the multiplication, addition, or inversion of exponents. In RSA groups, all such operations must be performed over integers, as the order of the group is not assumed to be known. In prime-order groups, all such operations will be done modulo *primeOrder*, as this value is always public. An example of such an operation is $g^{a*b \mod primeOrder} \mod primeModulus$.

Group Secrets:

When we emphasize that some party has not generated a group itself, we mean that that party may not know the secrets associated with the group generation. For an RSA group, the secrets will be p, q; the factorization of the modulus n. Furthermore, in both an RSA group and a prime-order group, the relative discrete logarithms of the generators are secrets.

4 Commitment Schemes

4.1 Fujisaki-Okamoto Commitment Scheme

Overview:

The Fujisaki-Okamoto commitment scheme [4, 3] is a statistically hiding, computationally binding commitment scheme. The Committer commits to something and sends the resulting commitment to the Verifier. At some later time, the Committer opens the commitment and the Verifier needs to verify that the opening matches the commitment sent before.

Setup:

This commitment scheme uses a special RSA group. In the case that an untrusted party (*e.g.*, the Verifier) generates the RSA group, he needs to prove to the Committer that each g_i is in the group generated by h, so that the commitment is statistically hiding. This can be done by proving in zero knowledge the knowledge of a_i such that $g_i = h^{a_i} \mod n$. The committer may not generate or know $p, q, p', q', a_1, \ldots, a_m$, as otherwise the scheme will not provide any meaningful binding property.

Assumptions:

The security of the scheme relies on the Strong RSA assumption.

Hiding:

The hiding property relies on the fact that g_i, h all generate the same group, so that when randomization is used the resulting commitment is a random group element.

Binding:

The binding property relies on the assumption that the Committer cannot find two different openings (using the same bases) that result in the same commitment. This follows from the Strong RSA assumption.

WARNINGS:
This commitment scheme requires at least two generators: g_1, h, otherwise it does not work.

Algorithm 3: Commitment procedure of the Fujisaki-Okamoto commitment scheme. This procedure is run by the Committer.

Input: Definition of the RSA group, number of secrets k, bases $g_1, \ldots, g_k, h$, secrets $x_1, \ldots, x_k$
Pre-conditions: The RSA group must be generated by a trusted third party or it must be proven that each g_i, h generates QR_n.
Output: commitment C, opening *open*

`Commit`

1 Pick a random number r from $\{0,1\}^{RSALength+stat}$.
2 Create the commitment $C = h^r \prod_{i=1}^{k} g_i^{x_i} \mod n$.
3 Output commitment C and opening $open = x_1, \ldots, x_k, r$.

COMMENTS:
The random number r actually needs to be relatively prime to $\phi(n)$, but since this will be the case with high probability we omit the check.

To open a commitment, the Committer just sends the opening *open* to the Verifier. After that, the Verifier needs to verify the opening of the commitment. This is done as follows:

Algorithm 4: Verification procedure of the Fujisaki-Okamoto commitment scheme. This procedure is run by the Verifier after receiving the opening *open* for a commitment C.

Input: Definition of the RSA group, number of secrets k, bases $g_1, \ldots, g_k, h$, commitment C, opening $open = x_1, \ldots, x_k, r$
Pre-conditions: The RSA group may NOT be generated by the Committer.
Output: ACCEPT or REJECT

`Verify`

1 **if** $C = h^r \prod_{i=1}^{k} g_i^{x_i} \mod n$ **then**
2 Output ACCEPT
3 **else**
4 Output REJECT

4.2 Pedersen Commitment Scheme

OVERVIEW:
The Pedersen commitment scheme [6] is a statistically hiding, computationally binding commitment scheme. It allows for commitments to values between 1 and $primeOrder - 1$.

SETUP:
This commitment scheme uses a prime-order group. If an untrusted party (*e.g.*, the Verifier) generates the prime-order group, then the participants (both the Committer and the Verifier) need to check that both *primeModulus* and *primeOrder* are primes (such that *primeOrder* divides $primeModulus - 1$) and that g_i, h have order *primeOrder* (which is equivalent to saying that $g_i \neq 1 \mod primeModulus$ and $g_i^{primeOrder} = 1 \mod primeModulus$). It is important that the Committer

does not know the relative discrete logarithms of the bases, or otherwise the commitment is no longer binding.

ASSUMPTIONS:

The security of the scheme relies on the Discrete Logarithm assumption.

HIDING:

The hiding property relies on the fact that g_i, h all generate the group with prime order *primeOrder*, so that when randomization is used the resulting commitment is a random element of the prime-order group.

BINDING:

The binding property relies on the Committer not being able to find two different openings, using the same bases, that result in the same commitment. This follows directly from the Discrete Logarithm assumption.

Algorithm 5: Commitment procedure of the Pedersen commitment scheme. This procedure is run by the Committer.

Input: Definition of the prime-order group, number of secrets k, bases $g_1, \ldots, g_k, h$, secrets $x_1, \ldots, x_k$

Pre-conditions: Group and the bases must adhere to guidelines in the SETUP in this section. Each x_i must be between 1 and $primeOrder - 1$.

Output: commitment C, opening *open*

Post-conditions: C should be sent to the Verifier.

`Commit`

1 Pick a random number r from $Z^*_{primeOrder}$.
2 Create the commitment $C = h^r \prod_{i=1}^{k} g_i^{x_i} \mod primeModulus$.
3 Output commitment C and opening $open = x_1, \ldots, x_k, r$.

To open a commitment, the Committer simply sends *open* to the Verifier. Upon receiving the opening, the Verifier needs to verify its validity. This is done as follows:

Algorithm 6: Verification procedure of the Pedersen commitment scheme. This procedure is run by the Verifier after receiving the opening *open* for a commitment C.

Input: Definition of the prime-order group, number of secrets k, bases $g_1, \ldots, g_k, h$, commitment C, opening $open = x_1, \ldots, x_k, r$

Pre-conditions: The group may not be generated by the Committer. Each x_i, r must be between 1 and $primeOrder - 1$.

Output: ACCEPT or REJECT

`Verify`

1 **if** $C = h^r \prod_{i=1}^{k} g_i^{x_i} \mod primeModulus$ **then**
2 Output ACCEPT
3 **else**
4 Output REJECT

COMMENTS:

If only one generator exists (namely g_1), then the commitment is only computationally hiding based on the Discrete Logarithm assumption, although it still works.

5 Honest-Verifier Zero Knowledge Σ-Proofs

In these Σ-protocols, both the Prover and the Verifier know the definition of the group used, as well as some common information (typically a commitment) C. In all honest-verifier zero knowledge Σ-proofs, the Prover first sends a randomized proof R. The Verifier then replies with a challenge c (which is always generated in the same way). The Prover then responds to that challenge with some value A. Finally, the Verifier verifies the whole proof.

Algorithm 7: Randomized Proof round of a Σ-protocol. This procedure is run by the Prover.

Input: Definition of the group, number of secrets k, bases $g_1, \ldots, g_k, h$,
Output: randomized proof R, its opening $openR$

```
Randomize Proof
1 Will be defined separately for each protocol.
```

Algorithm 8: Challenge round of a 3-round Σ-protocol. This procedure is run by the Verifier upon receipt of randomized proof R from the Prover.

Input: Definition of the group, number of secrets k, bases $g_1, \ldots, g_k, h$, commitment C, randomized proof R
Pre-conditions: In many protocols we use, the group may not be generated by the Prover. See the specific protocol specification for more on this.
Output: challenge c
Post-conditions: C, R and c should be kept for use in the verification procedure.

```
Challenge
1  Pick a random number c from the domain of challenges D_C.
2  Output challenge c
```

Domain of Challenges D_C:
If an RSA group is used, the domain of challenges is $D_C = \{0,1\}^{stat}$ (with the exception that the challenge must not be 0). If a prime-order group is used, the domain of challenges is $D_C = Z^*_{primeOrder}$ (actually $D_C = \{0,1\}^{stat}$ will also work). When using non-interactive proofs, $D_C = \{0,1\}^{2*stat}$ will be used, which will be the same as $D_C = Z^*_{primeOrder}$. This means that in practice it will make sense to simply use $D_C = \{0,1\}^{2*stat} - \{0\}$ for both groups.

Domain of Randomness D_R:
If an RSA group is used, the domain of randomness is $D_R = \{0,1\}^{RSALength+stat}$ (except again the randomness must not be 0). If a prime-order group is used, the domain of randomness is $D_R = Z^*_{primeOrder}$. Note that, unlike the challenge domains, these domains are very different for both groups, and that the prime-order group is more efficient in terms of both computation and communication.

Non-Interactive Version:
If non-interactive proofs will be employed using the Fiat-Shamir heuristic [**?**], the challenge will be computed as $c = hash$(definition of the group and bases used $||$ k $||$ C $||$ R). This computation can be carried out by the Prover and the Verifier separately, and then the Verifier can verify the proof as before. It is important to note that if the Fiat-Shamir heuristic is used, the resulting protocol is secure only in the Random Oracle model [**?**, **?**].

Full Zero Knowledge:
Techniques for converting honest-verifier zero-knowledge proofs to full zero-knowledge proofs (*i.e.*

using trapdoor commitments as in [**?**]) should be applied for the interactive versions. Non-interactive versions using the Fiat-Shamir heuristic do not require this conversion, since the Random Oracle model essentially forces the Verifier to be honest. Since all the proofs we will use are non-interactive, we do not provide full zero-knowledge protocols here.

Algorithm 9: Response round of a Σ-protocol. This procedure is run by the Prover.

Input: Definition of the group, number of secrets k, bases $g_1, \ldots, g_k, h$, opening of C: $openC$, randomized proof R and its opening $openR$, challenge c

Output: response A

`Respond`

1 Will be defined separately for each protocol.

Algorithm 10: Verification of a Σ-protocol. This procedure is run by the Verifier upon receipt of response A from the Prover.

Input: Definition of the group, number of secrets k, bases $g_1, \ldots, g_k, h$, commitment C, randomized proof R, challenge c, response A

Output: ACCEPT or REJECT

`Verify`

1 Will be defined separately for each protocol.

Non-Interactive Verification:
When verifying non-interactive proofs, the verification procedure is called with challenge $c = hash$(definition of the group and bases used $||$ k $||$ C $||$ R), and the rest proceeds the same as the interactive version.

5.1 Proof of Knowledge of Discrete Logarithm Representation

This is the protocol used for proving knowledge of the discrete logarithm representation of a number using some well-defined bases in an honest verifier zero knowledge way [7]. The Verifier knows the number and the bases. We call this protocol **PoKoDLR**. It has two versions:

The RSA group version uses Fujisaki-Okamoto commitments. Therefore it requires the same setup and the same assumptions. Everything said for the Fujisaki-Okamoto commitments also applies here.

The prime-order group version uses Pedersen commitments. Therefore it requires the same setup and the same assumptions. Everything said for the Pedersen commitments also applies here.

All operations will be done in respective groups.

Assumptions:
RSA group version makes the Strong RSA assumption.
Prime-order group version makes no assumptions.

Honest-Verifier Zero Knowledge:
RSA group version is honest verifier zero knowledge provided that the Fujisaki-Okamoto commitment is hiding. This means that the RSA group must be generated by a trusted third party or it must be proven to the Prover that each g_i, h generates QR_n.
Prime-order group version is honest verifier zero knowledge provided that the Pedersen commitment is hiding. This means that the prime-order group must be generated by a trusted third party or the Prover must verify that each g_i, h has order $primeOrder$.

Version	Commitments Used	HVZK Condition	Soundness Condition
RSA Group	Fujisaki-Okamoto	Fujisaki-Okamoto is hiding	Strong RSA
Prime-order Group	Pedersen	Pedersen is hiding	None or DLog*

Table 1: PoKoDLR Protocol Security Summary

SOUNDNESS:
In RSA group version, the extraction works under the Strong RSA assumption, which requires that the RSA group may NOT be generated by the Prover.
In prime-order group version, the extraction works without any assumption.

*WARNINGS:
In many uses of the prime-order group version of this protocol, for it to be meaningful, we would like the Pedersen commitments to be binding, which means the Prover may NOT generate g_i, h (and thus know the relative discrete logarithms of the bases), and that the Discrete Logarithm assumption holds.

Algorithm 11: Randomized Proof round of PoKoDLR protocol. This procedure is run by the Prover.

Input: Definition of the group, number of secrets k, bases $g_1, \ldots, g_k, h$

Pre-conditions: The group must be generated by a trusted third party, or it must be proven to the Prover that each g_i, h generates QR_n in a special RSA group, or the Prover must verify that each g_i, h has order *primeOrder* in a prime-order group.

Output: randomized proof R, its opening $openR$

`Randomize Proof`

1 **for** i : $1..k$ **do**
2 Pick a random number s_i from domain of randomness D_R.
3 Create a random element R using Randomize(group definition, $k+1$, $g_1, \ldots, g_k, h$) as in Algorithm 39. Let the random exponents returned by the Randomize procedure be $openR = s_1, \ldots, s_k, t$.
4 Output randomized proof R and its opening $openR$.

Algorithm 12: Response round of PoKoDLR protocol. This procedure is run by the Prover.

Input: Definition of the group, number of secrets k, bases $g_1, \ldots, g_k, h$, opening $open = x_1, \ldots, x_k, r$, randomized proof R and its opening $openR = s_1, \ldots, s_k, t$, challenge c

Pre-conditions: The group must be generated by a trusted third party, or it must be proven to the Prover that each g_i, h generates QR_n in a special RSA group, or the Prover must verify that each g_i, h has order *primeOrder* in a prime-order group.

Output: response A

`Respond`

1 **for** i : 1..k **do**
2 Compute $a_i = s_i + cx_i$
3 Compute $b = t + cr$
4 [In prime-order group version, let $a_i = a_i \mod primeOrder$ and $b = b \mod primeOrder$]
5 Output response $A = a_1, \ldots, a_k, b$

Algorithm 13: Verification of PoKoDLR protocol. This procedure is run by the Verifier upon receipt of response A from the Prover.

Input: Definition of the group, number of secrets k, bases $g_1, \ldots, g_k, h$, commitment C, randomized proof R, challenge c, response $A = a_1, \ldots, a_k, b$

Pre-conditions: The RSA group may NOT be generated by the Prover

Output: ACCEPT or REJECT

`Verify`

1 **if** $RC^c = h^b \prod_{i=1}^{k} g_i^{a_i}$ *(* $\mod n$ *for RSA group version,* $\mod primeModulus$ *for Prime-order group version)* **then**
2 Output ACCEPT
3 **else**
4 Output REJECT

WARNINGS:

In prime-order group version of this protocol, it's important for both parties to make sure that each x_i, r, s_i, t is between 1 and $primeOrder - 1$. Prover can only prove knowledge of x_i that is in that range.

5.2 Proof of equality of discrete logarithm representation

This is the protocol for proving knowledge of the equality of discrete logarithm representations of some numbers using some well-defined bases in an honest verifier zero knowledge way. The Verifier knows the bases. We call this protocol **PoEoDLR**. it has two versions:

The first version uses Fujisaki-Okamoto commitments. Therefore it requires the same setup and the same assumptions. Everything said for the Fujisaki-Okamoto commitments also applies there.

The second version uses Pederson commitments. Therefore it requires the same setup and the same assumptions. Everything said for the Pederson commitments also applies there.

All operations will be done in respective groups.

ASSUMPTIONS:

RSA group version makes the Strong RSA assumption.

Prime-order group version makes no assumptions.

HONEST-VERIFIER ZERO KNOWLEDGE:

RSA group version is honest verifier zero knowledge provided that the Fujisaki-Okamoto commitment is hiding. This means that the RSA group must be generated by a trusted third party or it must be proven to the Prover that each g_i, h generates QR_n.

Prime-order group version is honest verifier zero knowledge provided that the Pedersen commitment is hiding. This means that the prime-order group must be generated by a trusted third party or the Prover must verify that each g_i, h has order $primeOrder$.

SOUNDNESS:

In RSA group version, the extraction works under the Strong RSA assumption, which requires that the RSA group may NOT be generated by the Prover.

In prime-order group version, the extraction works without any assumption.

*WARNINGS:

In many uses of the prime-order group version of this protocol, for it to be meaningful, we would like the Pedersen commitments to be binding, which means the Prover may NOT generate g_i, h (and thus know the relative discrete logarithms of the bases), and that the Discrete Logarithm assumption holds.

Algorithm 14: Randomized Proof round of PoEoDLR protocol. This procedure is run by the Prover.

Input: Definition of the group, bases $g_1, ..., g_k, h$, number of common secrets k, number other secrets l

Output: randomized proof set R, its opening set $openR$

`Randomize Proof`

1 **for** i : 1..k **do**

2 Pick a random number s_i from domain of randomness D_R.

3 **for** i : 1..l **do**

4 Create the randomized proof R_i to using Randomize(group definition, 1 (number of random elements), k (number of fixed elements), $g_1, ...g_k, h$ (bases), $s_1, \ldots, s_k$ (fixed elements)). Let the opening returned by the Commit procedure be $openR_i = s_1, \ldots, s_k, t_i$.

5 Output randomized proof set R (set of R_i's) and opening set $openR$ (set of $openR_i$'s).

Algorithm 15: Response round of PoEoDLR protocol. This procedure is run by the Prover.

Input: Definition of the group, f the group, bases $g_1, ..., g_k, h$, number of common secrets k, number other secrets l, common secrets $x_1, \ldots, x_k$ and other secrets $r_1, \ldots, r_l$, openings of randomized proofs $openR_1 \ldots openR_l$ where $openR_j = s_1, \ldots, s_k, t_j$, challenge c

Output: set of responses $A = a_1, \ldots, a_m$

`Respond`

1 **for** i *:* $1..k$ **do**
2 Compute $a_i = s_i + cx_i$
3 **for** j *:* $1..l$ **do**
4 Compute $b_j = t_j + cr_j$
5 If no RSA group is involved, let $a_i = a_i \mod primeModulus$ and $b_i = b_i \mod primeModulus$
6 Output response $A = a_1, \ldots, a_k, b_1, \ldots, b_l$

Algorithm 16: Verification of PoEoDLR protocol. This procedure is run by the Verifier upon receipt of response A from the Prover.

Input: Definition of the group, number of common secrets k, number of other secrets l, bases $g_1, \ldots, g_k, h$, set of commitments $C = C_1, \ldots C_l$, set of randomized proofs $R = R_1, \ldots R_l$, challenge c, response $A = a_1, \ldots, a_k, b_1, \ldots, b_l$

Output: ACCEPT or REJECT

`Verify`

1 **if** $R_1 C_1^c = h^{b_1} \prod_{i=1}^{k} g_i^{a_i}$ *AND* $\ldots$ *AND* $R_l C_l^c = h^{b_l} \prod_{i=1}^{k} g_i^{a_i}$ **then**
2 Output ACCEPT
3 **else**
4 Output REJECT

The Simulator Here we present a simulator to simulate a 2-base equality of discrete logarithm representation protocol. The purpose of the simulator is to show that an entity starting with knowledge of the challenge but not the secrets can simulate the Prover's work to the Verifier. This is turn shows that with an honest Verifier, the process of proving is zero-knowledge and gives no extra information about the secrets to the Verifier, because the Simulator does not know any more about the secrets than the Verifier does.
a, b, d - the three secrets
g, h - the group bases/ generators
$A = g^a h^b$ - commitment 1
$B = g^a h^d$ - commitment 2
o - group order
m - group modulus
c - the challenge that the Verifier will send

the simulator	S		V	**the verifier**
starting information	g, h, A, B, o, m, c		g, h, A, B, o, m	starting information
S picks responses ahead of time	pick random r_1, r_2, r_3 s.t. $1 \leq r_i \leq o-1 \ \forall i$ $s_1 = r_1,\ s_2 = r_2,\ s_3 = r_3$			
S computes randomized proofs based on responses and challenge c	$R_A = g^{r_1} h^{r_2} A^{-c}$ $R_B = g^{r_1} h^{r_3} B^{-c}$	$\xrightarrow{R_A, R_B}$		
		$\xleftarrow{c}$	pick $c \in D_c$	
S sends back pre-computed responses		$\xrightarrow{r_1, r_2, r_3}$	$R_A A^c == g^{s_1} h^{s_2} \mod p$ $R_B A^c == g^{s_1} h^{s_3} \mod p$	check that all these equal
				if equal **->accept** if not **->reject**

5.3 Proof that a committed value is a multiplication of other values (and in particular, a square)

This is the protocol used for proving that a discrete logarithm representation is a product of two discrete logarithm representations in an honest verifier zero knowledge way [**?**]. In particular, this protocol can be used to prove that a committed number is a square, as in [**?**].

Suppose the Prover knows secrets x, y, z such that $x = y * z$ and wants to prove this to an honest verifier. The Verifier knows commitments to each of these numbers: C_x is a commitment to x, C_y is to y and C_z is to z. We call this protocol **Mult**. It uses Fujisaki-Okamoto commitments. Therefore it requires the same setup and the same assumptions. Everything said for the Fujisaki-Okamoto commitments also applies here.

Assumptions:

The security of the scheme relies on Strong RSA assumption.

Honest-Verifier Zero Knowledge:

The protocol is honest verifier zero knowledge provided that the Fujisaki-Okamoto commitment is hiding. This means that the RSA group must be generated by a trusted third party or it must be proven to the Prover that each g_i, h generates QR_n.

Soundness:

The extraction works under the Strong RSA assumption, which requires that the RSA group may

NOT be generated by the Prover.

Algorithm 17: Randomized Proof round of a Mult protocol for $x = yz$. This procedure is run by the Prover.

Input: Definition of the RSA group and bases: n, g_1, h
Output: randomized proof R, its opening $openR$

`Randomize Proof`

1 Pick a random number s from D_R.
2 Create a random element R_1 using Randomize(group definition, 1, 1, g_1, h, s) as in Algorithm 39 where s is a fixed element. Let the random exponent returned by the Randomize procedure be $openR_1 = t_1$.
3 Create a random element R_2 using Randomize(group definition, 1, 1, C_y, h, s) as in Algorithm 39 where s is again a fixed element. Note the use of C_y as one of the bases in the procedure (it will be used with the fixed element). Let the random exponent returned by the Randomize procedure be $openR_2 = t_2$.
4 Output randomized proof $R = R_1, R_2$ and opening $openR = s, t_1, t_2$.

Algorithm 18: Response round of a Mult protocol for $x = yz$. This procedure is run by the Prover.

Input: Definition of the RSA group and bases: n, g_1, h, openings to commitments C_x, C_y, C_z as $openX = x, r_x$, $openY = y, r_y$, $openZ = z, r_z$, randomized proof $R = R_1, R_2$ and its opening $openR = s, t_1, t_2$, challenge c
Output: response A

`Respond`

1 Compute $a = s + cz$, $b_1 = t_1 + cr_z$, $b_2 = t_2 + c(r_x - zr_y)$
2 Output response $A = a, b_1, b_2$

Algorithm 19: Verification of a Mult protocol for $x = yz$. This procedure is run by the Verifier upon receipt of response A from the Prover.

Input: Definition of the RSA group and bases: n, g_1, h, commitments C_x, C_y, C_z, randomized proof $R = R_1, R_2$, challenge c, response $A = a, b_1, b_2$
Output: ACCEPT or REJECT

`Verify`

1 **if** $R_1 C_z^c = g_1^a h^{b_1} \mod n$ AND $R_2 C_x^c = C_y^a h^{b_2} \mod n$ **then**
2 Output ACCEPT
3 **else**
4 Output REJECT

PROVING KNOWLEDGE OF x, y SUCH THAT $x = y^2$:
Using the proof above, it is very easy to prove knowledge of x, y such that $x = y^2$. Just set $z = y$, which also means getting rid of C_z, or in other words, setting $C_z = C_y$ and hence $openC_z = openC_y$.

5.4 Proof that a committed value lies within an interval [lo,hi]

We would like to prove that a secret value x lies within a publicly known interval $[lo, hi]$, as in [**?**, **?**]. The protocol uses a special RSA group. The verifier knows a commitment C_x to x, and

lo, hi, along with the RSA group definition. The prover additionally knows the secret x.

Observe that x is in the interval $[lo, hi]$ iff $x - lo \geq 0$ and $hi - x \geq 0$. Furthermore, any non-negative integer can be represented as a sum of 4 squares [**?**] (*e.g.*, $x - lo = v_1^2 + v_2^2 + v_3^2 + v_4^2$), whereas negative integers cannot. These numbers (v_i) can be computed efficiently [**?**, **?**]. Let W_i be v_i^2.

Further observe that a commitment to $x - lo$ can easily be obtained by computing $C_x g_1^{-lo}$, and a commitment to $hi - x$ can easily be obtained by computing $g_1^{hi} C_x^{-1}$.

We have seen that using the Mult algorithm, we can prove that a committed number is actually a square. Therefore, the basic idea is to prove that each W_i is a square (by committing to them so that the verifier does not learn W_i or v_i), and then their sum is equal to $x - lo$. A similar proof also needs to be given for $hi - x$.

We call this protocol **Range**. It uses Fujisaki-Okamoto commitments. Therefore it requires the same setup and the same assumptions. Everything said for the Fujisaki-Okamoto commitments also applies here.

Assumptions:
The security of the scheme relies on Strong RSA assumption.

Honest-Verifier Zero Knowledge:
The protocol is honest verifier zero knowledge provided that the Fujisaki-Okamoto commitment is hiding. This means that the RSA group must be generated by a trusted third party or it must be proven to the Prover that each g_i, h generates QR_n.

Soundness:
The extraction works under the Strong RSA assumption, which requires that the RSA group may NOT be generated by the Prover.

Below we provide the pseudocode for proving that a given value y is non-negative (≥ 0). We call this protocol **Non-Negative**. Then, proving the interval means running this protocol twice, one for $x - lo$ and one for $hi - x$.

In the Non-Negative protocol, the Prover is given a commitment C_y and its opening $open_y = y, r_y$, and the group definition. The Verifier is given C_y and the group definition.

1. The Prover computes v_1, v_2, v_3, v_4 using the four-squares algorithm [**?**, **?**]. Let $W_i = v_i^2$.
2. The Prover computes commitments C_i to W_i with their openings $open_i = W_i, r_i$, with the only rule that the last randomness is set to be $r_4 = r_y - (r_1 + r_2 + r_3)$.
3. The Prover proves that each C_i is a commitment to a square, using the special case of the Mult protocol.
4. The Verifier, in addition to checking that each C_i is a commitment to a square, checks if $C_y = \Pi_{i=1}^4 C_i$. If all checks pass, the Verifier accepts, otherwise he rejects.

5.5 Proof that a committed value lies within an interval [mean-delta,mean+delta]

The range proof given by the Range protocol can be made almost twice as efficient if the interval we want to prove has a special type: $[mean-delta, mean+delta]$ [**?**]. Call this protocol **SpecialRange**, which should be used instead of Range whenever possible, even if not stated explicitly. Note that,

x is in that interval iff $lo^2 - (x - mean)^2 \geq 0$. As before, the interval (and hence lo, hi) are publicly known. Let $A = delta^2 - (x - mean)^2 = delta^2 - mean^2 + 2 * mean * x - x^2$.

For ease of notation, let $D = g_1^{delta^2 - mean^2}$, $E = (g_1^{2hi} C_x^{-1})$. A commitment F to A can be computed as $D * E^x * h^{r_A}$. Note that the Prover can compute this, but the Verifier can only compute D and E.

The Verifier knows a commitment C_x to x, the interval (meaning lo, hi), and the RSA group definition, and can compute D and E. The Prover, in addition to these, also knows the opening $open_x = x, r_x$ to C_x.

1. The Prover computes $F = D * E^x * h^{r_A} = g_1^A h^{r'_A}$ and proves that F and C_x are commitments to the same number under the bases D, E, h and $1, g, h$ using the PoEoDLR protocol.

2. The Prover runs the Non-Negative protocol for A using F as the commitment to A, where $r'_A = r_A - x r_x$.

6 CL signatures

CL signatures is an example of blind signatures, where the signer may not know the values she signed. Besides, useful protocols such as proving existence of a CL signature on a secret value is also necessary. The version of CL signatures we will present here uses special RSA groups [**?**, **?**, **?**, **?**].

SETUP:

The key generation of CL signatures is exactly as generating a special RSA group as in Algorithm 1:

- We have the following generators: $f, g_1, \ldots, g_m, h$, and the key owner (signer) needs to provide a non-interactive proof that all these generators generate the same group (namely, QR_n). This can be done by proving the knowledge of a_i such that $g_i = h^{a_i} \mod n$ and a such that $f = h^a \mod n$.

- The public key (*i.e.* verification key) is $CLPK = n, f, g_1, \ldots, g_m, h$.

- The secret key (*i.e.* signing key) is $CLSK = p, q$.

ASSUMPTIONS:

The security of the scheme relies on Strong RSA assumption.

PARAMETERS:

This scheme uses the following (security) parameters: *RSALength* for the length of the RSA modulus, m for the number of bases (maximum number of messages that can be signed at once), l_x for the length of a message and D_x for the domain of messages (each message x_i is in $D_x = \{0,1\}^{l_x}$). We can also use messages in the domain $D_x = [-(2^{l_x} - 1), 2^{l_x} - 1]$. Further define $l_e = l_x + 2$ and

$l_v = RSALength + l_x + 2 * stat$.

Algorithm 20: Signing procedure for a CL signature. This is the signing procedure to sign a public message (not a blind signature yet). This procedure is run by the Signer.

Input: CL signature public key $CLPK = n, f, g_1, \ldots, g_m, h$, secret key $CLSK = p, q$, messages to be signed $x_1, \ldots, x_k$

Pre-conditions: Each message x_i needs to be in D_x.

Output: signature $\sigma = A, e, v$

`Sign`

1 Pick a random **prime** number e of length l_e.

2 Pick a random number v of length l_v.

3 Compute the value A such that $A^e = fh^v \Pi_{i=1}^k g_i^{x_i} \mod n$. This can be done with the knowledge of the $CLSK = p, q$ by setting $A = [fh^v \Pi_{i=1}^k g_i^{x_i}]^{1/e}$

4 Output the signature A, e, v.

Algorithm 21: Verification procedure for a CL signature. This is the verification procedure to verify a public message (not a blind signature yet). This procedure is run by the Verifier.

Input: CL signature public key $CLPK = n, f, g_1, \ldots, g_m, h$, messages $x_1, \ldots, x_k$, signature $\sigma = A, e, v$

Pre-conditions: Each message x_i needs to be in D_x. e needs to be of length l_e, v needs to be of length l_v.

Output: ACCEPT or REJECT

`Verify`

1 Check the pre-conditions (Length checks are important !!).

2 **if** $A^e = fh^v \Pi_{i=1}^k g_i^{x_i} \mod n$ **then**

3 Output ACCEPT

4 **else**

5 Output REJECT

6.1 Obtaining a Blind CL signature

Here, we present the protocol to obtain the CL signature on messages that are committed by the Recipient. Without loss of generality, let the first l out of k messages be the committed messages, and the rest be public messages.

Hence, both the Recipient and the Issuer knows the Fujisaki-Okamoto commitments $C_1, \ldots, C_l$ to messages $x_1, \ldots, x_l$ (note that another version can just use one commitment to all $x_1, \ldots, x_l$ under different bases. The extension is very straightforward, and hence not shown). The Issuer knows the public (or issuer-chosen) messages $x_{l+1}, \ldots, x_k$. The Recipient knows all the messages

$x_1, \ldots, x_k$.

Algorithm 22: This procedure is run by the Recipient of a blind CL signature to initiate the signature issuing process.

Input: CL signature public key $CLPK = n, f, g_1, \ldots, g_m, h$, messages $x_1, \ldots, x_k$, commitments $C_1, \ldots, C_l$ and their openings $openC_1, \ldots, openC_l$ which include $r_1, \ldots, r_l$ (if there was only one commitment to those messages, there's just on r)

Pre-conditions: Each message x_i needs to be in D_x.

`Receive`

1. Choose a random v' of length $RSALength + stat$.
2. Compute $C = h^{v'} \Pi_{i=1}^{l} g_i^{x_i}$.
3. Prove using PoEoDLR protocol that the discrete logarithm representation of each secret x_i in C corresponds to those in $C_1, \ldots, C_l$ and also prove that each secret x_i is in D_x (note that for $D_x = [-(2^{l_x} - 1), 2^{l_x} - 1]$, this is a SpecialRange proof).

Algorithm 23: This procedure is run by the Issuer of a blind CL signature to issue a blind CL signature.

Input: CL signature public key $CLPK = n, f, g_1, \ldots, g_m, h$, secret key $CLSK = p, q$, messages $x_{l+1}, \ldots, x_k$, commitments $C_1, \ldots, C_l$

Pre-conditions: Each message x_i needs to be in D_x.

Output: partial signature $\sigma' = A, e, v''$

`Issue`

1. Pick a random **prime** number e of length l_e.
2. Choose a random v'' of length $RSALength + l_x + stat$.
3. Compute $A = [fCh^{v''} \Pi_{i=l+1}^{k} g_i^{x_i})]^{1/e}$. Let us explain this step in detail. The given commitment is re-randomized using v''. Then, public messages are added to the signature in the Π clause. Finally, the signature is computed.
4. Send A, e, v'' to the Recipient and prove using PoKoDLR protocol the knowledge of $1/e$ in the equation above.

Algorithm 24: This procedure is run by the Recipient of a blind CL signature to construct a CL signature upon receipt of the partial signature.

Input: CL signature public key $CLPK = n, f, g_1, \ldots, g_m, h$, partial signature $\sigma' = A, e, v''$

Pre-conditions: Each message x_i needs to be in D_x. e must be a prime of length l_e, v'' needs to be of length l_v.

Output: signature $\sigma = A, e, v$

`Construct`

1. Check the range (and optionally primality) for e
2. Set $v = v' + v''$
3. Output signature $\sigma = \{A, e, v\}$

6.2 Proving a CL signature

Now that the Recipient has a blind CL signature, he wants to prove that fact to a verifier, without revealing the signature. Without loss of generality, let the first l out of k messages be the committed

messages, and the rest be public messages.

Hence, both the Recipient and the Verifier knows the Fujisaki-Okamoto commitments $C_1, \ldots, C_l$ to messages $x_1, \ldots, x_l$ (as before, there can be just one commitment to all those messages). The Verifier knows the public (or issuer-chosen) messages $x_{l+1}, \ldots, x_k$. The Recipient knows all the messages $x_1, \ldots, x_k$ and the signature $\sigma = A, e, v$. Of course, both parties know the CL signature public key $CLPK = n, f, g_1, \ldots, g_m, h$.

- The Recipient choses a random number r from $\{0,1\}^{RSALength+stat}$.
- The Recipient computes $A' = Ah^r$ and sends A' to the Verifier. Set $v' = v + r * e$.
- The Verifier and the Recipient both separately computes the public value $D = \Pi_{i=l+1}^{k} g_i^{x_i}$ using public messages $x_{l+1}, \ldots, x_k$.
- The Recipient computes the commitment $C = h^{r_C} \Pi_{i=1}^{l} g_i^{x_i}$ for secret messages (using a random r_C from $\{0,1\}^{RSALength+stat}$), and proves to the Verifier using PoEoDLR protocol that the discrete logarithm representation of each secret x_i in C corresponds to those in $C_1, \ldots, C_l$ and also proves that each secret x_i is in D_x (note that for $D_x = [-(2^{l_x} - 1), 2^{l_x} - 1]$, this is a SpecialRange proof), and of course the knowledge of r_C.
- Lastly, the Recipient proves using PoKoDLR protocol the knowledge of e, v' such that $A'^e h^{r_C} = fh^{v'}CD$ (actually, prove that $fCD = A'^e * h^{(r_C - v')}$ or equivalently $fCD = A'^e * (1/h)^{v' - r_C}$).

7 E-cash

In this section, we provide a description of the offline versions of compact e-cash [1] and endorsed e-cash [2]. We require that all connections between a user and the bank must be over an authenticated and secure channel (i.e., an SSL connection using the bank's certificate).

7.1 Compact E-Cash

Compact E-Cash [**?**] enables a user to withdraw a wallet containing many coins at once. But, the coins need to be spent one by one. Some extensions addressing this issue will be discussed later. Using offline Compact E-Cash, the Bank can find the public keys of double-spenders. Double deposits can easily be detected using serial numbers. The use of CL signatures assure that the serial numbers are not known to the Bank during the withdrawal process, and hence the anonimity of an honest user is guaranteed.

Compact E-Cash works with a prime-order group, and utilizes CL signatures, so uses a special RSA group for the purposes of CL signatures. It's important that the *orderLength* $\leq l_x$, the CL signature message length.

Below, we will present a slightly modified version of Compact E-Cash. The modifications keep the scheme secure, while improving its efficiency greatly.

Assumptions:

Compact E-cash works in the Random Oracle model and makes Discrete Logarithm and Strong RSA assumptions, and also the following assumptions:

q-Decisional Diffie-Hellman Inversion Assumption:

7.1.1 Register

Every user must register with the Bank her public key. This is a very simple protocol, where the User just picks a random secret key $1 < sk_u < primeOrder$ and sends her public key $pk_u = g^{sk_u} \bmod primeModulus$ to the Bank. The Bank registers the User's public key in a database, and associates it to an account.

Later on, whenever a user contacts the Bank and needs to prove her identity, the User proves knowledge of sk_u that corresponds to pk_u using the PoKoDLR protocol using only one base and Pedersen commitments.

7.1.2 Withdraw

To withdraw a coin, a user contacts the Bank. Before this, the User must have been registered with the Bank. First, the User proves her identity to the Bank. Then, the User and the Bank execute the following randomization procedure:

Algorithm 25: Randomization procedure between the User and the Bank. This procedure needs to be executed for each wallet, before the withdrawal.

Input: Both parties know the definition of the prime-order group. The User knows his secret key sk_u registered with the bank.

Pre-conditions: The group must be generated by a trusted third party, or the Prover must verify that each g_i, h has order $primeOrder$ in a prime-order group.

Output: The User's output is s, t, A, the Bank's output is A.

`Randomize Together`

1. The User picks secrets s' and t from $Z^*_{primeOrder}$.
2. The User creates a Pedersen commitment to sk_u, s', t. Call this commitment A'. The User sends A' to the Bank.
3. The Bank picks a random number r' from $Z^*_{primeOrder}$. The Bank sends r' to the User.
4. The User sets $s = s' + r'$. The User and the Bank independently compute $A = A' * g^{r'}$.
5. The User's output is s, t, A, the Bank's output is A.

After the Randomize Together protocol, the User gets a blind CL signature from the Bank. The User starts with the commitment A, which he needs to prove that the secrets in A correspond to the the secrets he committed to in the CL signature protocol, and the first secret corresponds to his secret key. Furthermore, the User also sends the Bank a wallet size W, which must be picked from a choice of wallet sizes (*e.g.*, 1,10,100,1000,10000) and must be at most the User's current account balance. After checking the User's balance, the Bank signs sk_u, s, t, W using blind CL signatures (where W is public input), and decrements the User's account balance. At the end of the Withdraw protocol, the User's wallet is composed of $sk_u, s, t, W, \sigma(sk_u, s, t, W)$ where $\sigma(sk_u, s, t, W)$ denotes the Bank's CL signature on sk_u, s, t, W, and a data structure to keep track of spent and remaining coins in the wallet (even though this can be a simple counter, we would like to spend coins with random indices, and so we need to keep a shuffled list of coin indices).

7.1.3 Spend

This is the protocol between a user and a merchant. To prevent man-in-the-middle attacks, before the exchange of the money, the User and the Merchant perform a secure key exchange protocol

without setup. This can be a simple Diffie-Hellman key exchange over an RSA group using fresh randomly generated keys for both parties. Let the session secret derived from the key exchange be $ses = hash(\text{session key})$.

At the beginning, the Merchant picks a random *info* and sends this to the User. Both the Merchant and the User computes $R = hash(ses, info)$. Note that only the User and the Merchant can compute this value. Furthermore, the Merchant needs to use a different *info* for every transaction. This is to prevent man-in-the-middle attacks.

Next, the User picks the next unused coin index (*i.e.* the next random index in the shuffled list of coin indices). Call this index J.

Algorithm 26: Spend-Earn procedure between the User and the Merchant for Compact E-Cash.

Input: Both parties know the definition of the prime-order group, and the Bank's CL signature public key $CLPK$. The User knows her wallet $sk_u, s, t, W, \sigma(sk_u, s, t, W)$, and the index J. Both parties know $R = hash(info, ses)$.

Pre-conditions: The User must have withdrawn a wallet from the Bank, and the wallet must contain an unused coin. Furthermore, R must be computed as directed above.

Output: The Merchant outputs $coin = B, C, D, W, S, T, J, \pi_{CL}, \pi_{ST}, info, ses$.

`Spend Compact E-Cash`

1 The User creates Pedersen commitments B to sk_u, C to s, and D to t. The User prepares a non-interactive proof of knowledge of a CL signature on these values, where W is a public value. Call this proof π_{CL}.

2 The User computes $S = g^{1/(s+J)}$ and $T = pk_u * g^{R/(t+J)}$.

3 The User prepares a non-interactive proof that S, T are formed correctly. This is done by proving the knowledge of $s, t, sk_u, r_B, \alpha, r_1, \beta, r_2$ such that $B = g^{sk_u} * h^{r_B}$, $g = (g^J * C)^{\alpha} h^{r_1}$ (the Prover knows $r_1 = -r_C/(s+J) \mod primeOrder$), $g = (g^J * D)^{\beta} h^{r_2}$ (the Prover knows $r_2 = -r_D/(t+J) \mod primeOrder$), $S = g^{\alpha}$, $T = g^{sk_u} * (g^R)^{\beta}$. Call this proof π_{ST}.

4 The User sends $B, C, D, W, S, T, J, \pi_{CL}, \pi_{ST}$ to the Merchant.

5 The Merchant checks if the coin index is correct: $0 \leq J < W$, the proof of knowledge of the CL signature π_{CL} verifies, and S, T are correct under the proof π_{ST}. If all these are satisfied, the Merchant stores $coin = B, C, D, W, S, T, J, \pi_{CL}, \pi_{ST}, info, ses$.

7.1.4 Deposit

Depositing the coin is very easy. The Merchant contacts the Bank over a secure and one-way authenticated channel (*i.e.* SSL), proves his identity to the Bank, and then sends the coin to the Bank ($coin = B, C, D, W, S, T, J, \pi_{CL}, \pi_{ST}, info, ses$). The Bank performs the exact check on line 5 of the Spend protocol in Algorithm 26. This verifies that the coin is formed correctly.

The bank then performs double-spending checks. First, the bank checks if S appears in the list of deposited coins. If it does not exist in the deposited coins database, the Bank credits the Merchant's account, and adds the coin to the database. If S indeed does exist, it exists together with an R'. If $R = R'$, then the Merchant is trying to deposit twice, and thus the Bank rejects the deposit. Otherwise, $R \neq R'$, and so the User double-spent the coin. The Bank runs the following Identify Double-Spender algorithm to find the public key (and hence the account) of the

double-spender. An appropriate punishment can be performed afterwards.

Algorithm 27: Identify Double-Spender procedure run by the Bank.

Input: Two coins $coin_1, coin_2$ with the same serial number S but different R_1, T_1 and R_2, T_2.
Output: the public key of the double-spender pk_u, and proof of double-spending $\pi_{DS} = coin_1, coin_2$.

`Identify Double-Spender`

1 Verify the coins, if not already verified.
2 The Bank computes $pk_u = (T_2^{R_1}/T_1^{R_2})^{(R_1-R_2)^{-1}} \mod primeModulus$.
3 The Bank outputs the public key of the double-spender pk_u, and proof of double-spending $\pi_{DS} = coin_1, coin_2$.

Given the proof of double-spending, anyone can run the Identify Double-Spender protocol and be assured that the user with public key pk_u has really double-spent. This is because the coins require the knowledge of the secret key of the user.

7.2 Endorsed E-Cash

The Withdraw, Deposit, and Identify Double-Spenders protocols of Endorsed E-Cash [**?**] are the same as the Compact E-Cash. It makes different assumptions, though.

Assumptions:

Endorsed E-cash works in the Random Oracle model and makes Discrete Logarithm and Strong RSA assumptions, and also the following assumptions:

Diffie-Hellman Assumption:

q-Diffie-Hellman Inversion Assumption:

7.2.1 Spend

The only different part of the Endorsed E-Cash is how to spend a coin. The spend algorithm below is run by the user, who, on input a valid set of values for $(info, ses)$, outputs the unendorsed coin

and its endorsment.

Algorithm 28: The algorithm whereby a user generates an unendorsed coin together with a valid endorsement

Input: The User, as well as the Merchant who will ultimately receive the coin, know the definition of the prime-order group, and the Bank's CL signature public key $CLPK$. The User knows her wallet $sk_u, s, t, W, \sigma(sk_u, s, t, W)$, and the index J. Both parties know $R = hash(info, ses)$.

Pre-conditions: The User must have withdrawn a wallet from the Bank, and the wallet must contain an unused coin. Furthermore, R must be computed as directed above.

Output: The unendorsed coin is $uecoin = (B, C, D, W, S, T, J, \pi_{CL}, \pi_{ST})$. The corresponding endorsement is $endorsement = (x_1, x_2, r_y)$

`Spend Endorsed E-Cash`

1 The User creates Pedersen commitments B to sk_u, C to s, and D to t. The User prepares a non-interactive proof of knowledge of a CL signature on these values, where W is a public value. Call this proof π_{CL}.

2 The User picks random x_1, x_2, r_y from $Z^*_{primeOrder}$. The User sets $y = g_1^{x_1} * g_2^{x_2} * f^{r_y}$; i.e., y is a Pedersen commitment to (x_1, x_2, x_3). (The bases used for y are also generators of the prime-order group, but they are different generators than used for B, C, D, S, T.)

3 The User computes $S = g^{1/(s+J)} * g^{x_1}$ and $T = pk_u * g^{R/(t+J)} * g^{x_2}$.

4 The User prepares a non-interactive proof that S, T, E are formed correctly. This is done by proving the knowledge of $s, t, sk_u, r_B, \alpha, r_1, \beta, r_2, x_1, x_2, r_y$ such that $y = g_1^{x_1} * g_2^{x_2} * f^{r_y}$, $B = g^{sk_u} * h^{r_B}$, $g = (g^J * C)^{\alpha} h^{r_1}$ (the Prover knows $r_1 = -r_C/(s+J) \mod primeOrder$), $g = (g^J * D)^{\beta} h^{r_2}$ (the Prover knows $r_2 = -r_D/(t+J) \mod primeOrder$), $S = g^{\alpha} * g^{x_1}$, $T = g^{sk_u} * (g^R)^{\beta} * g^{x_2}$. Call this proof π_{ST}.

5 The unendorsed coin is $uecoin = (B, C, D, W, S, T, J, \pi_{CL}, \pi_{ST}, y)$.

6 The endorsement is (x_1, x_2, r_y).

Algorithm 29: The algorithm to verify an unendorsed coin.

Input: The input is a purported unendorsed coin $uecoin = (B, C, D, W, S, T, J, \pi_{CL}, \pi_{ST})$.

Pre-conditions: The Merchant verifying the coin knows the definition of the prime-order group, the Bank's CL signature public key $CLPK$, and the values $info$, ses and $R = hash(info, ses)$.

Output: Accept or reject

1 To verify that $uecoin$ is a valid unendorsed coin for a given $R = hash(info, ses)$, check that (1) the coin index is correct: $0 \leq J < W$; (2) the proof of knowledge of the CL signature π_{CL} verifies; (3) S, T, y are correct under the proof π_{ST}. If all three checks pass, output Accept. Else, output Reject.

Algorithm 30: The algorithm to verify the endorsement for a given unendorsed coin.

Input: The input is an unendorsed coin $uecoin = (B, C, D, W, S, T, J, \pi_{CL}, \pi_{ST})$ and an endorsement $endorsement = (x_1, x_2, r_y)$.

Pre-conditions: The Merchant verifying the coin knows the definition of the prime-order group; the unendorsed coin $uecoin$ has already been verified using Algorithm 29.

Output: Accept or reject

1 Check that y (from the unendorsed coin) is as follows: $y = g_1^{x_1} g_2^{x_2} f^{r_y}$. If so, accept, else, reject.

The Deposit protocol consists of doing Algorithms 29 and 30 together. Notice that, the resulting coin is the same as in the Compact E-Cash, except that the proofs π_{CL}, π_{ST} are slightly different, and that we should use $S' = S/g^{x_1}, T' = T/g^{x_2}$ for double-spending detection and identification purposes.

The Withdraw protocol stays the same since no extra information needs to be stored in the wallet for the Endorsed E-Cash. The usefulness of Endorsed E-Cash will be clear once we see Verifiable Encryption and Fair Exchange protocols.

7.3 E-Cash FAQ

Q: Why do we have Algorithm 25 before the withdrawal to randomize s?
A: If multiple users use the same s, then their coins will have identical S values. This means, to identify a double-spender, the Bank needs to check all 2-combinations of those coins. Therefore, malicious users might mount a denial-of-service attack on the Bank this way.

Q: Why does the user pick the wallet size W from only a limited list of wallet sizes?
A: The problem occurs since for efficiency we send both W and the coin index J in clear when spending. Consider a wallet size of 1 million coins. Only very few people will have such a wallet, and once anyone sees a coin index that's very large, the spender can be identified as one of the rich guys. A fully flexible and secure version would commit to W and J and change the proofs accordingly. A range proof that proves a value is in a committed range (as opposed to a public range) is also required.

Q: Why do we need Endorsed E-Cash?
A: The values x_1, x_2, r_y can be easily used in verifiable encryption to prove that y, and hence the coin is formed correctly. Then, it is very easy and fast to fairly exchange x_1, x_2, r_y with the material being bought.

Q: What is the deal with the randomness R in the coin?
A: We use $R = hash(ses, info)$ for mainly two reasons. Use of *ses* ensures that no one other than the User and Merchant knows that value, and thus prevents man-in-the-middle attacks. The use of *info* ensures that the Merchant uses different R values for each transaction, which means all the coins he receives will be honored, and so the Bank can catch double-spenders.

Note that other types of contracts can also be used. What is required from a contract is that it is different for each transaction. This benefits the Merchant and the Bank, but not the User who wants to cheat. Therefore, the Merchant needs to input enough randomness (in the form of *info* above) into the contract. The other use of the contract prevents man-in-the-middle attacks. Consider the case above where *ses* is not used and only *info* is used. Then, an attacker can just forward the coin he gets from the User to the Merchant. One possible problem of this is that the man-in-the-middle gets the service in effect "for free". This may not present a problem in all the scenarios, but we choose to be on the safe side. With *ses* in use, the connections of the attacker to the User and the Merchant will be distinguished.

Q: We can identify the public keys of double-spenders, but can we do more?
A: Another version of e-cash is available where the Bank gets enough information to trace all the coins spent from a wallet of a double-spender. The idea is that during withdrawal, the Bank learns a verifiable encryption of the wallet secret under some other secret. Later on, when double-spending occurs, the Bank learns the secret encryption key similar to the way he learns the User's public key. Using the wallet decrypted secret, the Bank can create a blacklist for wallets/coins that the merchants should not accept. This way, further double-spending will be prevented online.

Unfortunately, this version reveals all purchases of the double-spender. The double-spending may be accidental (faulty software, stolen card, *etc.*), and so the privacy of an innocent user may be lost this way. There are "glitch protection" methods [**?**] to let some number of accidents happen, but this means all the adversarial users will double-spend some allowed number of coins every time. Therefore, blacklisting only "future" coins is an important open problem.

8 Verifiable Encryption

We will be using Caminisch-Shoup verifiable encryption [**?**]. This will later be the verifiable escrow method used in fair exchange.

ASSUMPTIONS:

The security of the scheme relies on Strong RSA assumption, Paillier's Decision Composite Residuosity assumption [**?**], and existence of a collision-resistant family of hash functions.

PARAMETERS:

The parameter m denotes the number of messages that can be verifiably encrypted at once. For Endorsed E-Cash, we need $m = 3$.

SETUP:

The key generation of CS verifiable encryption uses two runs of special RSA group generation as in Algorithm 1. The important differences are clearly identified below:

- Let N be the modulus returned by the first special RSA group generation. The generators will be picked in a completely different way. In fact, the group that will be in use will be a subgroup of $Z^*_{N^2}$ instead of Z^*_N.
- Pick a random number f' from $Z^*_{N^2}$. Compute $f = f'^{2n} \mod N^2$.
- Pick random numbers $x_1, \ldots, x_m, y, z$ from the interval $(0, N^2/4)$. Compute $a_i = f^{x_i} \mod N^2$ for $1 \leq i \leq m$, $d = f^y \mod N^2$, $e = f^z \mod N^2$.
- Compute $b = (1 + N) \mod N^2$. Notice that b is an element of order N in $Z^*_{N^2}$.
- Let the second special RSA group generation return a group with modulus n and generators $g_1, \ldots, g_m, h$.
- Also, pick a key hk as the key for a keyed hash function. For simplicity in presentation below, we will omit the key in hashes, but it will be used for every hash calculation.
- The public key (*i.e.* encryption and verification key) is composed of all public parts of the groups, and the hash function key: $VEPK = N, a_1, \ldots, a_m, b, d, e, f, n, g_1, \ldots, g_m, h, hk$.
- The secret key (*i.e.* decryption key) is the secret parts of the groups $VESK = P, Q, x_1, \ldots, x_m, y, z, p, q$ such that $N = PQ$ and $n = pq$.

The absolute value algorithm below will be used in the verifiable encryption scheme a lot.

Algorithm 31: Absolute Value procedure for Camenisch-Shoup verifiable encryption scheme.

Input: x in range $(0, N^2)$
Pre-conditions: x needs to be in range $(0, N^2)$
Output: $abs(x)$

```
abs
1  if x > N^2/2 then
2      Output (N^2 - x) mod N^2
3  else
4      Output x mod N^2
```

8.1 Encrypt

The encryption procedure creates a ciphertext. To turn it into verifiable encryption, we will need some specialized non-interactive zero knowledge proofs. Just as we need for fair exchange, this is a labeled encryption scheme [**?**]. The *label* is also known as *tag*, or *contract*.

Algorithm 32: Encryption procedure for Camenisch-Shoup verifiable encryption scheme. This procedure is run by the Encryptor. This is a sub-procedure; it's not verifiable yet.

Input: Verifiable encryption public key $VEPK = N, a_1, \ldots, a_m, b, d, e, f, n, g_1, \ldots, g_m, h, hk$, messages to be verifiably encrypted $x_1, \ldots, x_m$, a label L
Output: Ciphertext $u_1, \ldots, u_m, v, w$.

```
Encrypt
1  Pick a random number r from the interval (0, N/4).
2  for i : 1..m do
3      Compute u_i = b^{x_i} * a_i^r mod N^2.
4  Compute v = f^r mod N^2
5  Compute w = abs([d * e^{hash(u_1||...||u_m||v||L)}]^r mod N^2)
6  Output u_1, ..., u_m, v, w.
```

8.2 Verifiably Encrypt

We will now provide a conversion from regular encryption to verifiable encryption for Camenisch-Shoup verifiable encryption scheme. This will involve non-interactive zero knowledge proofs.

This verification can be used to prove correct encryption of discrete logarithms. When we consider Endorsed E-Cash, the Prover will verifiably encrypt x_1, x_2, r_y such that $y = g_1^{x_1} * g_2^{x_2} * f^{r_y}$ in the group chosen by the Endorsed E-Cash setup. Take a note that it is a different group than

the ones used in verifiable encryption here. X below will be y of the Endorsed E-Cash.

Algorithm 33: Verifiable Encryption procedure for Camenisch-Shoup verifiable encryption scheme. This procedure is run by the Encryptor/Prover.

Input: Verifiable encryption public key $VEPK = N, a_1, \ldots, a_m, b, d, e, f, n, g_1, \ldots, g_m, h, hk$, commitment X to all messages $x_1, \ldots, x_m$ (or a commitment X_i to each one of them), a label L

Output: Ciphertext and its proof $u_1, \ldots, u_m, v, w, X', \pi_{VE}$.

`Verifiably Encrypt`

1. Get the ciphertext $u_1, \ldots, u_m, v, w$ from the Encrypt function in Algorithm 32, along with the randomness r used in the encryption.
2. Pick a random number s from the interval $(0, N/4)$.
3. Compute a Fujisaki-Okamoto commitment X' to $x_1, \ldots, x_m$ using the randomness s in the group defined by $n, g_1, \ldots, g_m, h$ (the second RSA group generated for verifiable encryption purposes).
4. Prove knowledge of $r, s, x_1, \ldots, x_m$ such that $v^2 = f^{2r} \mod N^2$, $w^2 = [d * e^{hash(u_1||\ldots||u_m||v||L)}]^{2r} \mod N^2$, and that each $u_i^2 = b^{2x_i} * a_i^{2r} \mod N^2$, and that each x_i corresponds to the one in X' and X, and that the range of each x_i is $(-N/2, N/2)$ (SpecialRange). Call this proof π_{VE}.
5. Output $u_1, \ldots, u_m, v, w, X', \pi_{VE}$.

Any party with the verifiable encryption public key can verify the encryption.

Algorithm 34: Verification procedure for Camenisch-Shoup verifiable encryption scheme. This procedure is run by the Verifier.

Input: Verifiable encryption public key $VEPK = N, a_1, \ldots, a_m, b, d, e, f, n, g_1, \ldots, g_m, h, hk$, commitment X (or commitments $X_1, \ldots, X_m$), a label L, ciphertext and its proof $u_1, \ldots, u_m, v, w, X', \pi_{VE}$

Output: ACCEPTor REJECT.

`Verify`

1. Verify π_{VE}. If verification fails, output REJECT.
2. **if** $abs(w) = w \mod N^2$ **then**
3. Output ACCEPT.
4. **else**
5. Output REJECT.

8.3 Decrypt

Only a party equipped with the correct secret key can decrypt. In the fair exchange, this party will be the Arbiter.

Algorithm 35: Decryption procedure for Camenisch-Shoup verifiable encryption scheme. This procedure is run by the Decryptor/Arbiter.

Input: Verifiable encryption public key $VEPK = N, a_1, \ldots, a_m, b, d, e, f, n, g_1, \ldots, g_m, h, hk$, associated secret key $VESK = P, Q, x_1, \ldots, x_m, y, z, p, q$, a label L, ciphertext $u_1, \ldots, u_m, v, w$

Output: Plaintext $m_1, \ldots, m_m$ or ERROR.

```
Decrypt
1  if abs(w) ≠ w mod N^2 OR v^{y+z*hash(u_1||...||u_m||v||L)} ≠ w^2 mod N^2 then
2      Output ERROR.
3  Compute t = 2^{-1} mod N.
4  for i : 1..m do
5      Compute m'_i = (u_i / v^{x_i})^{2t} mod N^2.
6      Set m_i = (m'_i - 1)/N mod N^2.
7      if m_i is not in range (0, N) then
8          Output ERROR.
9  Output m_1, ..., m_m
```

8.4 Prove Decryption

This part is complicated, and for now, will not be employed since the Arbiter in the fair exchange will be a trusted entity.

9 Fair Exchange

Use regular encryption, signatures, public-key encryption, e-cash and verifiable encryption to perform fair exchange.

9.1 Buy

Let Alice be the *buyer* and Bob be the *seller*. As a precondition, the protocol assumes that Alice already knows *bhash*, which is the root of the Merkle hash tree computed on the block *block* that Alice wants to buy.

9.1.1 Merkle tree

First, we give the procedure $MHash(h, block, \ell)$ to compute *bhash*, the Merkle hash function that outputs the root of the depth-ℓ Merkle tree of a given *block* with hash function h. Next, we give (1) the procedure to compute the proof $MProve((h, block, \ell), i)$ that a particular value is the i^{th} leaf of the Merkle hash tree of which *bhash* is the root; the procedure $MVerify((h, \ell), (bhash, i, chunk))$ to verify the proof *MProof* that the value *chunk* is the i^{th} leaf of the Merkle hash tree of which *bhash* is the root.

First, let us explain the idea of the *MHash* procedure. Consider a rooted binary tree with 2^ℓ leaves (*i.e.* it is a binary tree of height ℓ). Associated with every node the tree, there is the address a of the node. Specifically, the label a associated with the root node is the empty string ε. The label of a left (resp. right) child is derived by concatenating 0 (resp., 1) to the label of the parent node. Thus, the label a associated with the i^{th} leaf is the integer i written in binary. Stored at a node labeled with a, is a value v_a. In a Merkle tree, stored at each leaf $0 \leq a \leq 2^\ell - 1$ is the value $v_a = h(chunk_a)$, where $chunk_a$ is the a^{th} chunk of the block; and stored at every internal node $0 \leq a \leq 2^i - 1$ at depth $i \geq 1$ is the value $v_a = h(v_{a||0}||v_{a||1})$. Finally, the value corresponding to the root node is $v_\varepsilon = h(v_0||v_1)$. The algorithm *MHash* outputs this value.

Algorithm 36: *MHash*: Generating a Merkle hash tree.

Input: A collision-resistant hash function $h : \{0,1\}^* \mapsto \{0,1\}^{hashLength}$, the data block *block*, the desired tree height ℓ.
Pre-conditions: None.
Output: The value *bhash*.
Post-conditions: None.

1 If $\ell = 0$, output $h(block)$.
2 Otherwise, divide the block *block* into $block_0$ and $block_1$, 2 strings of approximately equal byte lengths (this has to be done in a deterministic fashion so that running twice on same input gives same results) and output $h(MHash(h, block_0, \ell - 1)||MHash(h, block_1, \ell - 1))$

To prove that *chunk* is the i^{th} chunk of the block represented by $bhash = MHash(block)$, reveal the values v_{a_j} where for $1 \leq j \leq \ell$, the label a_j is obtained by taking the first $j - 1$ bits of the binary representation of i, and concatenating to them the negation of the j^{th} bit. (For example, if $i = 0101$, then $a_1 = 1$, $a_2 = 00$, $a_3 = 011$ and $a_4 = 0100$.) In the Merkle tree, the node a_j will be the sibling of the j^{th} node on the path from the root to the chunk in question. The procedure for doing this is as follows:

Algorithm 37: Generating the proof $MProof = (v_1, \ldots, v_j, chunk)$ that *chunk* is the i^{th} leaf of the Merkle tree.

Input: A collision-resistant hash function $h : \{0,1\}^* \mapsto \{0,1\}^{hashLength}$, the data block *block*, the desired tree height ℓ; the index i.
Pre-conditions: $0 \leq i < 2^\ell$
Output: The value *MProof*.
Post-conditions: None.

1 If $\ell = 0$, return *block*.
2 Otherwise, divide the block *block* into $block_0$ and $block_1$, 2 strings of approximately equal byte lengths (this has to be done in a deterministic fashion, the same as what is done by the algorithm *MHash*).
3 If $i \geq 2^{\ell-1}$, return $(MHash(h, block_0, \ell - 1), MProve((h, block_1, \ell - 1), i - 2^{\ell-1}))$.
4 Else return $(MHash(h, block_1, \ell - 1), MProve((h, block_0, \ell - 1), i))$.

To verify that $(v_1, ..., v_\ell)$ is a valid proof that *chunk* is the i^{th} chunk of *block* associated with *bhash*, we recompute the labels a_j (as above). We know that $v_j = v_{a_j}$ is the value that should be associated with node labelled a_j. Let $i = i_1 i_2 \ldots i_\ell$, *i.e.* i_j is the j^{th} bit of the ℓ-bit binary representation of i. Let $b_j = i_1 \ldots i_j$ be the j-bit prefix of i. First, we know that $v_i = h(chunk)$.

For each j, $\ell - 1 \geq j \geq 0$, compute $v_{b_j} = h(v_{b_j||0}||v_{b_j||1})$. We can do it because one of $(v_{b_j||0}, v_{b_j||1})$ is $v_{a_{j+1}}$, and the other one is computed in the previous step. Finally, verify that $v_\varepsilon = bhash$. The pseudocode for this procedure is as follows:

Algorithm 38: Verifying the proof $MProof = (v_1, \ldots, v_j, chunk)$ that $chunk$ is the i^{th} leaf of the Merkle tree.

Input: A collision-resistant hash function $h : \{0,1\}^* \mapsto \{0,1\}^{hashLength}$, the value $bhash$, the desired tree height ℓ; the proof $MProof$, the index i.
Pre-conditions: $0 \leq i < 2^\ell$
Output: Accept or reject.
Post-conditions: None.

```
1 TO BE WRITTEN
```

If two conflicting proofs can be constructed (*i.e.* for $chunk \neq chunk'$, there are proofs that each of them is the i^{th} chunk of $block$ associated with $bhash$), then a collision in h is found, contradicting the assumption that h is collision-resistant.

9.1.2 The Buy protocol

We present our protocol that lets the Buyer buy a file block from the Seller. Before the start of the protocol, the Buyer has (1) $bhash = MHash(block)$; and (2) has withdrawn a wallet from the Bank and is ready to create unendorsed coins together with their endorsements. from a trusted authority (*i.e.* tracker). Alice and Bob will agree on a *timeout* by when Bob must provide Alice with the block. The protocol works as follows:

9.2 Barter

9.2.1 Alice

9.2.2 Bob

9.2.3 Arbiter

A Common Functionalities

RANDOMIZATION:

Algorithm 39: Procedure to generate a random group element with associated random exponents. Call this procedure **Randomize**.

Input: Definition of the group, number of random elements k, number of fixed elements l, bases $g_1, \ldots, g_{k+l-1}, h$, fixed elements $x_1, \ldots, x_l$ if any.
Pre-conditions: $l < k$, $k \geq 1$, $l \geq 0$
Output: random element R, random exponenets $openR$

`Randomize`

1 Pick $k-1$ random numbers s_i from the domain of randomness D_R.
2 Pick another random number t from the domain of randomness D_R.
3 Compute $R = [\prod_{i=1}^{l} g_i^{x_i}][\prod_{i=1}^{k-1} g_{l+i}^{s_i}]h^t$ using group operations.
4 Output the random group element R and random exponents $openR = s_1, \ldots, s_{k-1}, t$.

References

[1] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. Compact e-cash. In *Proc. Eurocrypt '05*, volume 3494 of *Lecture Notes in Computer Science*, pages 302–321. Springer-Verlag, 2005.

[2] Jan Camenisch, Anna Lysyanskaya, and Mira Meyerovich. Endorsed e-cash. In *IEEE Symposium on Security and Privacy*, pages 101–115, 2007.

[3] Ivan Damgård and Eiichiro Fujisaki. A statistically-hiding integer commitment scheme based on groups with hidden order. In *Proc. Asiacrypt '02*, volume 2501 of *Lecture Notes in Computer Science*, pages 125–142. Springer-Verlag, 2002.

[4] Eiichiro Fujisaki and Tatsuaki Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In *Proc. Crypto '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 1997.

[5] Alfred J. Menezes, Paul van Oorschot, and Scott Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[6] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Proc. Crypto '91*, volume 576 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[7] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.