Fast Secure Multiparty ECDSA with Practical Distributed Key Generation and Applications to Cryptocurrency Custody*

Iftach Haitner†‡ Yehuda Lindell.§ Ariel Nof¶ Samuel Ranellucci†

May 29, 2023

Abstract

ECDSA is a standardized signing algorithm that is widely used in TLS, code signing, cryptocurrency and more. Due to its importance, the problem of securely computing ECDSA in a distributed manner (known as threshold signing) has received considerable interest. Despite this interest, however, as of the time of publication of the conference version of this paper ([LN18]), there had been no full threshold solution for more than two parties (meaning that any $t$-out-of-$n$ parties can sign, security is preserved for any $t-1$ or fewer corrupted parties, and $t \leq n$ can be any value) that supports practical key distribution. All previous solutions for this functionality utilized Paillier homomorphic encryption, and efficient distributed Paillier key generation for more than two parties is not known.

In this paper, we present the first (again, for the conference version publication time) truly practical full threshold ECDSA signing protocol that has fast signing and key generation. This solves an old open problem and opens the door to many practical uses of threshold ECDSA signing that are in demand today. One of these applications is the construction of secure cryptocurrency wallets (where key-shares are spread over multiple devices, and so are hard to steal) and cryptocurrency custody solutions (where large sums of invested cryptocurrency are strongly protected by splitting the key between a bank/financial institution, the customer who owns the currency, and possibly a third-party trustee, in multiple shares at each). There is growing practical interest in such solutions, but prior to our work, these could not be deployed due to the need for a distributed key generation.

*An extended abstract of this work (by the second and third authors) appeared as [LN18]. This full version includes full security proofs, and a new many-party multiplication protocol.

†Coinbase Ltd. Email: {iftach.haitner,yehuda.lindel,samuel.ranellucci}@coinbase.com.

‡The Blavatnik School of Computer Science at Tel-Aviv University.

§Part of this work was conducted while at the department of Computer Science at Bar-Ilan University, supported by the European Research Council under the ERC consolidators grant agreement n. 615172 (HIPS), by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Directorate in the Prime Minister’s Office, and by the Alter Family Foundation.

¶Department of Computer Science, Bar-Ilan University. Email: ariel.nof@biu.ac.il.
## Contents

1 Introduction
   1.1 Background and Prior Work ........................................ 4
   1.2 Our Results .................................................. 5
   1.3 Cryptocurrency Wallets and Custody ................................ 6
   1.4 Concurrent and Subsequent Work ................................... 6

2 Preliminaries
   2.1 Notation .................................................... 7
   2.2 Universal Composability (UC) .................................... 8
   2.3 The ECDSA Signature Scheme ..................................... 9
      2.3.1 The Standalone Scheme ....................................... 9
      2.3.2 The Multiparty Ideal Functionality .......................... 10
   2.4 Pedersen Commitments .......................................... 10
   2.5 ElGamal Commitments ........................................... 11
   2.6 Zero Knowledge ................................................. 11
   2.7 String Commitments ............................................ 13
      2.7.1 Committed Non-Interactive Zero Knowledge .................. 13

3 ECDSA Protocol
   3.1 Security .................................................... 18
   3.2 Using ElGamal Commitments ..................................... 22
   3.3 Many Honest Parties ............................................ 22
      3.3.1 Security .................................................. 23
   3.4 Threshold Signatures ............................................ 24
      3.4.1 Security .................................................. 25
   3.5 A More Efficient Protocol ...................................... 25

4 Many-Party Leaky Multiplication
   4.1 Security .................................................... 28
   4.2 Multiplication Equality Test .................................... 30
      4.2.1 Security .................................................. 31
      4.2.2 Many Honest Parties ....................................... 33
   4.3 Weak Two-Party Multiplication .................................. 34
      4.3.1 Using Weak Two-Party Multiplication in Protocol 4.3 ....... 35
   4.4 Using ElGamal Commitments ..................................... 36
   4.5 A More Efficient Protocol ...................................... 36

5 Experimental Results ............................................... 36

A Implementing Auxiliary Functionalities
   A.1 Zero Knowledge ............................................... 41
      A.1.1 Knowledge of Pedersen Commitment .......................... 43
      A.1.2 Knowledge of ElGamal Commitment ........................... 44
      A.1.3 ElGamal Equals Pedersen .................................... 46
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1.4</td>
<td>Knowledge of ElGamal Scalar Product</td>
<td>47</td>
</tr>
<tr>
<td>A.1.5</td>
<td>Product of ElGamal Commitments</td>
<td>48</td>
</tr>
<tr>
<td>A.2</td>
<td>String Commitments</td>
<td>50</td>
</tr>
<tr>
<td>A.2.1</td>
<td>Committed Non-Interactive Zero Knowledge</td>
<td>50</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Background and Prior Work

In the late 1980s and the 1990s, a large body of research emerged around the problem of threshold cryptography; e.g., [Boy86; CH89; Des87; DF89; Gen+01; SG98; Sho00; MR04]. In its most general form, this problem considers the setting of a private key shared between \( n \) parties with the property that any subset of \( t \) parties may be able to decrypt or sign, but any set of fewer than \( t \) parties can do nothing. This is a specific example of secure multiparty computation, where the functionality being computed is either decryption or signing. Note that trivial solutions like secret-sharing the private key and reconstructing to decrypt or sign do not work since, once the key is reconstructed, any single party can decrypt or sign by itself from that point onward. Rather, the requirement is that a subset of \( t \) parties is needed for every private-key operation.

Threshold cryptography can be used in applications where multiple signers are needed to generate a signature, and likewise where highly confidential documents should only be decrypted and viewed by a quorum. Furthermore, threshold cryptography can be used to provide a high level of key protection. This is achieved by sharing the key on multiple devices (or between multiple users) and carrying out private-key operations via a secure protocol that reveals nothing but the output. This provides key protection since an adversary must breach multiple devices in order to obtain the key. After intensive research on the topic in the 1990s and early 2000s, threshold cryptography received considerably less interest in the last decade. However, interest has recently been renewed. This can be seen from the fact that a number of startup companies are now deploying threshold cryptography for the purpose of key protection [Por; Sec; Sep; Cur; Fir]. Another reason is due to the fact that ECDSA signing is used in Bitcoin and other cryptocurrencies, and the theft of a signing key can immediately be translated into concrete financial loss. Bitcoin has a multisignature solution built in, which is based on using multiple distinct signing keys rather than a threshold signing scheme. However, the flexibility of the Bitcoin multisig is limited in that it does not support arbitrary and complex access structures. In addition, multisig solutions, like the one used in Bitcoin, introduce anonymity and scalability problems (as discussed in [Gen+16, Section 6.3]), and do not support revoking a party’s share (which can be a crucial feature in some applications). Thus, a more general solution may be obtained via threshold cryptography.

Fast threshold cryptography protocols exist for a wide variety of problems, including RSA signing and decryption, ElGamal and ECIES encryption, Schnorr signatures, Cramer–Shoup, and more. Despite the above successes, and despite the fact that DSA/ECDSA is a widely-used standard, for a long time DSA/ECDSA resisted attempts at constructing efficient protocols for threshold signing. This seems to be due to the need to compute \( k^{-1} \) without knowing \( k \), were \( k \) is the signature “nonce” (see discussion in Lindell [Lin21]). The first solution to overcome this difficulty in the honest minority setting was by MacKenzie and Reiter [MR04], who use Paillier additively homomorphic encryption in order to generate a signature between two parties. Their protocol required heavy zero-knowledge proofs, but was improved in [Gen+16] and later in [Lin21]. More significantly to our setting, [Gen+16; Bon+17] show how to generalize the MacKenzie and Reiter paradigm to any number of parties and with a full threshold. This means that for any \( n \) number of parties and any threshold \( t \leq n \) (even \( t = n \)) it is possible for any subset of \( t \) parties to sign, and security is preserved in the presence of any subset of \( t-1 \) corrupted parties. This is a significant breakthrough, but falls short of providing a full solution in practice, since it requires distributed Paillier key generation. Although two-party distributed Paillier key generation can work in practice [Fre+18] (albeit
requiring about 40 seconds between two powerful servers), it was still unknown as to whether this can be done practically for more than two parties. Thus, despite decades of research in threshold cryptography and secure multiparty computation, the following basic question remained open (as of the conference version of this paper [LN18] publication time):

*Is it possible to construct a full-threshold protocol for multiparty ECDSA, with practical distributed key generation and signing (based on well-studied hardness assumptions)?*

We answer this question in the affirmative.

1.2 Our Results

We present a full-threshold ECDSA signing protocol which is the first to have practical distributed key generation and fast signing. The protocol has seven rounds in the OT-hybrid model. We achieve this breakthrough by replacing Paillier additively homomorphic encryption with ElGamal-in-the-exponent encryption, which support additively homomorphic encryption. This change enables us to compute an encrypted signature in a similar way to that of [Gen+16], except that, upon decryption, the parties only receive $s \cdot G$ (where $G$ is the generator of the elliptic curve group) and not $s$ itself, where $s$ is the desired portion of the signature. This is due to the fact that we use ElGamal “in the exponent”, and so only obtain the result “in the exponent”.¹ We overcome this by computing the signature value $s$, in parallel, using a method that guarantees privacy, but not correctness. The combination of the above approaches yields a secure solution, since the encrypted signature is verified, and so $s$-in-the-clear is only revealed once equality with the encrypted signature is validated. The above method has many significant advantages over using Paillier; the major ones are:

1. We do not need distributed key generation of Paillier keys, which is hard, but just distributed key generation of ElGamal keys, which is very easy.

2. Elliptic curve operations themselves are far more efficient than Paillier operations.

3. Zero-knowledge proofs that are very expensive in Paillier are far more efficient in an elliptic curve group (it is well-known that zero-knowledge is easier in known-order groups).

4. By working in the same elliptic curve group as the signature, all prime-order homomorphic operations automatically take place modulo the required group order $q$. This removes many of the difficulties associated with Paillier (which requires adding randomness, to enable effectively working over the integers, and then proving in zero-knowledge that the “correct amount” was added).

5. Our ECDSA protocol enforces that the parties use the correct value of $k^{-1}$ using a method similar to that of [Gen+16], i.e., they multiplicatively mask $k$ with a random value $\rho$, and then reveal $k \cdot \rho$. By working with ElGamal and not Paillier, we achieve this far more efficiently than does [Gen+16], and without the expensive Paillier-based zero-knowledge proofs that they require.

¹This is called ElGamal “in-the-exponent” due to typical multiplicative group notation. Specifically, encryption of some $x$ using generator $G$ and public-key $D$ is carried out by computing $(G^r, D^r \cdot G^x)$. In elliptic curve notation, this becomes $(r \cdot G, r \cdot D + x \cdot G)$. It is easy to see that this scheme is additively homomorphic, but that decryption only returns $x \cdot G$ and not $x$ itself. Since obtaining $x$ requires solving the discrete log problem, this is not possible (except for very small values of $x$).
Finally, we remark that our entire protocol works over any group for which there exists an efficient injective mapping into $\mathbb{Z}_q$, and thus can be used to securely compute DSA in exactly the same way.

1.3 Cryptocurrency Wallets and Custody

As mentioned above, one important application of our protocol in practice today is in the protection of cryptocurrency. Although there are differing opinions on the benefit of existing cryptocurrencies to society, it is well accepted that honest investors should be protected from mass theft that we are already seeing in this space. On the end-user side, a secure cryptocurrency wallet should enable the user to split the signing key amongst multiple devices, and require all (or a subset) in order to transfer money. On the financial institution side, there is real interest in full cryptocurrency custody solutions for large customers. Such a solution is intended for use by investors who wish to protect very large amounts of cryptocurrency (even in the billions) as part of their investment portfolio. Due to the high amount of funds involved, it is not possible to enable any single party to have access to the signing key. Furthermore, neither the bank nor the customer should have the ability to single-handedly transfer funds (the bank cannot, due to liability, and the customer cannot due to the fact that its systems are typically less secure than the bank). Thus, a natural solution is to split the signing key into multiple parts, both in the bank and the customer (and potentially an additional third trustee) and then require some threshold in each entity to sign.\footnote{We remark that offline solutions requiring physical presence of representatives of both the bank and the customer, as could be achieved using physically protected HSMs, are not viable due to the requirement of fast transfer in case of a cryptocurrency crash.} A full-blown solution for this will typically have different roles both at the bank and the customer (one set of parties would authorize the signature itself as being requested from the customer, another would verify that the transfer meets the agreed-upon policy, and so on). Since such solutions require complex access structures for signing, our protocol, which provides the first real solution for this problem, is of very practical relevance.

1.4 Concurrent and Subsequent Work

Concurrently to this work, Gennaro and Goldfeder\cite{GG18} and Doerner, Kondi, Lee, and Shelat\cite{Doe+19} also present multiparty ECDSA protocols with practical key generation. Our protocol and the protocol of\cite{GG18} have some similarity, but the methods used to prevent adversarial behavior are very different. One significant difference between the two results is the hardness assumption and security model. We prove that our protocol is secure under simulation-based definitions, showing that it securely computes a standard ideal functionality for ECDSA. In addition, we prove the security of our protocol under the standard assumptions that the DDH problem is hard and the hardness of the standalone ECDSA signature. In contrast,\cite{GG18} prove the security of their protocol under a game-based definition, and require DDH as well as an ad-hoc but plausible assumption called Paillier-ECR (a weaker version of a similar assumption first introduced in\cite{Lin21}). However, we note that the running times given in\cite{GG18} (even in the updated version) are for the original insecure version without expensive range proofs. It is unclear how expensive their protocol is when the range proofs are run. The protocol of\cite{Doe+19} is an extension of\cite{Doe+18} to the multiparty setting. Their approach is very different than ours, as their main goal is to reduce the computational cost of the parties. Indeed, their protocol requires very few public-key operations compared to ours. However, their bandwidth is much higher, and their number of rounds grows
with the number of parties, and so is not constant. In contrast, in this work, as well as in [GG18], the protocols are constant-round. Thus, their protocol may be favorable when the parties are all located in the same local network, whereas our protocol is likely to outperform theirs when the parties are remotely located.

In a more recent work, Canetti, Gennaro, Goldfeder, Makriyannis, and Peled [Can+20a] present a four-round protocol (one round less than our protocol), featuring identifiable abort (our does not) and secure against adaptive corruptions (our protocol only withstands static corruptions). On the downside, their protocol uses Paillier homomorphic encryption, and thus suffers from the disadvantages this encryption scheme induces (see Section 1.2); moreover, the parties require, as an auxiliary input, safe bi-primes, which are costly to generate. Finally, Smart and Alaoui [SA19] and Dalskov et al. [Dal+20] developed frameworks for threshold ECDSA based on any MPC protocol over the field $\mathbb{Z}_q$.

### Paper Organization

Security notions and some basic building blocks used in our protocol are given in Section 2. In Section 3, we define and implement our distributed ECDSA signing protocol, which uses a many-party multiplication protocol described in Section 4. Experimental results of the running time of our ECDSA protocol are given in Section 5.

## 2 Preliminaries

### 2.1 Notation

We use calligraphic letters to denote sets, upper-case characters for group elements, and lower-case characters for scalars. We use $||$ to denote string concatenation. For a prime $q$, let $\mathbb{Z}_q$ denote the finite field of size $q$. For $t \in \mathbb{N}$, let $[t] = \{1, \ldots, t\}$ and $(t) = \{0, \ldots, t\}$. For a (binary) relation $R$, let $R(x, w) = 1$ iff $(x, w) \in R$. Let $\text{accept} = \text{true} = 1$, and $\text{reject} = \text{false} = 0$.

Throughout this paper, we fix an (elliptic-curve) group $G$ of prime order $q$ with additively written group operation, and fix a generator $G$ of $G$. Each point $A$ of $G$ is of the form $A = (a, \cdot)$ for some $a \in \mathbb{N}$. We let $\tilde{G} \overset{\text{def}}{=} G \times G$ denote the product group that inherits its additive operation from $G$ in an element-wise manner, and denote the elements of $\tilde{G}$ using the $\sim$ notation, e.g., $\tilde{A} \in \tilde{G}$. We also fix a security parameter $\kappa \in \mathbb{N}$. All arithmetic is done over the relevant group/field in which the elements in consideration reside.

**Remark 2.1** (Fixed security parameter). *In the following text, we mostly focus on a fixed group $G$ and a fixed security parameter $\kappa$. This is, however, merely for ease of notation. Since all the reductions given use the adversary as a black-box and their running time is some (fixed) polynomial in the security parameter, the following text readily extends to the case that the group in consideration is determined by a (non-fixed) security parameter, and the basic group operation as well as the description of the group element, are polynomial in the security parameter. In particular, when referring to the standard discrete logarithm and DDH (Decisional Diffie Hellman) hardness assumptions, we formally mean with respect to an infinite group ensemble.*

**Communication model.** Throughout, we assume each pair of parties have authenticated point-to-point channel. We assume no broadcast channel.
2.2 Universal Composability (UC)

We prove the security of our protocols in the UC (universally composable) framework [Can01], with the following specifications:

- Security with *abort*: after getting its output, the adversary can instruct the ideal functionality to send ⊥ instead of the actual output to some of the parties.

- *Static* corruptions: the adversary chooses which parties to corrupt before the interaction starts.

In the rest of the paper UC security stands for its variant with the above specifications. When proving UC security, we make the following assumptions:

1. Suffices to provide *non-rewinding* simulators. Since all protocols we use have an implicit *start synchronization* phase (all parties are active in the first round), to prove security, it suffices to provide a straight-line (i.e., non-rewinding) simulator [Kus+10, Theorem 5].

2. We assume for simplicity that the environment provides a fresh session id on each invocation it makes. (This assumption can be enforced by instructing the parties to choose fresh session ids using a single-round weak coin-flipping protocol.\(^3\))

3. We assume for simplicity that all honest parties agree on the common input. (This assumption can be forced by the parties sending hash of their common input in the first round of the protocol).

4. We prove security in different hybrid models, where the parties have access to an ideal implementation of several (more basic) functionalities. By composability of UC protocols, given UC secure implementation of these functionalities, the resulting protocol is UC secure.

5. The ideal functionalities are aware of the identity of the corrupted parties, which allowed to provided additional inputs (as specified by the functionality), e.g., to set the value of share of the secret key, and to cause abort (also after receiving their outputs).

In addition, throughout the paper we use the following convention:

1. A corrupted party never sends a message to the honest party (apart from an aborting message) or a messages to the ideal functionalities, that it knows, with probability one, it will make them abort. This is without loss of generality, since such an adversary can be emulated by a one that aborts the relevant honest party, by sending it an abort message, before sending such messages.

2. When an honest party receives an aborting message from another party or from an ideal functionality, it immediately informs all other parties and aborts.

3. When proving security, we do not address the all-corrupted party case, since the emulation in this case it trivial in an hybrid world in which all ideal functionalities are efficiently computable.

\(^3\)The parties send uniform strings to each other, and the session id is set to the hash (random oracle) of their concatenation.
Remark 2.2 (Concrete security). While we do not state concrete security bound in this paper, since all reductions considered are non-rewinding, the security loss is proportional (linearly) to the running time of the honest parties and the adversary. As such, as long as the concrete security of the underlying assumptions is good enough, then so is the security of the constructed protocol.

2.3 The ECDSA Signature Scheme

We start with defining the standalone ECDSA key generation, signing and verification algorithms, and then define the multiparty ideal functionality that enables distributed parties to jointly compute these operations. The ECDSA schemes below are defined with respect to arbitrary hash function $H : \{0, 1\}^* \mapsto \mathbb{Z}_q$ (in practice, SHA256 is used).

2.3.1 The Standalone Scheme

Algorithm 2.3 (KeyGenerator—ECDSA key generation).

1. Sample $x \leftarrow \mathbb{Z}_q$.
2. Output $(x, X = x \cdot G)$.

Algorithm 2.4 (Signer—ECDSA signing).

Input: $x \in \mathbb{Z}_q$ and $m \in \{0, 1\}^*$.

Operation:

1. Sample $k \leftarrow \mathbb{Z}_q$, and let $R = (r', \cdot) = k \cdot G$.
2. Let $r = r' \mod q$ and $s = k^{-1} \cdot (H(m) + r \cdot x)$.
3. Output $(r, s)$.

Algorithm 2.5 (Verifier—ECDSA verification).

Input: $X \in \mathbb{G}$, $m \in \{0, 1\}^*$ and signature $(r, s) \in \mathbb{Z}_q^2$.

Operation: accept, i.e., output 1, if $s^{-1} \cdot (H(m) \cdot G + r \cdot X) = (r, \cdot)$.

In the following, when saying that the standalone ECDSA is secure, we mean that the above signature scheme is existentially unforgeable secure.

\footnote{To get unique signatures, one can change the above algorithms to consider $\min\{s, -s\}$ instead of $s$.}
2.3.2 The Multiparty Ideal Functionality

We only define distributed key generation and signing protocols (verification is, locally, done using the standalone algorithm described above). We focus on the $t$-out-of-$n$ access structure ($t$ out of $n$ parties are needed for signing).

Let Store$_k(v)$ be the functionality that stores the value $v$ under key $k$. The functionality aborts if the key $k$ exists in the database. Analogously, Retrieve($k$) returns the value stored under key $k$, and aborts if no such key exists. We address these keys as object ids, and typically name them oid$_x$, for some id $x \in \{0,1\}^*$.

**Functionality 2.6 ($F_{Ecdsa}$—Distributed ECDSA).**

Parameters: $t \leq n \in \mathbb{N}$.

Parties: $P_1, \ldots, P_n$.

Operation:

- Upon receiving (sid, Keygen) from all $n$ parties:
  1. Let $(x, X) \leftarrow$ KeyGenerator().
  2. Store$_{sid}(x)$.
  3. Send (sid, $X$) to all parties.

- Upon receiving (sid, Sign, oid$_x$, $m$) from $t$ parties:
  1. $x = $ Retrieve(oid$_x$).
  2. Send (sid, ($r$, $s$) = Signer$_x(m)$) to all parties.

2.4 Pedersen Commitments

We utilize Pedersen commitments. A Pedersen commitment, of a value $m \in \mathbb{Z}_q$ with public key (parameter) $D \in \mathbb{G}$ and randomness $r \in \mathbb{Z}_q$, is defined by

$$\text{PedCom}_D(m;r) = r \cdot D + m \cdot G,$$

and PedCom$_D(m)$ stands for PedCom$_D(m;r)$, where $r \leftarrow \mathbb{Z}_q$. We typically denote elements of $\mathbb{G}$ that are outputs of PedCom using the $\hat{\cdot}$ symbol, e.g., $\hat{A} = \text{PedCom}_D(m)$.

**Fact 2.7 ([Ped91]).** PedCom is perfectly hiding. Assuming discrete log is hard over $\mathbb{G}$, it is computationally binding.

Pedersen commitments are additively homomorphic: for two commitments $\hat{A}_1 = \text{PedCom}_D(m_1; \cdot)$ and $\hat{A}_2 = \text{PedCom}_D(m_2; \cdot)$, it holds that $\hat{A}_1 + \hat{A}_2$ (where addition is over the group $\mathbb{G}$), is a commitment to $m_1 + m_2$. The latter can be combined with rerandomization by adding a fresh commitment to 0. We use the following syntactic sugar operation for $a \in \mathbb{Z}_q$ and $\hat{A} \in \mathbb{G}$:

- Let $\hat{A} +_D a$ stands for $\hat{A} + \text{PedCom}_D(a; 0)$. (Adding deterministic commitment to $a$.)
2.5 ElGamal Commitments

In Section 4.2, we utilize ElGamal in-the-exponent encryption\(^5\) for commitments, which we refer to as ElGamal commitments. This is done by putting the committed message is in the exponent of the generator. ElGamal commitments are twice as large comparing to Pedersen commitments (two group elements instead of one), but have an associated private key that can be used to open the commitment (if the committed values are taken from a (known) polynomial-size set). An ElGamal commitment of a value \(m \in \mathbb{Z}_q\) with public key \(E \in \mathbb{G}\), and randomness \(r \in \mathbb{Z}_q\) is defined by

\[
\text{EgCom}_E(m; r) = (r \cdot G, r \cdot E + m \cdot G),
\]

and \(\text{EgCom}_E(m)\) stands for \(\text{EgCom}_E(m; r)\) with uniform \(r \leftarrow \mathbb{Z}_q\). Recall that we denote elements of \(\mathbb{G} = \mathbb{G} \times \mathbb{G}\), and thus ElGamal commitments, using the \(\sim\) symbol, e.g., \(\tilde{A} = \text{EgCom}_E(m)\).

**Fact 2.8.** \(\text{EgCom}\) is perfectly binding. Assuming DDH is hard over \(\mathbb{G}\), it is computationally hiding.

Also ElGamal commitments are additively homomorphic: for two commitments \(e_{A_1} = \text{EgCom}_E(m_1; r)\) and \(e_{A_2} = \text{EgCom}_E(m_2; r)\), it holds that \(A_1 + A_2\) (where addition is over the group \(\mathbb{G}\), is a commitment to \(m_1 + m_2\). The latter can be combined with rerandomization by adding a fresh encryption of 0. We use the following syntactic sugar operation for \(r \in \mathbb{Z}_q\), \(E \in \mathbb{G}\) and \(e_{A} \in \mathbb{G}\):

- \(\text{EgRerand}_E(\tilde{A}; r) = \tilde{A} + \text{EgCom}_E(0; r)\).

2.6 Zero Knowledge

For a relation \(R\), we use the standard ideal zero-knowledge proof-of-knowledge functionality \(FR_{zk}\).

**Functionality 2.9** \((FR_{zk} - \text{Zero-knowledge for relation } R)\).

Party: \(P_i\).

Operation: Upon receiving \((\text{sid}, x, w, j)\) from \(P_i\), send \((\text{sid}, x, j)\) to \(P_j\) if \(R(x, w) = \text{true}\).

As a syntactic sugar, a call \(FR_{zk}(\text{sid}, x, w)\) stands for the parallel calls \(\{FR_{zk}(\text{sid}, x, w, j)\}_{j \neq i}\). We use zero-knowledge functionalities for the following relations (see Appendix A.1 regarding the UC implementation of these functionalities).

1. Knowledge of discrete log.

\[R_{DL} = \{(A, B, w) : B = w \cdot A\}\]

We write \((B, w) \in R_{DL}\) as a shorthand for \((G, B, w) \in R_{DL}\).

2. DH tuple.

\[R_{DH} = \{((\tilde{A} = (A_1, A_2), \tilde{B} = (B_1, B_2), w) : \tilde{B} = w \cdot \tilde{A})\}\]

We write \((A, \tilde{B}, w) \in R_{DH}\) as a shorthand for \(((G, A), \tilde{B}, w) \in R_{DH}\).

\(^5\)In our notation, see below, the message \(m\) is not actually in the exponent, but this is the name used since when using multiplicative group notation, a commitment to \(m\) will be of the form \(E^r \cdot G^m\).

\[ R_{\text{Ped Klwg}} = \{ ((D, \tilde{A}), (a, r)) : \tilde{A} = \text{PedCom}_D(a; r) \} \]

4. Pedersen commitment of a value.

\[ R_{\text{Ped Eq Val}} = \{ ((D, \tilde{A}, a), r) : \tilde{A} = \text{PedCom}_D(a; r) \} \]

5. Pedersen commitment of a value in the exponent.

\[ R_{\text{Ped Eq Exp}} = \{ ((D, \tilde{A}, A), (a, r)) : \tilde{A} = \text{PedCom}_D(a; r), A = G \cdot a \} \]

6. Knowledge of ElGamal commitment:

\[ R_{\text{Eg Klwg}} = \{ ((E, \tilde{A}), (a, r)) : \tilde{A} = \text{EgCom}_E(a; r) \} \]

7. ElGamal equals Pedersen:

\[ R_{\text{Eg Eq Ped}} = \{ ((E, \tilde{A}, D, \bar{A}), (a, \tilde{r}, \bar{r})) : \tilde{A} = \text{EgCom}_E(a; \tilde{r}) \land \bar{A} = \text{PedCom}_E(a; \bar{r}) \} \]

8. Knowledge of ElGamal scalar product

\[ R_{\text{Eg Prod Scalar}} = \{ ((E, \tilde{A}, \bar{B}), (r, c)) : \bar{B} = c \cdot \tilde{A} + \text{EgCom}_E(0; r) \} \]

That is, prove that you know \( c \) such that \( \bar{B} \) is a commitment to \( c \) times the value committed in \( \tilde{A} \).


\[ R_{\text{Eg Prod Eg}} = \{ ((E, \tilde{A}, \bar{B}, \bar{C}), (r^b, r^0, b)) : \bar{B} = \text{EgCom}_E(b; r^b) \land \bar{C} = b \cdot \tilde{A} + \text{EgCom}_E(0; r^0) \} \]

That is, \( \bar{C} \) is a fresh commitment to the product of the values committed in \( \tilde{A} \) and \( \bar{B} \).

10. ElGamal commitment of a value.

\[ R_{\text{Eg Eq Val}} = \{ ((E, \tilde{A}, a), r) : \tilde{A} = \text{EgCom}_E(a; r) \} \]

11. ElGamal commitment of a value in the exponent:

\[ R_{\text{Eg Eq Exp}} = \{ ((E, \tilde{A}, A), (a, r)) : \tilde{A} = \text{EgCom}_E(a; r), A = G \cdot a \} \]
2.7 String Commitments

We use the standard commitment functionality.

**Functionality 2.10** \((F_{com}—Commitment)\).

**Parties:** \(P_i\).

**Commit.** Upon receiving \((sid, commit, x, j)\) from party \(P_i\):

\[\text{Store}_{sid||i}(x) \text{ and send } (sid, i) \text{ to } P_j.\]

**Open.** Upon receiving \((sid, decommit, j)\) from party \(P_i\):

Set \(x = \text{Retrieve}(sid||i||j)\), and send \((sid, decommit, i, x)\) to \(P_j\).

Again, as a syntactic sugar, a call \(F_{com}(sid, commit, x)\) stand for the parallel calls \(\{F_{com}(sid, commit, x, j)\}_{j \neq i}\), and same for \(\text{Retrieve}\).

2.7.1 Committed Non-Interactive Zero Knowledge

The following functionality allows non-interactively committing to a statement together with a correctness proof.

**Functionality 2.11** \((F_{com-zk}^R—\text{Committed non-interactive zero-knowledge for relation } R)\).

**Parties:** \(P_i\).

**Commit.** Upon receiving \((sid, \text{ComProve}, x, w, j)\) from a party \(P_i\):

\[\text{Store}_{sid||i||j}(x) \text{ and send } (sid, \text{ProofReceipt}, i) \text{ to } P_j.\]

**Open.** Upon receiving \((sid, \text{DecomProof}, j)\) from a party \(P_i\):

Let \(x = \text{Retrieve}(sid||i||j)\). Send \((sid, \text{DecomProof}, i, x)\) to \(P_j\) if \(R(x, w) = \text{true}\).

We apply a similar syntactic sugar to that applied for \(F_{com}\).

3 ECDSA Protocol

In this section, we present a protocol that realizes the ECDSA functionality \(F_{Ecdsa}\), and prove its security (Section 3.1). We start with the the non-threshold (i.e., \(n\)-out-of-\(n\)) case against adversaries controlling all but one honest party. In Section 3.2, we present a variant of the protocol that uses ElGamal commitments instead of Pedersen commitments. In Section 3.3 we extend the protocol for adversaries controlling arbitrary number of parties, and in Section 3.4 we extend it for the threshold case. In Section 3.5 we present a more efficient variant of the signing phase.

We state and prove the security of the protocol in a hybrid-model with access to ideal (leaky) multiplication functionality, whose UC-realization is given in Section 4, and to zero-knowledge and commitment functionalities, whose UC-realizations are given in Appendix A.

Our distributed ECDSA protocol is fairly simple, and its basic form (non-threshold, single honest party) goes as follows: in the key-generation phase, the parties straightforwardly generate additive shares of random signing key \(x \in \mathbb{Z}_q\), each party holding a single share, and publish \(X = x \cdot G\) as the signature verification key. In the signing phase, on message \(m\), the parties
generate shares of two random values \( k, \rho \in \mathbb{Z}_q \), and jointly compute \( \tau = \rho \cdot k \) and \( R = (r, \cdot) = k \cdot G \). They then compute \( \beta = \rho \cdot (H(m) + x \cdot r) \), and output \( (r, s = \tau^{-1} \cdot \beta) \) as the signature. To enforce consistency, when the parties generate additive shares of a value, each party publishes a Pedersen commitment to its share.

The formal definition of the protocol (for the simple setting) is given next, using several sub-protocols that we defined below. Each of the parties maintains its state using a private database, that it naturally access using the \textit{Store} and \textit{Retrieve} commands.

Protocol 3.1 (Distributed ECDSA, non-threshold, single honest party).

**Common input:** \( D \in \mathbb{G} \). // A Pedersen Commitment key.

**Key generation.** On input \( (\text{sid}, \text{Keygen}) \), party \( P_i \) acts as follows:

1. Let \( \text{oid}_x = \text{sid} || \text{“x”} \).
2. The parties engage in \textit{CreateSharedVal} \((\text{sid}, D, \text{oid}_x)\).
3. The parties engage in \textit{OutExpOfSharedVal} \((\text{sid}, D, \text{oid}_x)\).
   
   Let \( X \) be the common output.
4. Output \( X \).

**Signing.** On input \( (\text{sid}, \text{Sign}, \text{oid}_x, m) \), party \( P_i \) acts as follows:

1. Abort if \( \text{oid}_x \) was not stored.
2. Let \( \text{oid}_k = \text{sid} || 1 \), \( \text{oid}_\rho = \text{sid} || 2 \), \( \text{oid}_{\rho \cdot k} = \text{sid} || 3 \), \( \text{oid}_{\rho \cdot x} = \text{sid} || 4 \), \( \text{oid}_{r \cdot \rho \cdot x} = \text{sid} || 5 \), \( \text{oid}_{\rho \cdot m} = \text{sid} || 6 \) and \( \text{oid}_\beta = \text{sid} || 7 \).
3. In parallel, the parties engage in: \textit{CreateSharedVal} \((\text{sid}, D, \text{oid}_\rho)\) and \textit{CreateSharedVal} \((\text{sid}, D, \text{oid}_k)\).
   //Let \( \rho \) and \( k \) denote the generated values.
4. In parallel, the parties engage in:
   
   (a) \textit{MultSharedVals} \((\text{sid}, D, \text{oid}_\rho, \text{oid}_k, \text{oid}_{\rho \cdot k})\).
   
   (b) \textit{MultSharedVals} \((\text{sid}, D, \text{oid}_\rho, \text{oid}_x, \text{oid}_{\rho \cdot x})\).
5. The parties engage in \textit{OutExpOfSharedVal} \((\text{sid}, D, \text{oid}_k)\).
   
   Let \( (r', \cdot) \) be the common output, and let \( r = r' \mod q \).
6. Each party invokes:
   
   (a) \textit{ScalarMultOfSharedVal} \((D, \text{oid}_{r \cdot \rho \cdot x}, \text{oid}_{r \cdot \rho \cdot x}, r)\).
   
   (b) \textit{ScalarMultOfSharedVal} \((D, \text{oid}_\rho, \text{oid}_{\rho \cdot m}, H(m))\).
   
   (c) \textit{AdditionOfSharedVals} \((D, \text{oid}_{r \cdot \rho \cdot x}, \text{oid}_{\rho \cdot m}, \text{oid}_\beta)\). // \textit{oid}_\beta \textit{ refers to } \rho \cdot (H(m) + x \cdot r).
7. In parallel, the parties engage in:
   
   (a) \textit{OutSharedVal} \((D, \text{sid}, \text{oid}_{\rho \cdot k})\); let \( \tau \) be the common output.
   
   (b) \textit{OutSharedVal} \((D, \text{sid}, \text{oid}_\beta)\); let \( \beta \) be the common output.
8. Output \((r, s = r^{-1} \cdot \beta) \mod q\).

The sub-protocols used above are defined below.

**Create shared value.** CreateSharedVal generates a Pedersen commitment of a uniformly chosen value, where each party is holding a (private) share of this value.

**Protocol 3.2 (CreateSharedVal).**

**Oracles:** \(F_{\text{PedKlg}}^{R}\).

**Common input:** \(D \in \mathbb{G}\) and \(\text{sid}, \text{oid} \in \{0, 1\}^*\).

**Operation:** Party \(P_i\) acts as follows:

1. Let \(\widehat{A}_i = \text{PedCom}_D(a_i; r_i)\), for \((a_i, r_i) \leftarrow \mathbb{Z}_q^2\).

2. Send \((\text{sid}, (D, \widehat{A}_i), (a_i, r_i))\) to \(F_{\text{PedKlg}}^{R}\).

3. Upon receiving \((\text{sid}, (D, \widehat{A}_\ell), \ell)\) from \(F_{\text{PedKlg}}^{R}\), for all \(\ell \in [n] \setminus \{i\}\), Store\(_{\text{oid}}\)(\(\widehat{A}_\ell^\text{\{\text{oid}\}}_{\ell=1}, a_i, r_i\)).

**Addition of shared values.** AdditionOfSharedVals adds two previously stored values. Since Pedersen commitments are additively homomorphic, this operation can be carried out locally.

**Algorithm 3.3 (AdditionOfSharedVals).**

**Common input:** \(D \in \mathbb{G}\), \(\text{oid}_a, \text{oid}_b, \text{oid} \in \{0, 1\}^*\).

**Operation:** Party \(P_i\) acts as follows:

1. Let \(\{\tilde{A}_\ell\}_{\ell=1}^n, a_i, r_i\) = Retrieve\(_{\text{oid}}\)(\(\text{oid}_a\)) and \(\{\tilde{B}_\ell\}_{\ell=1}^n, b_i, r_i^B\) = Retrieve\(_{\text{oid}}\)(\(\text{oid}_b\)).

2. Store\(_{\text{oid}}\)(\(\tilde{A}_\ell + \tilde{B}_\ell\)\{\text{oid}\}_{\ell=1}^n, a_i + b_i, r_i^A + r_i^B\).

**Scalar multiplication of shared value.** ScalarMultOfSharedVal multiplies of a previously stored value by a scalar. Again, since Pedersen commitments are additively homomorphic, this operation can be carried out locally.

**Algorithm 3.4 (ScalarMultOfSharedVal).**

**Common input:** \(D \in \mathbb{G}\), \(\text{oid}, \text{oid}' \in \{0, 1\}^*\) and \(\alpha \in \mathbb{Z}_q\).

**Operation:** Party \(P_i\) acts as follows:

1. Let \(\{\tilde{A}_\ell\}_{\ell=1}^n, a_i, r_i\) = Retrieve\(_{\text{oid}}\)(\(\text{oid}\)).

2. Store\(_{\text{oid}}\)(\(\alpha \cdot \tilde{A}_\ell\)\{\text{oid}'\}_{\ell=1}^n, \alpha \cdot a_i, \alpha \cdot r_i\).

---

\(6\)Here and after, each party should also verify that session id and instance it receives from the zero-knowledge functionality are consistent with the common input and the previous steps of the protocol (e.g., in Protocol 3.2, each party should verify that the value of \(\text{sid}\) and \(D\) are according to the common input). To keep the text more readable, however, we omit these simple verification steps.
Output shared value. OutSharedVal outputs a previously stored value.

Protocol 3.5 (OutSharedVal).
Oracles: \( F^{\text{PedEqVal}}_{zk} \).
Common input: \( D \in \mathbb{G} \) and \( \text{sid}, \text{oid} \in \{0,1\}^* \).
Operation: Party \( P_i \) acts as follows:

1. Let \( \{\widehat{A}_\ell\}_{\ell=1}^n, a_i, r_i \) = Retrieve(\( \text{oid} \)).
2. Send \( (\text{sid}, (D, \widehat{A}_i, a_i); r_i) \) to \( F^{\text{PedEqVal}}_{zk} \).
3. Upon receiving \( (\text{sid}, (D, \widehat{A}_\ell, a_\ell), \ell) \) from \( F^{\text{PedEqVal}}_{zk} \) for all \( \ell \in [n]\setminus\{i\} \), output \( \sum_{\ell \in [n]} a_\ell \).

Output exponent of shared value: OutExpOfSharedVal outputs \( A = a \cdot G \) for a previously stored \( a \).

Protocol 3.6 (OutExpOfSharedVal).
Oracles: \( F^{\text{PedEqExp}}_{zk} \).
Common input: \( D \in \mathbb{G} \) and \( \text{sid}, \text{oid} \in \{0,1\}^* \).
Operation: Party \( P_i \) acts as follows:

1. Let \( \{\widehat{A}_\ell\}_{\ell=1}^n, a_i, r_i \) = Retrieve(\( \text{oid} \)).
2. Send \( (\text{sid}, (D, \widehat{A}_i, A_i), (a_i, r_i)) \) to \( F^{\text{PedEqExp}}_{zk} \).
3. Upon receiving \( (\text{sid}, (D, \widehat{A}_\ell, A_\ell), \ell) \) from \( F^{\text{PedEqExp}}_{zk} \) for all \( \ell \in [n]\setminus\{i\} \), output \( \sum_{\ell \in [n]} A_\ell \).

Multiplication of shared values. MultSharedVals multiplies a pair of previously generated values and stores (additive) share of the outcome. The protocol below is merely a wrapper for the following functionality.

Functionality 3.7 (\( F^{\text{LeakyMult}}_{\text{Many-party leaky multiplication}} \)).
Parties: \( P_1, \ldots, P_n \).
Common input: \( \text{sid} \in \{0,1\}^* \), \( D \in \mathbb{G}, \{\widehat{A}_\ell, \widehat{B}_\ell\}_{\ell \in [n]} \).
\( P_i \)'s input: \( (a_i, r_i^A), (b_i, r_i^B) \in \mathbb{Z}_q^2 \).
Operation:

1. Abort if for some \( \ell \) it holds that \( \widehat{A}_\ell \neq \text{PedCom}_D(a_i, r_i^A) \) or \( \widehat{B}_\ell \neq \text{PedCom}_D(b_i, r_i^B) \).
2. Sample \( r_1, \ldots, r_n \) uniformly at random in \( \mathbb{Z}_q \).
   Sample \( c_1, \ldots, c_n \) uniformly at random in \( \mathbb{Z}_q \) conditioned on \( \sum_{\ell \in [n]} c_\ell = (\sum_{\ell \in [n]} a_\ell) \cdot (\sum_{\ell \in [n]} b_\ell) \).
3. Output \( \{ \hat{C}_\ell = \text{PedCom}_D(c_\ell; r_\ell) \}_{\ell \in [n]} \) as the common output.

4. Send \((r_\ell, c_\ell)\) to \(P_\ell\) for each \(\ell \in [n]\).

After getting their outputs, the corrupted parties \(C \subset [n]\) can make the following calls (at most one of each type):

**Linear test**: Send \(\delta \in \mathbb{Z}_q\) and \(\{(a^A_\ell, o_\ell^A) \in \mathbb{Z}_q^2 \}_{\ell \in [n] \setminus C}\). The functionality aborts if \(\sum_{\ell \in [n] \setminus C} (a^A_\ell \cdot a_\ell + a^B_\ell \cdot b_\ell) \neq \delta\).

**Commitment replacement**: Send \(\{(r'_\ell, c'_\ell)\}_{\ell \in C}\). The functionality aborts if \(\sum_{\ell \in C} c_\ell \neq \sum_{\ell \in C} c'_\ell\). Otherwise, the value of \(b^C_\ell\), for each \(\ell \in C\), in the common output is updated to \(\text{PedCom}_D(c'_\ell; r'_\ell)\).

That is, \(F_{\text{LeakyMult}}\) ideally multiplies the shared inputs, and shares the result. The linear-test call might leak to the corrupted parties whether a linear function over the honest parties’ inputs equals some value. The commitment-replacement call allows the corrupted parties to modify their commitment (though not their overall sum) based on the honest parties’ commitments. These backdoors enable us to implement the functionality efficiently. A UC-realization of \(F_{\text{LeakyMult}}\), assuming DDH is hard over \(\mathbb{G}\), is given in Section 4. Equipped with \(F_{\text{LeakyMult}}\), the multiplication protocol is defined as follows.

**Protocol 3.8** (MultSharedVals).

**Oracles**: \(F_{\text{LeakyMult}}\).

**Common input**: \(D \in \mathbb{G}\) and \(\text{oid}_a, \text{oid}_b, \text{oid} \in \{0, 1\}^*\).

**Operation**: Party \(P_\ell\) acts as follows:

1. Let \(\{(A_\ell)^n\}_{\ell=1}^n, a_\ell\) = \(\text{Retrieve}_(\text{oid}_a)\) and \(\{(B_\ell)^n\}_{\ell=1}^n, b_\ell, r^B_\ell\) = \(\text{Retrieve}_a(\text{oid}_b)\).

2. Send \((\text{sid}, D, \{(A_\ell), (B_\ell), (a_\ell, r^A_\ell, b_\ell, r^C_\ell)\})\) to \(F_{\text{LeakyMult}}\).

Let \(\{\hat{C}_\ell\}_{\ell=1}^n\) be the common output and \((c_\ell, r^C_\ell)\) be the private output.

3. Store\(\text{oid}_a(\{(C_\ell)^n\}_{\ell=1}^n, c_\ell, r^C_\ell)\).

**Pedersen commitments of the secret key shares.** Reading the security proof given below, it is easy to verify that we are only using the hiding of the Pedersen commitment of the secret key shares generated in the key-generation phase, to prevent malicious parties from controlling the secret key value. For the rest of the protocol, we could replace these commitments (but not the other commitments!) with the exponents of the shares (equivalently, with Pedersen commitments of randomness 1). Such a change becomes handy when refreshing the distribution of the secret key shares (while keeping the same secret key); it will not require the parties to generate, and prove knowledge of, Pedersen commitment of their shares, but only to generate the exponentiations of their shares.
Non-interactive signing. The above protocol naturally gives rise to the non-interactive signing phase in which all but the execution of protocols ScalarMultOfSharedVal, AdditionOfSharedVals and OutSharedVal is done in the preprocessing. The security of the resulting protocol, however, requires the following (stronger) security assumption on the standalone ECDSA: the security holds if many (“nonce”) r’s are published in advance, and the messages to be signed can be chosen (adversarially) as a function of these r’s. See [GS22] for more details (they refer to the publication of the nonce before the message to be signed was set as presignatures).

3.1 Security

In this section, we prove that Protocol Ecdsa UC-realizes the non-threshold $F_{\text{Ecdsa}}$ (Functionality 2.6) against adversaries corrupting at least $n-1$ parties. The following theorem assumes that the common input $D$ is sampled by $F_{\text{PedKeyGen}}$, a functionality that returns a uniform $D \leftarrow G$. The functionality $F_{\text{PedKeyGen}}$ can be implemented using a simple coin-flipping protocol followed by applying a hash-to-curve mapping on the resulting string.

**Theorem 3.9** (Security of Protocol 3.1). Assume the standalone ECDSA signature scheme is existentially unforgeable over $G$,\footnote{Actually, it suffices to assume that an adversary seeing polynomial many signatures cannot extract the signing key.} then on common input $D \leftarrow F_{\text{PedKeyGen}}()$, Protocol 3.1 UC-realizes the non-threshold $F_{\text{Ecdsa}}$ against adversaries corrupting at least $n-1$ parties, in the $(F_{\text{RPedKeyGen}}, F_{\text{RPedEqVal}}, F_{\text{RPedEqExp}}, F_{\text{LeakyMult}})$-hybrid model.

**Remark 3.10** (Security assumption). One might question the necessity of assuming the security of the standalone ECDSA signature for proving Theorem 3.9 (in the hybrid model). This assumption is not needed if we assume access to an ideal two-party multiplication oracle instead of $F_{\text{LeakyMult}}$ (a functionality for which we do not have an efficient implantation), but seems to be necessary when using $F_{\text{LeakyMult}}$. Clearly, for the security of final (non-hybrid) protocol this distinction makes no difference.

**Proof.** Fix an environment $E$ and an adversary $A$ corrupting the parties indexed by $C \subseteq [n]$ with $|C| = n-1$ (recall that the all-corruption case is trivial), and for ease of notation assume that $C = [n-1]$ . The ideal model simulator $S$ for $A$ is rather straightforward: it emulates a random execution of the protocol between the parties controlled by $A$ and the honest $P_n$, while the messages of $P_n$ are emulated as follows: the Pedersen commitments are generated arbitrarily (i.e., random commitments to zero), and in the parts where some values are unmasked, either in the clear via protocol OutExpOfSharedVal, or in the exponent via protocol OutExpOfSharedVal, the simulator manipulates the common output to match the values it gets from the ideal functionality.

In the following assume that in addition to $(r, s)$, the functionality $F_{\text{Ecdsa}}$ outputs the value of $R = (r', \cdot) = G \cdot k$ used to generate the signature (see Algorithm 2.4). This is without loss of generality since the value of $R$ can be efficiently computed from the $(r, s)$, the verification key and the message to sign.\footnote{Let $R^1, R^2 \in G$ be the two group elements of the form $(r', \cdot)$ with $r' \equiv r \mod q$ (these elements can be found by plugging $r$ into the elliptic curve equation.) Return the group element $R^j$ that satisfies $s \cdot R^j = m^j \cdot G + r' \cdot X$.} The following simulator is using several subroutines that we define below. For its own book-keeping, it maintains a data-structure that it naturally access using the Store and Retrieve commands.
Algorithm 3.11 (S—Simulator for A).

**Oracles:** $\mathcal{F}_{\text{Ecdsa}}$.

**Input:** $D \in \mathbb{G}$.

**Operation:** Start emulating a random execution of Ecdsa on common input $D$ with A controlling the parties in $C$. For each input inp sent by the environment $E$, act as follows:

$$inp = (\text{sid}, \text{Keygen})$$:

- Send (sid, Keygen) to $\mathcal{F}_{\text{Ecdsa}}$. Let $X$ be the returned output.
  1. Use sid to generate object id oid$_x$ according to the naming conventions given in Protocol 3.1.
  2. Call S.CreateSharedVal(sid, D, oid$_x$).

$$inp = (\text{sid}, \text{Sign}, \text{oid}_x, m)$$:

  1. Abort the emulation if oid$_x$ was not stored.
  2. Use sid to generate object ids oid$_k$, oid$_p$, oid$_{p-k}$, oid$_{p-x}$, oid$_{r-p-x}$, oid$_{p-m}$, oid$_{\beta}$, according to the naming conventions given in Protocol 3.1.
  3. Send (sid, Sign, oid$_x$, m) to $\mathcal{F}_{\text{Ecdsa}}$. Let $(R, \cdot, s)$ be the returned output.
  4. In parallel, call S.CreateSharedVal(sid, D, oid$_p$) and S.CreateSharedVal(sid, D, oid$_k$).
  5. In parallel, call S.MultSharedVals(sid, D, oid$_p$, oid$_k$) and S.MultSharedVals(sid, D, oid$_p$, oid$_x$, oid$_{p-x}$).
  6. Call S.ScalarMultOfSharedVal(D, oid$_{p-x}$, oid$_{r-p-x}$, r), S.ScalarMultOfSharedVal(D, oid$_p$, oid$_{p-m}$, H(m)), and S.AdditionOfSharedVals(D, oid$_{r-p-x}$, oid$_{p-m}$, oid$_{\beta}$).
  7. Sample $\tau \leftarrow \mathbb{Z}_q$.
  8. In parallel, call OutSharedVal(D, sid, oid$_{p-k}$, $\tau$) and S.OutSharedVal(D, sid, oid$_{\beta}$, $\tau \cdot s$).

The simulator sends abort to $\mathcal{F}_{\text{Ecdsa}}$ in case A send abort to $P_n$ in the emulation.\textsuperscript{10}

The subroutines used by S are defined below.

**Algorithm 3.12 (S.CreateSharedVal(D, sid, oid)).**

1. Emulate the execution of CreateSharedVal(sid, D, oid) with $P_n$ using uniform randomness.
   - Let $\{a_t, \hat{A}_t\} \in C$ and $\hat{A}_n$, be the values of these variables sent by the corrupted parties and $P_n$ to $\mathcal{F}_{\text{PedKlg}}$, respectively.
2. Store\textsuperscript{oid}($\{a_t\} \in C$, $\{\hat{A}_t\} \in \mathbb{N}$).

\textsuperscript{9}That is, the messages the simulators send to the corrupted parties are synchronized. If the honest party aborts in one of the executions it does so also in all executions.

\textsuperscript{10}Recall that we assume without loss of generality that a corrupted party never sends a message/make a call to an ideal functionality that would make an honest party to abort, with probability one, rather it sends the party an aborting message.
Algorithm 3.13 (S.AdditionOfSharedVals(D, sid, oid_a, oid_b, oid)).

1. Let \(\{a_t\}_{t \in C}, \{\hat{A}_t\}_{t \in [n]} = \text{Retrieve}(oid_a)\) and \(\{b_t\}_{t \in C}, \{\hat{B}_t\}_{t \in [n]} = \text{Retrieve}(oid)\).

2. \(\text{Store}_{oid}(\{(a_t + b_t)\}_{t \in C}, \{\hat{A}_t + \hat{B}_t\}_{t \in [n]}\).

Algorithm 3.14 (S.ScalarMultOfSharedVal(D, sid, oid, oid', \(\alpha\))).

1. Let \(\{a_t\}_{t \in C}, \{\hat{A}_t\}_{t \in [n]} = \text{Retrieve}(oid)\).

2. \(\text{Store}_{oid'}(\{\alpha \cdot a_t\}_{t \in C}, \{\alpha \cdot \hat{A}_t\}_{t \in [n]}\).

Algorithm 3.15 (S.OutSharedVal(sid, oid, a)).

1. Let \(\{a_t\}_{t \in C}, \{\hat{A}_t\}_{t \in [n]} = \text{Retrieve}(oid)\).

2. Emulate the execution of OutSharedVal(sid, oid) while setting the value of \(a_n\) in the message sent by \(\mathcal{F}_{2k}^{R_{\text{PedEqVal}}}\) in response to \(P_n\)'s call to \(a_n' = a - \sum_{t \in C} a_t\).

(I.e., the message sent by \(\mathcal{F}_{2k}^{R_{\text{PedEqVal}}}\) is set to \((\text{sid}, n, (D, \hat{A}_n, a_n'))\)).

Algorithm 3.16 (S.OutExpOfSharedVal(sid, D, oid, A)).

1. Let \(\{a_t\}_{t \in C}, \{\hat{A}_t\}_{t \in [n]} = \text{Retrieve}(oid)\).

2. Emulate the execution of OutExpOfSharedVal(sid, D, oid), while setting the value value of \(A_n\) sent by \(P_n\) and sent by \(\mathcal{F}_{2k}^{R_{\text{PedEqExp}}}\) in response to its the call, to \(A_n' = A - \sum_{t \in C} a_t \cdot G\).

(I.e., the message sent by \(\mathcal{F}_{2k}^{R_{\text{PedEqExp}}}\) is set to \((\text{sid}, n, (D, \hat{A}_n, A_n'))\)).

Algorithm 3.17 (S.MultSharedVals(sid, oid_a, oid_b, oid)).

1. Let \((., \{\hat{A}_t\}_{t \in [n]} = \text{Retrieve}(oid_a)\) and \((., \{\hat{B}_t\}_{t \in [n]} = \text{Retrieve}(oid_b)\).

2. Emulate the call to \(\mathcal{F}_{\text{LeakyMult}}\) with \(P_n\) using public input \((\text{sid}, D, \{\hat{A}_t, \hat{B}_t\}_{t \in [n]}\), and uses arbitrary private inputs, e.g., \((0, 0)\) and \((0, 0)\), and \(\mathcal{F}_{\text{LeakyMult}}\) does not check that \(P_n\)'s private inputs are consistent with the common input.

Send abort to all parties (as the answer of \(\mathcal{F}_{\text{LeakyMult}}\) if \(A\) makes a non-zero linear-test call, i.e., \((o^A_n, o^B_n) \neq (0, 0)\).
Since 5 is non-rewinding, i.e., does not rewind A, we only need to prove that, with respect to the same environment and uniformly chosen parameter D, the joint distribution of A’s view and the honest party outputs in the real execution, is computationally indistinguishable from the distribution of these values in the emulated execution: the view of the emulated A and the output of the honest party in the ideal execution induced by the simulator (see Section 2.2 for justification). In the following, we assume the corrupted parties never attempt to break the binding of a Pedersen commitment: in all accepting calls to the different functionalities with respect to a committed value \( \hat{A} \), e.g., \( F_{2k}^{R_{PedKlwg}} \), the party uses the same witness \((a, r)\) with \( \hat{A} = \text{PedCom}_D(a; r) \). Since the unforgeability of the of the standalone ECDSA, which implies that discrete-log is hard over \( \mathbb{G} \), implies that Pedersen commitments are computationally binding for uniform public key D, the above is without loss of generality: changing a party to sends aborts message to \( P_n \) before sending a different opening to a commitment, has only negligible effect on the execution. We also assume that A does not make linear test-call to \( F_{\text{LeakyMult}} \) with all linear coefficient \( \{(o^A_\ell, o^B_\ell)\} \) set to zero. (Indeed, by definition, such a call \( F_{\text{LeakyMult}} \) aborts iff \( \delta \neq 0 \), and thus can be easily emulated.)

We first prove that if A never makes a linear-test call to \( F_{\text{LeakyMult}} \), the distributions are identical. Since Pedersen commitments are perfectly hiding, and since A does not make a linear-test call, the only parts in A’s view that might be different are those related to the executions of OutSharedVal and OutExpOfSharedVal sub-protocols. Specifically, the value of \( a_n \) and \( A_n \) in these executions. By construction, these values in the real and emulated execution are identically distributed, making the two views identically distributed. For \( P_n \)’s outputs, since, by assumption, the corrupted parties never attempt to break the binding of the Pedersen commitments, the public key and the signatures generated in the emulated execution are according to the values returned by ideal key generation and signature functionality, making the two executions identical.

We complete the proof by showing that in the the real execution, a linear-test call aborts with all but negligible probability (making the real execution statistically close to the emulated one). Assume A makes a linear-test call in MultSharedVals done in Round 4a of the signing protocol. Note that in this execution of MultSharedVals, the key \( \text{sid}_a \) refers to the value of \( \rho \) and \( \text{sid}_b \) refers to the value of \( k \). Since until this step, the execution has leaked no information about the value of \( \rho_n \) and \( k_n \), values that were uniformly sampled in \( \mathbb{Z}_q \), the test-call aborts with all but negligible probability. Assume that A makes test-call in an execution of MultSharedVals done in Round 4b of the signing protocol. In this execution of MultSharedVals, the key \( \text{sid}_a \) refers to the value of \( \rho \) and \( \text{sid}_b \) refers to the value of \( x \). The same argument as in the \( \rho \cdot k \) case yields that unless \( o^A_n = 0 \), the functionality aborts with all but non-negligible probability (this holds even if the value of \( x \) is known to A). Assume that with non-negligible probability A makes a non-aborting test-call in Round 4b of the signing protocol with \( o^A_n = 0 \) and \( o^B_n \neq 0 \). We show that such A can be used to for violating the security of the standalone ECDSA. For ease of notation, assume that E only invokes the key-generation phase once (in the general case, we use one of these calls for the attack). We start by showing how to use such E and A for extracting the secret signing key (in the multi-party case) with non-negligible probability.

**Algorithm 3.18 (ExtractorDist: signing key extractor for the distributed ECDSA).**

**Oracles:** E, A.

**Operation:** Start a random execution of Ecdsa with A controlling the parties in C and the environment E. In the first execution of MultSharedVals done in Step 4b of the signing phase in which \( o^B_n \neq 0 \):
1. Let \( x_n = \delta / o_n^B \) (were \( \delta \) is the value of this parameter in the linear-test call).

2. Let \( \{x_\ell\}_{\ell \in C} \) be the the additive shares of the corrupted parties used in the (single) key-generation phase (can be extracted as done by \( S \)).

3. Output \( \sum_{\ell \in [n]} x_\ell \).

It is easy to verify that ExtractorDist indeed finds the signing key with non-negligible probability. Since, by the previous argument, \( S \) emulates \( A \) perfectly until the first testing call, applying the simulator \( S \) on ExtractorDist yields an algorithm that extracts the signing key in the \( F_{Ecdsa} \)-hybrid model. The latter algorithm can be straightforwardly transformed into an algorithm that wins the standalone ECDSA game, i.e., breaks the security of the standalone ECDSA scheme.\(^{11}\)

3.2 Using ElGamal Commitments

In some cases, see for instance Section 3.5, one might like to replace the perfectly hiding and computationally binding Pedersen commitments used in Protocol 3.1 with the computationally perfectly and computationally ElGamal commitments (see Section 2.5). That is, the ElGamal-based protocol follows that lines of Protocol 3.1, were in addition to replacing the Pedersen commitments with ElGamal commitments, the zero-knowledge proofs used for these commitments are replaced with their ElGamal analogs (all proofs appear in Appendix A.1). Since the the randomness used for generating the commitments is only used in the various zero-knowledge proofs, the security proof of the ElGamal-based protocol follows the same lines to the one given above for Theorem 3.9. Specifically, we replace the honest party’s commitments with commitments to zero. The computational hiding of the ElGamal commitments yields that adversary cannot tell this second variant from the first one, and thus from the real execution. The same line of the proof of Theorem 3.9 yields that the variant above can be simulated given access to \( F_{Ecdsa} \). Thus, the real execution can be simulated.

3.3 Many Honest Parties

In this section, we slightly adjust Protocol 3.1 so that its security holds against adversaries corrupting less than \( n - 1 \) parties. Let the public view of a party denotes the public information it sends and receives: inputs it received from the environment, messages it sends and receives, and outputs of the common output ideal functionalities (all functionalities but \( F_{LeakyMult} \)). More strictly, in a key-generation phase, the public view refers to the current key-generation execution, and in a signing phase, it relates to the key-generation execution in which the public key was created, and the current signing execution. Note that since we assume no broadcast channel, the public view of two honest parties might be different. An outputting round is a round in which the honest parties are suppose to output a value. The slight adjustment of Protocol 3.1 is given below.

Protocol 3.19 (Distributed ECDSA, non-threshold).

1. In each round that proceeds an outputting round, each honest party sends a hash of its public view to all parties. That is, \( H'(v) \) where \( H' \) is an arbitrary collision-resistant hash function, and \( v \) is its current public view.

\(^{11}\)We remark that ExtractorDist can only be used for extracting the signing key when having access to an ECDSA signer (for this key). Thus, it is not useful for finding discrete-log in the standard discrete-log game.

22
2. In each outputting round, an honest party first verifies that the hash of all other parties’ public view is consistent with its own.

3.3.1 Security

We prove that the above variant is secure against any number of corruptions.

**Theorem 3.20** (Security of Protocol 3.19). Assume standalone ECDSA signature scheme is existentially unforgeable against adaptive chosen-message attacks over \(G\), then on common input \(D \leftarrow \mathcal{F}_{\text{PedKeyGen}}()\), Protocol 3.19 UC-realizes the non-threshold \(\mathcal{F}_{\text{Ecdsa}}\), in the \((\mathcal{F}_{\text{LeakyMult}}, \mathcal{F}_{\text{R PedKlwg}}, \mathcal{F}_{\text{R PedEqVal}}, \mathcal{F}_{\text{R PedEqExp}})\)-hybrid model.

**Proof.** Let \(A\) be an adversary for \(\Pi = (P_1, \ldots, P_n)\), corrupting parties indexed by \(\mathcal{C}\). The proof for \(|\mathcal{C}| \geq n - 1\) is essentially like the one in Theorem 3.9, so we assume \(|\mathcal{C}| < n - 1\), let \(\mathcal{H} = [n] \setminus \mathcal{C}\), and assume for ease of notation that \(n \in \mathcal{H}\). The simulator \(S\) for \(A\) acts like the simulator for single-honest-party case, Algorithm 3.11, with the following adjustments:

- **S.CreateSharedVal:** Sample all honest parties’ input uniformly at random (and not just \(P_n\)’s).
- **S.AdditionOfSharedVals, S.ScalarMultOfSharedVal:** Same like in the single honest party case, i.e., keep track off corrupted parties shares, and all parties commitments.
- **S.OutSharedVal:** Choose the value of \(a_\ell\), for \(\ell \in \mathcal{H} \setminus \{n\}\), uniformly at random (and set \(a_n\) based on these choices and the corrupted parties shares).
- **S.OutExpOfSharedVal:** Choose the value of \(A_\ell\), for \(\ell \in \mathcal{H} \setminus \{n\}\), uniformly at random (and set \(A_n\) based on these choices and the corrupted parties shares).
- **S.MultSharedVals:** Abort the emulation, send abort to all parties, if \(A\) makes (an extended) non-zero linear-test call to \(\mathcal{F}_{\text{LeakyMult}}\): i.e., \((o^A_\ell, o^B_\ell) \neq (0, 0)\) for some \(\ell \in \mathcal{H}\).

**Aborts:** Instruct \(\mathcal{F}_{\text{Ecdsa}}\) to abort a party \(P_\ell\) if it receives an abort message.

A similar argument to that used in the single-honest-party case yields that in the real execution, a non-zero linear-test call to \(\mathcal{F}_{\text{LeakyMult}}\) aborts with all but negligible probability. Hence, in the following we assume without loss of generality that \(A\) never attempts to make such a call. We also assume that it never happens that two honest parties have different public history, and yet it is not reflected in the hash values exchanged. The very same argument as single-honest-party case, also yields that \(A\)’s view including \(P_n\)’s outputs in the real and emulated execution are identically distributed, so it only left to prove that the same holds when adding the other honest parties outputs. Since an honest party only outputs a (non-aborting) value if it its public history until (not including) this round is consistent with that of \(P_n\), it follows that a non-aborting output of this party in the real execution is consistent with its output of the ideal functionality, and this with its output in the emulated execution.
3.4 Threshold Signatures

In this section we extend our scheme to the threshold case, i.e., $t < n$. The extension follows the standard approach for achieving that, c.f. [Doe+19]: at the end of the key-generation phase, the parties use standard means to transform their additive shares $\{x_\ell\}_{\ell \in [n]}$ of the signing key $x$, into $t$-out-of-$n$ shares of $x$. In the signing phase, the active parties (locally) transform their $t$-out-of-$n$ shares into additive $t$-out-of-$t$ shares of $x$, and continue as in the non-threshold case we described above. The protocol is defined below.

**Protocol 3.21** (Distributed ECDSA, Threshold case).

**Paramter:** $t \in [n - 1]$.

**Common input:** $D \in \mathbb{G}$.

**Key generation.** On input $(\text{sid}, \text{Keygen})$, party $P_i$ acts as follows:

1. Interact in key-generation phase of Protocol 3.19 (with the same common input).
   
   Let $\{\hat{X}_\ell\}_{\ell \in [n]}$, $x_i$, $r_i^X$ be the value it stored in this call.

2. // Send commitments to the evaluations of a random $(t - 1)$-degree polynomial $p_\ell$ with $p_\ell(0) = x_i$, over the points in $[n]$, and reveal $p_i(\ell)$ to $P_\ell$.

   (a) Sample uniformly a degree $t$ polynomial $p_\ell$ with $p_i(0) = x_i$. Let $\{c_{i,j}\}_{j \in (t-1)}$ be the coefficients of $p_i$. // Note that $c_{i,0} = x_i$.

   (b) For each $j \in [t - 1]$, sample $r_{i,j} \leftarrow \mathbb{Z}_q$.

   (c) Send $\{\hat{C}_{i,j} = \text{PedCom}_D(c_{i,j}; r_{i,j}^C)\}_{j \in [t-1]}$ to all parties.

   (d) // For each $\ell$, compute a commitment to $\{p_\ell(\alpha)\}_{\alpha \in [n]}$ using $p_\ell$’s coefficients.

      Upon receiving $\{\hat{C}_{i,j}\}_{j \in [t-1]}$ from all $\ell \in [n] \setminus \{i\}$:

      - for each $\ell, \alpha \in [n]$, let $\hat{B}_{\ell,\alpha} = \sum_{j \in (t)} \hat{C}_{i,j} \cdot \alpha^j$, letting $\hat{C}_{\ell,0} = \hat{X}_\ell$.

   (e) // Send fresh commitments to $\{p_\ell(\alpha)\}_{\alpha}$ and prove their correctness.

      For each $\alpha \in [n]$:

      i. Send $\hat{Y}_{i,\alpha} = \text{PedCom}_D(p_i(\alpha); r_{i,\alpha}^Y)$ to all parties, for $r_{i,\alpha}^Y \leftarrow \mathbb{Z}_q$.

      ii. Send $(\text{sid}, \hat{Y}_{i,\alpha} - \hat{B}_{i,\alpha} - \sum_{j \in (t)} r_{i,j}^C \cdot x^j)$ to $\mathcal{F}_{\hat{X}_\ell}^\alpha$, letting $r_{i,0}^C = r_{i}^X$.

   (f) Upon receiving $\{\hat{Y}_{\ell,\alpha}\}_{\alpha \in [n]}$ from $P_\ell$ and $\{(\text{sid}, \hat{Y}_{\ell,\alpha} - \hat{B}_{\ell,\alpha})\}$ from $\mathcal{F}_{\hat{X}_\ell}^{\alpha}$, for all $\ell \in [n] \setminus \{i\}$:

      send $(p_i(\ell), r_{\ell,\ell}^Y)$ to $P_\ell$, for all $\ell \in [n] \setminus \{i\}$. // Send the opening of $\hat{Y}_{i,\ell}$ to $P_\ell$.

   (g) Upon receiving $(p_{\ell,\ell}, r_{\ell,\ell}^Y)$ for all $\ell \in [n] \setminus \{i\}$: abort if $\hat{Y}_{i,\ell} \neq \text{PedCom}(p_{\ell,\ell}; r_{\ell,\ell}^Y)$ for some $\ell$.

   (h) For each $\ell \in [n]$, let $\hat{Y}_{\ell} = \sum_{\ell \in [n]} \hat{Y}_{j,\ell}$.

      // $\hat{Y}_{\ell}$ is a commitment to $\sum_{\ell \in [n]} p_\ell(\ell)$.

   (i) Let $y_i = \sum_{\ell \in [n]} p_{\ell,\ell}$ and $r_i = \sum_{\ell \in [n]} r_{\ell,\ell}^Y$.

      // $(y_i, r_i)$ is an opening of $\hat{Y}_{i}$.

3. Store $\text{oid}_{\text{xtsh}}(\{\hat{Y}_{\ell}\}_{\ell \in [n]}, y_i, r_i)$, for $\text{oid}_{\text{xtsh}} = \text{sid} || \text{"xtsh"}$. 

24
Signing. On input \((\text{sid}, \text{Sign}, \text{oid}_x, m)\), party \(P_i\) acts as follows:

1. Let \(\mathcal{S} \subseteq [n]\) be the set of active parties. Abort if \(|\mathcal{S}| \neq t\).

2. // (Locally) Generate additive shares of \(x\).
   
   (a) For \(\ell \in \mathcal{S}\), let \(o_\ell = \prod_{j \in \mathcal{S} \setminus \{\ell\}} a_j - \ell j\).
   
   (b) Let \((\{\hat{A}_\ell\}_{\ell=1}^{n}, a_i, r_i) = \text{Retrieve}(\text{oid}_x)\).
   
   (c) Let \((a'_i, r'_i) = o_i \cdot (a_i, r_i)\), and for each \(\ell \in \mathcal{S}\) let \(b_{A'}^\ell = o_\ell \cdot b_{A}^\ell\).

3. // Sign using the additive shares.
   
   (a) Store \(\text{oid}' = \text{sid} || \gamma^\ell(\{\hat{A}_\ell\}_{\ell \in \mathcal{S} \setminus \{h\}}, a'_i, r'_i)\).
   
   (b) Interact in the signing phase of Protocol 3.19, while in the call to \(\text{MultSharedVals done in Round 4b}\), use the object-id \(\text{oid}' = \text{sid} || \text{"x"}\).

Namely, at the end of the key-generation phase all parties hold (fresh) commitments to the evaluation of a \((t-1)\)-degree polynomial \(p\) with \(p(0) = x\). Party \(P_i\) holds the opening of \(p(i)\). At the signing phase, the active parties locally transfer the above shares into additive shares, and continue as in the non-threshold case.

3.4.1 Security

We argue that the above protocol is secure.

**Theorem 3.22 (Security of Protocol 3.21).** Assume standalone ECDSA signature scheme is existentially unforgeable over \(\mathcal{G}\), then on common input \(D \leftarrow \mathcal{F}_{\text{PedKeyGen}}()\), Protocol 3.21 UC-realizes \(\mathcal{F}_{\text{Ecdsa}}\) in the \((\mathcal{F}_{\text{LeakyMult}}, \mathcal{F}_{\text{EquaExp}}, \mathcal{F}_{\text{EquaEqExp}})\)-hybrid model.

**Proof.** The proof is almost identical to the proof of Theorem 3.20. The only interesting difference is that in each emulation of the signing protocol, the simulator arbitrarily picks one of the active honest parties \(P_h\), and acts like Algorithm 3.11 (the non-threshold simulator) with party \(P_h\) taking the role of party \(P_n\), the commitments \(\{\hat{A}_\ell\}_{\ell \in \mathcal{S}}\) taking the role of \(\{\hat{A}_\ell\}_{\ell \in [n]}\) and \(\{(a'_\ell, r'_\ell)\}_{\ell \in \mathcal{S} \setminus \{h\}}\) taking the role of \(\{(a_\ell, r_\ell)\}_{\ell \in [n-1]}\), for \((a'_\ell, r'_\ell) = o_\ell \cdot (a_\ell, r_\ell)\), as the dishonest parties’ opening of the commitments. The logic of the proof now follows the same lines as that of Theorem 3.20.

3.5 A More Efficient Protocol

In its current form, the signing phase takes six rounds, which consists of two parallel executions of protocol \(\text{CreateSharedVal}\) (one round), two parallel executions of protocol \(\text{MultSharedVals}\) (one round), execution of protocol \(\text{OutExpOfSharedVal}\) (one round) and two parallel executions of protocol \(\text{OutSharedVal}\) (one round). When taking into account the five-round protocol \(\text{ManyPartyLeakyMult}\), see Section 4, that implements the \(\mathcal{F}_{\text{LeakyMult}}\) functionality called by protocol \(\text{MultSharedVals}\), we end up in a 8-round protocol. However, in the price of modularity, the round efficiency of the signing phase can be reduced to six rounds. The modification is simple, we avoid the additional two
rounds paid for the execution of protocols CreateSharedVal and OutExpOfSharedVal, by executing CreateSharedVal in parallel to the execution of the first round of protocol ManyPartyLeakyMult, and execute OutExpOfSharedVal in parallel to the execution of the last round of protocol ManyPartyLeakyMult. The security proof of this variant follows the very same line of that of the original variant. For improving the computation cost, one might move to using ElGamal commitments with shared secret key instead of Pedersen commitments, throughout the protocol. This change will save the costly transformation between Pedersen to ElGamal commitments done in protocol ManyPartyLeakyMult, but will slightly increase the communication cost of the other parts of the protocol. Finally, when instantiating the weak two-party multiplication functionality using a two-round protocol, we get a seven-round protocol (in the OT-hybrid model).

It is easy to see that in this number of rounds one can also squeeze the round required for verifying on the common inputs in the embedded call to Protocol 4.6 (such a call is not needed at the beginning of Protocol 4.3, which merely passes of its common input to Protocol 4.6), and the single-round protocol for generating high-entropy session ids.

4 Many-Party Leaky Multiplication

In this section, we present protocol ManyPartyLeakyMult, a UC-realization of the $F_{\text{LeakyMult}}$ functionality defined in Functionality 3.7. The implementation is using a two-party multiplication functionality $F_{\text{2PC-Mult}}$ and multiplication equality test $F_{\text{PedMultEqTest}}$, both defined below. The security of protocol ManyPartyLeakyMult is analyzed in Section 4.1. The UC-realization of $F_{\text{PedMultEqTest}}$ is given in Section 4.2. In Section 4.3, we present a weaker variant of $F_{\text{2PC-Mult}}$, which has a known efficient implementation, and prove that the security of protocol ManyPartyLeakyMult still holds when using this variant. Finally, in Section 4.5 we present a more efficient variant of protocol ManyPartyLeakyMult.

Starting with the helper functionalities, the two-party multiplication is defined as follows:

**Functionality 4.1 ($F_{\text{2PC-Mult}}$—Two-party multiplication).**

Parties: $P_1, P_2$.

Common input: $\text{sid} \in \{0, 1\}^*$.

$P_i$'s input: $a_i \in \mathbb{Z}_q$, and optional $s_i \in \mathbb{Z}_q$.

Operation:

1. Abort if both $s_1$ and $s_2$ are set.
2. If neither $s_1$ and $s_2$ is set, let $i = 1$ and sample $s_1 \overset{\$}{\leftarrow} \mathbb{Z}_q$.
   Else, let $i$ be the index for which $s_i$ is set.
3. Let $i' = \{1, 2\} \setminus \{i\}$ and let $s_{i'} = a_1 \cdot a_2 - s_i$.
4. For both $\ell \in \{2\}$: send $(\text{sid}, \ell, s_{i'})$ to $P_\ell$.

(Recall that, by convention, all operations are carried in the relevant field/group. In particular, $s_{i'} \in \mathbb{Z}_q$.) That is, $F_{\text{2PC-Mult}}$ on input $a_1, a_2 \in \mathbb{Z}_q$ returns a random additive share of $a_1 \cdot a_2$. The multiplication equality test is defined as follows:
**Functionality 4.2** \(F_{PedMultEqTest}\) — Pedersen commitment multiplication equality test.

**Parties:** \(P_1, \ldots, P_n\).

**Common input:** \(\text{sid} \in \{0, 1\}^*\) and \(D, \{\widehat{A}_\ell, \widehat{B}_\ell, \widehat{C}_\ell\}_{\ell=1}^n\).

**\(P_i\)'s input:** \(a_i, r_i^A, b_i, r_i^B, c_i, r_i^C \in \mathbb{Z}_q\).

**Operation:** Abort if (at least) one of the following conditions does not hold:

1. For all \(\ell \in [n]\): \(\{\widehat{X}_\ell = r_\ell^X \cdot D + x_\ell \cdot G\}_{(x, X) \in \{(a, A), (b, B), (c, C)\}}\).

2. \(\sum_{\ell \in [n]} c_\ell = (\sum_{\ell \in [n]} a_\ell) \cdot (\sum_{\ell \in [n]} b_\ell)\).

That is, \(F_{PedMultEqTest}\) enables the parties, each holding commitments to \(a_i, b_i, c_i\), to verify whether \(\sum_\ell c_\ell = \sum_\ell a_\ell \cdot b_\ell\). Equipped with the above functionalities, protocol \(ManyPartyLeakyMult\) acts as follows: let \(\{\widehat{A}_\ell, \widehat{B}_\ell\}_{\ell \in [n]}\) be the common input, let \((a_\ell, \cdot), (b_\ell, \cdot)\) be the private inputs of \(P_\ell\), and let \(a = \sum_\ell a_\ell\) and \(b = \sum_\ell b_\ell\). The protocol starts with the parties computing shares of \(a \cdot b\) using the two-party multiplication functionality \(F_{2PC-Mult}\) to pairwise multiply \(a_\ell \cdot b_\ell\) for every \(\ell \neq h \in [n]\). This step does not enforce the parties to use the correct (committed) inputs to the two-party multiplications, so the result might be utterly wrong. In the next step, the parties call multiplication equality test \(F_{PedMultEqTest}\) to verify that the value generated in the first step equals \(a \cdot b\) (according to the committed shares of \(a\) and \(b\)). Since the adversary might behave dishonestly in the \(F_{2PC-Mult}\) phase, the above test might leak some information about the honest parties’ input. This leak is inherent for our implementation of the protocol, and is reflected in the functionality \(F_{LeakyMult}\). Details below.\(^{12}\)

**Protocol 4.3** \(ManyPartyLeakyMult\) — Securely computing \(F_{LeakyMult}\).

**Oracles:** \(F_{\text{RandLog}}^{2PC}, F_{2PC-Mult}, F_{PedMultEqTest}\).

**Common input:** \(\text{sid} \in \{0, 1\}^*, D \in \mathbb{G}, \{\widehat{A}_\ell, \widehat{B}_\ell\}_{\ell \in [n]}\).

**\(P_i\)'s input:** \((a_i, r_i^A), (b_i, r_i^B) \in \mathbb{Z}_q^2\).

**Operation:** Party \(P_i\) acts as follows:

1. // Generate (additive) shares of \(\sum a_i \cdot \sum b_i\).

   (a) For each \(\ell \neq i \in [n]\):

   Send \((\text{sid}||i||\ell, a_i)\) and \((\text{sid}||\ell||i, b_i)\) to \(F_{2PC-Mult}\).

---

\(^{12}\)One might wonder why our protocol has to use the two-party multiplication functionality, given that the parties hold additive commitments of their shares. Indeed, the parties can use their shares to compute a (verified) commitment of \(c = a \cdot b\). The problem is that the latter commitment cannot be opened to reveal the value of \(c\). This still holds even if instead of an (in-the-exponent) Pedersen commitments, one uses ElGamal in-the-exponent commitment whose private key is shared among the parties. The ElGamal commitment to \(c\) can be jointly opened to reveal the value of \(C = c \cdot G\), but (in most cases) the value of \(c\) cannot be extracted from \(C\). To overcome that we use the non-verified candidate value computed in the first step of the protocol (using the two-party multiplication functionality). Once you have such a candidate, it is possible to check whether it is equal to the value inside the Pedersen commitment (see Section 4.2).
Let $d^a_{i,\ell}$ and $d^b_{i,\ell}$ denote the values $P_i$ received from these two calls, respectively.

(b) Set $c_i = a_i \cdot b_i + \sum_{\ell \neq i} (d^a_{i,\ell} + d^b_{i,\ell})$.

2. // Prove knowledge of committed values. (In parallel to the above round).

Send $(\text{sid}, (D, \hat{A}_i), (a_i, r_i^A))$ and $(\text{sid}, (D, \hat{B}_i), (b_i, r_i^B))$ to $F_{2k}^{R\text{PedKeyGen}}$.

3. // Send a commitment to $c_i$.

Upon receiving $(\text{sid}, (D, \hat{A}_i), \ell)$ and $(\text{sid}, (D, \hat{B}_i), \ell)$ from $F_{2k}^{R\text{PedKeyGen}}$ for all $\ell \in [n]\{i\}$:

(a) Send $\hat{C}_i = \text{PedCom}_D(c_i; r_i^C)$, for $r_i^C \leftarrow \mathbb{Z}_q$, to all parties.

(b) Send $(\text{sid}, (D, \hat{C}_i), (c_i, r_i^C))$ to $F_{2k}^{R\text{PedKeyGen}}$.

4. // Verify that $\sum \hat{C}_\ell$ is a commitment to $\sum a_i \cdot b_i$.

Upon receiving $(\text{sid}, (D, \hat{C}_i), \ell)$ from $F_{2k}^{R\text{PedKeyGen}}$ for all $\ell \in [n]\{i\}$:

Send $(\text{sid}, D, \{\hat{A}_\ell, \hat{B}_\ell, \hat{C}_\ell\}_{\ell \in [n]}; (a_i, r_i^A), (b_i, r_i^B), (c_i, r_i^C))$ to $F_{\text{PedMultEqTest}}$.

5. Output $\{\hat{C}_\ell\}_{\ell \in [n]}$ as the common output, and $(c_i, r_i^C)$ as the local output.

4.1 Security

**Theorem 4.4** (Security of Protocol 4.3). Assume discrete-log is hard over $G$, then on $D \leftarrow F_{\text{PedKeyGen}}()$, Protocol 4.3 UC-realizes Functionality 3.7 in the $(F_{2k}^{R\text{PedKeyGen}}, F_{2\text{PC-Mult}}, F_{\text{PedMultEqTest}})$-hybrid model.

*Proof.* Fix an adversary $A$ corrupting the parties indexed by $C \subseteq [n]$ (recall that the all-corruption case is trivial), let $H = [n] \setminus C$, and assume for ease of notation that $n \in H$. The following random variables are defined with respect to the real execution of the protocol: for $\ell \in [n]$, let $(r^A_\ell, a_\ell, \hat{A}_\ell), (r^B_\ell, b_\ell, \hat{B}_\ell), (r^C_\ell, c_\ell, \hat{C}_\ell)$ be the values provided by $P_\ell$ to the calls to $F_{2k}^{R\text{PedKeyGen}}$. If a corrupted party sends different values to different honest parties, set it according to the values it provides in the calls with $P_h$. For $\ell \in C$ and $h \in H$, let $(a_{\ell,h}, b_{\ell,h})$ be the values the corrupted party $P_\ell$ provides to $F_{2\text{PC-Mult}}$ in the two joint calls with $P_h$, and let $(d^a_{\ell,h}, d^b_{\ell,h})$ and $(d^a_{h,\ell}, d^b_{h,\ell})$ be the output returned by these two joint calls to $P_\ell$ and $P_h$, respectively. We write

$$
\left(\sum_{\ell \in [n]} a_\ell\right) \cdot \left(\sum_{\ell \in [n]} b_\ell\right) = \left(\sum_{\ell \in C} a_\ell\right) \cdot \left(\sum_{\ell \in C} b_\ell\right) + \left(\sum_{\ell \in H} a_\ell\right) \cdot \left(\sum_{\ell \in H} b_\ell\right) + \left(\sum_{\ell \in H} a_\ell\right) \cdot \left(\sum_{\ell \in C} b_\ell\right) + \left(\sum_{\ell \in C} a_\ell\right) \cdot \left(\sum_{\ell \in H} b_\ell\right) + (1)
$$

Let $c \overset{\text{def}}{=} \sum_{\ell \in [n]} c_\ell$. Assume $\sum_{\ell \in C} c_\ell = d^c + \sum_{\ell \in C, h \in H} (d^a_{\ell,h} + d^b_{\ell,h})$, i.e., the corrupt parties send arbitrary values $F_{2\text{PC-Mult}}$ when interacting with the honest parties, but otherwise compute the $c_i$

28
correctly. In this case,
\[ c = \sum_{\ell \in C, h \in H} (d_{\ell,h}^a + d_{\ell,h}^b + d_{h,\ell}^d) \]
\[ = \sum_{\ell \in C} c_{\ell} - \sum_{\ell \in C, h \in H} (d_{\ell,h}^a + d_{\ell,h}^b) \]
(2)

So in the actual computation, it holds that
\[ c = \sum_{\ell \in C} c_{\ell} - \sum_{\ell \in C, h \in H} (d_{\ell,h}^a + d_{\ell,h}^b) \]
(3)

Note that both \( \delta^c \) and \( \{\delta^A_h, \delta^B_h\} \), can be extracted from A’s view.

As in the proof of Theorem 3.9, we assume without loss of generality that a corrupted parties never attempt to break the binding of a Pedersen commitment. It follows that if \( F_{\text{PedMultEqTest}} \) does not abort, then \( c = (\sum_{\ell \in [n]} a_{\ell}) \cdot (\sum_{h \in [n]} b_h) \), and thus

\[ \delta^c = -\sum_{\ell \in C, h \in H} (d_{\ell,h}^a + d_{\ell,h}^b) \]
(4)

Equipped with the above understandings, the ideal model simulator \( S \) for A is defined as follows.

Algorithm 4.5 (S—Simulator for A).

**Oracles:** \( F_{\text{LeakyMult}} \).

**Operation:** On common input \( \text{sid}, D \) and \( \{A_{\ell}, B_{\ell}\}_{\ell \in [n]} \), emulate a random execution of \( \text{ManyPartyLeakyMult} \) on this common input with A controlling the parties in C as follows:

**Round 1:** For each \( \ell \in C \) and \( h \in H \): sample the values of \( d_{\ell,h}^a \) and \( d_{\ell,h}^b \) to return from the calls to \( F_{\text{2PC-Mult}} \) uniformly at random.

**Round 2:**
- For each \( \ell \in C \), send \( ((\text{sid}, D, \{A_{\ell}, B_{\ell}\}_\ell \in [n]), (a_\ell, r_{\ell}^A), (b_\ell, r_{\ell}^B)) \) to \( F_{\text{LeakyMult}} \) on behalf of \( P_\ell \).
- Let \( \{\hat{C}_\ell\}_{\ell \in [n]} \) be the common output of \( F_{\text{LeakyMult}} \):

**Round 3:** Use \( \{\hat{C}_\ell\}_{\ell \in H} \) as the commitments sent by the honest parties, and set the messages that \( F_{\text{PedKlwg}} \) sends on behalf of the honest parties accordingly.
- Make a commitment-replacement call to \( F_{\text{LeakyMult}} \) with inputs \( \{(c_\ell, r_{\ell}^C)\}_{\ell \in C} \).

**Round 4:**
- Make a linear-test call to \( F_{\text{LeakyMult}} \) with input \( (\delta^c, \{\delta^A_h, \delta^B_h\}_{h \in H}) \).
- Instruct \( F_{\text{LeakyMult}} \) to abort party \( P_h \) if the emulated \( P_h \) aborted.
Since $S$ is non-rewinding, we only need to prove that the joint distribution of $A$’s view and the honest parties output in the real an emulated executions are computationally indistinguishable. By the hiding of the Pedersen commitments and the definition of $\mathcal{F}_{2\text{PC-Mult}}$, the two views are identical until Round 4. By Equation (4), the aborting decision taken in Round 4 in both executions is the same (random) function of the $A$’s view in the first three rounds (determined by the private inputs of the honest parties, which was not leaked yet). Hence, $A$’s views in the real and emulated execution are identically distributed.

Moving to the honest parties’ output, since the common outputs of the parties, including the aborting decisions, is reflected in the real and emulated views, it is left to prove that this still holds when adding the private outputs $\{(r_h^C, c_h)\}_{h \in H}$ of the honest parties (assuming no abort). Since the value of $r_h^C$ is determined by $c_h$ and $\bar{C}_h$, where the latter appears in the $A$’s view, we focus only on the value of $\{c_h\}_{h \in H}$. By definition, in both executions $\sum_{h \in H} c_h = c - \sum_{h \in C} c_h$, so we finish the proof by showing that in both executions the value of $\{c_h\}_{h \in H}$ is uniformly distributed under this restriction. This is clearly the case in the emulated execution. For the real execution, recall that each pair of honest parties are engaged in (two) calls to $\mathcal{F}_{2\text{PC-Mult}}$, and the randomness used by $\mathcal{F}_{2\text{PC-Mult}}$ to determine the output of these calls is not exposed to the adversary. Hence, we can think as if before outputting the $\{c_h\}_{h \in H}$, a random shift $s_h$ is added to each $c_h$ with $\sum_{h \in H} s_h = 0$, making the distribution of $\{c_h\}_{h \in H}$ uniform under the restriction that $\sum_{h \in H} c_h = c - \sum_{h \in C} c_h$.  

4.2 Multiplication Equality Test

In this section, we present protocol $\text{PedMultEqTest}$, a UC-implementation of $\mathcal{F}_{\text{PedMultEqTest}}$ (the Pedersen multiplication equality test). In high level, protocol $\text{PedMultEqTest}$ goes as follows: let $\{(a_\ell, r_\ell^A), (b_\ell, r_\ell^B), (c_\ell, r_\ell^C)\}$ be the private inputs of the parties. The parties first transform their Pedersen commitments into ElGamal commitments whose private key is shared among them. They then use these commitments to generate an ElGamal commitment of $\sum a_\ell \cdot \sum b_\ell - \sum c_\ell$ and open (using the shared private key) to see if it is indeed zero. This approach, as written above, is not zero knowledge since it leaks information about the value of $\sum a_\ell \cdot \sum b_\ell - \sum c_\ell$ (the ideal functionality only tells whether it is zero or not). To overcome it, the parties first multiple the resulting commitment by a random value (not known to any subset of the parties), so that a non-zero value is mapped to a uniform value. Details below.

Protocol 4.6 (PedMultEqTest—Securely computing $\mathcal{F}_{\text{PedMultEqTest}}$).

Oracles: $\mathcal{F}_{\text{Com-zk}}, \mathcal{F}_{\text{EGKlew}}, \mathcal{F}_{\text{EGProd}}, \mathcal{F}_{\text{RDL}}$.

Parties: $P_1, \ldots, P_n$.

Common input: $\text{sid} \in \{0,1\}^*$, $D_i, \{\hat{A}_\ell\}_{\ell=1}^n, \{\hat{B}_\ell\}_{\ell=1}^n, \{\hat{C}_\ell\}_{\ell=1}^n$.

Private input of $P_i$: $(a_\ell, r_\ell^A), (b_i, r_i^B), (c_\ell, r_\ell^C)$.

Operation: Party $P_i$ acts as follows:

1. //Generate common ElGamal public key with shared private key.
   
   (a) Send $(\text{sid}, \text{ComProve}, E_i = e_i \cdot G, e_i)$ to $\mathcal{F}_{\text{Com-zk}}$, for $e_i \leftarrow \mathbb{Z}_q$. 

30
Theorem 4.7 (Security of Protocol 4.6, single honest party)

In this section, we state and prove the security of Protocol 4.6 for adversaries corrupting at least $n$ parties, and in Section 4.2.2 we extend it for arbitrary number of corruptions.

2. //Generate ElGamal commitment to $a_i, b_i, c_i$ and prove their correctness.
For each $(x, X) \in \{(a, A), (b, B), (c, C)\}$, in parallel,

(a) Send $\bar{X}_i = \text{EgCom}_E(x_i; r_i^X)$, for $r_i^X \leftarrow \mathbb{Z}_q^*$, to all parties.

(b) Send $(\text{sid}, (E, \bar{X}_i, D, \bar{X}_i), (x_i, r_i^X, r_i^X))$ to $\mathcal{F}^\text{RDL-com-zk}$.

(c) Upon receiving $(\text{sid}, \text{DecomProof}, E_{\ell}, \ell)$ from $\mathcal{F}^\text{RDL-com-zk}$ for all $\ell \in [n] \setminus \{i\}$: let $E = \sum_{\ell=1}^{n} E_{\ell}$.

3. // Generate ElGamal commitment to $(\sum a_t) : (\sum b_t) - (\sum c_t)$.
Upon receiving $(\text{sid}, (E, D, \bar{X}_\ell, \bar{X}_\ell), \ell)$ from $\mathcal{F}^\text{RDL-EgPed}_z$ for all $\ell \in [n] \setminus \{i\}$:

(a) Let $\bar{B} = \sum_\ell \bar{B}_\ell$.

(b) Send $\bar{F}_i = \text{EgRerand}_E(a_i \cdot \bar{B}; r_i^F)$, for $r_i^F \leftarrow \mathbb{Z}_q^*$, to all parties.

(c) Send $(\text{sid}, (E, \bar{B}, \bar{A}_i, \bar{F}_i), (r_i^A, r_i^F, a_i))$ to $\mathcal{F}^\text{RDL-EgProdEg}_z$.

(d) Upon receiving $(\text{sid}, (E, \bar{F}_i, \bar{A}_\ell, \bar{F}_\ell), \ell)$ from $\mathcal{F}^\text{RDL-EgProdPedEg}_z$ for all $\ell \in [n] \setminus \{i\}$:

let $\bar{Y} = \sum_\ell \bar{F}_\ell = \sum_\ell \bar{C}_\ell$.

4. //Generate a commitment $\bar{Z}$ to a random multiple of the value committed in $\bar{Y}$.

(a) Send $\bar{Z}_i = a_i \cdot \bar{Y} + \text{EgCom}_E(0; r_i^Z)$, for $(r_i^Z, a_i) \leftarrow \mathbb{Z}_q^2$, to all parties.

(b) Send $(\text{sid}, (E, \bar{Y}, \bar{Z}_i), (r_i^Z, a_i))$ to $\mathcal{F}^\text{RDL-EgProdScalar}_z$.

(c) Upon receiving $(\text{sid}, (E, \bar{Y}, \bar{Z}_\ell), \ell)$ from $\mathcal{F}^\text{RDL-EgProdScalar}_z$ for all $\ell \in [n] \setminus \{i\}$, let $\bar{Z} = \sum_\ell \bar{Z}_\ell$.

5. //Verify that $\bar{Z} = (\bar{Z}_L, \bar{Z}_R)$ is a cnt to 0 (i.e., $\bar{Z}_R = (\sum_\ell e_\ell) \cdot \bar{Z}_L$).

(a) Send $W_i = e_i \cdot \bar{Z}_L$ to all parties.

(b) Send $(\text{sid}, i, (\bar{Z}_L, E_i, W_i), e_i)$ to $\mathcal{F}^\text{RDL-DH}_z$.

(c) Upon receiving $(\text{sid}, (\bar{Z}_L, E_\ell, W_\ell), \ell)$ from $\mathcal{F}^\text{RDL-DH}_z$ for all $\ell \in [n] \setminus \{i\}$, let $W = \sum_\ell W_\ell$.

(d) Abort if $\bar{Z}_R \neq W$.

4.2.1 Security

In this section, we state and prove the security of Protocol 4.6 for adversaries corrupting at least $n - 1$ parties, and in Section 4.2.2 we extend it for arbitrary number of corruptions.

**Theorem 4.7** (Security of Protocol 4.6, single honest party). Assume DDH is hard over $G$, then Protocol 4.6 UC-realizes Functionality 4.2 against adversaries corrupting at least $n - 1$ parties, in the $(\mathcal{F}^\text{RDL}, \mathcal{F}^\text{RDL-com-zk}, \mathcal{F}^\text{RDL-EgKwG}, \mathcal{F}^\text{RDL-EgPed}, \mathcal{F}^\text{RDL-EgProdEg}, \mathcal{F}^\text{RDL-DH})$-hybrid model.
Proof. Let A be an adversary corrupting the parties in C. Since the all-corruption case is trivial, we assume without loss of generality that |C| = n − 1, and assume for ease of notation that C = [n − 1]. Not knowing the private inputs of P_n, the following simulator emulates the messages of P_n that are hidden in ElGamal commitments, according to arbitrary inputs. It then sets (non-hidden) value of W_n sent in Round 5, to match the output of F_{PedMultEqTest}.

Algorithm 4.8 (S—Simulator for A).

Oracles: F_{PedMultEqTest}.

Operation: On common input (sid, D, \(\{A_t\}_{t=1}^n\), \(\{B_t\}_{t=1}^n\), \(\{C_t\}_{t=1}^n\)), emulate a random execution of PedMultEqTest on this common input with A controlling the parties in C and arbitrary (valid) inputs for P_n, while modifying the messages of the honest party P_n and those sent in response to its calls by the ideal functionalities as follows:

**Round 1:**
- Let \(\{e_\ell\}_{\ell \in C}\) be the values sent by the corrupted parties to \(F_{\text{com-zk}}(\cdot, n)\).

**Round 2:** Change the value of (\(\hat{A}_n, \hat{B}_n, \hat{C}_n\)) according to the common input.
- Let \(\{(a_\ell, r_A^\ell), (b_\ell, r_B^\ell), (c_\ell, r_C^\ell)\}_{\ell \in C}\) be the values sent by the corrupted parties to \(F_{\text{zk}}^{R_{\text{PedEq}}}(\cdot, n)\).
- For each \(\ell \in C\): send (sid, D, \(\{A_h\}_{h \in [n]}\), \(\{B_h\}_{h \in [n]}\), \(\{C_h\}_{h \in [n]}\), \(a_\ell, r_A^\ell\), \(b_\ell, r_B^\ell\), \(c_\ell, r_C^\ell\)) to \(F_{\text{PedMultEqTest}}\) on behalf of party P_\ell.

**Round 5:** //Generate a value for W_n to match the output of F_{PedMultEqTest}.

If \(F_{\text{PedMultEqTest}}\) returned true, change the value of W_n to \(\tilde{Z}_R - \sum_{\ell \in C} e_\ell \cdot \tilde{Z}_L\).

Else, change it to a uniform value in \(G\).
- If the emulated P_n aborts, sends abort to \(F_{\text{PedMultEqTest}}\).

Since S is non-rewinding, we only need to prove that the joint distribution of A’s view and the output of P_n in the real an emulated executions are computationally indistinguishable. Actually, since (by construction) P_n accepts in the ideal execution only if does so in emulated view of A, we only need to show that for A’s view. Since we assume DDH, the ElGamal commitments used in the protocol are computationally hiding. Thus, it suffices to show that the values of W_n sent by P_n in Round 5, the only non hidden value in A’s view, in the two executions are computationally indistinguishable (given the rest of the view).

For both the real and emulated executions, let \(\{e_\ell\}_{\ell \in C}\) be the shares of the private key sent to \(F_{\text{com-zk}}(\cdot, n)\) by the corrupted parties in Round 1, let \(\{(a_\ell, b_\ell, c_\ell)\}_{\ell \in C}\) be the values sent by the corrupted parties to \(F_{\text{zk}}^{R_{\text{PedEq}}}(\cdot, n)\) in Round 2 and let \((a_n, b_n, c_n)\) be the, relevant part, of the private input of P_n. Assume that \(\sum_{\ell \in [n]} a_\ell \cdot \sum_{\ell \in [n]} b_\ell = \sum_{\ell \in [n]} c_\ell\). By construction, the value of W_n in both executions is equal to \(\tilde{Z}_R - \sum_{\ell \in C} e_\ell \cdot \tilde{Z}_L\). Hence, W_n is a deterministic function of A’s view, yielding that the distributions remain computationally indistinguishable also given this value. For the case \(\sum_{\ell \in [n]} a_\ell \cdot \sum_{\ell \in [n]} b_\ell \neq \sum_{\ell \in [n]} c_\ell\) we show that in the real execution the value of W_n is computationally indistinguishable from a uniform element of \(G\) (given the rest of
A’s view). This concludes the proof since \( W_n \) in the emulated execution is uniformly distributed in \( G \) (independently of the rest of the view). Assume there exists an adversary \( A \), a distinguisher \( D \), and a set of inputs for \text{PedMultEqTest} with \((\sum_{\ell \in [n]} a_{\ell}) \cdot (\sum_{\ell \in [n]} b_{\ell}) \neq \sum_{\ell \in [n]} c_{\ell})\), such that \( D \) can tell whether the value of \( W_n \) in \( A \)'s view was replaced by a uniform element of \( G \). We show that under this assumption, the following algorithm breaks DDH.

\textbf{Algorithm 4.9 (DDH-Breaker).}

Input: \((E, R, T) \in G^3\).

Operation: Start emulating a random execution of \text{PedMultEqTest} on the claimed inputs, while modifying the messages of the honest party \( P_n \), and those on its behalf by the ideal functionalities, as follows:

\textbf{Round 1–3:} Modify the value of \( E_n \) to \( E - \sum_{\ell \in \mathcal{C}} E_{\ell} \).

// So now the shared public key chosen by the protocol is equal to \( E \).

- Let \( \tilde{Y} \) be the value computed by \( P_n \) in Round 3d, and let \( r^Y \) be the discrete log of \( \tilde{Y}_L \), i.e., \( r^Y \cdot G = \tilde{Y}_L \).

\( (\text{Since } \tilde{Y}_L \text{ is independent of } E, \text{ the value of } r^Y \text{ can be efficiently computed from the values all parties sent to the zero-knowledge functionalities in rounds 2–3.)} \)

\textbf{Round 4:} Modify the value of \( Z_n \) to \((R, U) \), for \( U \leftarrow_G R \).

- Let \((o_\ell, r^Z_\ell)\) be the values the corrupted parties sent to \( \mathcal{F}_{D\ell}^{\mathcal{R}_{\text{EgProdScalar}}}(\cdot, n) \) in Round 4b.

\textbf{Round 5:} Modify the value of \( W_n \) to \( T + (\sum_{\ell \in \mathcal{C}} r^Z_\ell + o_\ell \cdot r^Y) \cdot E \).

Output the prediction of \( D \) on the generated view.

Since \((\sum_{\ell \in [n]} a_{\ell}) \cdot (\sum_{\ell \in [n]} b_{\ell}) \neq \sum_{\ell \in [n]} c_{\ell})\), the value of \( \tilde{Z}_n \) sent by \( P_n \) in the real execution is an encryption of a uniform value. Hence, the first four rounds in the emulation done by DDH-Breaker are distributed exactly like these rounds in the real execution (on these inputs). Let \( \{ \tilde{Z}_\ell \}_{\ell \in [n]} \) be the values sent by the parties in Round 4 (of the emulation) and let \( \tilde{Z} = (\tilde{Z}_L, \tilde{Z}_R) = \sum_{\ell \in [n]} \tilde{Z}_\ell \). Let \( e \) and \( r \) be the discrete-log of \( E \) and \( R \), respectively. By construction, \( \tilde{Z}_L = r \cdot G + \sum_{\ell \in \mathcal{C}} (r^Z_\ell + o_\ell \cdot r^Y) \cdot G \). Hence, if \( T = (e \cdot r) \cdot G \), then \( W_n = T + \sum_{\ell \in \mathcal{C}} (r^Z_\ell + o_\ell \cdot r^Y) \cdot E = e \cdot \tilde{Z}_L \), as it should be in a random execution of the protocol on these inputs. Where if \( T \) is a uniform (and independent) element of \( G \), then so is \( W_n \). Thus, DDH-Breaker tells \((E = e \cdot G, R = r \cdot G, (e \cdot r) \cdot G)\) from \((E = e \cdot G, R = r \cdot G, u \cdot G)\), for uniform \( e, r, u \leftarrow_G R \). Namely, it breaks DDH over \( G \). \( \square \)

\subsection{4.2.2 Many Honest Parties}

In this section, we state and prove the security of Protocol 4.6 to adversaries corrupting arbitrary number of parties.

\textbf{Theorem 4.10 (Security of Protocol 4.6).} Assume DDH is hard over \( G \), then Protocol 4.6 UC-realizes Functionality 4.2 in the \((\mathcal{F}_{\text{com-zk}}^{\mathcal{R}_{\text{Dil}}, \mathcal{F}_{\text{EGK}}^{\mathcal{R}_{\text{Eg}}}, \mathcal{F}_{\text{EgPedEq}}, \mathcal{F}_{\text{EgProdEq}}, \mathcal{F}_{\text{DH}}})\)-hybrid model.
Proof. Let A be an adversary corrupting the parties in C. By Theorem 4.7, we can assume that \( |C| < n - 1 \). Let \( \mathcal{H} = [n] \setminus C \), and assume for ease of notation that \( n \in \mathcal{H} \). We adjust the simulator \( S \) defined in Algorithm 4.8 into a simulator \( \hat{S} \) that emulates the messages of \( P_h \) with \( h \in \mathcal{H} \setminus \{n\} \), as follows: in the first four rounds, \( \hat{S} \) emulates all \( P_h \) messages like \( S \) emulates the messages of \( P_n \), here with respect to \( (A_h, B_h, C_h) \). In Round 5, \( \hat{S} \) always modifies the value of \( W_h \) to a uniform (and independent) element in \( G \). Finally, if an emulated party aborts, it instructs the ideal functionality to abort this party.

By construction, an honest party accepts in the ideal execution only if it does so in emulated view of \( A \). Hence, we only need to prove that \( A \)'s view in the real and ideal executions are computationally indistinguishable. Like in the single honest party case, assuming DDH, all but the messages sent in Round 5 of the emulated execution are computationally indistinguishable from the real one. Furthermore, the very same proof also yields that in the case \( (\sum_{\ell \in [n]} a_{\ell}) \cdot (\sum_{\ell \in [n]} b_{\ell}) \neq \sum_{\ell \in [n]} c_{\ell} \), the values of \( \{W_h\}_{h \in \mathcal{H}} \) in the real execution are (jointly) indistinguishable from uniform, and hence indistinguishability also holds in this case. So it is left to handle the case \( (\sum_{\ell \in [n]} a_{\ell}) \cdot (\sum_{\ell \in [n]} b_{\ell}) = \sum_{\ell \in [n]} c_{\ell} \). Since, in the real and emulated executions, \( W_n \) is a deterministic function of the rest of \( A \)'s view: \( Z_R = \sum_{\ell \in [n]} e_{\ell} \cdot Z_L = \sum_{h \in \mathcal{H} \setminus \{n\}} W_h \), we can safely ignore its contribution. We finish the proof showing that, without the value of \( W_n \), the values of \( \{W_h\}_{h \in \mathcal{H} \setminus \{n\}} \), given the rest of the view, are indistinguishable from uniform. Indeed, without the value of \( W_n \), the ElGamal commitments used in the protocol are hiding from \( A \)'s point of view.\(^{13}\) Hence, a simple reduction yields that \( A \) cannot distinguish the above case (where \( (\sum_{\ell \in [n]} a_{\ell}) \cdot (\sum_{\ell \in [n]} b_{\ell}) = \sum_{\ell \in [n]} c_{\ell} \) from the case that \( (\sum_{\ell \in [n]} a_{\ell}) \cdot (\sum_{\ell \in [n]} b_{\ell}) \neq \sum_{\ell \in [n]} c_{\ell} \). Thus, since \( A \) cannot distinguish \( \{W_h\}_{h \in \mathcal{H} \setminus \{n\}} \) from uniform in the latter case, it cannot do so also in the former.

\( \square \)

4.3 Weak Two-Party Multiplication

In this section, we present a weaker variant of the perfect two-party multiplication functionality \( \mathcal{F}_{\mathrm{2PC-Mult}} \), see Functionality 4.1, and prove that replacing \( \mathcal{F}_{\mathrm{2PC-Mult}} \) in Protocol 4.3 with this weaker variant, does not hurt its security. The advantage of using the weaker variant is that Haitner et al. [Hai+22] presented an OT-based implementation of this weak functionality that is almost as efficient as the best known OT-based semi-honest two-party multiplication.\(^{14}\)

We use the following definitions: for two vector \( x, y \in \mathbb{Z}_q^m \), let \( \langle x, y \rangle = \sum x_i \cdot y_i \) and \( x \ast y = (x_1 \cdot y_1, \ldots, x_m \cdot y_m) \). and let \( \text{Ham}(x, y) = \{|i : x_i \neq y_i|\} \). A vector is polychromatic if it is not close to being mono-chromatic (i.e., having the same element in all its coordinates).

Definition 4.11 (polychromatic vector). A vector \( d \in \mathbb{Z}_q^m \) is \( t \)-polychromatic if for every \( y \in \mathbb{Z}_q \) it holds that \( \text{Ham}(d, y^n) \geq t \).

Given the above definitions, the weak two-party multiplication functionality is defined as follows:

Functionality 4.12 (\( \mathcal{F}_{\mathrm{2PC-WeakMult}} \)—Two-party weak multiplication).

\(^{13}\) We could not use this simple argument in the previous reductions, since hiding against an adversary who knows \( W_{\ell} = e_{\ell} \cdot Z_L \) for all \( \ell \in [n] \), where \( e = \sum e_{\ell} \) is the secret key, is not captured by the security of the ElGamal commitment.

\(^{14}\) [Hai+22] do not prove the UC-security of their implementation for this functionality. However, since the simulator in their security proof is non-rewinding and since their protocol has an implicit start synchronization phase, their protocol UC-realizes the functionality in consideration (in the OT-hybrid model).
Security parameter: $\kappa \in \mathbb{N}$. Let $m = \lceil \log q \rceil + \kappa$.

Common input: $\text{sid} \in \{0,1\}^*$. 

$P_1$’s input: $a \in \mathbb{Z}_q$, and optional $d \in \mathbb{Z}_q^m$. 

$P_2$’s input: $b \in \mathbb{Z}_q$, and optional $s_2 \in \mathbb{Z}_q$. 

Operation:

Abort if $d$ is set but not $\kappa/2$-polychromatic. 

If $d$ is not set, act according to $F_{2\text{PC-Mult}}(\text{sid}, a, (b, s_2))$. 

Else:

1. Sample $(v, t) \leftarrow \mathbb{Z}_q^n \times \{-1,1\}^m$ such that $\langle v, t \rangle = b$. 
2. If $s_2$ is not set let $s_2 \leftarrow \mathbb{Z}_q$. 
3. Output $((s_1, v), s_2)$ for $s_1 = a \cdot b - s_2 + \langle v, d \ast t \rangle$. 

Haitner et al. [Hai+22] proved the following characterization of the above functionality.

**Lemma 4.13** ([Hai+22], Lemma 4.3). Let $q > 2$ be a prime number, let $\kappa \in \mathbb{N}$ and $m \overset{\text{def}}{=} \lceil \log q \rceil + \kappa$. Let $d \in \mathbb{Z}_q^m$, let $\ell = \min_{y \in \mathbb{Z}_q} \{\text{Ham}(d, y^m)\}$, let $\lambda = \min\{\ell, \kappa - 5, \log q, m/3\}$, and let $(V, T) \leftarrow \mathbb{Z}_q^n \times \{-1,1\}^m$. Then for every $b \in \mathbb{Z}_q$, with probability $1 - 2^{-\lambda/2+3}$ over $v \leftarrow \mathbb{Z}_q^n | \langle V, T \rangle = b$, it holds that 

$$H_\infty(\langle v, d \ast T \rangle | \langle v, T \rangle = b) \geq \lambda/2 + 4.$$ 

It follows that party $P_1$ can either act “honestly” in $F_{2\text{PC-WeakMult}}$, by not setting the optional vector $d$, and thus effectively acts like in the perfect multiplication. Otherwise, it makes the output of $P_2$ unpredictable from $P_1$ point of view, having min-entropy $\approx \kappa/4$ conditioned on $P_1$’s view, and this unpredictability holds even when given $P_2$’s input. ($P_2$ has exactly the same power on the outcome like it has in the perfect multiplication).

### 4.3.1 Using Weak Two-Party Multiplication in Protocol 4.3

We claim that Protocol 4.3 remains secure when using $F_{2\text{PC-WeakMult}}$ instead of $F_{2\text{PC-Mult}}$.

**Theorem 4.14** (Security of Protocol 4.3, when using weak two party multiplication). Assume discrete-log is hard over $\mathbb{G}$. Then on common input $D \leftarrow F_{\text{PedKeyGen}}()$, Protocol 4.3 UC-realizes Functionality 3.7 in the $(F_{2k}^{\text{PedKInt}}, F_{2\text{PC-WeakMult}}, F_{\text{PedMultEqTest}})$-hybrid model.

**Proof.** The only source of difference from using $F_{2\text{PC-WeakMult}}$ instead of $F_{2\text{PC-Mult}}$ might come from the party taking the part of $P_1$ in the call to $F_{2\text{PC-WeakMult}}$ uses a $\kappa/2$-polychromatic vector (see Definition 4.11). Lemma 4.13 yields that if a a corrupted party sends a $\kappa/2$-polychromatic, then the value of $\sum_{h \in \mathcal{H}} c_h$ becomes unpredictable from the adversary’s point of view (even it is given the honest parties inputs). We conclude that if such a vector was sent in the real execution, then with all but negligible probability $F_{\text{PedMultEqTest}}$ returns false.
Given the above, we modify the simulator described in Algorithm 4.5 so that in Round 1 it verifies that none of the calls to $F_{2PC-WeakMult}$ was done with a $\kappa/2$-polychromatic vector (it aborts the execution is such a call was made). By the above, and the security of the protocol when using $F_{2PC-Mult}$, the real and emulated executions are computationally indistinguishable also when using $F_{2PC-WeakMult}$.

4.4 Using ElGamal Commitments

As in Section 3, the perfectly hiding and computationally binding Pedersen commitments used in the above protocols can be replaced with the computationally hiding and perfectly binding ElGamal commitments. The security proof of this variant follows from that of the Pedersen commitments based protocols given above, using the same adaptation we describe in Section 3.2.

4.5 A More Efficient Protocol

In its current form, protocol $ManyPartyLeakyMult$ takes three rounds. When combined with the five rounds implementation of $F_{PedMultEqTest}$ done in protocol $PedMultEqTest$, see Protocol 4.6, and the two-round implementation of $F_{2PC-WeakMult}$, given by [Hai+22], makes it an 8-round protocol. However, in the price of modularity, the round efficiency of the signing phase can be reduced to five rounds.

First, we reuse the shared ElGamal keys generated in Protocol 4.6, and omit the ElGamal key-generation round from the counting. Next, we combine protocol $PedMultEqTest$ into $ManyPartyLeakyMult$, and move to using ElGamal commitments right from the beginning of protocol $ManyPartyLeakyMult$. Finally, we start the calculation the $F_{\ell}$'s done in Round 3 of $PedMultEqTest$, right after the first round of $ManyPartyLeakyMult$ is over. Overall, this makes it a five-round protocol. It is not hard to see that the security proof of the original protocol can be adjusted to argue the security of this variant.

5 Experimental Results

We implemented our protocol in C++ and ran it locally on an 2.3 GHz, 8-Core, Intel(R) i9. We ran experiments from 3 to 20 parties; each execution was run 16 times, and took the average. The results can be seen in Figure 1. This version uses the weak-two party multiplication of [Hai+22]. For the zero-knowledge POK protocols, we used the [Fis05] transformation. As can be clearly seen, the signing time is practical (especially for cryptocurrency applications): from 206ms for 3 parties, to about 1sec for 11 parties and less than 4sec for 20 parties. We remark that since we run on eight cores, there is a non-realistic slowdown when the number of parties exceeds 8 (since there is more than one party on each core). We also mention that the stated running time includes creating OT pairs from scratch, where a more efficient approach is to save a few such pairs from the last invocation and in the new invocation, only interact in the faster OT extension protocol. A comparison of the signing time of our protocol and three other protocols can be found in Figure 2. You can clearly see that our protocol outperforms the others by a factor of 4-5.
Figure 1: Key generation and signing time of our protocol. The running times, in milliseconds, for key generation (bottom line in blue) and signing (top line in red) for 2-20 parties. (See details in Table 1.)

![Graph showing key generation and signing times](image)

**Table 1: Key generation and signing time of our protocol, in milliseconds.**

<table>
<thead>
<tr>
<th>Number of Parties</th>
<th>Key generation</th>
<th>Signing</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>206</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>254</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>340</td>
</tr>
<tr>
<td>8</td>
<td>13</td>
<td>653</td>
</tr>
<tr>
<td>11</td>
<td>19</td>
<td>1138</td>
</tr>
<tr>
<td>14</td>
<td>28</td>
<td>1803</td>
</tr>
<tr>
<td>17</td>
<td>41</td>
<td>2662</td>
</tr>
<tr>
<td>20</td>
<td>55</td>
<td>3657</td>
</tr>
</tbody>
</table>
Table 2: Comparison of signing times, in milliseconds, between our protocol (Intel, 2.3 GHz, 8-Core, i9), [GG18; GG20] (Intel, 2.3 GHz, i5), and [Can+20b] (Intel, 2.8 GHz, 4-Core, i7). All times were measured without communication cost. The running time for CMP20 is for the pre-signing, and the running time for [GG18] is as reported in [GG20] (the times reported in [GG18] itself do not include zero-knowledge proofs necessary for security).
References


[Cur] Curv (cit. on p. 4).


A Implementing Auxiliary Functionalities

In this section, we discuss the implementations of the functionalities we use in Sections 3 and 4.2. We start with the zero-knowledge functionalities, Appendix A.1, and then handle the commitment functionalities, Appendix A.2.

A.1 Zero Knowledge

In this section, we present zero-knowledge proof-of-knowledge sigma protocols for the relations for which we use proof-of-knowledge (POK) zero-knowledge functionalities. In the random-oracle-model, one can use the transformation of [Fis05] to zero-knowledge with non-rewinding extraction, to get a non-interactive protocol that UC-realizes the zero-knowledge functionality in consideration. One can also just apply the plain Fiat-Shamir transform [FS86] and hope the resulting protocol is UC-secure, but there is no rigorous justification for this assumption.

We present sigma protocols for the following relations (all with respect to to the fix $q$-size group $G$, fixed $G \in G$ and $\overline{G} = G \times G$):
1. Knowledge of discrete log.

\[ R_{DL} = \{(A, B, w) : B = w \cdot A\} \]

The sigma protocol for this relation is just the standard Schnorr proof [Sch89], whose cost is one \((G)\) exponentiation for the prover and two for the verifier, and communication cost is one element \(G\) and one element of \(\mathbb{Z}_q\).

2. DH tuple:

\[ R_{DH} = \{(\tilde{A} = (A_1, A_2), \tilde{B} = (B_1, B_2), w) : \tilde{B} = w \cdot \tilde{A}\} \]

This is just knowledge of discrete log over the group \(\tilde{G} = G \times G\).


\[ R_{PedKlwg} = \{((D, \hat{A}), (a, r)) : \hat{A} = \text{PedCom}_D(a; r)\} \]

The sigma protocol for this relation is described in Appendix A.1.1.

4. Pedersen commitment of a value:

\[ R_{PedEqVal} = \{((D, \hat{A}, a), r) : \hat{A} = \text{PedCom}_D(a; r)\} \]

Since \(\text{PedCom}_D(a; r) = r \cdot D + a \cdot G\), it holds that \(\hat{A}\) is a commitment to \(a\) if \(\text{PedCom}_D(a; r) - a \cdot G = r \cdot D\). Therefore, proving that \(\text{PedCom}_D(a; r)\) is a commitment to \(a\) reduces to proving the knowledge of the discrete log of \((\hat{A} - a \cdot G)\) relative to \(D\).

5. Pedersen commitment of a value in the exponent:

\[ R_{PedEqExp} = \{((D, \hat{A}, A), (a, r)) : \hat{A} = \text{PedCom}_D(a; r), A = G \cdot a\} \]

The sigma protocol for this relation concatenates, with the same challenge \(e\) and answer \(z\), a proof of knowledge for the discrete log \(\text{PedCom}_D(a; r) - A\) relative to \(D\), and for the discrete log of \(A\) relative to \(G\).

6. Knowledge of ElGamal commitment:

\[ R_{EGKlwg} = \{((E, \tilde{A}), (a, r)) : \tilde{A} = \text{EgCom}_E(a; r)\} \]

The sigma protocol for this relation is described in Appendix A.1.2.

7. ElGamal equals Pedersen:

\[ R_{EgEqPed} = \{((E, \tilde{A}, D, \hat{A}), (a, \tilde{r}, \hat{r})) : \tilde{A} = \text{EgCom}_E(a; \tilde{r}) \land \hat{A} = \text{PedCom}_E(a; \hat{r})\} \]

The sigma protocol for this relation is described in Appendix A.1.3.
8. Knowledge of ElGamal scalar product:

\[ R_{EgProdScalar} = ((E, \tilde{A}, \tilde{B}), (r, c)) : \tilde{B} = c \cdot \tilde{A} + EgCom_E(0; r) \]

The protocol for this relation is described in Appendix A.1.4.

9. Product of ElGamal commitments:

\[ R_{EgProdEq} = \{(E, e^A, e^B, e^C) : \tilde{B} = EgCom_E(b; r^b) \land \tilde{C} = b \cdot \tilde{A} + EgCom_E(0; r^0) \} \]

The sigma protocol for this relation is described in Appendix A.1.5.

10. ElGamal commitment of a value:

\[ R_{EGEqVal} = \{(E, e^A, a) : e^A = EgCom_E(a; r) \} \]

Since \( EgCom_E(a; r) = (rG, r^G + aG) \), it holds that \( \tilde{A} \) is a commitment to \( a \) if \( EgCom_E(a; r) - (0, aG) = (rG, r \cdot E) \). Therefore, proving that \( (\tilde{A}_L, \tilde{A}_R) = \tilde{A} \) is a commitment to \( a \) reduces to proving the knowledge of the discrete log of \( A_R - a \cdot G \) relative to \( A_L \).

11. ElGamal commitment of a value in the exponent:

\[ R_{EGEqExp} = \{(E, e^A, A) : e^A = EgCom_E(a; r), \quad A = G \cdot a \} \]

Let \( (\tilde{A}_L, \tilde{A}_R) = \tilde{A} \). The sigma protocol for this relation concatenates, with the same challenge \( e \) and answer \( z \), a proof of knowledge for the discrete log of \( A_R - A \) relative to \( A_L \), and for the discrete log of \( A \) relative to \( G \).

**Notational conventions.** In the following we use \( r, s, t \in \mathbb{Z}_q \) to denote random coins uses for the ElGamal and Pedersen commitments, and \( a, b, c, z \in \mathbb{Z}_q \) for committed values.

**A.1.1 Knowledge of Pedersen Commitment**

We present a zero-knowledge sigma protocol for the relation

\[ R_{PedKlwg} = \{(D, \tilde{A}), (a, r)) : \tilde{A} = PedCom_D(a; r) \} \]

**Protocol A.1 (sigma protocol (P, V) for \( R_{PedKlwg} )).**

- **Security parameter:** \( \kappa \in \mathbb{N} \).
- **Common input:** \( (D, \tilde{A}) \).
- **P’s input:** \( (a, r) \).

**Operation:**

1. **P:** Sends \( \tilde{X} = s \cdot D + b \cdot G \), for \( b, s \leftarrow \mathbb{Z}_q \).
2. **V:** Send \( e \leftarrow \{0, 1\}^\kappa \).
3. **P:** Send \( (z = b + e \cdot a, t = s + e \cdot r) \).

- **V accepts if** \( PedCom_D(z; t) = \tilde{X} + e \cdot \tilde{A} \).
Completeness. Since
\[ \text{PedCom}_D(z; t) = t \cdot D + z \cdot G = (s + e \cdot r) \cdot D + (b + e \cdot a) \cdot G = \tilde{X} + e \cdot \tilde{A}, \]
\( V \) always accepts in an all-honest execution.

Special soundness. Assume there exists \((\hat{X}, e, e', z, t, z', t')\) such that \(e \neq e'\) (i.e., \( \text{mod } q \)) and both \((\hat{X}, e, b, t)\) and \((\hat{X}, e', b', t')\) are accepting transcripts. Hence,
\[ t \cdot D + z \cdot G = \hat{X} + e \cdot \hat{A} \quad \text{and} \quad t' \cdot D + z' \cdot G = \hat{X} + e' \cdot \hat{A} \]
Subtracting the equations from each other, we have
\[ (t - t') \cdot D + (z - z') \cdot G = (e - e') \cdot A \]
and thus
\[ \hat{A} = (t - t') \cdot (e - e')^{-1} \cdot D + (z - z') \cdot (e - e')^{-1} \cdot G \]  
(Note that since \(e \neq e'\), the value \((e - e')^{-1}\) exists and can be efficiently computed.) Hence, \( r = (t - t') \cdot (e - e')^{-1} \) and \( a = (z - z') \cdot (e - e')^{-1} \) form a valid opening of \( \hat{A} = \text{PedCom}(a; r) = a \cdot G + r \cdot D \).

Honest-verifier zero knowledge. Given \( e \), the simulator samples \( t, z \leftarrow \mathbb{Z}_q \), computes
\[ \tilde{X} = t \cdot D + z \cdot G - e \cdot \tilde{A} \]
and outputs \((\tilde{X}, t, z)\).
Assume there exist \((a, r)\) such that \( \tilde{A} = r \cdot D + a \cdot G \). Then
\[ \tilde{X} = (t - e \cdot r) \cdot D + (z - e \cdot a) \cdot G \]
Since \( s = t - e \cdot r \) and \( b = z - e \cdot a \) are uniformly distributed over \( \mathbb{Z}_q \), we deduce that the output distribution of the simulator is identical to the honest verifier view on such input.

A.1.2 Knowledge of ElGamal Commitment
We present a zero-knowledge sigma protocol for the relation
\[ \mathcal{R}_{\text{EGKlwg}} = \{((E, \tilde{A}), (a, r)): \tilde{A} = \text{EgCom}_E(a; r)\} \]

Protocol A.2 (sigma protocol for \( \mathcal{R}_{\text{EGKlwg}} \)).

Security parameter: \( \kappa \in \mathbb{N} \).
Common input: \((E, \tilde{A})\).
P’s input: \((a, \tilde{r})\).

Operation:
Completeness.

\[
\tilde{B} + e \cdot A = (\tilde{s} \cdot G, \tilde{s} \cdot E + b \cdot G) + e \cdot (\tilde{r} \cdot G, \tilde{r} \cdot E + a \cdot G)
= ((\tilde{s} + e \cdot \tilde{r}) \cdot G, (\tilde{s} + e \cdot \tilde{r}) \cdot E + (e \cdot a + b) \cdot G)
= \text{EgCom}_E(z; \tilde{t})
\]

\(V\) always accepts in an all-honest execution.

**Special soundness.** Assume there exists \((\tilde{B}, e, e', z, \tilde{t}, z', \tilde{t'})\) such that \(e \neq e'\) and both \((\tilde{B}, e, z, \tilde{t})\) and \((\tilde{B}, e', z', \tilde{t'})\) are accepting transcripts. Hence,

\[
\tilde{B} + e \cdot A = (\tilde{t} \cdot G, \tilde{t} \cdot E + z \cdot G)
\]

and

\[
\tilde{B} + e' \cdot \tilde{A} = (\tilde{t'} \cdot G, \tilde{t'} \cdot E + z' \cdot G)
\]

Subtracting the equations from each other, we have

\[
(e - e') \cdot \tilde{A} = ((\tilde{t} - \tilde{t'}) \cdot G, (\tilde{t} - \tilde{t'}) \cdot E + (z - z') \cdot G)
\]

Thus,

\[
\tilde{A} = (e - e')^{-1} \cdot ((\tilde{t} - \tilde{t'}) \cdot G, (\tilde{t} - \tilde{t'}) \cdot E + (z - z') \cdot G)
\]

Taking \(a = (z - z') \cdot (e - e')^{-1}\), \(\tilde{r} = (\tilde{t} - \tilde{t'}) \cdot (e - e')^{-1}\) and \(\tilde{r} = (\tilde{t} - \tilde{t'}) \cdot (e - e')^{-1}\), we have that \(\tilde{A} = \text{EgCom}_E(a; \tilde{r})\), as required.

**Honest-verifier zero knowledge.** Given \(e\), the simulator chooses random \(z, \tilde{t} \leftarrow \mathbb{Z}_q\), computes

\[
\tilde{B} = \text{EgCom}_E(z; \tilde{t}) - e \cdot \tilde{A}
\]

and outputs \((\tilde{B}, z, \tilde{t})\).

Assume there exists \((a, \tilde{r})\) such that \(\tilde{A} = \text{EgCom}_E(a; \tilde{r})\). Then

\[
\tilde{B} = \text{EgCom}_E(z - a \cdot e; \tilde{t} - e \cdot \tilde{r})
\]

Since \(b = z - e \cdot a\) and \(\tilde{s} = \tilde{t} - e \cdot \tilde{r}\) are uniform in \(\mathbb{Z}_q\), we deduce that the output distribution of the simulator is identical to the honest verifier view on such input.
A.1.3 ElGamal Equals Pedersen

We present a zero-knowledge sigma protocol for the relation

$$\mathcal{R}_{\text{EgEqPed}} = ((E, \tilde{A}, D, \tilde{\lambda}), (a, \tilde{r}, \tilde{\tau})) : \tilde{A} = \text{EgCom}_E(a; \tilde{r}) \land \tilde{A} = \text{PedCom}_D(a; \tilde{\tau}).$$

**Protocol A.3** (sigma protocol for $\mathcal{R}_{\text{EgEqPed}}$).

*Security parameter:* $\kappa \in \mathbb{N}$.

*Common input:* $(E, \tilde{A}, D, \tilde{\lambda})$.

*P’s input:* $(a, \tilde{r}, \tilde{\tau})$.

*Operation:*

1. **P:** Sends $(\tilde{B} = \text{EgCom}_E(b; \tilde{s}), \tilde{B} = \text{PedCom}_D(b; \tilde{s}))$, for $b, \tilde{s}, \tilde{s} \in \mathbb{Z}_q$.
2. **V:** Sends $e \in \{0, 1\}^\kappa$.
3. **P:** Send $(z = b + e \cdot a, \tilde{t} = \tilde{s} + e \cdot \tilde{r}, \tilde{t} = \tilde{s} + e \cdot \tilde{\tau})$.

   - **V** accepts if $\tilde{B} + e \cdot \tilde{A} = \text{EgCom}_E(z; \tilde{t})$, and $\tilde{B} + e \cdot \tilde{A} = \text{PedCom}_D(z; \tilde{t})$.

Note that the above protocol is just proof of knowledge for a Pedersen commitment and a proof knowledge for ElGamal commitment put together, using the same $z$.

**Completeness.**

$$\tilde{B} + e \cdot \tilde{A} = (\tilde{s} \cdot G, \tilde{s} \cdot E + b \cdot G) + e \cdot (\tilde{r} \cdot G, \tilde{r} \cdot E + a \cdot G)$$
$$= ((\tilde{s} + e \cdot \tilde{r}) \cdot G, (\tilde{s} + e \cdot \tilde{r}) \cdot E + (e \cdot a + b) \cdot G)$$
$$= \text{EgCom}_E(z; \tilde{t})$$

and

$$\tilde{B} + e \cdot \tilde{A} = \tilde{s} \cdot D + b \cdot G + e \cdot (\tilde{r} \cdot D + a \cdot G)$$
$$= (\tilde{s} + e \cdot \tilde{r}) \cdot D + (e \cdot a + b) \cdot G$$
$$= \text{PedCom}_D(z; \tilde{t}),$$

**V** always accepts in an all-honest execution.

**Special soundness.** Assume there exists $(\tilde{B}, \tilde{B}, e, e', z, \tilde{t}, \tilde{t}, z', \tilde{r}, \tilde{t})$ such that $e \neq e'$ and both $(\tilde{B}, \tilde{B}, e, z, \tilde{t}, \tilde{t})$ and $(\tilde{B}, \tilde{B}, e', z', \tilde{r}, \tilde{t})$ are accepting transcripts. Hence,

$$\tilde{B} + e \cdot \tilde{A} = (\tilde{t} \cdot G, \tilde{t} \cdot E + z \cdot G), \tilde{B} + e \cdot \tilde{A} = \tilde{t} \cdot D + z \cdot G$$

and

$$\tilde{B} + e' \cdot \tilde{A} = (\tilde{t'} \cdot G, \tilde{t'} \cdot E + z' \cdot G), \tilde{B} + e' \cdot \tilde{A} = \tilde{t'} \cdot D + z' \cdot G$$
Subtracting the equations from each other, we have
\[(e - e') \cdot \tilde{A} = ((\bar{t} - \bar{t}') \cdot G, (\bar{t} - \bar{t}') \cdot E + (z - z') \cdot G)\]
and
\[(e - e') \cdot \tilde{A} = (\bar{t} - \bar{t}') \cdot D + (z - z') G\]
Thus,
\[\tilde{A} = (e - e')^{-1} \cdot ((\bar{t} - \bar{t}') \cdot G, (\bar{t} - \bar{t}') \cdot E + (z - z') \cdot G)\]
and
\[\tilde{A} = (e - e')^{-1} \cdot (\bar{t} - \bar{t}') \cdot D + (z - z') \cdot G.\]
Taking \(a = (z - z') \cdot (e - e')^{-1}, \tilde{r} = (\bar{t} - \bar{t}') \cdot (e - e')^{-1}\) and \(\hat{r} = (\bar{t} - \bar{t}') \cdot (e - e')^{-1}\), we have that \(\hat{A} = \text{EgCom}_E(a; \tilde{r})\) and \(\hat{A} = \text{PedCom}_D(a; \hat{r})\), as required.

**Honest-verifier zero knowledge.** Given \(e\), the simulator chooses random \(z, \tilde{t}, \tilde{t} \leftarrow \mathbb{Z}_q\), computes
\[\hat{B} = \text{EgCom}_E(z; \tilde{t}) - e \cdot \tilde{A} = \text{and} \hat{B} = \text{PedCom}_D(z; \tilde{t}) - e \cdot \tilde{A},\]
and outputs \((\hat{B}, \hat{B}, z, \tilde{t}, \tilde{t})\).
Assume there exists \((a, \tilde{r}, \hat{r})\) such that \(\hat{A} = \text{EgCom}_E(a; \tilde{r})\) and \(\hat{A} = \text{PedCom}_D(a; \hat{r})\). Then
\[\hat{B} = \text{EgCom}_E(z - a \cdot e; \tilde{t} - e \cdot \tilde{r})\]
and
\[\hat{B} = \text{PedCom}_D(z - a \cdot e; \tilde{t} - e \cdot \tilde{r})\]
Since \(b = z - e \cdot a, \tilde{s} = \tilde{t} - e \cdot \tilde{r}\), and \(\tilde{s} = \tilde{t} - e \cdot \hat{r}\) are uniform in \(\mathbb{Z}_q\), we deduce that the output distribution of the simulator is identical to the honest verifier view on such input.

**A.1.4 Knowledge of ElGamal Scalar Product**

We present a zero-knowledge sigma protocol for the relation
\[\mathcal{R}_{\text{EgProdScalar}} = ((E, \tilde{A}, \hat{B}), (r, c)): \hat{B} = c \cdot \tilde{A} + \text{EgCom}_E(0; r)\]

**Protocol A.4** (sigma protocol for \(\mathcal{R}_{\text{EgProdScalar}}\)).

*Security parameter:* \(\kappa \in \mathbb{N}\).
*Common input:* \((E, \tilde{A}, \hat{B})\).
*P’s input:* \((r, c)\).
*Operation:*

1. P: Send \(\tilde{X} = g \cdot \tilde{A} + \text{EgCom}_E(0; s), \) for \(g, s \leftarrow \mathbb{Z}_q\).
2. V: Send \(e \leftarrow \{0, 1\}^\kappa\).
3. P: Send \((z = g + e \cdot c, t = s + e \cdot r)\).
   • V accepts if \(\tilde{X} + e \cdot \hat{B} = z \cdot \tilde{A} + \text{EgCom}_E(0; t)\)
Completeness. Since
\[ z \cdot \tilde{A} + \text{EgCom}_E(0; t) - e \cdot \tilde{B} = (g + e \cdot c) \cdot \tilde{A} + \text{EgCom}_E(0; s + e \cdot r) - e \cdot (c \cdot \tilde{A} + \text{EgCom}_E(0; r)) \]
\[ = (g + e \cdot c) \cdot \tilde{A} + (s + e \cdot r) \cdot (G, E) - e \cdot (c \cdot \tilde{A} + r \cdot (G, E)) \]
\[ = g \cdot \tilde{A} + s \cdot (G, E) \]
\[ = \tilde{X}, \]
\[ V \text{ always accepts in an all-honest execution.} \]

Special soundness. Assume there exists \((\tilde{X}, e, e', z, t, z', t')\) such that \(e \neq e'\) and both \((X, e, z, t)\) and \((X', e', z', t')\) are accepting transcripts. Hence,
\[ e \cdot \tilde{B} = z \cdot \tilde{A} + t \cdot \tilde{B} - \tilde{X} \]
and
\[ e' \cdot \tilde{B} = z' \cdot \tilde{A} + t' \cdot \tilde{B} - X \]
Subtracting the equations from each other, we have
\[ (e - e') \cdot \tilde{B} = (z - z') \cdot \tilde{A} + (t - t') \cdot G \]
Taking \(r = (z - z') \cdot (e - e')^{-1}\) and \(c = (t - t') \cdot (e - e')^{-1}\), we have that \(\tilde{B} = c \cdot \tilde{A} + r \cdot (G, E)\) as required.

Honest-verifier zero knowledge. Given \(e\), the simulator samples \(z, t \overset{\$}{\leftarrow} \mathbb{Z}_q\) and computes
\[ \tilde{X} = z \cdot E + z_2 \cdot \tilde{A} - e \cdot \tilde{A}' \]
If there exist \((c, r)\) such that \(\tilde{B} = c \cdot \tilde{A} + r \cdot (G, E)\), then
\[ \tilde{X} = (z - e \cdot r) \cdot (G, E) + (z_2 - e \cdot c) \cdot \tilde{A} \]
Since \(g = z - e \cdot r\) and \(s = z_2 - e \cdot d\) are uniform in \(\mathbb{Z}_q\), we deduce that the output distribution of the simulator is identical to the honest verifier view on such input.

A.1.5 Product of ElGamal Commitments

We present a zero-knowledge sigma protocol for the relation
\[ \mathcal{R}_{\text{EgProdEg}} = \{(E, \tilde{A}, \tilde{B}, \tilde{C}), (r^b, r^0) : \tilde{B} = \text{EgCom}_E(b; r^b) \land \tilde{C} = b \cdot \tilde{A} + \text{EgCom}_E(0; r^0)\} \]

Protocol A.5 (sigma protocol for \(\mathcal{R}_{\text{EgProdEg}}\)).

Security parameter: \(\kappa \in \mathbb{N}\).
Common input: \((E, \tilde{A}, \tilde{B}, \tilde{C})\).
P’s input: \((r^b, r^0, b)\).
Operation:
1. P: Send $\tilde{X} = \text{EgCom}_E(f; s^f), \tilde{Y} = f \cdot \tilde{A} + \text{EgCom}(0; s^0)$, for $f, s^f, s^0 \leftarrow \mathbb{Z}_q$.

2. V: Send $e \leftarrow \{0, 1\}^\kappa$.

3. P: Send $(z = f + e \cdot b, t_1 = s^f + e \cdot r^b, t_2 = s^0 + e \cdot r^0)$.  
   - V accepts if $\text{EgCom}_E(z; t_1) = \tilde{X} + e \cdot \tilde{B}$, and $z \cdot \tilde{A} + \text{EgCom}_E(0; t_2) = \tilde{Y} + e \cdot \tilde{C}$.

---

**Completeness.** Since 
$$\text{EgCom}_E(z; t_1) = (t_1 \cdot G, z \cdot G + t_1 \cdot D) = \tilde{X} + e \cdot \tilde{B}$$
and
$$z \cdot \tilde{A} + \text{EgCom}_E(0; t_2) = (f + e \cdot b) \cdot \tilde{A} + t_2 \cdot (G, E) = \tilde{Y} + e \cdot \tilde{C},$$
V always accepts in an all-honest execution.

**Special soundness.** Assume there exist $(\tilde{X}, \tilde{Y}, e, e', z, t_1, t_2, z', t_1', t_2')$ such that $e \neq e'$ and both $(\tilde{X}, \tilde{Y}, e, z, t_1, t_2)$ and $(\tilde{X}, \tilde{Y}, e', z', t_1', t_2')$ are accepting transcripts. Hence,
$$(t_1 \cdot G, t_1 \cdot D + z \cdot G) = \tilde{X} + e \cdot \tilde{B} \quad \text{and} \quad z \cdot \tilde{A} + t_2 \cdot (G, E) = \tilde{Y} + e \cdot \tilde{C}$$
and
$$(t_1' \cdot G, t_1' \cdot D + z' \cdot G) = \tilde{X} + e' \cdot \tilde{B} \quad \text{and} \quad z' \cdot \tilde{A} + t_2' \cdot (G, E) = \tilde{Y} + e' \cdot \tilde{C}.$$ 

Subtracting the equations from each other, we have 
$$((t_1 - t_1') \cdot G, (z - z') \cdot G + (t_1 - t_1') \cdot E) = (e - e') \cdot \tilde{B}$$
and
$$(z - z') \cdot \tilde{A} + (t_2 - t_2') \cdot (G, E) = (e - e') \cdot \tilde{C}.$$ 
Thus,
$$\tilde{B} = (e - e')^{-1} \cdot ((t_1 - t_1') \cdot G, (z - z') \cdot G + (t_1 - t_1') \cdot E)$$
and
$$\tilde{C} = (e - e')^{-1} \cdot (z - z') \cdot \tilde{A} + (t_2 - t_2') \cdot (G, E)).$$

Setting $b = (z - z') \cdot (e - e')^{-1}, r^b = (t_1 - t_1') \cdot (e - e')^{-1}$ and $r^0 = (t_2 - t_2') \cdot (e - e')^{-1}$, we have that $\tilde{B} = \text{EgCom}_E(b; r^b) \wedge \tilde{C} = b \cdot \tilde{A} + \text{EgCom}_E(0; r^0)$, as required.

**Honest-verifier zero knowledge.** Given $e$, the simulator chooses random $z, t_1, t_2 \leftarrow \mathbb{Z}_q$, computes 
$$\tilde{X} = \text{EgCom}_E(z; t_1) \cdot e \cdot \tilde{B} \quad \text{and} \quad \tilde{Y} = z \cdot \tilde{A} + \text{EgCom}_E(0; t_2) - e \cdot \tilde{C}.$$ 
and outputs $(\tilde{X}, \tilde{Y}, z, t_1, t_2)$.

Assume there exists $(r^b, r^0, b)$ such that $\tilde{B} = \text{EgCom}_E(b; r^b) \wedge \tilde{C} = b \cdot \tilde{A} + \text{EgCom}_E(0; r^0)$. Then 
$$\tilde{X} = ((t_1 - e \cdot r^b) \cdot G, (z - e \cdot b) \cdot G + (t_1 - e \cdot r^b) \cdot E) = \text{EgCom}_E(z - e \cdot b; (t_1 - e \cdot r^b))$$ and similarly 
$$\tilde{Y} = (z - e \cdot b) \cdot \tilde{A} + \text{EgCom}_E(0; t_1 - e \cdot r^0)$$ 
Since $f = z - e \cdot b$, $s^f = t_1 - e \cdot r^b, s^0 = t_2 - e \cdot r^0$ are uniform in $\mathbb{Z}_q$, we deduce that the output distribution of the simulator is identical to the honest verifier view on such input.
A.2 String Commitments

Any UC-secure commitment scheme, e.g., [Lin11; Bla+13; Fuj21], UC-realizes $F_{\text{com}}$. In the random-oracle model, $F_{\text{com}}$ can be trivially realized (with static security) by simply defining $c = \text{Com}(x, \text{sid}, i, j) = \text{SHA256}(x||\text{sid}||i||j, r)$ with $r \leftarrow \{0,1\}^κ$ and $\text{SHA256}: \{0,1\}^* \rightarrow \{0,1\}^{256}$ is the SHA256 hash function, and sending $c$ to $P_j$.

A.2.1 Committed Non-Interactive Zero Knowledge

Given non-interactive zero-knowledge proofs of knowledge, $F_{\text{com-zk}}^R$ can be securely realized by having the prover commit to the statement together with its proof, using the ideal commitment functionality $F_{\text{com}}$. 