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

Yehuda Lindell† Ariel Nof† Samuel Ranellucci‡

October 14, 2018

**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. However, despite this interest, there is still no *full* threshold solution for more than 2 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 thus supporting an honest minority) that has *practical key distribution*. This is due to the fact that all previous solutions for this utilize Paillier homomorphic encryption, and efficient distributed Paillier key generation for more than two parties is not known.

In this paper, we present the *first truly practical* full threshold ECDSA signing protocol that has both fast signing and fast key distribution. This solves a years-old open problem, and opens the door to 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 today due to the need for distributed key generation.

## 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*; cf. [3, 9, 11, 12, 17, 34, 33, 29]. In its most general form, this problem considers the

*An extended abstract of this work (by the first two authors) appeared at ACM CCS 2018. This paper includes the full proofs of security, as well as a performance improvement to the Paillier-based private multiplication.

†Dept. of Computer Science, Bar-Ilan University, Israel. lindell@biu.ac.il, nofarie@cs.biu.ac.il. Some of this work was carried out for Unbound Tech Ltd. This work was also 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.

‡Unbound Tech, Ltd. samuel.ranellucci@unboundtech.com

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 less 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 on. 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 needs to 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 past decade. However, interest has recently been renewed. This can be seen by the fact that a number of startup companies are now deploying threshold cryptography for the purpose of key protection [35, 36, 37]. 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, not supporting arbitrary and complex access structures. In addition, plain multisig solutions introduce anonymity and scalability problems (as discussed in [18, 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, DSA/ECDSA has resisted attempts at constructing efficient protocols for threshold signing. This seems to be due to the need to compute $k$ and $k^{-1}$ without knowing $k$, as discussed in detail in [28]. The first solution to overcome this difficulty in the *honest minority setting* was by Mackenzie and Reiter in [29] who use Paillier additively homomorphic encryption in order to generate a signature between *two parties*. Their protocol required heavy zero-knowledge proofs, but this was improved in [18] and later in [28]. More significant to our setting, [18, 4] show how to generalize the Mackenzie-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 [15] (albeit requiring about 40 seconds between two strong servers), it is still unknown as to whether this can be done practically for more than two parties. We remark that an extremely fast two-party ECDSA signing protocol was recently presented by [13] (the protocol of [13] is significantly faster in signing time, but has much higher bandwidth than [28]). However, it is unclear how to generalize their method to the multiparty setting. Thus, despite decades of research in threshold cryptography and secure multiparty computation, the following basic question

remains open:

> *Is it possible to construct a full-threshold protocol for multiparty ECDSA, with practical distributed key generation and signing?*

We answer this question in the affirmative.

### 1.2 Our Results

In this paper, we present the first *full threshold* ECDSA signing protocol that has *practical distributed key generation* and fast signing. We achieve this breakthrough by replacing the Paillier additively homomorphic encryption with ElGamal in-the-exponent that also supports additive homomorphism. This enables us to compute an encrypted signature in a similar way to that of [18], except that upon decryption the parties are only able to receive $s \cdot G$ (where $G$ is the generator point 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 we 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 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. First, 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. Second, Elliptic curve operations themselves are far more efficient than Paillier operations. Third, zero-knowledge proofs that are very expensive in Paillier are far more efficient in the Elliptic curve group (it is well known that zero-knowledge is easier in known-order groups). Fourth, by working in the same Elliptic curve group as the signature, all homomorphic operations are automatically modulo the required group order $q$. This removes many of the difficulties in using Paillier, that required adding randomness to enable effectively working over the integers, and then proving in zero-knowledge that the "correct amount" was added. Another issue that arises when securely computing ECDSA is how to force the parties to use the correct $k^{-1}$. We use a similar method as that of [18] which is to multiplicative mask $k$ with a random value $\rho$, and then to reveal $k \cdot \rho$. This enables each party to locally invert and obtain $k^{-1} \cdot \rho^{-1}$ which can be used in generating the signature. Once again, by working within ElGamal and not Paillier, we can achieve this far more efficiently than [18] and without the expensive Paillier-based zero-knowledge proofs that they require.

We remark that our entire protocol works over any group, 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

[1] This is called ElGamal in-the-exponent due to typical multiplicative group notation. Specifically, encryption of some $x$ using generator $g$ and public-key $h$ is carried out by computing $(g^r, h^r \cdot g^x)$. In Elliptic curve notation, this becomes $(r \cdot G, r \cdot \mathcal{P} + x \cdot G)$ for public-key $\mathcal{P}$. It is easy to see that this scheme is additively homomorphic, but that decryption only returns $x \cdot G$ but not $x$ itself. Since obtaining $x$ requires solving the discrete log problem, this is not possible except for very small values of $x$.

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 their 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 banks and other institutions to offer full cryptocurrency custody solutions to 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 singlehandedly 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.[2] 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). Thus, such solutions require complex access structures for signing. We discuss this in more detail in Section 5.4 and show that our protocol fully supports these requirements. Thus, our protocol provides the first real solution for this problem that is of very practical relevance today.

### 1.4 Concurrent Work

Concurrently to this work, [19] also present a multiparty ECDSA protocol with practical key generation. The protocols 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 that Paillier encryption is indistinguishable. In contrast, [19] 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-EC (first introduced in [28]).

## 2 Preliminaries

We denote the security parameter by $\kappa$ and the number of parties by $n$. We denote by $[n]$ the set $\{1, \ldots, n\}$.

### 2.1 The ECDSA Signing Algorithm

The ECDSA signing algorithm is defined as follows. Let $\mathbb{G}$ be an Elliptic curve group of order $q$ with base point (generator) $G$. The private key is a random value $x \leftarrow \mathbb{Z}_q$ and the public key is $Q = x \cdot G$. ECDSA signing on a message $m \in \{0,1\}^*$ is defined as follows:

1. Compute $m'$ to be the $|q|$ leftmost bits of $SHA256(m)$, where $|q|$ is the bit-length of $q$. Denote this operation by $H_q(m)$.

[2] 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.

2. Choose a random $k \leftarrow \mathbb{Z}_q^*$

3. Compute $R \leftarrow k \cdot G$. Let $R = (r_x, r_y)$.

4. Compute $r = r_x \bmod q$ and $s \leftarrow k^{-1} \cdot (m' + r \cdot x) \bmod q$.

5. Output $(r, s)$

It is a well-known fact that for every valid signature $(r, s)$, the pair $(r, -s)$ is also a valid signature. In order to make $(r, s)$ unique (which will help in formalizing security), we mandate that the "smaller" of $\{s, -s\}$ is always output (where the smaller is the value between 0 and $\frac{q-1}{2}$). We denote $(r, s) \leftarrow \mathsf{Sign}(x, m)$ to be the signing algorithm, and $\mathsf{Verify}(Q, m, (r, s))$ to be the verification procedure (that outputs 0 for "invalid" and 1 for "valid").

## 2.2 ElGamal in the Exponent

Let $\mathbb{G}$ denote a group of order $q$ where the DDH assumption is assumed to be hard, and let $G$ be a generator of the group. We will use addition as the group operation, upper-case characters for group elements, and lower-case characters for scalars in $\mathbb{Z}_q$. This is consistent with Elliptic curve notation, although all of our protocols work equivalently in finite field groups.

We utilize ElGamal decryption "in the exponent". An encryption of a value $m \in \mathbb{Z}_q$ with public key $\mathcal{P} \in \mathbb{G}$ is denoted $\mathsf{EGexpEnc}_{\mathcal{P}}(m)$, and is formally defined by

$$\mathsf{EGexpEnc}_{\mathcal{P}}(m) = (A, B) = (r \cdot G, r \cdot \mathcal{P} + m \cdot G),$$

where $r \leftarrow \mathbb{Z}_q$ is random. When we wish to specify the randomness in the encryption, we denote the above by $\mathsf{EGexpEnc}_{\mathcal{P}}(m; r)$. Note that $m$ is *not* actually in the exponent here, but this is the name used since when using multiplicative group notation (as in finite fields), a ciphertext is $(g^r, h^r \cdot g^m)$, in which case $m$ *is* in the exponent.

Observe that this encryption is additively homomorphic. Specifically, two ciphertexts $(A, B)$ and $(C, D)$ can be added by computing $(A+C, B+D)$. If $(A, B) = (r \cdot G, r \cdot \mathcal{P} + m \cdot G)$ and $(C, D) = (s \cdot G, s \cdot \mathcal{P} + m' \cdot G)$ then $(A+C, B+D) = ((r+s) \cdot G, (r+s) \cdot \mathcal{P} + (m+m') \cdot G) = \mathsf{EGexpEnc}_{\mathcal{P}}(m+m')$. In addition, multiplication of a ciphertext $(A, B) = (r \cdot G, r \cdot \mathcal{P} + m \cdot G)$ by a scalar $c$ is computed by $(c \cdot A, c \cdot B) = ((c \cdot r) \cdot G, (c \cdot r) \cdot \mathcal{P} + (c \cdot m) \cdot G) = \mathsf{EGexpEnc}(c \cdot m)$. This can be combined with rerandomization by computing $(c \cdot A + s \cdot G, c \cdot B + s \cdot P) = ((c \cdot r + s) \cdot G, (c \cdot r + s) \cdot \mathcal{P} + (c \cdot m') \cdot G)$, which is a "fresh" random encryption of $c \cdot m'$. We stress that ElGamal in-the-exponent is *not* a valid encryption scheme since decryption requires solving the discrete log problem. In particular, given $d$ where $\mathcal{P} = d \cdot G$ and $\mathsf{EGexpEnc}_{\mathcal{P}}(m)$, it is possible to efficiently compute $m \cdot G$. The next step of computing $m$ requires solving the discrete log, and so can only be achieved if $m$ is relatively small.

## 2.3 Private Multiplication

Our protocol for secure multiplication utilizes a subprotocol $\pi_{\text{mult}}^{\text{priv}}$ that computes the product of additive shares with *privacy* but *not correctness*. Specifically, we consider the multiplication functionality defined by $f((a_1, b_1), \ldots, (a_n, b_n)) = (c_1, \ldots, c_n)$, where the output values $c_1, \ldots, c_n$ are random under the constraint that $\sum_{\ell=1}^{n} c_\ell = (\sum_{\ell=1}^{n} a_\ell) \cdot (\sum_{\ell=1}^{n} b_\ell) \bmod q$, where $q$ is a large prime (say, 256 bits). We require that the protocol for computing $f$ be private in the presence of malicious

adversaries; this means that the adversary cannot learn anything more than allowed, but correctness is not guaranteed. This is actually problematic to define, since the adversary receives output, and thus saying that it learns nothing more than allowed would typically require an ideal-model type of definition. However, since the outputs of the parties are random in $\mathbb{Z}_q$ under the constraint, nothing is revealed unless the shares are reconstructed. We therefore require the following two properties (let $I \subseteq [n]$ denote the set of corrupted parties and let $J = [n] \setminus I$ denote the set of honest parties):

1. *Privacy:* For any probabilistic polynomial-time adversary *malicious* $\mathcal{A}$ running the protocol, and any two sets of inputs $\{(a_j, b_j)\}_{j \in J}$ and $\{(a'_j, b'_j)\}_{j \in J}$ for the honest parties, the view of $\mathcal{A}$ (including its input, randomness, message transcript and output shares $\{c_i\}_{i \in I}$) when executing the protocol with honest parties using inputs $\{(a_j, b_j)\}_{j \in J}$ is computationally indistinguishable from its view when executing the protocol with honest parties using inputs $\{(a'_j, b'_j)\}_{j \in J}$.

2. *Input indistinguishability:* Define an implicit-input function on the views of all parties in the execution; this function outputs the implicit inputs of the adversary so that the output of the protocol is the function being computed applied to these implicit adversary inputs and the honest parties' inputs. Then, input indistinguishability requires that for any probabilistic polynomial-time adversary *malicious* $\mathcal{A}$ running the protocol, and any two sets of inputs $\{(a_j, b_j)\}_{j \in J}$ and $\{(a'_j, b'_j)\}_{j \in J}$ for the honest parties, if the output of the function applied to $\mathcal{A}$'s implicit inputs and $\{(a_j, b_j)\}_{j \in J}$ equals the output of the function applied to $\mathcal{A}$'s implicit inputs and $\{(a'_j, b'_j)\}_{j \in J}$, then $\mathcal{A}$'s view is indistinguishable even given $c_1, \ldots, c_n$. This notion was formally defined in [30] in order to achieve concurrent security, but can also be applied to other settings.

Observe that the first property mandates privacy irrespective of what inputs are used by the parties and irrespective of $\mathcal{A}$'s strategy. However, this only holds as long as the shares are not reconstructed to obtain $\sum_{\ell=1}^{n} c_\ell$, since indistinguishability is trivially impossible in such a case. Thus, the second property guarantees that *if* two sets of inputs result in the same output, then the views of $\mathcal{A}$ even given $\sum_{\ell=1}^{n} c_\ell$ are indistinguishable. Thus, as long as the output is only reconstructed when the output is correct, the protocol guarantees full privacy. Our protocol for secure multiplication (used to compute ECDSA) will ensure this exact property; the output of the private multiplication is securely verified against encrypted and validated values, and the result is only revealed if it is correct. We stress that we require input indistinguishability of multiplication for the result of the multiplication *modulo* $q$.

We show how private multiplication can be instantiated in Section 6. One instantiation is based on oblivious transfer and has low computation cost but higher bandwidth, while our second instantiation is based on Paillier encryption and has higher computation cost but much lower bandwidth.

# 3 Definition of Security

## 3.1 The ECDSA Ideal Functionality

We show how to securely compute the functionality $\mathcal{F}_{\text{ECDSA}}$. The functionality is defined with two functions: key generation and signing. The key generation is called once, and then any arbitrary number of signing operations can be carried out with the generated key. The functionality is defined in Figure 3.1.

**FUNCTIONALITY 3.1 (The ECDSA Functionality $\mathcal{F}_{\text{ECDSA}}$)**

Functionality $\mathcal{F}_{\text{ECDSA}}$ works with parties $P_1, \ldots, P_n$, as follows:

- Upon receiving $\mathsf{KeyGen}(\mathbb{G}, G, q)$ from all parties $P_1, \ldots, P_n$, where $\mathbb{G}$ is an Elliptic-curve group of order $q$ with generator $G$:
  1. Generate an ECDSA key pair $(Q, x)$ by choosing a random $x \leftarrow \mathbb{Z}_q^*$ and computing $Q = x \cdot G$. Then, store $(\mathbb{G}, G, q, x)$.
  2. Send $Q$ to all $P_1, \ldots, P_n$.
  3. Ignore future calls to $\mathsf{KeyGen}$.
- Upon receiving $\mathsf{Sign}(sid, m)$ from all $P_1, \ldots, P_n$, if $\mathsf{KeyGen}$ was already called and $sid$ has not been previously used, compute an ECDSA signature $(r, s)$ on $m$, and send it to all $P_1, \ldots, P_n$. (Specifically, choose a random $k \leftarrow \mathbb{Z}_q^*$, compute $(r_x, r_y) = k \cdot G$, $r = r_x \bmod q$, and $s' \leftarrow k^{-1}(H_q(m) + r \cdot x)$. Then, set $s = \min\{s', q - s'\}$ so that the signature $(r, s)$ always has $s < q/2$.)

We defined $\mathcal{F}_{\text{ECDSA}}$ using Elliptic curve (additive) group notation, although all of our protocols work for *any* prime-order group.

## 3.2 Security Model

**Security in the presence of malicious adversaries.** We prove security according to the standard simulation paradigm with the real/ideal model [5, 23], in the presence of *malicious adversaries* and *static corruptions*. As is standard for the case of no honest majority, we consider security with abort meaning that a corrupted party can learn output while the honest party does not. In our definition of functionalities, we describe the instructions of the trusted party. Since we consider security with abort, the ideal-model adversary receives output first and then sends either $(\mathsf{continue}, j)$ or $(\mathsf{abort}, j)$ to the trusted party, for every $j \in [n]$ to instruct the trusted party to either deliver the output to party $P_j$ (in case of $\mathsf{continue}$) or to send $\mathsf{abort}$ to party $P_j$. This means that honest parties either receive the correct output or abort, but some honest parties may receive output while others abort. This was termed *non-unanimous abort* in [24]. As described in Section 5 (after Protocol 5.1), in this case of secure signing, it is easy to transform the protocol so that all parties receive output if any single honest party received output.

We remark that when all of the zero-knowledge proofs are UC secure [6], then our protocol can also be proven secure in this framework.

**Security, the hybrid model and composition.** We prove the security of our protocol in a hybrid model with ideal functionalities that securely compute $\mathcal{F}_{\mathsf{com}}, \mathcal{F}_{\mathsf{zk}}, \mathcal{F}_{\mathsf{com\text{-}zk}}$, defined next in Section 3.3. The soundness of working in this model is justified in [5] (for stand-alone security) and in [6] (for security under composition). Specifically, as long as subprotocols that securely compute the functionalities are used (under the definition of [5] or [6], respectively), it is guaranteed that the output of the honest and corrupted parties when using real subprotocols is indistinguishable to when calling a trusted party that computes the ideal functionalities.

### 3.3 Ideal Functionalities

As in [28] and [13], we prove the security of our protocol using the ideal zero-knowledge functionality, denoted $\mathcal{F}_{\mathsf{zk}}$, and an ideal commit-and-prove functionality, denoted $\mathcal{F}_{\mathsf{com\text{-}zk}}$. In practice, these proofs are instantiated using Fiat-Shamir on highly efficient Sigma protocols. In this section, we define these ideal functionalities, and the relations for the zero-knowledge proofs.

**The ideal commitment functionality $\mathcal{F}_{\mathsf{com}}$.** In order to realize $\mathcal{F}_{\mathsf{com\text{-}zk}}$ defined below, we use an ideal commitment functionality $\mathcal{F}_{\mathsf{com}}$, formally defined in Functionality 3.2. Any UC-secure commitment scheme fulfills $\mathcal{F}_{\mathsf{com}}$; e.g., [27, 1, 16]. In the random-oracle model, $\mathcal{F}_{\mathsf{com}}$ can be trivially realized with static security by simply defining $\mathsf{Com}(x) = H(x, r)$ where $r \leftarrow \{0,1\}^{\kappa}$ is random, and sending $\mathsf{Com}(x)$ to all parties. Since $\mathcal{F}_{\mathsf{com}}$ is defined so that all parties receive the same commitment, the value $\mathsf{Com}(x)$ needs to be broadcasted. However, as shown in [24], a simple echo-broadcast suffices here for the case of non-unanimous abort (this takes two rounds of communication). In order to ensure this, the parties send a hash of all the commitments that they received in the round after the commitments were sent. If any two parties received different commitments, then they notify all parties to abort and then halt. This adds very little complexity and ensures the same view for any committed values.

**FUNCTIONALITY 3.2 (The Commitment Functionality $\mathcal{F}_{\mathsf{com}}$)**

Functionality $\mathcal{F}_{\mathsf{com}}$ works with parties $P_1, \ldots, P_n$, as follows:

- Upon receiving $(\mathsf{commit}, sid, i, x)$ from party $P_i$ (for $i \in [n]$), record $(sid, i, x)$ and send $(\mathsf{receipt}, sid, i)$ to all $P_1, \ldots, P_n$. If some $(\mathsf{commit}, sid, i, *)$ is already stored, then ignore the message.
- Upon receiving $(\mathsf{decommit}, sid, i)$ from party $P_i$, if $(sid, i, x)$ is recorded then send $(\mathsf{decommit}, sid, i, x)$ to party $P_1, \ldots, P_n$.

**The ideal zero knowledge functionality $\mathcal{F}_{\mathsf{zk}}$.** We use the standard ideal zero-knowledge functionality defined by $((x, w), \lambda) \to (\lambda, (x, R(x, w)))$, where $\lambda$ denotes the empty string. For a relation $R$, the functionality is denoted by $\mathcal{F}_{\mathsf{zk}}^{R}$. Note that any zero-knowledge proof of knowledge fulfills the $\mathcal{F}_{\mathsf{zk}}$ functionality [25, Section 6.5.3]; non-interactive versions can be achieved in the random-oracle model via the Fiat-Shamir paradigm; see Functionality 3.3 for the formal definition.

**FUNCTIONALITY 3.3 (The Zero-Knowledge Functionality $\mathcal{F}_{\mathsf{zk}}^{R}$ for Relation $R$)**

Upon receiving $(\mathsf{prove}, sid, i, x, w)$ from a party $P_i$ (for $i \in [n]$): if $sid$ has been previously used then ignore the message. Otherwise, send $(\mathsf{proof}, sid, i, x, R(x, w))$ to all parties $P_1, \ldots, P_n$, where $R(x, w) = 1$ iff $(x, w) \in R$.

We define Sigma protocols for five languages; these can be compiled to zero-knowledge proofs of knowledge in the standard manner. Namely, for non-interactive zero-knowledge in the random-oracle model, the Fiat-Shamir paradigm [14] can be used, whereas standard techniques using efficient commitments can be used to obtain proofs in the standard model with interaction. We define the following relations:

1. *Knowledge of the discrete log of an Elliptic-curve point:* Define the relation

$$R_{DL} = \{(\mathbb{G}, G, q, \mathcal{P}, w) \mid \mathcal{P} = w \cdot G\}$$

of discrete log values (relative to the given group). We use the standard Schnorr proof for this [32]. The cost of this proof is one exponentiation for the prover and two for the verifier, and communication cost of two elements of $\mathbb{Z}_q$.

2. *Diffie-Hellman tuple of Elliptic-curve points:* Define the relation

$$R_{DH} = \{(\mathbb{G}, G, q, (A, B, C), w) \mid B = w \cdot G \ \wedge \ C = w \cdot A\}$$

of Diffie-Hellman tuples (relative to the given group). We use the well-known proof that is an extension of Schnorr's proof for discrete log. The cost of this proof is two exponentiations for the prover and four for the verifier, and communication cost of two elements of $\mathbb{Z}_q$.

3. *Rerandomization of tuple:* Define the rerandomization relation

$$R_{RE} = \{(\mathbb{G}, G, q, (\mathcal{P}, A, B, A', B'), (r, s)) \mid A' = r \cdot G + s \cdot A \ \wedge \ B' = r \cdot \mathcal{P} + s \cdot B\}.$$

Observe that if $(G, \mathcal{P}, A, B)$ is a Diffie-Hellman tuple, then $(G, \mathcal{P}, A', B')$ is a uniformly distributed and independent Diffie-Hellman tuple (with the same $\mathbb{G}, \mathcal{P}$). In contrast, if $(G, \mathcal{P}, A, B)$ is not a Diffie-Hellman tuple, then $A', B'$ are uniform and independent random group elements. See Eq. Eq. (1) in the proof of Theorem B.1 for a proof of this fact. The Sigma protocol for this relation is described in Appendix A.1; the cost of this proof is four exponentiations for the prover and six for the verifier, and communication cost of three elements of $\mathbb{Z}_q$.

4. *Knowledge of $x$ in $\mathsf{EGexpEnc}_{\mathcal{P}}(x)$:* we define the relation

$$R_{EG} = \{((\mathbb{G}, G, q, \mathcal{P}, A, B), (x; r)) \mid (A, B) = \mathsf{EGexpEnc}_{\mathcal{P}}(x; r)\}$$

of the encrypted value in an ElGamal encryption-in-the-exponent ciphertext. Note that this is very different from knowing a regular ElGamal plaintext since here $x$ is "in the exponent", and the knowledge extractor must be able to extract $x$ itself (and not just $x \cdot G$). The Sigma protocol for this relation is described in Appendix A.2; the cost of the proof is three exponentiations for the prover and five for the verifier, and communication cost of three elements of $\mathbb{Z}_q$.

5. *Scalar product with encrypted values:* we define the relation

$$\begin{aligned} R_{\text{prod}} = \{((\mathbb{G}, G, q, \mathcal{P}, A, B, C, D, E, F), (t, r, y)) \mid \\ (C, D) = \mathsf{EGexpEnc}_P(y; t) \ \wedge \ E = y \cdot A + r \cdot G \ \wedge \ F = y \cdot B + r \cdot \mathcal{P}\} \end{aligned}$$

which means that $(E, F)$ is generated by multiplying $(A, B)$ by the scalar $y$ which is encrypted-in-the-exponent in $(C, D)$, and then rerandomizing the result. The Sigma protocol for this relation is described in Appendix A.3; the cost of this proof is eleven exponentiations for the prover and twelve for the verifier, and communication cost of six elements of $\mathbb{Z}_q$.

For clarity, we remove the $(\mathbb{G}, G, q)$ part from the input to the zero-knowledge proofs below, with the understanding that these parameters are fixed throughout.

**The committed non-interactive zero knowledge functionality $\mathcal{F}_{\mathsf{com\text{-}zk}}$.** In our protocol, we will have parties send commitments to a statement together with a non-interactive zero-knowledge proof of the statement. As in [28], we model this formally via a commit-zk functionality, denoted $\mathcal{F}_{\mathsf{com\text{-}zk}}$, defined in Functionality 3.4. Given non-interactive zero-knowledge proofs of knowledge, this functionality is securely realized by just having the prover commit to the statement together with its proof, using the ideal commitment functionality $\mathcal{F}_{\mathsf{com}}$. As in $\mathcal{F}_{\mathsf{com}}$, consistency of views is validated by all parties sending a hash of the commitments that they received.

**FUNCTIONALITY 3.4 (The Committed NIZK Functionality $\mathcal{F}^{R}_{\mathsf{com\text{-}zk}}$ for Relation $R$)**

Functionality $\mathcal{F}_{\mathsf{com\text{-}zk}}$ works with parties $P_1, \ldots, P_n$, as follows:

- Upon receiving $(\mathsf{ComProve}, sid, i, x, w)$ from a party $P_i$ (for $i \in [n]$): if $sid$ has been previously used then ignore the message. Otherwise, store $(sid, i, x)$ and send $(\mathsf{ProofReceipt}, sid, i)$ to $P_1, \ldots, P_n$.
- Upon receiving $(\mathsf{DecomProof}, sid)$ from a party $P_i$ (for $i \in [n]$): if $(sid, i, x)$ has been stored then send $(\mathsf{DecomProof}, sid, i, x, R(x, w))$ to $P_1, \ldots, P_n$.

# 4 Secure Multiplication – $\mathcal{F}_{\text{mult}}$

## 4.1 Functionality Definition

A basic multiplication functionality (from additive shares) can be defined by having each party $P_i$ provide $a_i$ and $b_i$ for input, and then returning $c_i$ to party $P_i$, where $c_1, \ldots, c_n$ are random under the constraint that $\sum_{\ell=1}^{n} c_\ell = (\sum_{\ell=1}^{n} a_\ell) \cdot (\sum_{\ell=1}^{n} b_\ell) \bmod q$

We will actually need an extended version of this multiplication functionality, where the value $a \cdot G$ is also returned to the parties, where $a = \sum_{\ell=1}^{n} a_\ell$ ($(\mathbb{G}, G, q)$ denotes the description of a group of order $q$, with generator $G$). Although this looks "out of place", this additional value can be computed efficiently while computing $\mathcal{F}_{\text{mult}}$. For this reason, we combine them together. In addition, in our protocol, we need to enable multiplication with the same value more than once. This is achieved by having an "input" command, and then enabling multiplications between inputs. Finally, we define the functionality for *random input shares only* for the honest parties, whereas corrupted parties can choose their own shares.[3] This is needed in our proof of security (see Game 2 in the proof of Theorem B.1) and is anyway the way the functionality is used in order to securely compute ECDSA. Finally, we add a local affine transformation on an encrypted/shared input, that is needed for computing ECDSA. This is a local operation only, and so is formalized by the honest parties providing these values. See Functionality 4.1 for the specification of $\mathcal{F}_{\text{mult}}$ that we use in our protocol.

Observe that $\mathcal{F}_{\text{mult}}$ as described here is not a "standard" multiplication functionality and is tailored to what we need for securely computing $\mathcal{F}_{\text{ECDSA}}$. Arguably, as such, one should call it an *ECDSA helper functionality*. However, since it's main operation (and most difficult one) is multiplications, we call it *extended multiplication*.

[3] As such, the functionality is "corruption aware", meaning that it behaves differently for honest parties and corrupted parties. This is standard in ideal-model formalizations.

**FUNCTIONALITY 4.1 (The Extended Multiplication Functionality $\mathcal{F}_{\text{mult}}$)**

The functionality $\mathcal{F}_{\text{mult}}$ is given the set of indices of corrupted parties $\mathcal{C} \subseteq [n]$, and works with parties $P_1, \ldots, P_n$ as follows:

- Upon receiving $(\mathsf{init}, \mathbb{G}, G, q)$ from all parties, $\mathcal{F}_{\text{mult}}$ stores $(\mathbb{G}, G, q)$. If some $(\mathbb{G}, G, q)$ has already been stored, then ignore the message.
- Upon receiving $(\mathsf{input}, sid, a_i)$ from a party $P_i$ with $i \in \mathcal{C}$, if no value $(sid, i, \cdot)$ value is stored, then $\mathcal{F}_{\text{mult}}$ stores $(sid, i, a_i)$. Else, the message is ignored.
- Upon receiving $(\mathsf{input}, sid)$ from a party $P_i$ with $i \notin \mathcal{C}$, if no value $(sid, i, \cdot)$ value is stored, then $\mathcal{F}_{\text{mult}}$ chooses a random $a_i \leftarrow \mathbb{Z}_q$, returns $(\mathsf{input}, sid, a_i)$ to $P_i$, and stores $(sid, i, a_i)$. Else, the message is ignored.
- If some $(sid, i, a_i)$ has been stored for all $i \in [n]$ then $\mathcal{F}_{\text{mult}}$ computes $a = \sum_{\ell=1}^{n} a_\ell \bmod q$, stores $(sid, a)$ and sends $(\mathsf{input}, sid)$ to all parties.
- Upon receiving $(\mathsf{mult}, sid_1, sid_2)$ from all parties, $\mathcal{F}_{\text{mult}}$ checks that some $(sid_1, a)$ and $(sid_2, b)$ values have been stored. If yes, then $\mathcal{F}_{\text{mult}}$ sets $c = a \cdot b \bmod q$ and sends $(\mathsf{mult\text{-}out}, sid_1, sid_2, c)$ to all parties.
- Upon receiving $(\mathsf{affine}, sid_1, sid_2, x, y)$ from the honest parties with $x, y \in \mathbb{Z}_q$, functionality $\mathcal{F}_{\text{mult}}$ checks that some $(sid_1, a)$ has been stored. If yes, $\mathcal{F}_{\text{mult}}$ computes $b = a \cdot x + y \bmod q$, stores $(sid_2, b)$, and sends $(\mathsf{affine}, sid_1, sid_2, x, y)$ to $\mathcal{A}$.
- Upon receiving $(\mathsf{element\text{-}out}, sid)$ from all parties, functionality $\mathcal{F}_{\text{mult}}$ checks that some $(sid, a)$ has been stored. If yes, $\mathcal{F}_{\text{mult}}$ computes $A = a \cdot G$ and sends $(\mathsf{element\text{-}out}, sid, A)$ to all parties. In addition, $\mathcal{F}_{\text{mult}}$ sends $A_1, \ldots, A_n$ to the ideal adversary, where $A_i = a_i \cdot G$ (with $a_i$ as received in $\mathsf{input}$ for $sid$).

## 4.2 Checking Diffie-Hellman Tuples

In order to securely compute $\mathcal{F}_{\text{mult}}$, one step involves the parties securely checking that a tuple $(G, \mathcal{P}, U, V)$ is a Diffie-Hellman tuple or not. As part of the initialization of $\mathcal{F}_{\text{mult}}$, each party has a share $d_i$ such that $\mathcal{P} = d \cdot G$, with $d = \sum_{\ell=1}^{n} d_\ell$. Thus, $(G, \mathcal{P}, U, V)$ is a Diffie-Hellman tuple if and only if $V = d \cdot U$. This can be verified by the parties each sending $U_i = d_i \cdot U$ to each other, and then all parties simply check that $V = \sum_{\ell=1}^{n} U_i$. However, this will not be secure since we require that nothing be learned if the input is not a Diffie-Hellman tuple, and $\sum_{\ell=1}^{n} U_i$ reveals the value $T$ such that $(G, \mathcal{P}, U, V - T)$ is a Diffie-Hellman tuple. Thus, the parties first each rerandomize the tuple and then continue as above. This is rather standard, and the full specification is therefore deferred to Section 7. We remark that as part of the protocol, the parties need to verify values using $\mathcal{P}_1, \ldots, \mathcal{P}_n$ such that $\mathcal{P}_i = d_i \cdot G$ for every $i \in [n]$. (In our multiplication protocol, these values are learned in the initialization phase.) We define the ideal functionality $\mathcal{F}_{\text{checkDH}}$ in Functionality 4.2, and show how to securely compute it in Section 7. The cost of securely computing $\mathcal{F}_{\text{checkDH}}$ is $11 + 10(n-1)$ group exponentiations per party, each party sending 7 group elements (or equivalent) to each other party, and 3 rounds of communication.

**FUNCTIONALITY 4.2 ($\mathcal{F}_{\text{checkDH}}$ - Check DH Tuple)**

$\mathcal{F}_{\text{checkDH}}$ runs with parties $P_1, \ldots, P_n$, as follows:

1. Upon receiving $(\mathsf{init}, i, \mathcal{P}_i, d_i)$ from party $P_i$, $\mathcal{F}_{\text{checkDH}}$ verifies that $\mathcal{P}_i = d_i \cdot G$. If not, it sends abort to all parties. Otherwise, it stores $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ and sends $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ to all parties.
2. Upon receiving $(\mathsf{check}, i, U, V, d_i)$ from party $P_i$ for every $i \in [n]$, $\mathcal{F}_{\text{checkDH}}$ verifies that $\mathcal{P}_i = d_i \cdot G$ for all $i \in [n]$ and that $V = d \cdot U$ where $d = \sum_{\ell=1}^{n} d_\ell \bmod q$. If all conditions were satisfied, $\mathcal{F}_{\text{checkDH}}$ sends accept to all the parties. Otherwise, it sends reject to all parties.

### 4.3 Securely Computing $\mathcal{F}_{\text{mult}}$

We describe separate subprotocols for init, input, element-out, affine and mult. We remark that init must be called first, and all other calls to $\mathcal{F}_{\text{mult}}$ are ignored until init is completed. In our description of the protocol, each round is structured as follows: **(a)** each party receives the messages sent by the other parties and/or ideal functionalities in the previous round, **(b)** each party carries out local computation, and **(c)** each party sends the messages from this round to other parties and/or to ideal functionalities.

**Init:** The init procedure is used to generate an ElGamal (in-the-exponent) public key $\mathcal{P}$, with each party holding shares $d_1, \ldots, d_n$ of the private key (meaning that $\sum_{\ell=1}^{n} d_\ell = d$ where $\mathcal{P} = d \cdot G$). In addition to each party storing its share $d_i$ of the private key, and the public key $\mathcal{P}$, the parties also store all values $\mathcal{P}_1, \ldots, \mathcal{P}_n$ where $\mathcal{P}_i = d_i \cdot G$. These latter values are used inside the protocol for $\mathcal{F}_{\text{checkDH}}$ in order to enforce the parties to use their correct value $d_i$. Specifically, if a party is supposed to compute $Y = d_i \cdot X$ for some value $X$, then it can prove that it behaved honestly by proving that $(\mathbb{G}, X, \mathcal{P}_i, Y)$ is a Diffie-Hellman tuple (since $\mathcal{P}_i = d_i \cdot G$ and $Y = d_i \cdot X$). Thus, the parties need to have stored $\mathcal{P}_i$.

The protocol for init is basically a type of simulatable coin tossing. In the first round, each party generates a value $\mathcal{P}_i = d_i \cdot G$ and prepares a zero-knowledge proof of knowledge of $d_i$, and commits to the value and proof. Then, in the second round, the parties decommit. Finally, $\mathcal{P}$ is set to be the sum of all $\mathcal{P}_i$. This is simulatable in the ideal commitment hybrid model using standard simulation techniques. See Protocol 4.3 for a full specification.

**Input:** The input procedure is used to generate an ElGamal in-the-exponent encryption of $a = \sum_{\ell=1}^{n} a_i \bmod q$, where each $P_i$ chooses $a_i$; the honest parties choose $a_i$ randomly whereas the corrupted parties can choose $a_i$ as they wish (as prescribed in $\mathcal{F}_{\text{mult}}$). In order to prevent the corrupted parties from making their $a_i$ values dependent on the honest parties' values (and thus influencing the result $a$), each party encrypts its $a_i$ and provides a zero-knowledge proof of knowledge of the encrypted value. The parties generate an encryption of $a$ by using the additively homomorphic properties of ElGamal in the exponent. In addition to storing the final ciphertext, they store each party's encryption and this is used to enforce correct behavior later. See Protocol 4.4 for a full specification.

**PROTOCOL 4.3 (Initialization Subprotocol of $\mathcal{F}_{\text{mult}}$)**

Upon joint input $(\mathsf{init}, \mathbb{G}, G, q)$, number of parties $n$, and a unique session identifier $sid$, the parties run any initialization steps needed for $\pi^{\text{priv}}_{\text{mult}}$ (e.g., base oblivious transfers). In parallel, each party $P_i$ works as follows:

1. *Round 1:* Party $P_i$ chooses a random $d_i \leftarrow \mathbb{Z}_q$, and computes $\mathcal{P}_i = d_i \cdot G$. Then, $P_i$ sends $(\mathsf{ComProve}, init, i, \mathcal{P}_i, d_i)$ to $\mathcal{F}^{R_{DL}}_{\mathsf{com\text{-}zk}}$ (i.e., $P_i$ sends a commitment to $\mathcal{P}_i$ and a POK of its discrete log).
2. *Round 2:* Upon receiving $(\mathsf{ProofReceipt}, init, j)$ from $\mathcal{F}^{R_{DL}}_{\mathsf{com\text{-}zk}}$ for every $j \in [n]$, party $P_i$ sends $(\mathsf{DecomProof}, init, i)$ to $\mathcal{F}^{R_{DL}}_{\mathsf{com\text{-}zk}}$. In addition, $P_i$ sends $(\mathsf{init}, i, \mathcal{P}_i, d_i)$ to $\mathcal{F}_{\text{checkDH}}$.
3. *Output:* $P_i$ receives $(\mathsf{DecomProof}, init, j, \mathcal{P}_j, \beta_j)$ from $\mathcal{F}^{R_{DL}}_{\mathsf{com\text{-}zk}}$ for every $j \in [n]$, and receives $(\mathcal{P}_1, \ldots, \mathcal{P}_n)$ from $\mathcal{F}_{\text{checkDH}}$. If some $\beta_j = 0$ (meaning that a proof is not valid), or the $\mathcal{P}_j$ values received from $\mathcal{F}^{R_{DL}}_{\mathsf{com\text{-}zk}}$ do not match the $\mathcal{P}_j$ values received from $\mathcal{F}_{\text{checkDH}}$ then it aborts. Otherwise, $P_i$ locally computes the ElGamal public-key $\mathcal{P} = \sum_{\ell=1}^{n} \mathcal{P}_\ell$ and stores $(d_i, \mathcal{P}, \{\mathcal{P}_\ell\}_{\ell=1}^{n})$.

**PROTOCOL 4.4 (Input Subprotocol of $\mathcal{F}_{\text{mult}}$)**

Upon input $(\mathsf{input}, sid, a_i)$ to $P_i$ for all $i \in [n]$, each party $P_i$ works as follows:

1. *Round 1:* $P_i$ chooses a random $s_i \leftarrow \mathbb{Z}_q$ and computes $(U_i, V_i) = \mathsf{EGexpEnc}_{\mathcal{P}}(a_i; s_i)$. Then, $P_i$ sends $(\mathsf{proof}, sid, i, (\mathcal{P}, U_i, V_i), (a_i, s_i))$ to $\mathcal{F}^{R_{EG}}_{\mathsf{zk}}$.
2. *Output:* $P_i$ receives $(\mathsf{proof}, sid, j, (\mathcal{P}, U_j, V_j), \beta_j)$ from $\mathcal{F}^{R_{EG}}_{\mathsf{zk}}$, for every $j \in [n] \setminus \{i\}$. If some $\beta_j = 0$, then $P_i$ aborts. Otherwise, $P_i$ computes $U = \sum_{\ell=1}^{n} U_\ell$ and $V = \sum_{\ell=1}^{n} V_\ell$ and stores $(sid, (U, V), a_i, s_i, \{(U_\ell, V_\ell)\}_{\ell=1}^{n})$.
   $P_i$ outputs $(\mathsf{input}, sid)$.

In the point-to-point model, it is necessary for the parties to ensure that they all received the same input. This is achieved by sending hashes of all received ciphertexts $(U_1, V_1), \ldots, (U_n, V_n)$ after receiving output. This can be done in parallel to the continuation of the protocol, and we therefore do not add it as a separate round.

**Get Element:** As defined in the $\mathcal{F}_{\text{mult}}$ ideal functionality, the $\mathsf{element\text{-}out}$ procedure is used by the parties to obtain $A = a \cdot G$ for a value $a$ that was input. Recall that in order to get input $a$, each party $P_i$ encrypted $a_i$ and the sum of all of these values equal $a$. As such, all that is needed here is for each party $P_i$ to provide $A_i = a_i \cdot G$ and prove that this is generated using the same value $a_i$ as was encrypted in the input phase. Recall that the parties store each party's encryption $(U_i, V_i)$ of $a_i$, meaning that $U_i = s_i \cdot G$ and $V_i = s_i \cdot \mathcal{P} + a_i \cdot G$. However, this means that $(G, \mathcal{P}, U_j, V_j - A_j) = (G, \mathcal{P}, s_i \cdot G, s_i \cdot \mathcal{P})$ and so is a Diffie-Hellman tuple. Thus, $P_i$ can provide $A_i$ and prove that it provided the correct $A_i$ very efficiently by sending a zero-knowledge proof of a Diffie-Hellman tuple. The final value $A$ is obtained easily by summing all $A_i$ values. See Protocol 4.5 for a full specification.

**PROTOCOL 4.5 (Element-Out Subprotocol of $\mathcal{F}_{\text{mult}}$)**

Upon input $(\mathsf{element\text{-}out}, sid)$, if $P_i$ has some $(sid, (U, V), a_i, s_i, \{(U_\ell, V_\ell)\}_{\ell=1}^n)$ stored, then it proceeds as follows (otherwise it ignores the input):

1. *Round 1:* $P_i$ computes $A_i = a_i \cdot G$, and sends $A_i$ to $P_j$ for all $j \in [n] \setminus \{i\}$, and sends $(\mathsf{proof}, sid, i, (G, \mathcal{P}, U_i, V_i - A_i), s_i)$ to $\mathcal{F}_{\mathsf{zk}}^{R_{DH}}$.
2. *Output:* $P_i$ receives $(\mathsf{proof}, sid, j, (G, \mathcal{P}, U_j, V_j - A_j), \beta_j)$ from $\mathcal{F}_{\mathsf{zk}}^{R_{DH}}$, for all $j \in [n] \setminus \{i\}$. If some $\beta_j = 0$, then $P_i$ aborts. Otherwise, $P_i$ computes $A = \sum_{\ell=1}^n A_\ell$. Then, $P_i$ outputs $(\mathsf{element\text{-}out}, sid, A)$.

**Affine:** The affine procedure merely involves scalar multiplication and scalar addition to a given ciphertext. Since ElGamal in-the-exponent is additively homomorphic, these operations can be easily carried out on any value that was input. We remark that since the parties store their local shares and randomness in all input values, and the encryptions of all the parties' shares, these must also be adjusted. However, this is also straightforward using the same homomorphic operations. See Protocol 4.6 for a full specification.

**PROTOCOL 4.6 (Affine Subprotocol of $\mathcal{F}_{\text{mult}}$)**

Upon input $(\mathsf{affine}, sid_1, sid_2, x, y)$, if $P_i$ has some $(sid_1, (U, V), a_i, s_i, \{(U_\ell, V_\ell)\}_{\ell=1}^n)$ stored and $sid_2$ has not yet been used, then it proceeds as follows (otherwise it ignores the input):

1. $P_i$ locally computes $U' = x \cdot U$ and $V' = x \cdot V + y \cdot G$, for all $\ell \in [n]$ it computes $U'_\ell = x \cdot U_\ell$ and $V'_\ell = x \cdot V_\ell + \frac{y}{n} \cdot G$. Then, $P_i$ stores $\left(sid_2, (U', V'), a_i \cdot x + \frac{y}{n}, x \cdot s_i, \{(U'_\ell, V'_\ell)\}_{\ell=1}^n\right)$.

**Mult:** The main procedure of $\mathcal{F}_{\text{mult}}$ is the multiplication procedure, and this is also the most involved. The idea behind the protocol is as follows. From the input procedure, each party holds and encryption $(X, Y)$ of $b$, along with encryptions $(U_1, V_1), \ldots, (U_n, V_n)$ of $a_1, \ldots, a_n$, respectively. In addition, each party knows its own share $a_i$. As a result, it is possible for each party $P_i$ to use scalar multiplication on ElGamal in-the-exponent ciphertexts in order to generate an encryption of $a_i \cdot b$. In addition, each party proves in zero-knowledge that the ciphertext was generated in this way. This proof can be carried since each party already holds an encryption of $a_i$ and $b$, and so this is a well-defined NP-statement. In addition, due to the clean properties of ElGamal, the proof is very efficient; see Section A.3. By summing all of these resulting ciphertexts, the parties obtain an encryption of $a \cdot b \bmod q$ that is *provably* $a \cdot b$ and nothing else (due to the zero-knowledge proofs). Unfortunately, since the encryption is ElGamal in-the-exponent, the parties cannot decrypt, since this would yield $(a \cdot b) \cdot G$ and not $a \cdot b \bmod q$. Thus, in parallel to the above, the parties run a private multiplication subprotocol $\pi_{\text{mult}}^{\text{priv}}$ that is not necessarily correct. In addition, nothing forces the parties from inputting the correct $a_i, b_i$ values. However, given that they hold a proven encryption of $a \cdot b$, and now hold shares $c_1, \ldots, c_n$ that are supposed to sum to $a \cdot b$, they can reveal these shares to each other and verify their correctness against the proven encrypted value. *This is the main observation that enables us to replace Paillier encryption used in previous protocols with ElGamal in the exponent.* The actual implementation of this is more difficult since the parties cannot reveal $c_1, \ldots, c_n$ until *after* they have verified that the values are correct, or else this can leak

private information of the honest parties. This therefore requires a method of verifying equality without revealing anything.

**PROTOCOL 4.7 (Mult Subprotocol of $\mathcal{F}_{\text{mult}}$)**

Upon input $(\mathsf{mult}, sid_1, sid_2)$, if $P_i$ has some $(sid_1, (U, V), a_i, s_i, \{(U_\ell, V_\ell)\}_{\ell=1}^n)$ and some $(sid_2, (X, Y), b_i, t_i, \{(X_\ell, Y_\ell)\}_{\ell=1}^n)$ stored, then it defines $sid = sid_1 \| sid_2$ and proceeds as follows (otherwise it ignores the input):

0. In parallel to the below, the parties run a private multiplication protocol $\pi_{\text{mult}}^{\text{priv}}$. Party $P_i$ inputs $(a_i, b_i)$; denote $P_i$'s output by $c_i$. (In fact, this can even be run in parallel to $(\mathsf{input}, sid_1, a_i)$ and/or $(\mathsf{input}, sid_2, b_i)$.)

1. *Round 1:*

   (a) $P_i$ chooses a random $s_i'$ and computes

   $$(E_i, F_i) = (a_i \cdot X + s_i' \cdot G, a_i \cdot Y + s_i' \cdot \mathcal{P}).$$

   (b) $P_i$ sends $(\mathsf{proof}, sid, i, (\mathcal{P}, X, Y, U_i, V_i, E_i, F_i), (s_i, s_i', a_i))$ to $\mathcal{F}_{\mathsf{zk}}^{R_{\text{prod}}}$, and sends $(E_i, F_i)$ to all parties.

2. *Round 2:*

   (a) $P_i$ receives $(\mathsf{proof}, sid, j, (\mathcal{P}, X, Y, U_j, V_j, E_j, F_j), \beta_j)$ from $\mathcal{F}_{\mathsf{zk}}^{R_{\text{prod}}}$, for all $j \in [n] \setminus \{i\}$. If some $\beta_j = 0$ (meaning that a proof is not valid), then it aborts.

   (b) $P_i$ sets $(E, F) = (\sum_{\ell=1}^n E_\ell, \sum_{\ell=1}^n F_\ell)$.

   (c) $P_i$ chooses a random $\hat{s}_i \leftarrow \mathbb{Z}_q$ and computes $(A_i, B_i) = \mathsf{EGexpEnc}_{\mathcal{P}}(c_i; \hat{s}_i)$, where $c_i$ is its output from $\pi_{\text{mult}}^{\text{priv}}$.

   (d) $P_i$ sends $(\mathsf{proof}, sid, i, (\mathcal{P}, A_i, B_i), (c_i, \hat{s}_i))$ to $\mathcal{F}_{\mathsf{zk}}^{R_{EG}}$, and sends $(A_i, B_i)$ to all parties.

3. *Round 3:*

   (a) $P_i$ receives $(\mathsf{proof}, sid, j, (\mathcal{P}, A_j, B_j), \beta_j')$ from $\mathcal{F}_{\mathsf{zk}}^{R_{EG}}$, and $(A_j, B_j)$ from $P_j$, for every $j \in [n] \setminus \{i\}$. If some $\beta_j' = 0$, then $P_i$ aborts.

   (b) $P_i$ computes $A = E - \sum_{\ell=1}^n A_\ell$ and $B = F - \sum_{\ell=1}^n B_\ell$.

   (c) $P_i$ sends $(\mathsf{check}, i, A, B, d_i)$ to $\mathcal{F}_{\text{checkDH}}$.

4. *Round 4:*

   (a) If $P_i$ receives $\mathsf{reject}$ from $\mathcal{F}_{\text{checkDH}}$, then it aborts.

   (b) $P_i$ sends $(\mathsf{proof}, sid, i, (\mathcal{P}, A_i, B_i - c_i \cdot G), \hat{s}_i)$ to $\mathcal{F}_{\mathsf{zk}}^{R_{DH}}$, and sends $c_i$ to all parties.

5. *Output:* $P_i$ receives $(\mathsf{proof}, sid, j, (\mathcal{P}, A_j, B_j - c_j \cdot G), \beta_j'')$ from $\mathcal{F}_{\mathsf{zk}}^{R_{DH}}$, and $c_j$ from $P_j$, for every $j \in [n] \setminus \{i\}$. If some $\beta_j'' = 0$ (where $P_i$ verifies the proof using the $c_j$ value received here, and $A_j, B_j$ from Step 3a), then $P_i$ aborts. Otherwise, $P_i$ computes $c = \sum_{\ell=1}^n c_\ell \bmod q$ and outputs $c$.

We achieve this by having each party encrypt its $c_i$ value; the parties then sum these encryptions together and subtract the result from the proven encryption of $a \cdot b$. If the $c_i$ values were correct, then this would become an encryption of 0; stated differently, the result would be a Diffie-Hellman

tuple. Thus, the parties run a subprotocol that checks if a tuple is a Diffie-Hellman tuple without revealing *anything else*. Informally, this is achieved by each party rerandomizing the tuple so that if it is a Diffie-Hellman then it remains one, but if it is not then it becomes purely random. This subprotocol is formalized in the $\mathcal{F}_{\text{checkDH}}$ functionality and presented in Section 7. Finally, if the check passes, then the parties can send $c_i$ to each other, sum the result and output it. Of course, this last step must also be verified, but this is easy to do since the parties already provided encryptions of $c_i$ in order to check the result, and they can therefore prove that the $c_i$ provided is the one that they previously encrypted.

The full proof of security of Protocols 4.3–4.7 is presented in Appendix B and follows the intuition given above.

# 5 Securely Computing ECDSA

In this section, we present our protocol for distributed ECDSA signing. We separately describe the key generation phase (which is run once) and the signing phase (which is run multiple times). In the description of the protocol, we denote by $P_i$ the party carrying out the instructions, by $P_j$ the other parties, and we use $\ell$ as a running index from 1 to $n$.

## 5.1 The Protocol for $\mathcal{F}_{\text{ECDSA}}$

Given $\mathcal{F}_{\text{mult}}$, it is very easy to construct a protocol for securely computing $\mathcal{F}_{\text{ECDSA}}$. In particular, $\mathcal{F}_{\text{mult}}$ provides the ability to securely multiply values together. Thus, it is possible for the parties to choose random $x_i, k_i$ values (to define $x = \sum_{\ell=1}^{n} x_i$ and $k = \sum_{\ell=1}^{n} k_i$) and use $\mathcal{F}_{\text{mult}}$ to obtain $R = k \cdot G$ (using element-out), to compute $H(m) + r \cdot x \bmod q$ using affine, and finally to multiply this with $k$ to obtain $k \cdot (H(m) + r \cdot x) \bmod q$. (Note that all operations are modulo $q$ already and so all computations are correct.) However, this is *not* the computation needed! Rather, ECDSA signing requires computing $\boldsymbol{k^{-1}} \cdot (H(m) + r \cdot x)$, with $k$-inverse. Note that $\mathcal{F}_{\text{mult}}$ does not enable securely inverting an element that was input, and this is not an operation that is typically efficient in MPC. We overcome this problem by having the parties input an addition *random masking* element $\rho$ and then use mult to reveal $\tau = \rho \cdot k$. Given this value in the clear, each party can locally compute $\tau^{-1} = \rho^{-1} \cdot k^{-1}$. In addition, the parties can use mult a second time to compute $\beta = \rho \cdot (H(m) + r \cdot x)$. The key observation is that the product of these values $\beta$ and $\tau^{-1}$ (which party can locally compute) is exactly $\rho^{-1} \cdot k^{-1} \cdot \rho \cdot (H(m) + r \cdot x) = k^{-1} \cdot (H(m) + r \cdot x)$ which is the $s$-part of a valid signature. By using $\mathcal{F}_{\text{mult}}$, all of the above operations are guaranteed to be correct, and so the adversary cannot cheat. In addition, since $\rho$ is random, the values $\tau$ and $\beta$ reveal nothing more than the signature; as we will show, they are random values under the constraint that $\frac{\beta}{\tau}$ equals the valid signature. This can therefore be simulated.

We remark that key generation here merely involves running init for $\mathcal{F}_{\text{mult}}$ and input for values $x_1, \ldots, x_n$ in order to define a random $x$ that is the ECDSA private key. By calling element-out, the parties also obtain $Q = x \cdot G$ which is the ECDSA public key. As such, distributed key generation is very efficient, and scales easily to a large number of parties.

Observe also that all operations in $\mathcal{F}_{\text{mult}}$ are modulo $q$, where $q$ is the order of the ECDSA group itself, by the fact that we use ElGamal in-the-exponent over the same Elliptic curve as defined for the signing algorithm. This significantly simplifies the protocol. See Protocol 5.1 for a full specification.

**PROTOCOL 5.1 (Securely Computing $\mathcal{F}_{\text{ECDSA}}$)**

**Auxiliary input:** Each party has the description $(\mathbb{G}, G, q)$ of a group, and the number of parties $n$.

**Key generation:** Upon input $\mathsf{KeyGen}(\mathbb{G}, G, q)$, each party $P_i$ works as follows:

1. $P_i$ sends $(\mathsf{init}, \mathbb{G}, G, q)$ to $\mathcal{F}_{\text{mult}}$ to run the initialization phase.
2. $P_i$ sends $(\mathsf{input}, sid_{\text{gen}})$ to $\mathcal{F}_{\text{mult}}$, and receives back $(\mathsf{input}, sid_{\text{gen}}, x_i)$. (Denote $x = \sum_{\ell=1}^{n} x_\ell$ and $Q = x \cdot G$.)
3. $P_i$ waits to receive $(\mathsf{input}, 0)$ to $\mathcal{F}_{\text{mult}}$.
4. $P_i$ sends $(\mathsf{element\text{-}out}, 0)$ to $\mathcal{F}_{\text{mult}}$.
5. $P_i$ receives $(\mathsf{element\text{-}out}, 0, Q)$ from $\mathcal{F}_{\text{mult}}$.
6. *Output:* $P_i$ locally stores $Q$ as the ECDSA public-key.

**Signing:** Upon input $\mathsf{Sign}(sid, m)$, where $sid$ is a unique session id $sid$, each party $P_i$ works as follows:

1. $P_i$ sends $(\mathsf{input}, sid\|1)$ and $(\mathsf{input}, sid\|2)$ to $\mathcal{F}_{\text{mult}}$, and receives back $(\mathsf{input}, sid\|1, k_i)$ and $(\mathsf{input}, sid\|2, \rho_i)$. (Denote $k = \sum_{\ell=1}^{n} k_\ell$ and $\rho = \sum_{\ell=1}^{n} \rho_\ell$.)
2. After receiving $(\mathsf{input}, sid\|1)$ and $(\mathsf{input}, sid\|2)$ from $\mathcal{F}_{\text{mult}}$, $P_i$ sends $(\mathsf{mult}, sid\|1, sid\|2)$ and $(\mathsf{element\text{-}out}, sid\|1)$ to $\mathcal{F}_{\text{mult}}$.
3. $P_i$ receives $(\mathsf{mult\text{-}out}, sid\|1, sid\|2, \tau)$ and $(\mathsf{element\text{-}out}, sid\|1, R)$ from $\mathcal{F}_{\text{mult}}$ (note that $\tau = k \cdot \rho$ and $R = k \cdot G$).
4. $P_i$ computes $R = (r_x, r_y)$ and $r = r_x \bmod q$.
5. $P_i$ sends $(\mathsf{affine}, 0, sid\|3, r, m')$ to $\mathcal{F}_{\text{mult}}$ (recall that identifier 0 is associated with the private-key $x$, and thus $sid\|3$ will be associated with $m' + x \cdot r \bmod q$).
6. $P_i$ sends $(\mathsf{mult}, sid\|2, sid\|3)$ to $\mathcal{F}_{\text{mult}}$.
7. $P_i$ receives $(\mathsf{mult\text{-}out}, sid\|2, sid\|3, \beta)$ from $\mathcal{F}_{\text{mult}}$ (note that $\beta = \rho \cdot (m' + x \cdot r) \bmod q$).
8. $P_i$ computes $s' = \tau^{-1} \cdot \beta \bmod q$ and $s = \min\{s, q - s\}$.
9. *Output:* $P_i$ outputs $(r, s)$.

**Output to all parties.** As we have described above, our protocol as described is not secure with unanimous abort, since some honest parties may abort while others receive output. However, in this case of ECDSA signing, it is easy to transform the protocol so that if one honest party receives output then so do all. This is achieved by having any party who receives $(r, s)$ as output send it to all other parties. Then, if a party who otherwise aborted receives $(r, s)$, it can check that $(r, s)$ is a valid signature on $m$ and output it if yes.

**Correctness.** In order to validate correctness, observe that $\tau$ is the product of $k = \sum_{\ell=1}^{n} k_i$ and $\rho = \sum_{\ell=1}^{n} \rho_i$, and that $\beta$ is the product of the same $\rho$ with $\alpha = \sum_{\ell=1}^{n} \alpha_i$. Furthermore, $R = k \cdot G$ for the same $k$ as above. Given the above, and noting that $\alpha_i = \frac{m'}{n} + x_i \cdot r$, it follows that $s' = \tau^{-1} \cdot \beta = k^{-1} \cdot \rho^{-1} \cdot \rho \cdot \alpha = k^{-1} \cdot (m' + x \cdot r)$, where $r = r_x \bmod q$ for $R = k \cdot G = (r_x, r_y)$. Thus, $(r, s)$ is a valid ECDSA signature with private-key $x = \sum_{\ell=1}^{n} x_i$.

**Security.** Since $\mathcal{F}_{\text{mult}}$ is used for all the operations, the adversary cannot deviate from the protocol at all. Thus, all that is required is to show that the $\tau$ and $\beta$ values revealed leak no information beyond the signature itself. This follows from the fact that $\rho$ is random. We show this formally in the proof of security in Section 5.3.

**Assumptions.** As pointed out in [13], requiring Paillier as an additional assumption for ECDSA signing can be viewed as a disadvantage. If this is a concern, then the private multiplication protocol $\pi^{\text{priv}}_{\text{mult}}$ in Protocol 4.7 can be instantiated with the OT-based protocol of Section 6.1 with the result that the only assumption required is DDH (since oblivious transfer can also be instantiated under this assumption). Although this is not strictly a minimal assumption (since ECDSA does not strictly require DDH), it is a much closer assumption than Paillier.

## 5.2 Efficiency and Experimental Results

In this section, we analyze the theoretical complexity of our protocol, and describe its concrete running time based on our implementation. For the cost, we count the number of exponentiations and communication of group elements (we don't count the cost of commitments since this involves only computing a hash function, and sending small bandwidth; we count the cost of sending an element of $\mathbb{Z}_q$ as the same as a group element even though it's less).

### 5.2.1 Theoretical Complexity

The $\mathcal{F}_{\text{mult}}$ protocol is comprised of subprotocols for init, input, element-out and mult (affine is a local computation only). In addition, it includes a call to $\mathcal{F}_{\text{checkDH}}$ and to $\pi^{\text{priv}}_{\text{mult}}$. Recall that we have two instantiations of $\pi^{\text{priv}}_{\text{mult}}$, one based on OT and the other on Paillier; see Section 6. The costs of each of these subprotocols is summarized in Table 1 (where mult of $\mathcal{F}_{\text{mult}}$ includes the cost of $\mathcal{F}_{\text{checkDH}}$ and $\pi^{\text{priv}}_{\text{mult}}$; note that $\pi^{\text{priv}}_{\text{mult}}$ is run in parallel to the rest and so does not add rounds). We also remark that the base OT computations for $\pi^{\text{priv}}_{\text{mult}}$ based on OT are run in the ECDSA key generation phase, as is the Paillier key-generation and proof of correctness of the key for the Paillier-based protocol. We count these under init of $\mathcal{F}_{\text{mult}}$. For this, we use the OT protocol of [8] that costs 3 exponentiations for the sender and 2 for the receiver (and sending 4 group elements), and use the OT extension of [26] that requires $\kappa$ base OTs for security parameter $\kappa$ (this is the same as used in [13]). We therefore count $2.5 \times 128 = 320$ exponentiations between each pair of parties (averaging that each party is sender half the time and receiver the other half).

The protocol for computing $\mathcal{F}_{\text{ECDSA}}$ is comprised of KeyGen and Sign. The KeyGen phase consists of one call of each of init, input and element-out of $\mathcal{F}_{\text{mult}}$. The overall cost appears in Table 1 (obtained by summing the costs from $\mathcal{F}_{\text{mult}}$); note that these must be called sequentially, and thus the round complexity is also summed. The Sign phase consists of two parallel calls to input of $\mathcal{F}_{\text{mult}}$, followed by a parallel call to element-out and mult, and an additional call to mult. Note that the second call to mult can begin as soon as the result of element-out is received, thereby reducing the round complexity. Note also that when using $\pi^{\text{priv}}_{\text{mult}}$ based on Paillier, the second multiplication requires only 2 Pailler exponentiations and sending a single Paillier ciphertext; this is explained in Section 6.2.

Table 1 clearly shows the communication/computation tradeoff between the OT-based and Paillier-based variants. In particular, the OT-based protocol has much higher bandwidth, but lower computation. This is due to the fact that once the base OTs have been computed in the

key generation phase, the OT extensions used in the actual signing have almost zero cost and are not noticeable relative to the rest of the protocol. In contrast, the Paillier-based protocol requires $7(n-1)$ additional large-integer exponentiations (which are much more expensive than the Elliptic curve operations). In Section 5.2.2, we show the actual running-time of the variant of our protocol that uses Paillier-based private multiplication.

| **Protocol** | **EC Mult.** | **Paillier Exp.** | **Communic.** | **Rounds** |
|---|---|---|---|---|
| $\mathcal{F}_{\text{checkDH}}$ | $11+10n$ | 0 | 8EC | 3 |
| init of $\pi^{\text{priv}}_{\text{mult}}$ (OT) | $320n$ | 0 | 40KiB | 2 |
| mult of $\pi^{\text{priv}}_{\text{mult}}$ (OT) | 0 | 0 | 97KiB | 2 |
| init of $\pi^{\text{priv}}_{\text{mult}}$ (Paillier) | 0 | $11+11n$ | 11N | 1 |
| mult of $\pi^{\text{priv}}_{\text{mult}}$ (Paillier) | 0 | $14n$ | 16N | 2 |
| init of $\mathcal{F}_{\text{mult}}$ (OT) | $2+322n$ | 0 | 3EC + 40KiB | 2 |
| init of $\mathcal{F}_{\text{mult}}$ (Paillier) | $2+2n$ | $11+11n$ | 3EC + 11N | 2 |
| input of $\mathcal{F}_{\text{mult}}$ | $6+5n$ | 0 | 5EC | 1 |
| element-out of $\mathcal{F}_{\text{mult}}$ | $3+4n$ | 0 | 3EC | 1 |
| mult of $\mathcal{F}_{\text{mult}}$ (OT) | $34+32n$ | 0 | 23EC + 97KiB | 6 |
| mult of $\mathcal{F}_{\text{mult}}$ (Paillier) | $34+32n$ | $14n$ | 23EC + 16N | 6 |
| **Totals – OT** | | | | |
| KeyGen of $\mathcal{F}_{\text{ECDSA}}$ | $\mathbf{11+331n}$ | 0 | **11EC + 40KiB** | **5** |
| Sign of $\mathcal{F}_{\text{ECDSA}}$ (OT) | $\mathbf{83+78n}$ | 0 | **59EC + 194KiB** | **8** |
| **Totals – Paillier** | | | | |
| KeyGen of $\mathcal{F}_{\text{ECDSA}}$ | $\mathbf{11+11n}$ | $\mathbf{11+11n}$ | **11EC + 11N** | **5** |
| Sign of $\mathcal{F}_{\text{ECDSA}}$ | $\mathbf{83+78n}$ | $\mathbf{21n}$ | **59EC + 24N** | **8** |

Table 1: Theoretical counts of all costs in our protocols; the communication cost given is in *group elements* (denoted EC) and elements of $\mathbb{Z}_N$ (denoted N) that *each* party sends to *each other* party. The cost of $\pi^{\text{priv}}_{\text{mult}}$ for OT is taken from [13, Sec. VI-D]; these concrete numbers are based on computational security parameter $\kappa = 128$ and statistical security parameter 80. The key generation for the Paillier variant includes additional costs not counted here (like local Paillier key generation, and verification of the zero-knowledge proof of correctness of the Paillier key).

| **Variant** | **Our Protocol** | **[18]** |
|---|---|---|
| Key generation 256-bit curve (OT) | 40.3KiB | theoretical |
| Key generation 521-bit curve (OT) | 40.6KiB | theoretical |
| Sign 256-bit curve (OT) | 196KiB | 6KiB |
| Sign 521-bit curve (OT) | 198KiB | 13KiB |
| Key generation 256-bit curve (Paillier) | 3.1KiB | theoretical |
| Key generation 521-bit curve (Paillier) | 6.4KiB | theoretical |
| Sign 256-bit curve (Paillier) | 7.8KiB | 6KiB |
| Sign 521-bit curve (Paillier) | 9.8KiB | 13KiB |

Table 2: Concrete bandwidth for our protocol and [18], for different curve sizes, in ***kilobytes***. The communication is how much *each* party sends to *each other* party. In all cases, we use a 2048-bit modulus for Paillier.

We compare the concrete communication costs to that of [18] in Table 2. When using the OT-based private multiplication, the communication is significantly greater than [18], whereas our Paillier-based protocol almost twice the communication of [18] for a 256-bit curve, and approxi-

mately the same communication of [18] for a 521-bit curve.

### 5.2.2 Experimental Results

We implemented our protocol in C++, and ran it on AWS with all machines of type Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz with 1 GB RAM (RedHat 7.2) and a 1Gb per second network card. We ran experiments from 2 to 20 parties (all in the same region); each execution was run 16 times and we took the average. The results can be seen in the graph in Figure 1, and in Table 3. We implemented the version with the Paillier-based private multiplication of Section 6.2 (although the OT version has much faster computation, its significantly higher communication makes it less attractive for most real-world scenarios). As is clearly seen, the signing time is practical (especially for cryptocurrency applications): from 304ms for 2 parties to about 3sec for 10 parties and about 5sec for 20 parties. We stress that the implementation is *single threaded*, and the running time can be significantly reduced by using multiple threads on multicore machines.

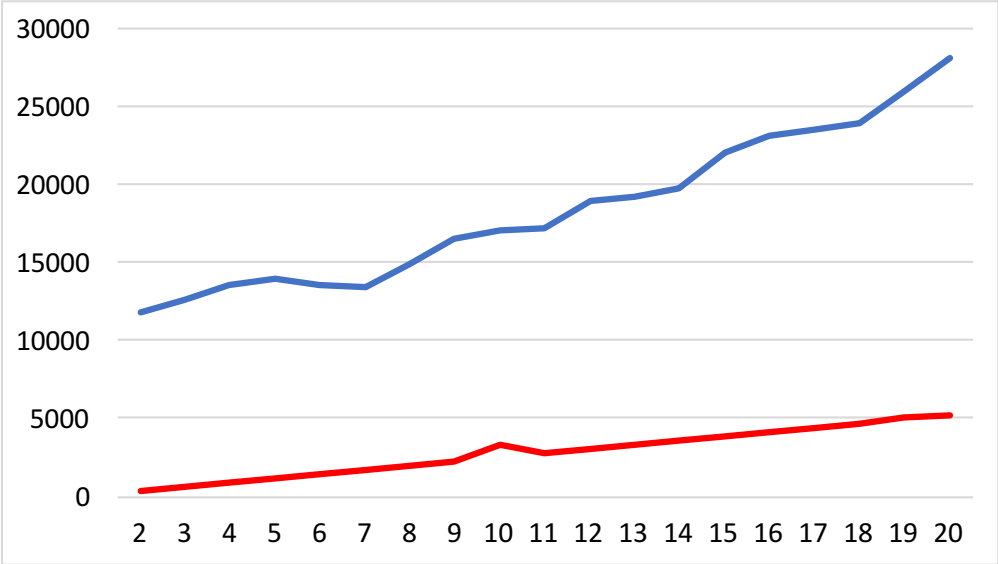


Figure 1: The running times in milliseconds for key generation (top line in blue) and signing (bottom line in red) for 2-20 parties, for the Pailler variant of the protocol.

As we have stressed, the primary contribution of this paper is to achieve practical signing and key generation. Indeed, as shown in Figure 1, key generation takes 11sec for 2 parties, 17sec for 10 parties and 28 seconds for 20 parties. This is not very fast, but is clearly practical, especially since key generation need only be run once.

We provide a comparison of our *signing phase* with [18]; we stress that we cannot compare key generation, since this has not been implemented for [18] due to the fact that key generation requires multiparty Paillier key generation (although two-party Paillier key generation can be feasible [15], there is currently no practical multiparty variant). The comparison appears in Table 3. The protocol of [18] was run on a Ubuntu desktop with an Intel quad-core i7-6700 CPU @ 3.40GHz and 64GB of RAM (although only a single core was used). The times for [18] are taken from [22], which refers to a reimplementation of [18] that is much faster than the times reported in [18] itself. We stress that the times from [18, 22] include *local computation only*, and no communication. In

comparison, our signing phase *including communication* is 2–5 times slower than [18] (depending on the number of parties). Since our goal is multiparty ECDSA with *practical key generation and signing*, this demonstrates that our goal is achieved.

| **Number of Parties** | **GGN16 [18]** (local comp. only) | **Our protocol – Sign** (including communication) |
|---|---|---|
| 2 | 205 | 304 |
| 3 | 256 | 575 |
| 4 | 312 | 840 |
| 5 | 369 | 1112 |
| 6 | 418 | 1395 |
| 7 | 453 | 1648 |
| 8 | 505 | 1929 |
| 9 | 557 | 2196 |
| 10 | 609 | 3267 |
| 11 | 661 | 2774 |
| 12 | 714 | 3011 |
| 13 | 765 | 3284 |
| 14 | 818 | 3572 |
| 15 | 876 | 3852 |
| 16 | 949 | 4144 |
| 17 | 977 | 4385 |
| 18 | 1029 | 4636 |
| 19 | 1083 | 5061 |
| 20 | 1136 | 5194 |

Table 3: Comparison of signing times (in milliseconds) between [18] and our protocol. The times for [18] are of *local computation only* (without communication), whereas the times for our protocol are in AWS with real communication.

### 5.3 Proof of Security of Protocol 5.1

**Theorem 5.2** *Protocol 5.1 securely computes $\mathcal{F}_{\text{ECDSA}}$ with abort in the $\mathcal{F}_{\text{mult}}$-hybrid model, in the presence of a malicious adversary corrupting any $t < n$ parties, with point-to-point channels.*

**Proof:** We have already provided the intuition regarding the security of the protocol in Section 5.1, and so proceed directly with the formal proof.

Let $\mathcal{A}$ be an adversary and let $I \subseteq [n]$ be the set of corrupted parties. If all parties are corrupted, then the simulation is trivial. We therefore consider $I \subset [n]$ (with $|I| < n$) and we denote the set of honest parties by $J = [n] \setminus I$. Throughout the proof, we denote corrupted parties by $P_i$ (i.e., $i \in I$), honest parties by $P_j$ (i.e., $j \in J$), and a running index over $[n]$ by $\ell$.

We construct a simulator $\mathcal{S}$ who invokes $\mathcal{A}$ internally and simulates an execution of the real protocol, while interacting with $\mathcal{F}_{\text{ECDSA}}$ in the ideal model.

**Key generation:** For the key generation, $\mathcal{S}$ instructs all $P_i$ with $i \in I$ to send $\mathsf{KeyGen}(\mathbb{G}, G, q)$ to $\mathcal{F}_{\text{ECDSA}}$. Upon receiving back $Q$ from $\mathcal{F}_{\text{ECDSA}}$, simulator $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ receives $\mathsf{init}$ messages that $\mathcal{A}$ sends to $\mathcal{F}_{\text{mult}}$, for every $i \in I$.

2. $\mathcal{S}$ receives the messages $(\mathsf{input}, sid_{\text{gen}}, x_i)$ that $\mathcal{A}$ sends to $\mathcal{F}_{\text{mult}}$, for every $i \in I$.

3. $\mathcal{S}$ simulates $\mathcal{F}_{\text{mult}}$ sending $(\mathsf{input}, sid_{\text{gen}})$ to $P_i$, for every $i \in I$.

4. $\mathcal{S}$ receives the messages $(\mathsf{element\text{-}out}, 0)$ that $\mathcal{A}$ sends to $\mathcal{F}_{\text{mult}}$, for every $i \in I$.

5. $\mathcal{S}$ simulates $\mathcal{F}_{\text{mult}}$ sending $(\mathsf{element\text{-}out}, 0, Q)$ to $P_i$, for every $i \in I$, where $Q$ is the public-key received by $\mathcal{S}$ from $\mathcal{F}_{\text{ECDSA}}$. In addition $\mathcal{S}$ computes $Q_i = x_i \cdot G$ for every $i \in I$, and chooses random $\{Q_j\}_{j \in J}$ under the constraint that $\sum_{\ell=1}^{n} Q_\ell = Q$. Then, $\mathcal{S}$ simulates $\mathcal{F}_{\text{mult}}$ sending $Q_1, \ldots, Q_n$ to $\mathcal{A}$.

6. $\mathcal{S}$ stores all of the values $\{x_i\}_{i \in I}$, as well as $Q$.

This completes the simulation of the key generation phase.

**Signing:** For the signing phase, upon input $\mathsf{Sign}(sid, m)$, if $sid$ has not been used previously, then $\mathcal{S}$ instructs all $P_i$ with $i \in I$ to send $\mathsf{Sign}(sid, m)$ to $\mathcal{F}_{\text{ECDSA}}$. Upon receiving back a signature $(r, s)$, simulator $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ runs the ECDSA verify procedure on signature $(r, s)$ and public key $Q$, and obtains the group element $R$ such that $R = (r_x, r_y)$ and $r = r_x \bmod q$.

2. $\mathcal{S}$ receives $(\mathsf{input}, sid\|1, k_i)$ and $(\mathsf{input}, sid\|1, \rho_i)$ that $\mathcal{A}$ instructs $P_i$ to send to $\mathcal{F}_{\text{mult}}$, for all $i \in I$.

3. $\mathcal{S}$ simulates $\mathcal{F}_{\text{mult}}$ sending $(\mathsf{input}, sid\|1)$ and $(\mathsf{input}, sid\|2)$ to $P_i$, for all $i \in I$.

4. $\mathcal{S}$ receives messages $(\mathsf{mult}, sid\|1, sid\|2)$ and $(\mathsf{element\text{-}out}, sid\|1)$ that $\mathcal{A}$ instructs $P_i$ to send to $\mathcal{F}_{\text{mult}}$, for all $i \in I$.

5. $\mathcal{S}$ chooses a random $\tau \in \mathbb{Z}_q$ and simulates $\mathcal{F}_{\text{mult}}$ sending $(\mathsf{mult\text{-}out}, sid\|1, sid\|2, \tau)$ to $P_i$ for all $i \in I$.

6. $\mathcal{S}$ simulates $\mathcal{F}_{\text{mult}}$ sending $(\mathsf{element\text{-}out}, sid\|1, R)$ to $P_i$ for all $i \in I$, where $R$ is as computed in Step 1 above. In addition $\mathcal{S}$ computes $R_i = k_i \cdot G$ for every $i \in I$, and chooses random $\{R_j\}_{j \in J}$ under the constraint that $\sum_{\ell=1}^{n} R_\ell = R$. Then, $\mathcal{S}$ simulates $\mathcal{F}_{\text{mult}}$ sending $R_1, \ldots, R_n$ to $\mathcal{A}$.

7. $\mathcal{S}$ simulates $\mathcal{F}_{\text{mult}}$ sending $(\mathsf{affine}, 0, sid\|3, r, m')$ to $\mathcal{A}$.

8. $\mathcal{S}$ receives messages $(\mathsf{mult}, sid\|2, sid\|3)$ that $\mathcal{A}$ instructs $P_i$ to send to $\mathcal{F}_{\text{mult}}$, for all $i \in I$.

9. $\mathcal{S}$ sets $s' = s$ with probability $1/2$, and otherwise $s' = q - s$.

10. $\mathcal{S}$ computes $\beta = \tau \cdot s' \bmod q$.

11. $\mathcal{S}$ simulates $\mathcal{F}_{\text{mult}}$ sending $(\mathsf{mult\text{-}out}, sid\|2, sid\|3, \beta)$ to $P_i$, for all $i \in I$.

12. If no abort has happened, then $\mathcal{S}$ sends $(\mathsf{continue}, sid, j)$ to $\mathcal{F}_{\text{ECDSA}}$ for all $j \in J$ to instruct honest parties to receive output.

We now show that the simulation by $\mathcal{S}$ in the ideal model results in a distribution that is identical to that of an execution of Protocol 5.1 in the $\mathcal{F}_{\text{mult}}$-hybrid model. Observe that the only actual values received by the corrupted parties during the execution are $Q$ (in the key generation), $R$ (as the output of element-out) and values $\tau$ and $\beta$. The distribution over these values in a real execution are

$$Q = x \cdot G, \quad R = k \cdot G, \quad \tau = \rho \cdot k, \quad \beta = \rho \cdot (m' + x \cdot r)$$

where $k, \rho \in \mathbb{G}$ are random, and $r = r_x \bmod q$ where $R = (r_x, r_y)$. (Note that $x$ is also random, but it is fixed in the key generation phase, and the same in all signing executions.) In contrast, the distribution over these values in the simulated execution are

$$Q = x \cdot G, \quad R = k \cdot G, \quad \tau, \quad \beta = \tau \cdot s'$$

where $k, \tau \in \mathbb{G}$ are random, and $s' = k^{-1} \cdot (m + x \cdot r)$. (Observe that $\mathcal{S}$ does not know $x$ and $k$, but this is the distribution since it is derived from the output received from $\mathcal{F}_{\text{ECDSA}}$.) In the real execution, since $\rho$ is random, we have that $\tau$ and $\beta$ are *random under the constraint* that

$$\frac{\beta}{\tau} = \frac{\rho \cdot (m' + x \cdot r)}{\rho \cdot k} = k^{-1} \cdot (m' + x \cdot r) = s'.$$

Similarly, in the simulated execution, since $\tau$ is random, we have that $\tau$ and $\beta$ are *random under the constraint* that

$$\frac{\beta}{\tau} = \frac{\tau \cdot s'}{s'} = s'.$$

Thus, these distributions are identical.

The only issue that we have ignored is the distribution over $s'$ being less than or greater than $q/2$. Specifically, $\mathcal{F}_{\text{ECDSA}}$ always takes $s$ in the signature to be less than $q/2$ (by defining $s = \min\{s', q - s'\}$). The parties in the protocol do the same. However, the distribution over $s'$ in the simulation is such that $s' < q/2$ with probability $1/2$ and $s' > q/2$ with probability $1/2$ (see Step 9 in the simulation description). We therefore have to show that this is also the same in a real execution. Nevertheless, this follows immediately from the fact that for every $k \in \mathbb{Z}_q$ resulting in $s' = k^{-1} \cdot (m' + r \cdot x)$ it follows that $-k$ results in $s' = -k^{-1} \cdot (m + r \cdot x) = q - k^{-1} \cdot (m + r \cdot x)$; in both cases $r$ is the same since if $k \cdot G = (r_x, r_y)$ then $-k \cdot G = (r_x, -r_y)$. This implies that the probability of receiving $s' < q/2$ is $1/2$ and equal to the probability of receiving $s' > q/2$. This completes the proof. ■

### 5.4 Extending the Protocol to Threshold (and General) Access Structures

Our protocol is described using $n$-out-of-$n$ additive sharing of the ECDSA private key $x$. This therefore yields a protocol that is secure for any $t < n$ corruptions, but also requires all $n$ parties to sign. For a full custody solution, it must be possible to define a more general access structure. This is due to the fact that when currency needs to be transferred, not all parties may be available. Using known techniques, it is possible for the $n$ parties to generate a Shamir sharing of $x$ with threshold $t$, in order to enable a subset of any $t$ parties to sign while preserving security for any $t' < t$ corruptions. Once the private key $x$ is shared in this way, our protocol for ECDSA signing proceeds in almost exactly the same way as described in Protocol 5.1, with the sole change being that the affine operation called to generate shares of $\beta$ includes the appropriate Lagrange coefficients so that $\beta$ equals $\rho \cdot (m' + x \cdot r)$. (Note that inside $\mathcal{F}_{\text{mult}}$ each party must hold a simple *additive* share

$\beta_i$ of $\beta$, but this is achieved by including the Lagrange coefficients) Of course, it is also necessary to generate a Shamir sharing of $d$ (as in the init phase of $\mathcal{F}_{\text{mult}}$), to enable computing $\mathcal{F}_{\text{checkDH}}$, which requires shares of the private ElGamal key associated with $\mathcal{P}$.

The above requires a modification to the key generation phase in order to generate these shares. Fortunately, it is not difficult to generate $t$-out-of-$n$ shares in a robust way for ECDSA, as described in [13] (although their signing protocol only works for 2-out-of-$n$). We sketch this now, using the $\mathcal{F}_{\text{mult}}$ functionality. The idea is for the parties to generate a sharing of a random degree-$(t-1)$ polynomial $p(\cdot)$ where the ECDSA private key is $x = p(0)$. Denote $p(z) = \sum_{\ell=0}^{t-1} c_\ell \cdot z^\ell$; note that $x = p(0) = c_0$. This polynomial is generated by the parties calling $(\mathsf{input}, sid\|\ell)$ for $\ell = 0, \ldots, t-1$, to define the coefficients ($c_\ell$ is the coefficient with identifier $sid\|\ell$). Next, the parties call $(\mathsf{element\text{-}out}, sid\|0)$ to obtain the ECDSA public key $Q = c_0 \cdot G = x \cdot G$. Finally, each party $P_i$ needs to obtain its Shamir share $p(i)$ of the private key. This can be achieved by a variant of the affine operation, for the parties to compute ElGamal encryptions and shares of $p(i)$ for $i = 1, \ldots, n$; this is just computing a local linear combination of the shares of the coefficients and the appropriate value $i^0, i^1, i^2, \ldots, i^{t-1}$. Once the parties hold these shares and the associated ElGamal encryptions, they can run a variant of element-out that provides the value to only one party, in order for each $P_i$ to receive $p(i) \cdot G$. Finally, the parties can all send their shares of $p(i)$ to $P_i$, who sums them and verifies that the result is correct by multiplying $G$ by the sum and comparing the result to the value received in element-out. This prevents any party from providing an incorrect share. In order to ensure that $P_i$ learns nothing but the sum, each $P_j$ can randomly mask its share and send shares of the mask to all other parties.

The above can be generalized to *any* linear secret sharing scheme for *any* access structure, as long as such a scheme has an efficient method of robustly sharing $x$ (and obtaining $Q$).

**An important access structure.** One specific access structure that is of practical interest for cryptocurrency custody is the following. Define $\ell$ sets of users $A_1, \ldots, A_\ell$ of sizes $n_1, \ldots, n_\ell$, respectively, and define authorized thresholds $t_1, \ldots, t_\ell$ for each set. Then, define the access structure to be any combination of AND/OR over these sets. For example, one could define $(A_1 \vee A_2) \wedge (A_3 \vee A_4) \wedge A_5$, which means that in order to sign, an appropriate threshold is needed in $A_1$ or $A_2$, and in $A_3$ or $A_4$ and in $A_5$. To be concrete, $A_5$ could denote a set of servers at a trustee (that rely on automatic authorization and not human verification), $A_3$ and $A_4$ could be two sets of shift workers at the financial institution offering the custody solution, and $A_1$ and $A_2$ could be analogous workers at the customer who owns the cryptocurrency. Each of the sets may have its own threshold which trades off security and the difficulty of contacting a large enough set when funds need to be transferred. Our protocol easily supports such an access structure by using the well-known method of traversing the circuit from the output to the input, and additively sharing the key at an AND gate while replicating the key at an OR gate. Finally, Shamir sharing is used to split the key shares at each node amongst the defined subset with the defined threshold.

# 6 Private Multiplication Instantiations

## 6.1 Private Multiplication Using OT

A multiplication protocol that achieves the above properties can be efficiently constructed using the well-known oblivious-transfer based approach for multiplication. This approach was introduced by [20] but suffers from selective-bit attacks. This problem can be overcome by encoding the

receiver's inputs, as was recently shown in [13] also in the context of ECDSA. When running this protocol with oblivious transfer that is secure in the presence of malicious adversaries, the corrupted parties' inputs to the oblivious are actually explicitly defined (and can be extracted). Thus, when correct inputs are used, this protocol can be simulated, implying input indistinguishability. In contrast, if the adversary inputs incorrect values to the OTs that do not define *any* real input (i.e., are inconsistent between the OTs), then the result will not be the valid $c = \sum_{\ell=1}^{n} c_\ell$ except with negligible probability. Thus, in such a case, we merely need the first privacy requirement.

**Complexity.** When using OT extension [26], private multiplication requires the parties to carry out a setup step, where a public key is established which is used to compute seeds for the sender and receiver. Then, these seeds are expanded by a PRG (pseudo-random generator) to generate random correlated pads to mask the transmitted messages. The setup step involves each party sending 40KiB and carrying out 320 elliptic-curve multiplications. (We have 320 multiplications since [26] requires running $\kappa$ base OTs, that cost 3 multiplications for the sender and 2 for the receiver, and thus 2.5 on average. In practice, we take $\kappa = 128$, thereby requiring 320 multiplications.) The extension part is cheap in terms of computation but requires each party to transmit 97KiB to each of the other parties for each multiplication.

## 6.2 Private Multiplication Using Paillier

### 6.2.1 The Multiplication Protocol

We now present an alternative method of achieving private multiplication which has much lower communication complexity but higher computation (but still very reasonable). This protocol is based on the folklore multiplication protocol based on additively homomorphic encryption, with additional zero-knowledge proofs to prevent malicious parties from cheating. In particular, one of the main problems that we face in this protocol is that we wish to multiply inside $\mathbb{Z}_q$, but are using Paillier with message space $\mathbb{Z}_N$. This opens the door to multiple attacks, that we discuss below.

Each party $P_i$ works as follows:

1. *Setup (run during key generation):* Party $P_i$ generates a Paillier public/private key pair [31], denoted $(pk_i, sk_i)$, and sends $pk_i$ to all parties. We denote the Paillier encryption of $x$ using randomness $r$ and public-key $pk$ by $\mathsf{Enc}_{pk}(x; r)$. We stress that each party generates its own Paillier key *locally*, and no distributed generation is needed. In addition, each party generates a non-interactive proof that $pk_i$ was generated correctly (more exactly, that $\gcd(N, \phi(N) = 1$ and so the scheme is indeed additively homomorphic); see Section 6.2.3.

   In addition, each party sets parameters $s = t = 128$ and $\ell = 80$ (the parameters $t, \ell$ are utilized in the zero-knowledge proofs in Sections 6.2.6 and 6.2.7), whereas $s$ is used to ensure that $\delta_{j\to i}$ sent in Step 2 is large enough to hide $P_j$'s private input.

2. *Multiplication:* each party $P_i$ holds inputs $x_i, y_i$ and the aim is for the parties to obtain $z_1, \ldots, z_n$ such that $\sum_{i=1}^{n} z_i = (\sum_{i=1}^{n} x_i) \cdot (\sum_{i=1}^{n} y_i) \bmod q$. The protocol is as follows:

   (a) *Step 1 (message 1):* $P_i$ computes $c_i = \mathsf{Enc}_{pk_i}(x_i)$, where $pk_i$ is its own public-key. In addition, for every $j$, party $P_i$ prepares a non-interactive zero-knowledge range proof $\pi^1_{i\to j}$ using $N_j$ proving that $x_i \in \mathbb{Z}_q$ (with soundness as long as $x_i \in (-2^{t+\ell} \cdot q, 2^{t+\ell} \cdot q)$ for

$t, \ell$ security parameters as defined in Section 6.2.6 (the proof requires a modulus from the verifier, and thus is different for each $P_j$). For every $j \in [n]\backslash\{i\}$, $P_i$ sends $(c_i, \pi^1_{i\to j})$ to $P_j$.

(b) *Step 2 (message 2):* For every $j \in [n]\backslash\{i\}$, party $P_i$ receives $c_j = \mathsf{Enc}_{pk_j}(x_j)$ and $\pi_{j\to i}$ from $P_j$. Party $P_i$ verifies the proof $\pi_{j\to i}$ and then uses the homomorphic operations—where $\odot, \oplus$ are homomorphic multiplication and addition, respectively—to generate

$$c_{i\to j} = (c_j \oplus 2^{t+\ell} \cdot q) \odot y_i \oplus \delta_{i\to j} = \mathsf{Enc}_{pk_j}(x_j \cdot y_i + 2^{t+\ell} \cdot q \cdot y_j + \delta_{i\to j})$$

for a randomly chosen $\delta_{i\to j} \in \mathbb{Z}_{q^2 \cdot 2^{t+\ell+s}}$. In addition, $P_i$ generates a non-interactive zero-knowledge proof of knowledge $\pi^2_{i\to j}$ of the pair of values $(y_j, \delta_{j\to i})$ such that $c_{i\to j} = ((c_j \oplus 2^{t+\ell} \cdot q) \odot y_j) \oplus \delta_{j\to i}$ with $y_j \in \mathbb{Z}_q$ and $\delta_{j\to i} \in \mathbb{Z}_{q^2 \cdot 2^{t+\ell+s}}$ (with soundness as long as $y_j \in (-2^{t+\ell} \cdot q, 2^{t+\ell} \cdot q)$ and $\delta_{j\to i} \in (-q^2 \cdot 2^{2t+2\ell+s}, q^2 \cdot 2^{2t+2\ell+s})$). See Section 6.2.7 for the proof specification. $P_i$ sends $(c_{i\to j}, \pi^2_{i\to j})$ to $P_j$.

(c) *Step 3 (output):* $P_i$ receives all $c_{j\to i} = \mathsf{Enc}_{pk_i}(x_i \cdot y_j + 2^{t+\ell} \cdot q \cdot y_j + \delta_{j\to i})$ and $\pi^2_{j\to i}$ values from $P_j$ for every $j \in [n]\backslash\{i\}$. Then, for each $c_{j\to i}$, $P_i$ verifies the proof $\pi^2_{j\to i}$, decrypts $c_{j\to i}$ and adds $(x_i + 2^{t+\ell} \cdot q) \cdot 2^{t+\ell} \cdot q + 2^{t+\ell} \cdot q^2 \cdot 2^{t+\ell+s}$ mod $N$. Finally, $P_i$ sums the results, adds $x_i \cdot y_i$ and subtracts all its $\delta_{i\to j}$ values, and reduces the result modulo $q$. The result is $P_i$'s output $z_i$.

Before proceeding, we explain some of the ideas behind the above protocol. First, we explain why $P_i$ and $P_j$ add the $2^{t+\ell} \cdot q$ values. Note that the soundness of the zero-knowledge proof provided by $P_i$ only guarantees that $x_i \in (-2^{t+\ell} \cdot q, 2^{t+\ell} \cdot q)$. Thus, $P_j$ first adds $2^{t+\ell} \cdot q$ to the ciphertext $c_i$ before proceeding, ensuring that it encrypts a value that is greater than 0 (and less than $2 \cdot 2^{t+\ell} \cdot q$). Likewise, the proof provided by $P_j$ only guarantees that $b \in (-2^{t+\ell} \cdot q, 2^{t+\ell} \cdot q)$ and $\Delta \in (-2^{t+\ell} \cdot q^2 \cdot 2^{t+\ell+s}, 2^{t+\ell} \cdot q^2 \cdot 2^{t+\ell+s})$. Thus, $P_i$ adds $(x_i + 2^{t+\ell} \cdot q) \cdot 2^{t+\ell} \cdot q$ mod $N$, converting the multiplication by $b$ to a multiplication by $b + 2^{t+\ell} \cdot q$, ensuring that even if $-2^{t+\ell} \cdot q < b < 0$ then the result is multiplication by a positive value in the range $[0, 2 \cdot 2^{t+\ell} \cdot q)$. Furthermore, $P_i$ adds $2^{t+\ell} \cdot q^2 \cdot 2^{t+\ell+s}$ mod $N$ so that if $-2^{t+\ell} \cdot q^2 \cdot 2^{t+\ell+s} < \Delta < 0$ then the result is as if a positive value in the range $[0, 2 \cdot 2^{t+\ell} \cdot q^2 \cdot 2^{t+\ell+s})$ was added. Note that these operations are modulo $N$ since this is the domain of the Paillier encryption.

Next, we explain the choice of $\delta_{j\to i} \in \mathbb{Z}_{q^2 \cdot 2^{t+\ell+s}}$. In our use of this protocol, all $x_i, y_i \in \mathbb{Z}_q$. In order to ensure that $c_{j\to i}$ reveals nothing about $y_j$, the value $\delta_{j\to i}$ needs to be longer than $x_i \cdot y_j + 2^{t+\ell} \cdot q \cdot y_j$ (this is because no modulo reduction takes place and so $P_i$ receives the integer value). Now, as stated above, the zero-knowledge proof regarding $x_i$ only guarantees that it is less than $q \cdot 2^{t+\ell}$ and so $x_i \cdot y_j < q^2 \cdot 2^{t+\ell}$. We therefore choose $\delta_{j\to i} \in \mathbb{Z}_{q^2 \cdot 2^{t+\ell+s}}$, making it $s$-bits longer than $x_i \cdot y_j + 2^{t+\ell} \cdot q \cdot y_j$ and so making the statistical distance between $x_i \cdot y_j + 2^{t+\ell} \cdot q \cdot y_j + \delta_{j\to i}$ and a random element $r \in \mathbb{Z}_{q^2 \cdot 2^{t+\ell+s}}$ be at most $2^{-s}$.

**Correctness:** The protocol is correct as long as there is no reduction modulo $N$. In order to see this, note that

$$x_i \cdot y_j + 2^{t+\ell} \cdot q \cdot y_j + \delta_{j\to i} + (x_i + 2^{t+\ell} \cdot q) \cdot 2^{t+\ell} \cdot q + 2^{t+\ell} \cdot q^2 \cdot 2^{t+\ell+s} = x_i \cdot y_j + \delta_{j\to i} \bmod q.$$

Thus, by each party $P_i$ subtracting the $\delta_{i\to j}$ values they chose, the overall sum equals $x \cdot y$. Now, we have that no reduction modulo $N$ takes place in the Paillier encryption as long as

$$x_i \cdot y_j + 2^{t+\ell} \cdot q \cdot y_j + \delta_{j\to i} + (x_i + 2^{t+\ell} \cdot q) \cdot 2^{t+\ell} \cdot q + 2^{t+\ell} \cdot q^2 \cdot 2^{t+\ell+s} < N.$$

Now, since $x_i, y_j \in \mathbb{Z}_q$ and $\delta_{j\to i} \in \mathbb{Z}_{q^2\cdot 2^{t+\ell+s}}$, we have that the above is upper bound by

$$q^2 + 2^{t+\ell}\cdot q^2 + q^2\cdot 2^{t+\ell+s} + 2^{2t+2\ell}\cdot q^2 + 2^{t+\ell}\cdot q^2\cdot 2^{t+\ell+s} < 2\cdot q^2\cdot 2^{2t+2\ell+s}.$$

Taking $s = \ell = 80$ (which suffices for statistical distance) and $t = 128$, we have that as long as $N > q^2\cdot 2^{497}$ there is no reduction modulo $N$ in the computation. This implies that for $q < 775$ (which includes all typically used curves), it suffices to use $N$ of length 2048 (which is the minimum reasonable size for Paillier in any case).

**On the necessity of the zero-knowledge proofs.** We first prove why it is necessary to have the parties prove that the values they used in the homomorphic encryption are within certain ranges. Consider a corrupted party $P_i$ who sends an encryption $c_i = \mathsf{Enc}_{pk_i}(\mu\cdot q^\nu + x_i)$ for some $\mu, \nu \in \mathbb{N}$ with the property that if $y_j$ is small then $(\mu\cdot q^\nu + x_i)\cdot y_j < N_i$ and if $y_j$ is large then $(\mu\cdot q^\nu + x_i)\cdot y_j \geq N_i$. In the former case, the protocol will proceed with everything correct (since $\mu\cdot q^\nu\cdot y_j + x_i\cdot y_j = x_i\cdot y_j \bmod q$), whereas in the latter case the overall protocol will fail (since $[\mu\cdot q^\nu\cdot y_j \bmod N_i] + x_i\cdot y_j \neq x_i\cdot y_j \bmod q$). This will reveal one bit of information about $y_j$ that should not be revealed, and enables an adversary to carry out a binary search on private values. In a similar way, a corrupted $P_j$ can also use $y_j + \mu\cdot q^\nu$ and achieve the same effect. In order to prevent this attack, we have both parties prove that their values are in range in zero-knowledge. Since $N$ is much larger than $q$, we are able to use range proofs that have a lot of slack regarding soundness. This is important since tight range proofs are much more expensive.

In addition, recall that $P_j$ proves that it computed $c_{j\to i}$ by multiplying by some $b$ and adding $\Delta$ (within range). This part of the proof is needed due to the following. Assume that the encryption scheme used (Paillier) is actually *fully homomorphic* and not just additively homomorphic. Then, $P_j$ could generate the ciphertext $c_{j\to i}$ by computing a circuit that says: if the value encrypted modulo $q$ is less than $q/2$ then output the correct value; else output an incorrect value. This would leak information about $P_i$'s private value, as discussed above. Thus, in order to prove security *without* the zero-knowledge proof, one would need to assume that Paillier is *not* fully homomorphic, which would be a non-standard assumption on Paillier. We stress that it may be possible to prove the ECDSA protocol secure under a game-based definition without making this assumption (although we do not know if this is the case). Nevertheless, our aim is to achieve simulation-based security which seems to require this stronger property.

**Multiple multiplications.** In our use of multiplication in our ECDSA protocol, the parties need to carry out two multiplications, where one of the values is the same. Let $x_i$ be the input of $P_i$ and let $y_j, y'_j$ be the inputs of $P_j$. Then, $P_i$ sends a single first message, and $P_j$ computes its second message once with $y_j$ and $\delta_{j\to i}$ and a second time with $y'_j$ and a different random $\delta'_{j\to i}$. Thus, the second multiplication requires only a single round of communication, and only a single ciphertext. Note that this second multiplication can also be run later and does not need to be run at the same time as the first.

**Complexity.** We count the computation and communication complexity of this protocol. In the setup (key generation) phase, each party runs Paillier key generation, computes $11n$ exponentiations in $\mathbb{Z}_N$ (11 to generate the proof and $11(n-1)$ to verify the other proofs) and sends 12 elements of $\mathbb{Z}_N$ (for the zero-knowledge proof); see Section 6.2.3. (There is an additional cost of checking divisibility of $N$ as described in [21].) Thus, the overall cost is computing approximately $11n$

exponentiations in $\mathbb{Z}_N$ and sending 11 elements of $\mathbb{Z}_N$ to each party. Then, for multiplication, each party works independently with each other party to pairwise multiply their shares. The cost of this is carrying out one Paillier encryption and decryption (when playing $P_i$), homomorphic operations on Paillier which are not significant (when playing $P_j$), and proving and verifying both proofs of Sections 6.2.6 and 6.2.7. Counting the cost of one Paillier encryption to be approximately two Pedersen commitments, we have that each of these proofs costs approximately 3 Paillier encryption by the prover and verifier each, and the transmission of approximately 6 elements of $\mathbb{Z}_N$. Thus, the overall cost is computing approximately $14n$ Paillier encryptions, and each party sending each other party approximately 16 elements of $\mathbb{Z}_N$ (12 for the proofs, plus two Paillier ciphertext which are equivalent to two elements of $\mathbb{Z}_N$ each). For $N$ of length 2048, this comes to approximately 4KiB for multiplication and 2.75KiB for setup (key generation).

As discussed above, in our ECDSA protocol we carry out two multiplication with the same value from $P_i$. In this second multiplication, no ciphertexts or proofs need to be sent or verified for Step 1. Thus, the cost of the second multiplication is exactly half the above, and so computing approximately $7n$ Paillier encryptions and each party sending each other party approximately 8 elements of $\mathbb{Z}_N$.

**Bandwidth versus computation.** The Paillier-based solution presented here has much lower bandwidth than the OT-based solution of Section 6.1. As such, it is better suited for scenarios where some of the participants are more limited (e.g., may be mobile phones). However, in a setting where all parties are in the same region and have fast connections, the OT-based approach may be preferable.

### 6.2.2 Security

We now show that this protocol satisfies our definition of private multiplication in Section 2.3. First, observe that the first privacy property easily follows from the fact that each party $P_j$ sees only encryptions of the other $P_i$'s inputs under $P_i$'s public key (which are therefore indistinguishable), and decrypted values $x_i \cdot y_j + \delta_{j\to i}$ (plus other terms divisible by $q$) which statistically hides the value, since $\delta_{j\to i}$ is random in $\mathbb{Z}_{q^2 \cdot 2^{t+\ell+s}}$ and this is $2^s$ times larger than the values encrypted. (See more details regarding this in the explanation about why $\delta_{j\to i}$ is chosen in $\mathbb{Z}_{q^2 \cdot 2^{t+\ell+s}}$ following the protocol description in Section 6.2.1.) Formally, privacy can be shown by the fact that for every two sets of honest parties' inputs $\{x_j, y_j\}_{j\in J}$ and $\{x'_j, y'_j\}_{j\in J}$, the Paillier ciphertexts are indistinguishable (via reduction to the semantic security of the encryption), and the distribution over $x_i \cdot y_j + 2^{t+\ell} \cdot q \cdot y_j + \delta_{j\to i}$ is statistically close to the distribution over $x_i \cdot y'_j + 2^{t+\ell} \cdot q \cdot y'_j + \delta_{j\to i}$.

For input indistinguishability, observe that once again, for every two sets of inputs $\{x_\ell, y_\ell\}_{\ell\in[n]}$ and $\{x'_\ell, y'_\ell\}_{\ell\in[n]}$, the Paillier encryptions of the honest parties' inputs are indistinguishable, and the $x_i \cdot y_j + 2^{t+\ell} \cdot q \cdot y_j + \delta_{j\to i}$ and $x'_i \cdot y'_j + 2^{t+\ell} \cdot q \cdot y'_j + \delta'_{j\to i}$ are statistically close. Now, since the sum $[c \bmod q]$ is the same in both cases, this means that there exist $\delta_{j\to i}, \delta'_{j\to i}$ such that $z_{i,j} = x_i \cdot y_j + 2^{t+\ell} \cdot q \cdot y_j + \delta_{j\to i}$ and $z_{i,j} = x'_i \cdot y'_j + 2^{t+\ell} \cdot q \cdot y'_j + \delta'_{j\to i}$ for *all* $i, j \in [n]$ (with $i \neq j$). Thus indistinguishability holds, even given all output shares $c_1, \ldots, c_n$. We stress that the above holds only if there is no reduction modulo $N_j$ of any value; this is crucial since otherwise the fact that the two sets of inputs define the same output modulo $q$ does not imply that they define the same output in the Paillier decryptions. In order to see that no reduction modulo $N$ takes place at any point, recall that the zero-knowledge proofs guarantee that all values provided by parties are

in the defined ranges *and* that the parties add appropriate values ($2^{t+\ell} \cdot q \cdot y_j$ and so on) to make sure that any negative values used are converted to positive values that are equivalent modulo $q$. See more discussion on this after the protocol description in Section 6.2.1.

### 6.2.3 The Zero-Knowledge Proof of Correct Paillier Generation

We use the new proof of [21] in order to certify that the Paillier key was created correctly. Observe that their proof certifies that RSA is a permutation by proving that $\gcd(N, \phi(N)) = 1$ and $x^e \bmod N$ is a 1–1 permutation over $\mathbb{Z}_N$. We actually don't need this second part of the statement. However, they only prove a promise problem so that the proof is only guaranteed to fail if $x^e \bmod N$ is *not* a 1–1 permutation over $\mathbb{Z}_N$. For our application, we need soundness to hold if $\gcd(N, \phi(N)) \neq 1$. Fortunately, we can easily adapt this by take $e = N$ and then it suffices to take $m_1 = m_2$, meaning that we just need to show that $m_2$ random values have $N$th roots. (This also means that we only need to compute the $\sigma_j$ values taking $e = N$; see Step 3 of the basic protocol in [21]). Concretely, we used the parameters $\alpha = 6370$ and $m_2 = 11$, meaning that the cost of the protocol is 11 modular exponentiations, and the bandwidth is 11 elements of $\mathbb{Z}_N$.

### 6.2.4 Zero-Knowledge Equality of Paillier Encryption and Pedersen Commitment

In existing efficient range proofs, soundness requires that the prover use a commitment to the value over a group of unknown order (this essentially forces the prover to work over the integers and not modulo the order of the group). Since we wish to have the prover prove that a value in a Paillier encryption is within a range, we first need the prover to commit to the same value in a Pedersen commitment (modulo a modulus for which it does not know the factorization) and prove that it committed to the same value. Then, the range proof can be carried out on the Pedersen commitment. In this section, we present a zero-knowledge proof that the same value was used in a Paillier encryption and Pedersen commitment; this proof is taken from [2, Appendix A] with appropriate modifications for Paillier.

Formally, we describe a Sigma protocol for the relation:

$$\mathcal{R}_{\mathsf{eq}} = \left\{ \Big( (C, \tilde{C}, N, \tilde{N}, g, h), (x, r, \rho) \Big) \Big| C = (1+N)^x \cdot r^N \bmod N^2 \ \wedge \ \tilde{C} = g^x \cdot h^\rho \bmod \tilde{N} \right\}$$

where $g, h \in \mathbb{Z}^*_{\tilde{N}}$ are random (with the discrete log of $h$ relative to $g$ and vice versa unknown to the prover). The zero-knowledge property holds for $x \in \mathbb{Z}_q$ (as is the case for our application); technically, we do not include this requirement in the relation since that would mean that soundness must ensure that $x$ is in this range but this is what is covered by the range proof that follows this one. The proof is parameterized by security parameter $t$; concretely we take $t = 128$ and compile the Sigma protocol into a zero-knowledge proof of knowledge using the Fiat-Shamir transform (this is true for all Sigma protocols that we use).

1. **Prover $P$'s first message:** $P$ chooses random $\alpha \in \mathbb{Z}_{q \cdot 2^{t+\ell}}$, $\beta \in \mathbb{Z}_N$ and $\gamma \in \mathbb{Z}_{\tilde{N}}$. Then, $P$ computes $A = (1+N)^\alpha \cdot \beta^N \bmod N^2$ and $B = g^\alpha \cdot h^\gamma \bmod \tilde{N}$, and sends $(A, B)$ to $V$.

2. **Verifier $V$'s challenge:** $V$ chooses a random $e \in \{0,1\}^{2t}$ and sends it to $P$.

3. **$P$'s second message:** $P$ computes $z_1 = \alpha + e \cdot x$ (over the integers), $z_2 = \beta \cdot r^e \bmod N$, and $z_3 = \gamma + \rho \cdot e$. $P$ sends $(z_1, z_2, z_3)$ to $V$.

4. $V$**'s verification:** $V$ accepts if and only if

$$(1+N)^{z_1} \cdot z_2^N = A \cdot C^e \bmod N^2 \quad \text{and} \quad g^{z_1} \cdot h^{z_3} = B \cdot (\tilde{C})^e \bmod \tilde{N}.$$

We remark that when using the Fiat-Shamir paradigm, $P$ computes $e = H(C, \tilde{C}, N, \tilde{N}, A, B)$ and defines the proof to be $\pi = (e, z_1, z_2, z_3)$ only. Then, $V$ derives $A = (1+N)^{z_1} \cdot {z_2}^N \cdot (C^e)^{-1} \bmod N^2$ and $B = g^{z_1} \cdot h^{z_3} \cdot (\tilde{C}^e)^{-1} \bmod \tilde{N}$, and verifies the hash.

### 6.2.5 Range Proof for Pedersen with Slack (ZK-CFT)

This proof is from [7], as described in [2, Section 1.2.3].

**Notation.** Let $t, \ell, s$ be security parameters (in the implementation, we take $t = s = 128$ and $\ell = 80$). Let $\tilde{N}$ be the modulus for Pedersen, and let $g, h \in \mathbb{Z}_{\tilde{N}}$ be random (with the discrete log of $h$ to base $g$ and vice versa unknown to the prover). We denote by $\mathsf{Ped}_{g,h,\tilde{N}}(x;\rho)$ the Pedersen commitment of $x$ with randomness $\rho$; i.e., $\mathsf{Ped}_{g,h,\tilde{N}}(x;\rho) = g^x \cdot h^\rho \bmod \tilde{N}$.

The prover wishes to prove that $x$ committed to in the Pedersen commitment lies in the interval $[0, q)$. Formally, we describe a Sigma protocol for the relation:

$$\mathcal{R}_{\mathsf{PedRange}} = \left\{ \left( (\tilde{C}, \tilde{N}, g, h), (x, \rho) \right) \middle| \tilde{C} = g^x \cdot h^\rho \bmod \tilde{N} \ \wedge \ x \in \mathbb{Z}_q \right\}$$

although the soundness of the protocol only guarantees that $x \in \left(-2^{t+\ell} \cdot q, 2^{t+\ell} \cdot q\right)$; this is what is meant by "slack" in the protocol.

1. $P$**'s first message:** $P$ chooses random $a \leftarrow \mathbb{Z}_{2^{t+\ell} \cdot q}$ and $\alpha \leftarrow \mathbb{Z}_{\tilde{N}}$ and computes the commitment $A = \left[ g^a \cdot h^\alpha \bmod \tilde{N} \right] = \mathsf{Ped}_{g,h,\tilde{N}}(a;\alpha)$. $P$ sends $A$ to $V$.
2. $V$**'s challenge:** $V$ chooses a random $e \leftarrow \mathbb{Z}_{2^t}$ and sends it to $P$.
3. $P$**'s second message:** $P$ computes $z_1 = a + x \cdot e$ (over the integers) and $z_2 = \alpha + e \cdot \rho$. $P$ sends $(z_1, z_2)$ to $V$.
4. $V$**'s verification:** $V$ verifies that $z_1 \in [2^t \cdot q, 2^{t+\ell} \cdot q)$ and that $g^{z_1} \cdot h^{z_2} = A \cdot \tilde{C}^e \bmod \tilde{N}$.

We remark that when using the Fiat-Shamir paradigm, the proof is $\pi = (e, z_1, z_2)$, and $V$ derives $A = g^{z_1} \cdot h^{z_2} \cdot (\tilde{C}^e)^{-1} \bmod \tilde{N}$ and verifies the hash.

### 6.2.6 Range Proof for Paillier with Slack

In this section, we describe the full range proof for Paillier (with slack). The prover wishes to prove that $x$ encrypted in a Paillier ciphertext $C$ lies in the interval $[0, q)$. Formally, we describe a Sigma protocol for the relation:

$$\mathcal{R}_{\mathsf{PaillRange}} = \left\{ ((C, N), (x, r)) \middle| C = (1+N)^x \cdot r^N \bmod N^2 \ \wedge \ x \in \mathbb{Z}_q \right\}.$$

As above, the soundness of the protocol only guarantees that $x \in \left(-2^{t+\ell} \cdot q, 2^{t+\ell} \cdot q\right)$.

This proof simply combines the proofs of Section 6.2.4 and 6.2.5. That is, the prover generates a Pedersen commitment, proves that it commits to the same value as encrypted in the Paillier ciphertext and then proves that the committed value is in the appropriate range. Formally:

1. **Verifier message:** The verifier $V$ sends the prover $P$ parameters $\tilde{N}, g, h$ for Pedersen commitments ($V$ chooses $\tilde{N} = p \cdot q$ where $p = 2p'+1, q = 2q'+1$ are safe primes, and sets $g, h$ to be random elements $QR_{\tilde{N}}$; since $\tilde{N}$ is a safe prime, the number of generators of $QR_{\tilde{N}}$ is $\phi(p' \cdot q')$ and so the probability of a random quadratic residue not being a generator is negligible).

2. **Prover message:** $P$ chooses a random $\rho \leftarrow \mathbb{Z}_{\tilde{N}}$ and computes $\tilde{C} = g^x \cdot h^\rho \bmod \tilde{N}$. Then, $P$ proves in zero-knowledge that $\Big((C, \tilde{C}, N, \tilde{N}, g, h), (x, r, \rho)\Big) \in \mathcal{R}_{\mathsf{eq}}$ (using the proof of Section 6.2.4) and that $\Big((\tilde{C}, N, \tilde{N}, g, h), (x, \rho)\Big) \in \mathcal{R}_{\mathsf{PedRange}}$ (using the proof of Section 6.2.5).

### 6.2.7 Zero-Knowledge Proof of Pailler-Pedersen Range-Bounded Affine Operation

In this section, we describe a zero-knowledge proof that shows that a value $D$ was generated from $C$ by carrying out a homomorphic affine operation using values $y, \delta$ in a given range (such an affine operation is defined by $x \cdot y + \delta$ where $x$ is the value encrypted in $C$). Formally, we are interested in the relation:

$$\mathcal{R}_{\mathsf{AffineRange}} = \left\{ ((C, D, N), (y, \delta)) \middle| D = C^y \cdot (1+N)^\delta \bmod N^2 \ \wedge \ y \in \mathbb{Z}_q \ \wedge \ \delta \in \mathbb{Z}_{q^2 \cdot 2^{t+\ell+s}} \right\}.$$

The proof is a generalization of the proofs in Sections 6.2.4 and 6.2.5, and works as follows:

1. **Verifier first message:** The verifier $V$ sends the prover $P$ parameters $\tilde{N}, g, h$ for Pedersen commitments (as in Section 6.2.6). (Note: in our specific usage, we can set $N = \tilde{N}$ since the Paillier private key belongs to the verifier.)

2. **First prover message:** $P$ chooses random $\alpha \leftarrow \mathbb{Z}_{q \cdot 2^{t+\ell}}$, $\beta \leftarrow \mathbb{Z}_{q^2 \cdot 2^{2t+2\ell+s}}$, and $\rho_1, \rho_2, \rho_3, \rho_4 \in \mathbb{Z}_{\tilde{N}}$. Then, $P$ computes $A = C^\alpha \cdot (1+N)^\beta \bmod N^2$ and $B_1 = g^\alpha \cdot h^{\rho_1} \bmod \tilde{N}$, $B_2 = g^\beta \cdot h^{\rho_2} \bmod \tilde{N}$, $B_3 = g^y \cdot h^{\rho_3} \bmod \tilde{N}$ and $B_4 = g^\delta \cdot h^{\rho_4} \bmod \tilde{N}$. $P$ sends $(A, B_1, B_2, B_3, B_4)$ to $V$.

3. **Verifier challenge:** $V$ sends the prover a random $e \in \mathbb{Z}_t$.

4. **Second prover message:** $P$ computes $z_1 = \alpha + e \cdot y$, $z_2 = \beta + e \cdot \delta$, $z_3 = \rho_1 + e \cdot \rho_3$ and $z_4 = \rho_2 + e \cdot \rho_4$. Send $(z_1, z_2, z_3, z_4)$ to the verifier.

5. **Proof verification:** Accept if and only if

   (a) $z_1 \in [2^t \cdot q, 2^{t+\ell} \cdot q)$

   (b) $z_2 \in [2^t \cdot q^2 \cdot 2^{2t+\ell+s}, 2^{t+\ell} \cdot q^2 \cdot 2^{2t+\ell+s}) = [q^2 \cdot 2^{3t+\ell+s}, q^2 \cdot 2^{3t+2\ell+s})$

   (c) $C^{z_1} \cdot (1+N)^{z_2} = A \cdot D^e \bmod N^2$

   (d) $g^{z_1} \cdot h^{z_3} = B_1 \cdot {B_3}^e \bmod \tilde{N}$.

   (e) $g^{z_2} \cdot h^{z_4} = B_2 \cdot {B_4}^e \bmod \tilde{N}$.

We remark that here $B_3, B_4$ need to be sent, as well as $(e, z_1, z_2, z_3, z_4)$ when using Fiat-Shamir. Observe that the range proof is based on verifying the ranges of $z_1, z_2$, and it is valid since these are proven to be the values committed in $B_3, B_4$ (for which the prover does not know the order of the group).

# 7 Checking Diffie-Hellman Tuples

In this section, we show how to securely compute the $\mathcal{F}_{\text{checkDH}}$ functionality defined in Section 4.2. The idea behind the protocol is as follows. Each party holds $\mathcal{P}_1, \ldots, \mathcal{P}_n$ such that $\mathcal{P} = \sum_{\ell=1}^{n} \mathcal{P}_\ell$, a tuple $(G, \mathcal{P}, U, V)$, and a share $d_i$ such that $\mathcal{P}_i = d_i \cdot G$. The aim of the parties is to verify that $V = d \cdot U$ where $d = \sum_{\ell=1}^{n} d_i$. Naively, each party $P_i$ can send $U_i = d_i \cdot G$ to all other parties with a zero-knowledge proof that $(G, U, \mathcal{P}_i, U_i)$ is a Diffie-Hellman tuple (using witness $d_i$). This ensures that $P_i$ computed $U_i = d_i \cdot G$ with the same $d_i$ that defines $\mathcal{P}_i = d_i \cdot G$. Then, using all $U_i$ received, each party $P_i$ can compute $\sum_{\ell=1}^{n} U_\ell = \sum_{\ell=1}^{n} d_\ell \cdot U = d \cdot U$ and check that it equals $V$. If the input is indeed a Diffie-Hellman tuple, then this equality will hold.

Unfortunately, however, the above method is not secure. This is due to the fact that by the definition of $\mathcal{F}_{\text{checkDH}}$ (and what we need for securely computing $\mathcal{F}_{\text{mult}}$), the parties should learn nothing but whether or not the input is a Diffie-Hellman tuple. In order to achieve this, the parties first need to randomize the ciphertext, and only then can they proceed as above. This randomization is carried out by having each party $P_i$ choosing random $\alpha_i, \rho_i \in \mathbb{Z}_q$ and computing $(U_i, V_i) = (\alpha_i \cdot U + \rho_i \cdot G, \alpha_i \cdot V + \rho_i \cdot \mathcal{P})$. As described in Section 3.3 for relation $R_{RE}$ (and proven in the proof of Theorem B.1; see Eq. Eq. (1)), this has the property that if $(G, \mathcal{P}, U, V)$ is a Diffie-Hellman tuple, then all $(G, \mathcal{P}, U_i, V_i)$ are Diffie-Hellman tuples, and so $(G, \mathcal{P}, \sum_{\ell=1}^{n} U_\ell, \sum_{\ell=1}^{n} V_\ell)$ is a Diffie-Hellman tuple. In contrast, if $(G, \mathcal{P}, U, V)$ is not a Diffie-Hellman tuple, then the $(G, \mathcal{P}, U_i, V_i)$ tuples generated by the honest parties are such that $U_i$ and $V_i$ are truly random and independent. Thus, the resulting sum will be a Diffie-Hellman tuple with probability only $1/q$. The protocol for securely computing $\mathcal{F}_{\text{checkDH}}$ is described in Protocol 7.1.

**PROTOCOL 7.1 (Securely Computing $\mathcal{F}_{\text{checkDH}}$ in the $\mathcal{F}_{\mathsf{zk}}, \mathcal{F}_{\mathsf{com\text{-}zk}}$-Hybrid Model)**

**Input:** Each party $P_i$ holds a private key $d_i$.

**Auxiliary Input:** Each party holds public keys $\vec{\mathcal{P}} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ and a pair $(U, V)$; denote $\mathcal{P} = \sum_{\ell=1}^{n} \mathcal{P}_\ell$. In addition, each party holds a unique session identifier *sid*.

**The** init **subprotocol:** The init subprotocol is *identical* to init of $\mathcal{F}_{\text{mult}}$; see Protocol 4.3. (Since it is exactly the same protocol and exactly the same values, this is run once for both functionalities.)

**The** check **subprotocol:** Each party $P_i$ works as follows:

1. *Round 1:* Party $P_i$ chooses random $\alpha_i, \rho_i \in \mathbb{Z}_q$ and computes $(U_i, V_i) = (\alpha_i \cdot U + \rho_i \cdot G, \alpha_i \cdot V + \rho_i \cdot \mathcal{P})$. Then, $P_i$ sends $(\mathsf{ComProve}, sid, i, (U, V, U_i, V_i), (\alpha_i, \rho_i))$ to $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{RE}}$.
2. *Round 2:* Upon receiving $(\mathsf{ProofReceipt}, sid, j)$ for all $j \in [n]$, party $P_i$ sends $(\mathsf{DecomProof}, sid)$ to $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{RE}}$.
3. *Round 3:*
   (a) $P_i$ receives $(\mathsf{DecomProof}, sid, j, (U, V, U_j, V_j), \beta_j)$ from $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{RE}}$ for all $j \in [n]$. If some $\beta_j = 0$, then $P_i$ aborts.
   (b) $P_i$ locally computes $U' = \sum_{i=1}^{n} U_i$ and $V' = \sum_{i=1}^{n} V_i$.
   (c) $P_i$ computes $U_i' = d_i \cdot U'$ and sends $(\mathsf{proof}, sid, i, (G, U', \mathcal{P}_i, U_i'), d_i)$ to $\mathcal{F}_{\mathsf{zk}}^{R_{DH}}$.
4. *Output:* $P_i$ receives $(\mathsf{proof}, sid, j, (G, U', \mathcal{P}_i, U_i'), \beta_j')$ from $\mathcal{F}_{\mathsf{zk}}^{R_{DH}}$ for all $j \in [n]$. If any $\beta_j' = 0$ then $P_i$ aborts. $P_i$ checks that $V' = \sum_{\ell=1}^{n} U_\ell'$. If equality holds, it outputs accept; else, it outputs reject.

**Complexity.** The cost of Protocol 7.1 is $11 + 10(n-1)$ exponentiations per party, each party sending 7 group elements (or equivalent) to each other party, and 3 rounds of communication.

**Proposition 7.2** *Assume that the DDH problem is hard in $\mathbb{G}$. Then, Protocol 7.1 securely computes Functionality 4.2 with abort in the $(\mathcal{F}_{\mathsf{zk}}^{R_{RE}}, \mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{DH}})$-hybrid model, in the presence of a malicious adversary corrupting any $t < n$ parties, with point-to-point channels.*

**Proof:** Let bad be the event that $(G, \mathcal{P}, U, V)$ is not a Diffie-Hellman tuple but the honest parties output accept. This event happens when $V \neq d \cdot U$ but $V' = d \cdot U'$ (where $d = \sum_{i=1}^{n} d_i$). Since the corrupted parties are committed to their $U_i, V_i$ values before seeing the honest parties' values $U_j, V_j$, it follows that $U', V'$ is a true rerandomization of $U, V$. Thus, if $(G, \mathcal{P}, U, V)$ is not a Diffie-Hellman tuple, it follows that $U', V'$ are uniformly and independently distributed in $\mathbb{G}$. Thus $V' = d \cdot U'$ with probability exactly $1/q$, implying that $\Pr[\mathsf{bad}] = 1/q$.

We now describe the simulator $\mathcal{S}$. Let $\mathcal{A}$ be the adversary in the real protocol. As in all our previous proofs, we use $I$ to denote the set of indexes of the corrupted parties and $J$ for the set of the honest parties. The simulation of init is identical to that of init in $\mathcal{F}_{\text{mult}}$ and so is not repeated here. We describe now the simulator for check. $\mathcal{S}$ sends $(\mathsf{check}, i, U, V, d_i)$ to $\mathcal{F}_{\text{checkDH}}$ for every $i \in I$, using the same $d_i$ as in the init subprotocol. $\mathcal{S}$ receives back accept or reject.

1. $\mathcal{S}$ invokes $\mathcal{A}$ and simulates $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{RE}}$ sending $P_i$ the message $(\mathsf{ProofReceipt}, sid, j)$ for every $i \in I$ and $j \in J$.

2. $\mathcal{S}$ receives the messages $(\mathsf{ComProve}, sid, i, (U, V, U_i, V_i), (\alpha_i, \rho_i))$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{RE}}$ for every $i \in I$.

3. If $\mathcal{S}$ received accept from $\mathcal{F}_{\text{checkDH}}$, then it chooses a random $\gamma \in \mathbb{Z}_q$ and sets $U' = \gamma \cdot G$ and $V' = \gamma \cdot \mathcal{P}$ (and so $(G, \mathcal{P}, U', V')$ is a random Diffie-Hellman tuple). Let $j^* \in J$. Then, for every $j \in J \setminus \{j^*\}$, simulator $\mathcal{S}$ computes $U_j, V_j$ like an honest $P_j$. Then, $S^*$ computes $U_{j^*} = U' - \sum_{\ell \neq j^*} U_\ell$ and $V_{j^*} = V' - \sum_{\ell \neq j^*} V_\ell$.

4. If $\mathcal{S}$ received reject from $\mathcal{F}_{\text{checkDH}}$, then $\mathcal{S}$ chooses independent random $U'_j, V'_j \in \mathbb{G}$ for every $j \in J$ (and sets $U' = \sum_{\ell=1}^{n} U'_\ell$ and $V' = \sum_{\ell=1}^{n} V'_\ell$).

5. $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{zk}}^{R_{RE}}$ sending $P_i$ the message $(\mathsf{DecomProof}, sid, j, (U, V, U_j, V_j), 1)$ for every $j \in J$.

6. $\mathcal{S}$ receives the message $(\mathsf{DecomProof}, sid, i)$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{RE}}$ for every $i \in I$. If $(U_i, V_i) \neq (\alpha_i \cdot U + \rho_i \cdot G,\ \alpha_i \cdot V + \rho_i \cdot \mathcal{P})$ for some $i \in I$, then $\mathcal{S}$ simulates the honest parties aborting in the real world, sends abort to $\mathcal{F}_{\text{checkDH}}$ and halts.

7. $\mathcal{S}$ receives the messages $(\mathsf{DecomProof}, sid)$ from $\mathcal{A}$ for each corrupted party. Then, if $\mathcal{S}$ received accept from $\mathcal{F}_{\text{checkDH}}$, it computes $U'_i = d_i \cdot G$ for every $i \in I$, and chooses random $U'_j \in \mathbb{G}$ for each $j \in J$ under the constraint that $\sum_{\ell=1}^{n} U'_\ell = V'$. In contrast, if $\mathcal{S}$ received reject from $\mathcal{F}_{\text{checkDH}}$, then $\mathcal{S}$ just chooses random $U'_j \in \mathbb{G}$ for every $j \in J$. If $V' = \sum_{\ell=1}^{n} U'_\ell$, then $\mathcal{S}$ outputs fail and halts.

8. $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{zk}}^{R_{DH}}$ handing $\mathcal{A}$ the messages $(\mathsf{proof}, sid, j, (G, U', \mathcal{P}_j, U'_j), 1)$ for every $j \in J$.

9. $\mathcal{S}$ receives the messages $(\mathsf{proof}, sid, i, (G, U', \mathcal{P}_i, U'_i), d_i)$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{zk}}^{R_{DH}}$ for every $i \in I$. Simulator $\mathcal{S}$ checks that $sid$ is correct, $U'$ and $\mathcal{P}_i$ are correct and that $\mathcal{P}_i = d_i \cdot G$ and $U'_i = d_i \cdot U'$ for every $i \in I$. If not, it simulates the honest parties aborting in the protocol, sends $\mathsf{abort}$ to $\mathcal{F}_{\text{checkDH}}$, and aborts.

10. If no $\mathsf{abort}$ took place (or $\mathsf{fail}$), then $\mathcal{S}$ sends $(\mathsf{continue}, j)$ to $\mathcal{F}_{\text{checkDH}}$ for every $j \in J$ (for the honest parties to receive output).

The proof below relies on the fact that if $(G, \mathcal{P}, U, V)$ is a Diffie-Hellman tuple, then $(G, \mathcal{P}, U', V')$ is a random Diffie-Hellman tuple, and if $(G, \mathcal{P}, U, V)$ is not a Diffie-Hellman tuple then $U', V'$ are independent random group elements. This is guaranteed in the protocol in the $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{RE}}$-hybrid model, since the adversary is (perfectly) committed to its $U_i, V_i$ before seeing the honest parties' $U_j, V_j$ values. We consider two cases:

*Case 1 – the output from $\mathcal{F}_{\text{checkDH}}$ is $\mathsf{accept}$:* This means that $V = \sum_{\ell=1}^{n}(d_\ell \cdot U)$ and so $V' = \sum_{\ell=1}^{n}(d_\ell \cdot U') = \sum_{\ell=1}^{n} U'_\ell$. Observe that if there are $t = n - 1$ corrupted parties and exactly one honest party $P_j$, the simulation is perfect. This follows from the fact that $\mathcal{S}$ can compute $U'_j = V' - \sum_{\ell \in [n] \setminus \{j\}} U'_\ell$ as it knows in advance all $U'_\ell$s of the corrupted parties and $(G, \mathcal{P}, U', V')$ is a random Diffie-Hellman tuple, exactly like in a real execution. Thus, we focus only on the case where there are at least *two* honest parties. In this case, we need to show that the output distributions from the real and simulated executions are indistinguishable.

We prove this by reducing it to the DDH assumption. Let $\mathcal{D}$ be a distinguisher who receives $(\mathbb{G}, G, q)$ and a series of $|J| - 1$ tuples $(G, \hat{U}, \hat{P}_j, \hat{U}_j)$ for $j \in J \setminus \{j^*\}$, where all tuples are either random, or are Diffie-Hellman tuples (with $\hat{U} \in \mathbb{G}$ is random and the same in all tuples). We denote by $j^*$ the index of one of the honest parties. The distinguisher $\mathcal{D}$ works as follows:

1. $\mathcal{D}$ works exactly like $\mathcal{S}$ in the $\mathsf{init}$ phase, setting $\mathcal{P}_i = d_i \cdot G$ for every $i \in I$. However, for $j \in J \setminus \{j^*\}$, $\mathcal{D}$ sets $\mathcal{P}_j = \hat{\mathcal{P}}_j$ instead of choosing it at random. In addition, $\mathcal{D}$ sets $(U, V)$ to be a random Diffie-Hellman tuple. Finally, $\mathcal{D}$ chooses a random $d \in \mathbb{Z}_q$ and sets $\mathcal{P}_{j^*} = d \cdot G - \sum_{\ell \neq j^*} \mathcal{P}_\ell$.

2. $\mathcal{D}$ invokes $\mathcal{A}$ on input $(\mathsf{check}, i, U, V)$ and runs the simulator instructions, withe following changes:

   (a) Instead of choosing a random Diffie-Hellman tuple $(U', V')$ in Step 3 of the simulation, $\mathcal{D}$ defines $U' = \hat{U}$ and $V' = d \cdot U'$. Then, it chooses the $U_j, V_j$ values of the honest parties as described in the simulation.

   (b) When defining $U'_j$ for $j \in J$ in Step 8 of the simulation, $\mathcal{D}$ sets $U'_j = \hat{U}_j$ for every $j \in J \setminus \{j^*\}$ and sets $U'_{j^*}$ like in the simulation (i.e, when choosing the $U'_j$ under the constraint, all $U'_j$ for $j \neq j^*$ are set to $\hat{U}_j$, and $U'_{j^*}$ is chosen to fulfill the constraint).

Observe that if the tuples $\mathcal{D}$ received are *not* Diffie-Hellman tuples, then the distribution generated is exactly that of the simulator. This is because all $U'_j$ for $j \in J \setminus \{j^*\}$ are random, exactly like for $\mathcal{S}$, and this is the only difference. (Note that $U', V'$ generated by $\mathcal{D}$ are exactly like $\mathcal{S}$ because they are a random Diffie-Hellman tuple, computing using a random $d$. Likewise, all $\mathcal{P}_i$ have the same distribution.)

In contrast, if the tuples $\mathcal{D}$ received are Diffie-Hellman tuples, then the distribution is exactly like a real execution for case that the output is $\mathsf{accept}$. This is due to the fact that all values are Diffie-Hellman tuples, just like honest parties produce.

Case 2: the output from $\mathcal{F}_{\text{checkDH}}$ is reject: This case differs from the previous one in that the simulator chooses $U', V'$ to be random elements (and not a random Diffie-Hellman tuple). As above, the distribution is identical to a real execution, except for the $U'_j$ values for $j \in J$ which are randomly chosen by $\mathcal{S}$. The reduction to DDH here is very similar to above. The only difference in this case can occur when the bad event happens, in which case the parties accept in a real execution but reject in the ideal world, indicated by $\mathcal{S}$ outputting fail. As we have noted above, this happens with probability only $1/q$ which is negligible.

This concludes the proof. ■

## Acknowledgements

We thank Valery Osheter from Unbound Tech Ltd. for the implementation of the ECDSA protocol and for running the experiments, and Rosario Gennaro and Steven Goldfeder for helpful discussions.

## References

[1] O. Blazy, C. Chevalier, D. Pointcheval and D. Vergnaud. Analysis and Improvement of Lindell's UC-Secure Commitment Schemes. In *ACNS 2013*, Springer (LNCS 7954), pages 534–551, 2013.

[2] F. Boudot. Efficient Proofs That a Committed Number Lies in an Interval. In *EUROCRYPT 2000*, Springer (LNCS 1807), pages 431–444, 2000.

[3] C. Boyd. Digital Multisignatures. In *Cryptography and Coding*, pages 241–246, 1986.

[4] D. Boneh, R. Gennaro and S. Goldfeder. Using Level-1 Homomorphic Encryption To Improve Threshold DSA Signatures For Bitcoin Wallet Security In *Latincrypt 2017*.

[5] R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[6] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In 42*nd FOCS*, pages 136–145, 2001. Full version available at `http://eprint.iacr.org/2000/067`.

[7] A. Chan, Y. Frankel and Y. Tsiounis. Easy Come - Easy Go Divisible Cash. Updated version with corrections, GTE Tech. Rep. (1998).

[8] T. Chou and C. Orlandi. The Simplest Protocol for Oblivious Transfer. In *LATINCRYPT* 2015.

[9] R.A. Croft and S.P. Harris. Public-Key Cryptography and Reusable Shared Secrets. In *Cryptography and Coding*, pages 189–201, 1989.

[10] I. Damgård and M. Jurik. A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System. In *PKC 2001*, Springer (LNCS 1992), pages 119–136, 2001.

[11] Y. Desmedt. Society and Group Oriented Cryptography: A New Concept. In *CRYPTO'87*, Springer (LNCS 293), pages 120–127, 1988.

[12] Y. Desmedt and Y. Frankel. Threshold Cryptosystems. In *CRYPTO'89*, Springer (LNCS 435), pages 307–315, 1990.

[13] J. Doerner, Y. Kondi, E. Lee and a. shelat. Secure Two-party Threshold ECDSA from ECDSA Assumptions In the 39*th IEEE Symposium on Security and Privacy*, 2018.

[14] A. Fiat and A. Shamir: How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *CRYPTO 1986*, Springer (LNCS 263), pages 186–194, 1986.

[15] T. Frederiksen, Y. Lindell, V. Osheter and B. Pinkas. Fast Distributed RSA Key Generation for Semi-Honest and Malicious Adversaries. To appear at *CRYPTO 2018*.

[16] E. Fujisaki. Improving Practical UC-Secure Commitments Based on the DDH Assumption. In *SCN 2016*, Springer (LNCS 9841), pages 257–272, 2016.

[17] R. Gennaro, S. Jarecki, H. Krawczyk and T. Rabin. Robust Threshold DSS Signatures. In *EUROCRYPT'96*, Springer (LNCS 1070), pages 354–371, 1996.

[18] R. Gennaro, S. Goldfeder and A. Narayanan: Threshold-Optimal DSA/ECDSA Signatures and an Application to Bitcoin Wallet Security. In *ACNS 2016*, pages 156–174, 2016.

[19] R. Gennaro and S. Goldfeder. Fast Multiparty Threshold ECDSA with Fast Trustless Setup. In *ACM CCS 2018*.

[20] N. Gilboa. Two Party RSA Key Generation. In *CRYPTO 1999*, Springer (LNCS 1666), pages 116–129, 1999

[21] S. Goldberg, L. Reyzin, O. Sagga and F. Baldimtsi. Certifying RSA Public Keys with an Efficient NIZK. *Cryptology ePrint Archive: Report 2018/057*, 2018.

[22] S. Goldfeder. Personal communication, April 2018.

[23] O. Goldreich. *Foundations of Cryptography: Volume 2 – Basic Applications*. Cambridge University Press, 2004.

[24] S. Goldwasser and Y. Lindell. Secure Computation Without Agreement. *Journal of Cryptology,* 18(3):247–287, 2005.

[25] C. Hazay and Y. Lindell. *Efficient Secure Two-Party Protocols: Techniques and Constructions*. Springer, November 2010.

[26] M. Keller, E. Orsini, and P. Scholl. Actively Secure OT Extension With Optimal Overhead. In *CRYPTO 2015*, Springer (LNCS 9215), 724–741, 2015.

[27] Y. Lindell. Highly-Efficient Universally-Composable Commitments Based on the DDH Assumption. In *EUROCRYPT 2011*, Springer (LNCS 6632), pages 446–466, 2011.

[28] Y. Lindell. Fast Secure Two-Party ECDSA Signing. In *CRYPTO 2017*, Springer (LNCS 10402), pages 613–644, 2017.

[29] P.D. MacKenzie and M.K. Reiter. Two-party generation of DSA signatures. *International Journal of Information Security*, 2(3-4):218–239, 2004. An extended abstract appeared at *CRYPTO 2001*.

[30] S. Micali, R. Pass and A. Rosen. Input-Indistinguishable Computation. In the 47*th FOCS*, pages 367–378, 2006.

[31] P. Paillier. Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT99*, Springer (LNCS 1592), pages 223–238, 1999.

[32] C.P. Schnorr. Efficient Identification and Signatures for Smart Cards. In *CRYPTO 1989*, Springer (LNCS 435), pages 239–252, 1990.

[33] V. Shoup. Practical Threshold Signatures. In *EUROCRYPT 2000*, Springer (LNCS 1807), pages 207–220, 2000.

[34] V. Shoup and R. Gennaro. Securing Threshold Cryptosystems against Chosen Ciphertext Attack. In *EUROCRYPT 1998*, Springer (LNCS 1403), pages 1–16, 1998.

[35] Porticor, www.porticor.com.

[36] Unbound Tech, www.unboundtech.com.

[37] Sepior, www.sepior.com.

# A Zero-Knowledge Proofs

In this section, we present Sigma protocols for the non-standard proofs that we need for our protocol. These Sigma protocols can be transformed into interactive zero-knowledge proofs of knowledge using standard techniques (e.g., see [25, Sec. 6.5.2]), and into non-interactive zero-knowledge proofs of knowledge in the random-oracle model using the Fiat-Shamir transform [14].

## A.1 Rerandomization of Tuple – $R_{RE}$

We present a Sigma protocol for the relation

$$R_{RE} = \{(\mathbb{G}, G, q, (\mathcal{P}, A, B, A', B'), (r, s)) \mid A' = r \cdot G + s \cdot A \ \wedge \ B' = r \cdot \mathcal{P} + s \cdot B\}.$$

that proves that $(G, \mathcal{P}, A', B')$ is a rerandomization of $(G, \mathcal{P}, A, B)$. See Protocol A.1 for the full description.

**PROTOCOL A.1 (Sigma Protocol for Relation $R_{RE}$)**

Upon joint input $(\mathbb{G}, G, q, \mathcal{P}, A, B, A', B')$ and witness $(r, s)$ for the prover $P$, the parties work as follows:

**First prover message:**

1. The prover $P$ chooses random $\sigma, \tau \leftarrow \mathbb{Z}_q$, and computes $X = \sigma \cdot G + \tau \cdot A$ and $Y = \sigma \cdot \mathcal{P} + \tau \cdot B$.
2. $P$ sends $(X, Y)$ to the verifier $V$.

**Verifier challenge:** $V$ chooses a random $e \in \{0,1\}^{\kappa_s}$ (for statistical security parameter $\kappa_s$), and sends it to $P$.

**Second prover message:**

1. $P$ computes $z_1 = \sigma + e \cdot r \bmod q$ and $z_2 = \tau + e \cdot s \bmod q$.
2. $P$ sends $(z_1, z_2)$ to $V$.

**Verification:** $V$ accepts if and only if the following holds

1. $X = z_1 \cdot G + z_2 \cdot A - e \cdot A'$, and
2. $Y = z_1 \cdot \mathcal{P} + z_2 \cdot B - e \cdot B'$

We sketch the proof of security, proving completness, special soundness and honest-verifier zero-knowledge.

**Completeness:**

$$\begin{aligned} & z_1 \cdot G + z_2 \cdot A - e \cdot A' \\ &= (\sigma + e \cdot r) \cdot G + (\tau + e \cdot s) \cdot A - e \cdot (r \cdot G + s \cdot A) \\ &= \sigma \cdot G + e \cdot (r \cdot G) + \tau \cdot A + e \cdot (s \cdot A) - e \cdot (r \cdot G) - e \cdot (s \cdot A) \\ &= \sigma \cdot G + \tau \cdot A \\ &= X, \end{aligned}$$

and

$$\begin{aligned} & z_1 \cdot \mathcal{P} + z_2 \cdot B - e \cdot B' \\ &= (\sigma + e \cdot r) \cdot \mathcal{P} + (\tau + e \cdot s) \cdot B - e \cdot (r \cdot \mathcal{P} + s \cdot B) \\ &= \sigma \cdot \mathcal{P} + e \cdot (r \cdot \mathcal{P}) + \tau \cdot B + e \cdot (s \cdot B) - e \cdot (r \cdot \mathcal{P}) - e \cdot (s \cdot B) \\ &= \sigma \cdot P + \tau \cdot B \\ &= Y, \end{aligned}$$

and so the verifier accepts.

**Special soundness:** We show that given $(X, Y, e, e', z_1, z_2, z_1', z_2')$ such that $e \neq e' \bmod q$ and both $(X, Y, e, z_1, z_2)$ and $(X, Y, e', z_1', z_2')$ are accepting transcripts, it is possible to compute $(s, r)$ such that $A' = r \cdot G + s \cdot A$ and $B' = r \cdot \mathcal{P} + s \cdot B$. By the assumption that both transcripts are accepting we have that both

$$e \cdot A' = z_1 \cdot G + z_2 \cdot A - X \quad \text{and} \quad e \cdot B' = z_1 \cdot \mathcal{P} + z_2 \cdot B - Y,$$

and

$$e' \cdot A' = z_1' \cdot G + z_2' \cdot A - X \quad \text{and} \quad e' \cdot B' = z_1' \cdot \mathcal{P} + z_2' \cdot B - Y,$$

Subtracting the equations from each other, we have

$$(e - e') \cdot A' = (z_1 - z_1') \cdot G + (z_2 - z_2') \cdot A$$

and

$$(e - e') \cdot B' = (z_1 - z_1') \cdot \mathcal{P} + (z_2 - z_2') \cdot B$$

Thus, setting $r = (z_1 - z_1') \cdot (e - e')^{-1} \bmod q$ and $s = (z_2 - z_2') \cdot (e - e')^{-1} \bmod q$, we have that

$$A' = r \cdot G + s \cdot A \quad \text{and} \quad B' = r \cdot \mathcal{P} + s \cdot B$$

as required. (Observe that since $e \neq e' \bmod q$, the value $(e-e')^{-1} \bmod q$ exists and can be efficiently computed.)

**Honest-verifier zero knowledge:** Given $e$, the simulator chooses random $z_1, z_2 \leftarrow \mathbb{Z}_q$ and computes

$$X = z_1 \cdot G + z_2 \cdot A - e \cdot A' \quad \text{and} \quad Y = z_1 \cdot \mathcal{P} + z_2 \cdot B - e \cdot B'.$$

If indeed there exist $(s, r)$ such that $A' = r \cdot G + s \cdot A$ and $B' = r \cdot \mathcal{P} + s \cdot B$, then

$$X = (z_1 - e \cdot r) \cdot G + (z_2 - e \cdot s) \cdot A$$

and

$$Y = (z_1 - e \cdot r) \cdot \mathcal{P} + (z_2 - e \cdot s) \cdot B.$$

Now, set $\sigma = z_1 - e \cdot r$ and $\tau = z_2 - e \cdot s$, and observe that if $z_1, z_2$ are chosen at random (as indeed is the case by the simulator), then $\sigma$ and $\tau$ are also random. This implies that $z_1 = \sigma + e \cdot r$ and $z_2 = \tau + e \cdot s$ for random $\sigma, \tau$ and thus the distribution is *identical* to that of a real proof.

**Complexity.** The cost of the proof is four exponentiations for the prover and six for the verifier. When applying the Fiat-Shamir transform, the communication cost is three elements of $\mathbb{Z}_q$.

### A.2 Knowledge of $x$ in $\mathsf{EGexpEnc}_{\mathcal{P}}(x)$ – $R_{EG}$

We present a Sigma protocol for the relation

$$R_{EG} = \{((\mathbb{G}, G, q, \mathcal{P}, A, B), (x, r)) \mid (A, B) = \mathsf{EGexpEnc}_{\mathcal{P}}(x; r)\}$$

that expresses knowledge of the encrypted value in an ElGamal encryption-in-the-exponent ciphertext. See Protocol A.2 for the full description. Recall that $(A, B) = \mathsf{EGexpEnc}_{\mathcal{P}}(x; r)$ means that $A = r \cdot G$ and $B = r \cdot \mathcal{P} + x \cdot G$.

**PROTOCOL A.2 (Sigma Protocol for Relation $R_{EG}$)**

Upon joint input $(\mathbb{G}, G, q, \mathcal{P}, A, B)$ and witness $(x, r)$ for the prover $P$, the parties work as follows:

**First prover message:**

1. The prover $P$ chooses random $\sigma, \rho \leftarrow \mathbb{Z}_q$, and computes $X = \sigma \cdot G$ and $Y = \sigma \cdot \mathcal{P} + \rho \cdot G$.
2. $P$ sends $(X, Y)$ to the verifier $V$.

**Verifier challenge:** $V$ chooses a random $e \in \{0,1\}^{\kappa_s}$ (for statistical security parameter $\kappa_s$), and sends it to $P$.

**Second prover message:**

1. $P$ computes $z_1 = \sigma + e \cdot r \bmod q$ and $z_2 = \rho + e \cdot x \bmod q$.
2. $P$ sends $(z_1, z_2)$ to $V$.

**Verification:** $V$ accepts if and only if the following holds

1. $z_1 \cdot G = X + e \cdot A$, and
2. $z_1 \cdot \mathcal{P} + z_2 \cdot G = Y + e \cdot B$

We sketch the proof of security, proving completness, special soundness and honest-verifier zero-knowledge.

**Completeness:** If all messages are computed correctly, then:

$$z_1 \cdot G = (\sigma + e \cdot r) \cdot G = \sigma \cdot G + e \cdot (r \cdot G) = X + e \cdot A,$$

and

$$\begin{aligned} z_1 \cdot \mathcal{P} + z_2 \cdot G &= (\sigma + e \cdot r) \cdot P + (\rho + e \cdot x) \cdot G \\ &= \sigma \cdot \mathcal{P} + e \cdot (r \cdot \mathcal{P}) + \rho \cdot G + e \cdot (x \cdot G) \\ &= (\sigma \cdot \mathcal{P} + \rho \cdot G) + e \cdot (r \cdot \mathcal{P} + x \cdot G) \\ &= Y + e \cdot B, \end{aligned}$$

and so the verifier accepts.

**Special soundness:** We show that given $(X, Y, e, e', z_1, z_2, z_1', z_2')$ such that $e \neq e' \bmod q$ and both $(X, Y, e, z_1, z_2)$ and $(X, Y, e', z_1', z_2')$ are accepting transcripts, it is possible to compute $(x, r)$ where $(A, B) = \mathsf{EGexpEnc}_{\mathcal{P}}(x; r)$. By the assumption that both transcripts are accepting we have that both

$$z_1 \cdot G = X + e \cdot A \quad \text{and} \quad z_1 \cdot \mathcal{P} + z_2 \cdot G = Y + e \cdot B,$$

and

$$z_1' \cdot G = X + e' \cdot A \quad \text{and} \quad z_1' \cdot \mathcal{P} + z_2' \cdot G = Y + e' \cdot B.$$

Subtracting the equations from each other, we have

$$(z_1 - z_1') \cdot G = (e - e') \cdot A$$

and

$$(z_1 - z_1') \cdot \mathcal{P} + (z_2 - z_2') \cdot G = (e - e') \cdot B.$$

Thus,

$$A = [(z_1 - z_1') \cdot (e - e')^{-1} \bmod q] \cdot G$$

and

$$B = [(z_1 - z_1') \cdot (e - e')^{-1} \bmod q] \cdot \mathcal{P} + [(z_2 - z_2') \cdot (e - e')^{-1} \bmod q] \cdot G.$$

Observe that since $e \neq e' \bmod q$, the value $(e - e')^{-1} \bmod q$ exists and can be efficiently computed. Setting $r = (z_1 - z_1') \cdot (e - e')^{-1} \bmod q$ and $x = (z_2 - z_2') \cdot (e - e')^{-1} \bmod q$, the above shows that $(A, B) = \mathsf{EGexpEnc}_{\mathcal{P}}(x; r)$, as required.

**Honest-verifier zero knowledge:** Given $e$, the simulator chooses random $z_1, z_2 \leftarrow \mathbb{Z}_q$ and computes

$$X = z_1 \cdot G - e \cdot A \quad \text{and} \quad Y = z_1 \cdot \mathcal{P} + z_2 \cdot G - e \cdot B.$$

If indeed there exist $(x, r)$ such that $A = r \cdot G$ and $B = r \cdot \mathcal{P} + x \cdot G$, then

$$X = (z_1 - e \cdot r) \cdot G \quad \text{and} \quad Y = (z_1 - e \cdot r) \cdot \mathcal{P} + (z_2 - e \cdot x) \cdot G.$$

Now, set $\sigma = z_1 - e \cdot r$ and $\rho = z_2 - e \cdot x$, and observe that if $z_1, z_2$ are chosen at random (as indeed is the case by the simulator), then $\sigma$ and $\rho$ are also random. This implies that $z_1 = \sigma + e \cdot r$ and $z_2 = \rho + e \cdot x$ for random $\sigma, \rho$ and thus the distribution is *identical* to that of a real proof.

**Complexity.** The cost of the proof is three exponentiations for the prover and five for the verifier. When applying the Fiat-Shamir transform, the communication cost is three elements of $\mathbb{Z}_q$.

### A.3 Scalar Product on Encrypted Values – $R_{\text{prod}}$

We present a Sigma protocol for the relation

$$\begin{aligned} R_{\text{prod}} = \{((\mathbb{G}, G, q, \mathcal{P}, A, B, C, D, E, F), (t, r, y)) \mid \\ (C, D) = \mathsf{EGexpEnc}_P(y; t) \ \wedge \ E = y \cdot A + r \cdot G \ \wedge \ F = y \cdot B + r \cdot \mathcal{P}\} \end{aligned}$$

that validates that the same $y$ encrypted in $(C, D)$ is used in the scalar multiplication and rerandomization of $(A, B)$ in order to obtain $(E, F)$. Recall that $(C, D) = \mathsf{EGexpEnc}_{\mathcal{P}}(y; t)$ means that $C = t \cdot G$ and $D = t \cdot \mathcal{P} + y \cdot G$.

**PROTOCOL A.3 (Sigma Protocol for Relation $R_{\text{prod}}$)**

Upon joint input $(\mathbb{G}, G, q, \mathcal{P}, A, B, C, D, E, F)$ and witness $(t, r, y)$ for the prover $P$, the parties work as follows:

**First prover message:**

1. The prover $P$ chooses random $\sigma, \rho \leftarrow \mathbb{Z}_q$, and computes $X = \sigma \cdot A + \rho \cdot G$ and $Y = \sigma \cdot B + \rho \cdot \mathcal{P}$ and $W = \sigma \cdot G$.
2. $P$ sends $(X, Y, W)$ to the verifier $V$.

**Verifier challenge:** $V$ chooses a random $e \in \{0,1\}^{\kappa_s}$ (for statistical security parameter $\kappa_s$), and sends it to $P$.

**Second prover message:**

1. $P$ computes $z_1 = \sigma + e \cdot y \bmod q$ and $z_2 = \rho + e \cdot r \bmod q$.
2. $P$ sends $(z_1, z_2)$ to $V$.

**Additional Sigma protocol:** $P$ proves to $V$ that $(G, \mathcal{P}, e \cdot C, e \cdot D - z_1 \cdot G + W)$ is a Diffie-Hellman tuple (with witness $e \cdot t \bmod q$) using the standard Sigma protocol.

**Verification:** $V$ accepts if and only if the following holds

1. $z_1 \cdot A + z_2 \cdot G = X + e \cdot E$, and
2. $z_1 \cdot B + z_2 \cdot \mathcal{P} = Y + e \cdot F$
3. The Diffie-Hellman tuple Sigma protocol is accepted.

Observe that when applying the Fiat-Shamir transform, the entire proof is non-interactive. This is due to the fact that the first proof can be generated, at which point $e$ is obtained and so the statement for the second Sigma protocol for Diffie-Hellman tuples can be defined and proven.

We sketch the proof of security, proving completness, special soundness and honest-verifier zero-knowledge.

**Completeness:** If all messages are computed correctly, then:

$$\begin{aligned} z_1 \cdot A + z_2 \cdot G &= (\sigma + e \cdot y) \cdot A + (\rho + e \cdot r) \cdot G \\ &= (\sigma \cdot A + \rho \cdot G) + e \cdot (y \cdot A + r \cdot G) \\ &= X + e \cdot E \end{aligned}$$

and

$$\begin{aligned} z_1 \cdot B + z_2 \cdot \mathcal{P} &= (\sigma + e \cdot y) \cdot B + (\rho + e \cdot r) \cdot \mathcal{P} \\ &= (\sigma \cdot B + \rho \cdot \mathcal{P}) + e \cdot (y \cdot B + r \cdot \mathcal{P}) \\ &= Y + e \cdot F, \end{aligned}$$

and so the verifier accepts the first two conditions. In addition

$$\begin{aligned} e \cdot D - z_1 \cdot G + W &= e \cdot t \cdot \mathcal{P} + e \cdot y \cdot G - \sigma \cdot G - e \cdot y \cdot G + \sigma \cdot G \\ &= e \cdot t \cdot \mathcal{P} \end{aligned}$$

and so

$$(G, \mathcal{P}, e \cdot C, e \cdot D - z_1 \cdot G + W) = (G, \mathcal{P}, (e \cdot t) \cdot G, (e \cdot t) \cdot \mathcal{P})$$

is a Diffie-Hellman tuple with witness $e \cdot t \bmod q$. Thus, the verifier accepts.

**Special soundness:** We show that given $(X, Y, W, e, e', z_1, z_2, z_1', z_2', \pi, \pi')$ such that $e \neq e' \bmod q$ and both $(X, Y, W, e, z_1, z_2, \pi)$ and $(X, Y, W, e', z_1', z_2', \pi')$ are accepting transcripts (where $\pi, \pi'$ are the Diffie-Hellman proofs), it is possible to compute $(x, r)$ where $(A, B) = \mathsf{EGexpEnc}_{\mathcal{P}}(x; r)$. By the assumption that both transcripts are accepting we have that both

$$z_1 \cdot A + z_2 \cdot G = X + e \cdot E \quad \text{and} \quad z_1 \cdot B + z_2 \cdot \mathcal{P} = Y + e \cdot F,$$

and

$$z_1' \cdot A + z_2' \cdot G = X + e' \cdot E \quad \text{and} \quad z_1' \cdot B + z_2' \cdot \mathcal{P} = Y + e' \cdot F,$$

Subtracting the equations from each other, we have

$$(z_1 - z_1') \cdot A + (z_2 - z_2') \cdot G = (e - e') \cdot E$$

and

$$(z_1 - z_1') \cdot B + (z_2 - z_2') \cdot \mathcal{P} = (e - e') \cdot F.$$

Thus,

$$E = (z_1 - z_1') \cdot (e - e')^{-1} \cdot A + (z_2 - z_2') \cdot (e - e')^{-1} \cdot G$$

and

$$F = (z_1 - z_1') \cdot (e - e')^{-1} \cdot B + (z_2 - z_2') \cdot (e - e')^{-1} \cdot \mathcal{P}.$$

Observe that since $e \neq e' \bmod q$, the value $(e - e')^{-1} \bmod q$ exists and can be efficiently computed. Setting $y = (z_1 - z_1') \cdot (e - e')^{-1} \bmod q$ and $r = (z_2 - z_2') \cdot (e - e')^{-1} \bmod q$, the above shows that $E = y \cdot A + r \cdot G$ and $F = y \cdot B + r \cdot \mathcal{P}$ as required.

It remains to validate that the value $y$ obtained is indeed the same $y$ as encrypted in $(E, D)$. The special soundness of the Diffie-Hellman tuple proof (for the two accepting transcripts with $e, e'$ that contain two different proofs $\pi, \pi'$) provides us with a knowledge extractor to obtain $w, w' \in \mathbb{Z}_q$ such that

$$e \cdot C = w \cdot G \quad \text{and} \quad e \cdot D - z_1 \cdot G + W = w \cdot \mathcal{P}.$$

and

$$e' \cdot C = w' \cdot G \quad \text{and} \quad e' \cdot D - z_1' \cdot G + W = w' \cdot \mathcal{P}.$$

This implies that

$$C = (w - w') \cdot (e - e')^{-1} \cdot G$$

and

$$D = (w - w') \cdot (e - e')^{-1} \cdot \mathcal{P} + (z_1 - z_1') \cdot (e - e')^{-1} \cdot G.$$

From above, we have already determined that $y = (z_1 - z_1') \cdot (e - e')^{-1} \bmod q$ and thus setting $t = (w - w') \cdot (e - e')^{-1} \bmod q$, we conclude that $C = t \cdot G$ and $D = t \cdot \mathcal{P} + y \cdot G$, as required.

**Honest-verifier zero knowledge:** Given $e$, the simulator chooses random $z_1, z_2 \leftarrow \mathbb{Z}_q$ and computes

$$X = z_1 \cdot A + z_2 \cdot G - e \cdot E \quad \text{and} \quad Y = z_1 \cdot B + z_2 \cdot \mathcal{P} - e \cdot F,$$

and generates a simulated proof $\pi$ for the Diffie-Hellman tuple.

If indeed there exist $(y, r)$ such that $E = y \cdot A + r \cdot G$ and $F = y \cdot B + r \cdot \mathcal{P}$, then

$$X = z_1 \cdot A + z_2 \cdot G - e \cdot y \cdot A - e \cdot r \cdot G = (z_1 - e \cdot y) \cdot A + (z_2 - e \cdot r) \cdot G$$

and

$$Y = z_1 \cdot B + z_2 \cdot \mathcal{P} - e \cdot y \cdot B - e \cdot r \cdot \mathcal{P} = (z_1 - e \cdot y) \cdot B + (z_2 - e \cdot r) \cdot \mathcal{P}.$$

Setting $\sigma = z_1 - e \cdot y$ and $\rho = z_2 - e \cdot r$, we have that the distribution is identical (using the same argument as in Section A.2 for $R_{EG}$). (The simulation of the Diffie-Hellman tuple is perfect, by the fact that a Sigma protocol is used for that as well.)

**Complexity.** The cost of the proof is seven exponentiations for the prover (5 for the operations described in Protocol A.3 and 2 more for the Diffie-Hellman proof) and ten for the verifier (6 for the operations described in Protocol A.3 and 4 more for the Diffie-Hellman proof) . When applying the Fiat-Shamir transform, the communication cost is 6 elements of $\mathbb{Z}_q$ (3 for Protocol A.3 and 3 more for the Diffie-Hellman proof).

# B Full Proof of Security – $\mathcal{F}_{\text{mult}}$

We prove that Protocols 4.3–4.7 securely compute $\mathcal{F}_{\text{mult}}$. We prove the theorem in the $(\mathcal{F}_{\mathsf{zk}}, \mathcal{F}_{\mathsf{com\text{-}zk}}, \mathcal{F}_{\text{checkDH}})$-hybrid model. As described in Section 3.3, $\mathcal{F}_{\mathsf{zk}}$ and $\mathcal{F}_{\mathsf{com\text{-}zk}}$ can be efficiently instantiated non-interactively in the random oracle model. These functionalities also have efficient interactive instantiations without relying on a random oracle, but they both increase the round complexity, and the commitment also increases the computational complexity. We show how to securely compute Functionality $\mathcal{F}_{\text{checkDH}}$ in Section 7.

**Theorem B.1** *Assume that the Decisional Diffie-Hellman problem is hard in the group $(\mathbb{G}, G, q)$, and that $\pi^{\text{priv}}_{\text{mult}}$ is a private multiplication protocol for malicious adversaries as defined in Section 2.3. Then, Protocols 4.3 to 4.7 securely compute the $\mathcal{F}_{\text{mult}}$ functionality with abort in the $(\mathcal{F}_{\mathsf{zk}}, \mathcal{F}_{\mathsf{com\text{-}zk}}, \mathcal{F}_{\text{checkDH}})$-hybrid model, in the presence in the presence of a malicious adversary corrupting any $t < n$ parties, with point-to-point channels.*

**Proof:** The intuition behind the security is given in Section 4.3, and so we proceed directly to describe the simulator. Let $\mathcal{A}$ be an adversary and let $I \subseteq [n]$ be the set of corrupted parties. If all parties are corrupted, then the simulation is trivial. We therefore consider $I \subset [n]$ (with $|I| < n$) and we denote the set of honest parties by $J = [n] \setminus I$. Throughout the proof, we denote corrupted parties by $P_i$ (i.e., $i \in I$), honest parties by $P_j$ (i.e., $j \in J$), and a running index over $[n]$ by $\ell$. We remark that in the simulation, the simulation of each round begins by simulating the messages that the corrupted parties receives from the honest in *this* round, and only then receiving this round messages from the adversary. This is due to the fact that the adversary is rushing and so may receive the messages that the honest parties send in round $i$ before sending its own round messages in round $i$ (this is in contrast to the protocol presentation, where the receipt of round $i$ messages

by honest parties is processed at the beginning of round $i+1$, since this is the way that honest parties work). Unless stated otherwise, all operations on scalar values are modulo $q$.

We construct a simulator $\mathcal{S}$ who invokes $\mathcal{A}$ internally and simulates an execution of the real protocol, while interacting with $\mathcal{F}_{\text{mult}}$ in the ideal model. We describe the simulator steps separately for each subprotocol of $\mathcal{F}_{\text{mult}}$.

**Initialization:** Upon receiving $(\mathsf{init}, \mathbb{G}, G, q)$, simulator $\mathcal{S}$ invokes $\mathcal{A}$ upon input $(\mathsf{init}, \mathbb{G}, G, q)$ and works as follows:

1. $\mathcal{S}$ chooses a random group element $\mathcal{P} \in \mathbb{G}$.
2. $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{DL}}$ sending $(\mathsf{ProofReceipt}, init, j)$ to $P_i$, for every $i \in I$ and $j \in J$.
3. $\mathcal{S}$ receives the messages $(\mathsf{ComProve}, init, i, \mathcal{P}_i, d_i)$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{DL}}$ for every $i \in I$.
4. $\mathcal{S}$ chooses random group elements $\{P_j\}_{j\in J}$ under the constraint that $\sum_{j\in J} \mathcal{P}_j = \mathcal{P} - \sum_{i\in I} \mathcal{P}_i$. (Specifically, $\mathcal{S}$ specifies some $j^* \in J$ and then chooses random $\mathcal{P}_j$ for all $j \in J \setminus \{j^*\}$. Finally, $\mathcal{S}$ sets $\mathcal{P}_{j^*} = \mathcal{P} - \sum_{\ell \neq j^*} \mathcal{P}_\ell$.)
5. $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{DL}}$ sending $(\mathsf{DecomProof}, init, j, \mathcal{P}_j, 1)$ to $P_i$, for every $i \in I$ and $j \in J$.
6. $\mathcal{S}$ receives the messages $(\mathsf{DecomProof}, init, i)$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{DL}}$ for every $i \in I$.
7. If for some $i \in I$ it holds that a $\mathsf{DecomProof}$ message was received and $\mathcal{P}_i \neq d_i \cdot G$ in the associated $\mathsf{ComProve}$ message of Step 3 above, then $\mathcal{S}$ sends $\mathsf{abort}$ to $\mathcal{F}_{\text{mult}}$, outputs whatever $\mathcal{A}$ outputs and halts. Else, $\mathcal{S}$ proceeds to the next step.
8. $\mathcal{S}$ stores all of the values $\{d_i\}_{i\in I}$, as well as $\mathcal{P}$ and all $\{\mathcal{P}_\ell\}_{\ell=1}^n$.

**Input:** Upon receiving $(\mathsf{input}, sid, \tilde{a}_i)$ for all $i \in I$, simulator $\mathcal{S}$ invokes $\mathcal{A}$ upon the same input and works as follows:

1. $\mathcal{S}$ chooses random group elements $\{U_j, V_j\}_{j\in J}$.
2. $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{zk}}^{R_{EG}}$ sending $(\mathsf{proof}, sid, j, (\mathcal{P}, U_j, V_j), 1)$ to $P_i$, for every $i \in I$ and $j \in J$.
3. $\mathcal{S}$ receives the messages $(\mathsf{proof}, sid, i, (\mathcal{P}, U_i, V_i), (a_i, s_i))$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{zk}}^{R_{EG}}$ for every $i \in I$.
4. If for some $i \in I$ it holds that $U_i \neq s_i \cdot G$ or $V_i \neq s_i \cdot \mathcal{P} + a_i \cdot G$, then $\mathcal{S}$ sends $\mathsf{abort}$ to $\mathcal{F}_{\text{mult}}$, outputs whatever $\mathcal{A}$ outputs and halts. Else, $\mathcal{S}$ proceeds to the next step.
5. $\mathcal{S}$ computes $U = \sum_{\ell=1}^n U_\ell$ and $V = \sum_{\ell=1}^n V_\ell$, and stores $(sid, (U, V), \{U_i, V_i, a_i, s_i\}_{i\in I}, \{U_j, V_j\}_{j\in J})$.

**Element-out:** Upon receiving $(\mathsf{element\text{-}out}, sid)$, if some $sid$ has been stored, then simulator $\mathcal{S}$ sends $(\mathsf{element\text{-}out}, sid)$ to $\mathcal{F}_{\text{mult}}$ from every $P_i$ with $i \in I$. Upon receiving back $(\mathsf{element\text{-}out}, sid, A)$ and points $(A_1, \ldots, A_n)$, $\mathcal{S}$ invokes $\mathcal{A}$ upon the same input and works as follows:

1. Let $(U_j, V_j)$ be the pair of points stored by $\mathcal{S}$ associated with $sid$ for each honest party $P_j$ (if none exists, then $\mathcal{S}$ does nothing).

2. $\mathcal{S}$ simulates $P_j$ sending $A_j$ to $P_i$, and $\mathcal{F}_{\mathsf{zk}}^{R_{DH}}$ sending $(\mathsf{proof}, sid, j, (G, \mathcal{P}, U_j, V_j - A_j), 1)$ to $P_i$, for every $i \in I$ and $j \in J$.

3. $\mathcal{S}$ receives the messages $(\mathsf{proof}, sid, i, (G, \mathcal{P}, U_i, V_i - A_i), s_i)$ and $A_i$, that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{zk}}^{R_{DH}}$ and to $P_j$, for every $i \in I$ and $j \in J$. If for some $i \in I$ it holds that $U_i \neq s_i \cdot G$ or $V_i \neq s_i \cdot \mathcal{P} + a_i \cdot G$, then $\mathcal{S}$ sends $\mathsf{abort}$ to $\mathcal{F}_{\text{mult}}$, outputs whatever $\mathcal{A}$ outputs and halts.

**Affine:** This operation involves local operations by the parties only. Thus, $\mathcal{S}$ carries out the local transformations on the values that it has stored as well, in the same way as the honest parties.

**Multiply:** Upon receiving $(\mathsf{mult}, sid_1, sid_2)$, if these have been stored, then simulator $\mathcal{S}$ sends $(\mathsf{mult}, sid_1, sid_2)$ to $\mathcal{F}_{\text{mult}}$ from every $P_i$ with $i \in I$. Upon receiving back $(\mathsf{mult\text{-}out}, sid_1, sid_2, c)$, $\mathcal{S}$ sets $sid = sid_1 \| sid_2$, invokes $\mathcal{A}$ and works as follows:

1. Let $(sid_1, (U, V), \{U_i, V_i, a_i, s_i\}_{i \in I}, \{U_j, V_j\}_{j \in J})$ and $(sid_2, (X, Y), \{X_i, Y_i, b_i, t_i\}_{i \in I}, \{X_j, Y_j\}_{j \in J})$ be as stored.

2. $\mathcal{S}$ chooses random values $a_j, b_j$ under the constraint that $(\sum_{\ell=1}^{n} a_\ell) \cdot (\sum_{\ell=1}^{n} b_\ell) = c$, and plays the honest parties in protocol $\pi_{\text{mult}}^{\text{priv}}$ using inputs $\{a_j, b_j\}_{j \in J}$. (The idea here is that if $\mathcal{A}$ uses the "correct" inputs then this will be indistinguishable. Otherwise, the parties will abort before revealing anything anyway. Note that the corrupted parties' values $\{a_i, b_i\}_{i \in I}$ were extracted by $\mathcal{S}$ in the simulation of the $\mathsf{input}$ phase, and possibly modified in a known way using $\mathsf{affine}$. Thus, there values are known to $\mathcal{S}$.)

3. $\mathcal{S}$ chooses random group elements $E_j, F_j \in \mathbb{G}$ for every $j \in J$.

4. $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{zk}}^{R_{\text{prod}}}$ sending $(\mathsf{proof}, sid, j, (\mathcal{P}, X, Y, U_j, V_j, E_j, F_j), 1)$ to $P_i$, for every $i \in I$ and $j \in J$.

5. $\mathcal{S}$ receives the messages $(\mathsf{proof}, sid, i, (\mathcal{P}, X, Y, U_i, V_i, E_i, F_i), (a_i, s_i, s_i'))$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{zk}}^{R_{\text{prod}}}$ for every $i \in I$. If for some $i \in I$ it holds that the proof is incorrect (checked via the witness), then $\mathcal{S}$ sends $\mathsf{abort}$ to $\mathcal{F}_{\text{mult}}$, outputs whatever $\mathcal{A}$ outputs and halts. Else, $\mathcal{S}$ proceeds to the next step.

6. $\mathcal{S}$ chooses random $A_j, B_j \in \mathbb{G}$ and simulates $\mathcal{F}_{\mathsf{zk}}^{R_{EG}}$ sending $(\mathsf{proof}, sid, j, (\mathcal{P}, A_j, B_j))$ to $P_i$, for every $j \in J$ and $i \in I$.

7. $\mathcal{S}$ receives $(\mathsf{proof}, sid, i, (\mathcal{P}, A_i, B_i), (c_i, \hat{s}_i))$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{zk}}^{R_{EG}}$ for every $i \in I$. If for some $i \in I$ it holds that $(A_i, B_i) \neq \mathsf{EGexpEnc}(c_i'; \hat{s}_i)$, then $\mathcal{S}$ sends $\mathsf{abort}$ to $\mathcal{F}_{\text{mult}}$, outputs whatever $\mathcal{A}$ outputs and halts.

8. $\mathcal{S}$ computes $A = E - \sum_{\ell=1}^{n} A_\ell$ and $B = F - \sum_{\ell=1}^{n} B_\ell$.

9. $\mathcal{S}$ receives messages $(\mathsf{check}, i, A, B, d_i)$ that $\mathcal{A}$ sends to $\mathcal{F}_{\text{checkDH}}$ for every $i \in I$.

   (a) If $d_i \cdot G \neq \mathcal{P}_i$ for any $i \in I$, then $\mathcal{S}$ simulates $\mathcal{F}_{\text{checkDH}}$ sending $\mathsf{reject}$ to all parties, sends $\mathsf{abort}$ to $\mathcal{F}_{\text{mult}}$, outputs whatever $\mathcal{A}$ outputs and halts.

(b) $\mathcal{S}$ verifies that $\sum_{\ell=1}^{n} c_\ell = c$, where $\{c_i\}_{i\in I}$ are the values that $\mathcal{S}$ receives from $\mathcal{A}$ in the proof messages that it sends to $\mathcal{F}_{\mathsf{zk}}^{R_{EG}}$ in Step 7, and $\{c_j\}_{j\in J}$ is the output that the honest parties simulated by $\mathcal{S}$ receive from the $\pi_{\text{mult}}^{\text{priv}}$ execution in Step 2. If no, then $\mathcal{S}$ simulates $\mathcal{F}_{\text{checkDH}}$ sending reject to all parties, sends abort to $\mathcal{F}_{\text{mult}}$, outputs whatever $\mathcal{A}$ outputs and halts.

If reject was not sent, then $\mathcal{S}$ simulates $\mathcal{F}_{\text{checkDH}}$ sending accept to all parties.

10. $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{zk}}^{R_{DH}}$ sending $(\mathsf{proof}, sid, j, (\mathcal{P}, A_j, B_j - c_j \cdot G), 1)$ to $P_i$, and $P_j$ sending $c_j$ to $P_i$, for every $i \in I$ and $j \in J$.

11. $\mathcal{S}$ receives the messages $(\mathsf{proof}, sid, i, (\mathcal{P}, A_i, B_i - c_i \cdot G), \hat{s}_i)$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{zk}}^{R_{DH}}$ for every $i \in I$. If for some $i \in I$ it holds that $A_i \neq \hat{s}_i \cdot G$ or $B_i - c_i \cdot G \neq \hat{s}_i \cdot B_i$, then $\mathcal{S}$ sends abort to $\mathcal{F}_{\text{mult}}$, outputs whatever $\mathcal{A}$ outputs and halts.

12. $\mathcal{S}$ receives the values $c_i$ that $\mathcal{A}$ instructs $P_i$ to send to $P_j$, for every $i \in I$ and $j \in J$.

13. For every $j \in J$, $\mathcal{S}$ verifies if $c = \sum_{\ell=1}^{n} c_\ell$, with the $c_i$ values received specifically by $P_j$. If yes, then $\mathcal{S}$ sends $(\mathsf{continue}, j)$ to $\mathcal{F}_{\text{mult}}$ (to instruct that $P_j$ receives output); if not, then $\mathcal{S}$ sends $(\mathsf{abort}, j)$ to $\mathcal{F}_{\text{mult}}$ (to instruct that $P_j$ receives abort).

This completes the description of the simulator. We now proceed to show that the output distribution generated by an execution of $\mathcal{S}$ in the ideal world is computationally indistinguishable from a real execution of the protocol with $\mathcal{A}$ in the real world. Observe that the main difference is that the ciphertexts seen by $\mathcal{A}$ in the real execution are valid encryptions whereas in the simulation they are random pairs of group elements, and the inputs used in the subprotocol $\pi_{\text{mult}}^{\text{priv}}$. The proof will therefore rely on the DDH assumption in group $\mathbb{G}$ and on the assumption that $\pi_{\text{mult}}^{\text{priv}}$ is a private multiplication protocol (see Section 2.3).

We prove via a series of games.

**Game 0:** This is an ideal-world execution with $\mathcal{S}$ and $\mathcal{F}_{\text{mult}}$.

**Game 1:** In this game, we modify $\mathcal{F}_{\text{mult}}$ so that $\mathcal{F}_{\text{mult}}$ sends all honest party inputs $\{a_j\}_{j\in J}$ to $\mathcal{S}$, whenever input is called. Since $\mathcal{S}$ is unchanged, this clearly makes no difference to the output distribution.

**Game 2:** In this game, we modify the simulator $\mathcal{S}$ so that instead of running $\pi_{\text{mult}}^{\text{priv}}$ with $a_j, b_j$ chosen randomly under the specified constraint, it uses the correct $a_j, b_j$ received from $\mathcal{F}_{\text{mult}}$. The fact that the output distributions of Game 1 and Game 2 are indistinguishable follows from the privacy properties of $\pi_{\text{mult}}^{\text{priv}}$. In particular, if the implicit inputs $\{a'_i\}_{i\in I}$ and $\{b'_i\}_{i\in I}$ used by the adversary are such that $\sum_{i\in I} a'_i = \sum_{i\in I} a_i \bmod q$ and $\sum_{i\in I} b'_i = \sum_{i\in I} b_i \bmod q$ then this guarantees that the output $c = (\sum_{\ell=1}^{n} a_\ell) \cdot (\sum_{\ell=1}^{n} b_\ell) \bmod q$, is the same in both cases (when the honest parties inputs are correct, or chosen randomly under the simulator-specified constraint), as long as no modular reduction took place. Thus, by input indistinguishability the view of the adversary is indistinguishable, even given all $c_1, \ldots, c_n$ as in the last step of the protocol.

In contrast, if the implicit inputs $\{a'_i\}_{i\in I}$ and $\{b'_i\}_{i\in I}$ used by the adversary are such that $\sum_{i\in I} a'_i \neq \sum_{i\in I} a_i$ *or* $\sum_{i\in I} b'_i \neq \sum_{i\in I} b_i$, then the adversary will be able to supply $\{c'_i\}_{i\in I}$ such that all sum to the correct $c$ with probability only $1/q$, since all the honest parties' inputs $\{a_j, b_j\}_{j\in J}$

are uniformly distributed shares. Thus, in this case, the adversary will not view $c_1, \ldots, c_n$. This is therefore indistinguishable by the privacy property of $\pi^{\text{priv}}_{\text{mult}}$. Furthermore, this argument guarantees that the fact of $c$ being correct or incorrect does not reveal any information to the adversary (since this fact is revealed in the protocol).

We stress that in the entire simulation by $\mathcal{S}$, the private-keys for Paillier are only needed within $\pi^{\text{priv}}_{\text{mult}}$. Thus, $\mathcal{S}$ can run the simulation in Games 1 and 2 without knowing the Paillier private keys

**Game 3:** In this game, $\mathcal{S}$ generates all encryptions of corrupted parties *correctly* using the $a_j, b_j$ values it received (this includes the $(U_j, V_j)$ values in $\mathsf{input}$, and the $(E_j, F_j)$ and $(A_j, B_j)$ values in $\mathsf{mult}$).

In order to show that the output distributions of Games 2 and 3 are computationally indistinguishable, we show that there exists a distinguisher $\mathcal{D}$ who receives a tuple $(G, \hat{\mathcal{P}}, \Lambda, \Omega)$ of group elements of $\mathbb{G}$ for input. Then, $\mathcal{D}$ generates the output distribution of Game 2 if they are a random tuple, but generates the output distribution of Game 3 if they are a Diffie-Hellman tuple (i.e., there exists some $r$ such that $\Lambda = r \cdot G$ and $\Omega = r \cdot \hat{\mathcal{P}}$. This implies indistinguishability of the output distribution of the games, under the DDH assumption.

The idea behind the reduction is that $\mathcal{D}$ can define $\mathcal{P} = \hat{\mathcal{P}}$ and then use $\Lambda, \Omega$ to construct ciphertexts that are either random or valid, depending on whether or not its input was random or a Diffie-Hellman tuple. We use a rerandomization method, defined by

$$\mathsf{Rerand}(\Lambda, \Omega; s, t) = (s \cdot \Lambda + t \cdot G,\ \ s \cdot \Omega + t \cdot \hat{\mathcal{P}}). \tag{1}$$

We denote by $\mathsf{Rerand}(\Lambda, \Omega)$ the above procedure when $s, t \leftarrow \mathbb{Z}_q$ are random. Clearly, if $(G, \mathcal{P}, \Lambda, \Omega)$ is a Diffie-Hellman tuple, then $(G, \mathcal{P}, A, B)$ where $(A, B) = \mathsf{Rerand}(\Lambda, \Omega)$ is a *random* Diffie-Hellman tuple (where by random we mean that it is independent of $(G, \mathcal{P}, \Lambda, \Omega)$). This is due to the fact that if $\Lambda = r \cdot G$ and $\Omega = r \cdot \hat{\mathcal{P}}$ then $A = (s \cdot r + t) \cdot G$ and $B = (s \cdot r + t) \cdot \mathcal{P}$. Since $s, t$ are random, this is an independent and uniformly distributed Diffie-Hellman tuple. In contrast, if $(G, \mathcal{P}, \Lambda, \Omega)$ is not a Diffie-Hellman tuple, then $(G, \mathcal{P}, A, B)$ where $(A, B) = \mathsf{Rerand}(\Lambda, \Omega)$ is a *random* tuple, independent of $(G, \mathcal{P}, \Lambda, \Omega)$. In order to see this, write $\Lambda = r_1 \cdot G$ and $\Omega = r_2 \cdot G$. Then, for any $\rho_1, \rho_2$, we have that $A = \rho_1 \cdot G$ and $B = \rho_2 \cdot G$ if and only if $s \cdot r_1 + t = \rho_1$ and $s \cdot r_2 + t = \rho_2$. Since $r_1 \neq r_2$, there is a single solution $(s, t)$ to this system of equations. Since $s, t$ are random, this proves that the result is an independent random tuple.

We proceed to describe $\mathcal{D}$. Distinguisher $\mathcal{D}$ receives all of the parties' inputs and plays the trusted party computing $\mathcal{F}_{\text{ECDSA}}$ in Games 2/3 ($\mathcal{F}_{\text{ECDSA}}$ is the same in both of these games), as well as the ideal-world honest parties. In addition, $\mathcal{D}$ invokes $\mathcal{A}$ and runs the simulator $\mathcal{S}$ like in Games 2/3 with the exception of how the encryptions are generated, as described above. $\mathcal{D}$ works as follows:

**Initialization:** $\mathcal{D}$ invokes $\mathcal{A}$ upon input $(\mathsf{init}, \mathbb{G}, G, q)$ and works in exactly the same way as $\mathcal{S}$ except that it defines $\mathcal{P} = \hat{\mathcal{P}}$ instead of choosing it randomly itself. This is therefore exactly the same distribution as $\mathcal{S}$.

**Input:** $\mathcal{D}$ works exactly like the simulator $\mathcal{S}$ except that instead of choosing random group elements $\{U_j, V_j\}_{j \in J}$ it sets $(U'_j, V'_j) = \mathsf{Rerand}(\Lambda, \Omega)$ for every $j \in J$ (using independent randomness in each call to $\mathsf{Rerand}$. Then, $\mathcal{D}$ sets $U_j = U'_j$ and $V_j = V'_j + a_j \cdot G$. $\mathcal{D}$ stores $\big(sid, (U, V), \{U_\ell, V_\ell, a_\ell\}_{\ell \in [n]}, \{s_i\}_{i \in I}\big)$.

Observe that if the tuple $(G, \mathcal{P}, \Lambda, \Omega)$ is a Diffie-Hellman tuple, then all $(U_j, V_j)$ are valid encryptions of the honest parties' inputs as in Game 3. In contrast, if the tuple $(G, \mathcal{P}, \Lambda, \Omega)$ is random, then $(U_j, V_j)$ is random and so exactly is Game 2 (adding $a_j \cdot G$ makes no difference to the distribution since $U'_j, V'_j$ are truly random and independent).

**Element-out:** $\mathcal{D}$ works exactly like the simulator $\mathcal{S}$.

**Affine:** This is a local operation and thus the $\mathcal{D}$ carry out the local transformation on all values it holds.

**Multiply:** $\mathcal{D}$ invokes $\mathcal{A}$ on $(\mathsf{mult}, sid_1, sid_2)$ and works as follows.

1. Let $(sid_1, (U, V), \{U_\ell, V_\ell, a_\ell\}_{\ell=1}^n, \{s_i\}_{i \in I}, )$ and $\big(sid_2, (X, Y), \{X_\ell, Y_\ell, b_\ell\}_{\ell \in [n]}, \{s_i\}_{i \in I}\big)$ be as stored.

2. $\mathcal{D}$ plays the honest parties in protocol $\pi^{\text{priv}}_{\text{mult}}$ using the inputs $\{a_j, b_j\}_{j \in J}$ exactly like $\mathcal{S}$ in Game 2/3.

3. For every $j \in J$, $\mathcal{D}$ computes $(E'_j, F'_j) = \mathsf{Rerand}(\Lambda, \Omega)$ and defines $E_j = E'_j$ and $F_j = F'_j + (a_j \cdot b) \cdot G$ (where $a_j, b$ are the correct values known to $\mathcal{D}$).

   Observe that if $(\Lambda, \Omega)$ are Diffie-Hellman, then the distribution over all $(E_j, F_j)$ pairs is correct and so exactly like Game 3. In order to see that they are correct, observe that all values provided by $\mathcal{A}$ involved in computing $E_j, F_j$ are those provided in $\mathsf{input}$ and are validated with zero-knowledge proofs. Thus, computing an encryption of $a_j \cdot b$ directly as carried out by $\mathcal{D}$ yields the same result as the simulator in Game 3 who uses the honest parties' inputs.

   In contrast, if $(\Lambda, \Omega)$ are random, then then all $(E_j, F_j)$ pairs are independently random, just like $\mathcal{S}$ in Game 2.

4. $\mathcal{D}$ simulates $\mathcal{F}_{\mathsf{zk}}^{R_{\text{prod}}}$ sending the zero-knowledge proofs, exactly like $\mathcal{S}$.

5. $\mathcal{D}$ simulates receives the zero-knowledge proofs for $\mathcal{F}_{\mathsf{zk}}^{R_{\text{prod}}}$, exactly like $\mathcal{S}$.

6. Let $\{c_j\}_{j \in J}$ be the outputs received by $\mathcal{D}$ when running the honest parties in $\pi^{\text{priv}}_{\text{mult}}$ with $\mathcal{A}$, above. For every $j \in J$, $\mathcal{D}$ computes $(A'_j, B'_j) = \mathsf{Rerand}(\Lambda, \Omega)$ and defines $A_j = A'_j$ and $B_j = B'_j + c_j \cdot G$.

   As above, $(A_j, B_j)$ is a valid encryption of $c_j$ as in Game 3 (and a real execution) if $(\Lambda, \Omega)$ is a Diffie-Hellman tuple, and is a random pair as in Game 2 otherwise.

7. $\mathcal{D}$ proceeds and follows the simulation instructions to the end.

8. $\mathcal{D}$ outputs whatever $\mathcal{A}$ outputs, together with the outputs of the honest parties (it knows these values since it also plays the trusted party computing $\mathcal{F}_{\text{ECDSA}}$).

As explained throughout the description above, when the tuple $(G, \hat{\mathcal{P}}, \Lambda, \Omega)$ received by $\mathcal{D}$ is a Diffie-Hellman tuple, then the output distribution generated by $\mathcal{D}$ is distributed as in Game 2. In contrast, when $(G, \hat{\mathcal{P}}, \Lambda, \Omega)$ is a random tuple, the output distribution generated by $\mathcal{D}$ is distributed as in Game 3. Thus, the output distributions of Games 2 and 3 are indistinguishable under the assumption that the DDH assumption holds in $\mathbb{G}$.

**Game 4:** Game 3 is exactly the same as a real execution, with the exception that $\mathcal{P}$ is chosen by $\mathcal{S}$ and not the result of the computation by the parties in init. We therefore modify $\mathcal{S}$ so that it plays honestly in init instead of running the simulation. Observe that since the $\mathcal{P}_j$ values are chosen randomly by $\mathcal{S}$ in Game 4 and not revealed to $\mathcal{A}$ until *after* it has sent the $\mathcal{P}_i$ values, the sum $\mathcal{P} = \sum_{\ell=1}^{n} \mathcal{P}_\ell$ is uniformly distributed, exactly as in Game 3. Thus, this makes no difference.

**Summing up.** Game 4 is exactly the same as a real execution, except for the technicality that $\mathcal{F}_{\text{mult}}$ is involved and hands the honest parties their outputs. However, in Game 4, the simulator $\mathcal{S}$ runs the instructions of the honest parties exactly, with the addition of perfectly simulating the $\mathcal{F}_{\mathsf{zk}}$, $\mathcal{F}_{\mathsf{com\text{-}zk}}$ and $\mathcal{F}_{\text{checkDH}}$ functionalities. Thus, the output distribution of Game 4 is identical to the real execution. We conclude that the output distribution of the simulation (Game 1) is computationally indistinguishable from the output distribution of the real execution (Game 4) under the DDH difficulty assumption in $\mathbb{G}$. This concludes the proof.

**Consistency in the point-to-point model.** The above proof essentially assumes that all honest parties receive the same values in each round, as if there is a broadcast channel. As described after Protocol 4.4, the parties exchange hashes of all received input encryptions in the round following each input. Since the simulator can simulate the first round of any subprotocol following input even if the adversary was not consistent, this suffices (since in the following round, all parties abort). In addition, note that consistency is guaranteed in the init phase since $\mathcal{F}_{\mathsf{com\text{-}zk}}$ is used. Now, within element-out and mult, all messages generated by the parties are proven in zero knowledge. Thus, even if the adversary sends different ciphertexts to different honest parties, these will always be of the same correct results. Thus, the simulator can follow the same strategy as above, and provide the expected honest party messages back, for each honest party. Thus, our protocol is secure in the point-to-point model of communication. ■