Fast Secure Two-Party ECDSA Signing*

Yehuda Lindell**

Dept. of Computer Science
Bar-Ilan University, Israel
lindell@biu.ac.il

Abstract. ECDSA is a standard digital signature schemes that is widely used in TLS, Bitcoin and elsewhere. Unlike other schemes like RSA, Schnorr signatures and more, it is particularly hard to construct efficient threshold signature protocols for ECDSA (and DSA). As a result, the best-known protocols today for secure distributed ECDSA require running heavy zero-knowledge proofs and computing many large-modulus exponentiations for every signing operation. In this paper, we consider the specific case of two parties (and thus no honest majority) and construct a protocol that is approximately *two orders of magnitude faster* than the previous best. Concretely, our protocol achieves good performance, with a single signing operation for curve P-256 taking approximately 37ms between two standard machine types in Azure (utilizing a single core only). Our protocol is proven secure under standard assumptions using a game-based definition. In addition, we prove security by simulation under a plausible yet non-standard assumption regarding Paillier.

1 Introduction

1.1 Background

In the late 1980s and the 1990s, a large body of research emerged around the problem of *threshold cryptography*; cf. [3,8,10,11,14,26,25,22]. In its most general form, this problem considers the setting of a private key shared between n parties with the property that any subset of t parties may be able to decrypt or sign, but any set of 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 after the first operation the key is reconstructed, and 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.

* An extended abstract of this work appeared at *CRYPTO 2017*. This version contains a faster and much simpler zero-knowledge proof for the key-generation phase, and corrects an error in the definition of the ad-hoc assumption enabling a simulation-based proof.

** Much of this work was done for Unbound Tech Ltd.

Threshold cryptography can be used in applications where multiple signators 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 [27,28,29]. Another reason is due to the fact that ECDSA signing is used in bitcoin, 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. Nevertheless, a more general solution may be obtained via threshold cryptography (for the more general t-out-of-n threshold case).

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. We explain this by comparing ECDSA signing to EC-Schnorr signing. In both cases, the public verification key is an elliptic curve point Q and the private signing key is x such that $Q = x \cdot G$, where G is the generator point of an EC group of order q.

EC-Schnorr signing	**ECDSA signing**
Choose a random $k \leftarrow \mathbb{Z}_q$	Choose a random $k \leftarrow \mathbb{Z}_q$
Compute $R = k \cdot G$	Compute $R = k \cdot G$
Compute $e = H(m \| R)$	Compute $r = r_x \bmod q$ where $R = (r_x, r_y)$
Compute $s = k - x \cdot e \bmod q$	Compute $s = k^{-1} \cdot (H(m) + r \cdot x) \bmod q$
Output (s, e)	Output (r, s)

Observe that Schnorr signing can be easily distributed since the private key x and the value k are both used in a linear equation. Thus, two parties with shares x_1, x_2 such that $Q = (x_1 + x_2) \cdot G$ can each locally choose k_1, k_2, and set $R = k_1 \cdot G + k_2 \cdot G = (k_1 + k_2) \cdot G$. Then, each can locally compute e and $s_i = k_i - x_i \cdot e \bmod q$ and send to each other, and each party can sum $s = s_1 + s_2 \bmod q$ and output a valid signature (s, e). In the case of malicious adversaries, some zero knowledge proofs are needed to ensure that R is uniformly distributed, but these are highly efficient proofs of knowledge of discrete log. In contrast, in ECDSA signing, the equation for computing s includes k^{-1}. Now, given shares k_1, k_2 such that $k_1 + k_2 = k \bmod q$ it is very difficult to compute k'_1, k'_2 such that $k'_1 + k'_2 = k^{-1} \bmod q$.

As a result, beginning with [22] and more lately in [15], two-party protocols for ECDSA signing use *multiplicative sharing* of x and of k. That is, the parties hold x_1, x_2 such that $x_1 \cdot x_2 = x \bmod q$, and in each signing operation they generate k_1, k_2 such that $k_1 \cdot k_2 = k \bmod q$. This enables them to easily compute k^{-1} since each party can locally compute $k'_i = k_i^{-1} \bmod q$, and then k'_1, k'_2 are multiplicative shares of k^{-1}. The parties can then use additively homomorphic encryption – specifically Paillier encryption [23] – in order to combine their equations. For example, P_1 can compute $c_1 = \mathsf{Enc}_{pk}((k_1)^{-1} \cdot H(m))$ and $c_2 = \mathsf{Enc}_{pk}(k_1^{-1} \cdot x_1 \cdot r)$. Then, using scalar multiplication (denoted $\odot$) and homomorphic addition (denoted $\oplus$), P_2 can compute $(k_2^{-1} \odot c_1) \oplus [(k_2^{-1} \cdot x_2) \odot c_2]$ which will be an encryption of

$$k_2^{-1} \cdot (k_1^{-1} \cdot H(m)) + k_2^{-1} \cdot x_2 \cdot (k_1^{-1} \cdot x_1 \cdot r) = k^{-1} \cdot (H(m) + r \cdot x),$$

as required. However, proving that each party worked correctly is extremely difficult. For example, the first party must prove that the Paillier encryption includes k_1^{-1} when the second party only has $R_1 = k_1 \cdot G$, it must prove that the Paillier encryptions are to values in the expected range, and more. This can be done, but it results in a protocol that is very expensive.

1.2 Our Results

As in previous protocols, we use Paillier homomorphic encryption (with a key generated by P_1), and multiplicative sharing of both the private key x and the random value k. However, we make the novel observation that if P_2 already holds a Paillier encryption c_{key} of P_1's share of the private key x_1, then P_1 need not do anything except participate in the generation of $R = k \cdot G$. Specifically, assume that the parties P_1 and P_2 begin by generating $R = k_1 \cdot k_2 \cdot G$ (this is essentially accomplished by just running a Diffie-Hellman key exchange with basic knowledge-of-discrete-log proofs which are highly efficient). Then, given $c_{key} = \mathsf{Enc}_{pk}(x_1)$, R and k_2, x_2, party P_2 can singlehandedly compute an encryption of $k_2^{-1} \cdot H(m) + k_2^{-1} \cdot r \cdot x_2 \cdot x_1$ using the homomorphic properties of Paillier encryption. This ciphertext can be sent to P_1 who decrypts and multiplies the result by k_1^{-1}. If P_2 is honest, then the result is a valid signature.

The crucial issue that must be dealt with is what happens when P_1 or P_2 is corrupted. If P_1 is corrupted, it cannot do anything since the only message that it sends P_2 is in the generation of R which is protected by an efficient zero-knowledge proof. Thus, no expensive proofs are needed. Furthermore, if P_2 is corrupted, then the only way it can cheat is by encrypting something incorrect and sending it to P_1. However, here we can utilize the fact that we are specifically computing a digital signature that can be *publicly verified*. That is, since all P_1 does is locally decrypt the ciphertext received from P_2 and multiply by k_1^{-1}, it can locally check if the signature obtained is valid. If yes, it outputs it, and if not it detects P_2 cheating. Thus, no zero-knowledge proofs are required for P_2 either (again, beyond the zero-knowledge proof in the generation of R).

As a result, we obtain a signing protocol that is extremely simple and efficient. As we show, our protocol is approximately two orders of magnitude faster

than the previous best. Before proceeding, we remark that there are additional elements needed in the protocol (like P_2 adding random noise in the ciphertext it sends), but these have little effect on the efficiency.

We remark that since the security of the signing protocol rests upon the assumption that P_2 holds an encryption of x_1, which is P_1's share of the key, this must be proven in the key generation phase. Thus, the key generation phase of our protocol is more complicated than the signing phase, and includes a proof that P_1 generated the Paillier key correctly and that c_{key} is an encryption of x_1, given $R_1 = x_1 \cdot G$. This latter proof is of interest since it connects between Paillier encryption and discrete log, and we present a novel efficient proof in the paper. We remark that since key generation is run only once, having a more expensive key-generation phase is a worthwhile tradeoff. This is especially the case since it is still quite reasonable (concretely taking about 2.5 seconds between standard machines in Azure and running with a single thread, which is much faster than the key-generation phase of [15]). Furthermore, it can easily be parallelized to further bring down the cost.

DSA vs ECDSA. In this paper, we refer to ECDSA throughout and we use Elliptic curve (additive group) notation. However, our entire protocol translates easily to the DSA case, since we do nothing but standard group operations.

Caveat. The only caveat of our work is that it focuses specifically on the two-party case, whereas prior works considered general thresholds as well. The two-party case is in some ways the most difficult case (since there is no honest majority), and we therefore believe that our techniques may be useful for the general case as well. We leave this for future research.

1.3 Related Work and a Comparison of Efficiency

The first specific protocol for threshold DSA signing with proven security was presented in [14]. Their protocol works as long as more than 1/3 of the parties are honest. The two party case (where there is no honest majority) was later dealt with by [22]. The most recent protocol by [15] contains efficiency improvements for the two-party case, and improvements regarding the thresholds for the case of an honest majority.

Efficiency comparison with [15]. The previous best DSA/ECDSA threshold signing protocol is due to [15]. Their signing protocol requires the following operations by each party: 1 Paillier encryption, 5 Paillier homomorphic scalar multiplications, 5 Paillier homomorphic additions, and 46 exponentiations (the vast majority of these modulo N or N^2 for the Paillier modulus). Furthermore, they require the Paillier modulus to be greater than q^8 where q is the group order. Now, for P-256, this makes no difference since anyway a 2048-bit modulus is minimal. However, for P-384 and P-521 respectively, this requires a modulus of size 3072 and 4168 respectively, which severely slows down the computation. Regarding the key generation phase, [15] need to run a protocol for distributed key generation for Paillier. This outweighs all other computations and is very

expensive for the case of malicious adversaries. (They did not implement this phase in their prototype, but the method they refer to [19] has a reported time of 15 minutes for generating a 2048-bit modulus for the semi-honest case alone.)

In contrast, the cost of our key-generation protocol is dominated by approximately 350 Paillier encryptions/exponentiations by each party; see Section 3.4 for an exact count. Furthermore, as described in Section 3.4, in the signing protocol, party P_1 computes 7 Elliptic curve multiplications and 1 Paillier decryption, and party P_2 computes 5 Elliptic curve multiplications and 1 Paillier encryption, 1 homomorphic scalar multiplication and 1 Paillier homomorphic addition. Furthermore, the Paillier modulus needs only to be greater than $2q^4 + q^3$, where q is the ECDSA group order. Thus, a 2048-bit modulus can be taken for P-256 and P-384, and a 2086-bit modulus only is needed for P-521. We therefore conclude that the cost of our signing protocol is approximately *two orders of magnitude* faster than their protocol.[1] This theoretical estimate is validated by our experimental results.

Experimental results and comparison. The running-time reported for the protocol of [15] for curve P-256 is approximately 12 seconds per signing operation between a mobile and PC. An improved optimized implementation using parallelism and *4 cores* on a 2.4GHz machine achieves approximately 1 second per signing operation (these measurements are only for the *computation time* and do not include communication). In contrast, as we describe in Section 3.4, for curve P-256 our signing protocol takes approximately 37ms, using a single core (measuring the actual full running time, including communication). This validates the theoretical analysis of approximately *two orders of magnitude difference*, when taking into account the use of multiple cores. Specifically, on 4 cores, we can achieve a throughput of over 100 signatures per second, in contrast to a single signing operation for [15]. Full details of our experiments, for curves P-256, P-384 and P-521 appear in Section 3.4.

Finally, the key generation phase of our protocol for curve P-256 takes approximately 2.5 seconds, using a single core. In contrast, [15] requires distributed Paillier key generation which is extremely expensive, as described above.

2 Preliminaries

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 $Q = x \cdot G$.

The ECDSA signing operation on a message $m \in \{0,1\}^*$ is defined as follows:

[1] We base this estimate on an OpenSSL speed test that puts the speed of the entire ECDSA signing operation for P-256 (which consists of one EC multiplication and more) at more than 10 times faster than a single RSA2048 private-key exponentiation. Note that for P-521 and RSA4096 the gap is even larger with the entire ECDSA signing operation being more than 30 times faster than a single RSA4096 private-key exponentiation.

1. Compute m' to be the $|q|$ leftmost bits of $SHA256(m)$, where $|q|$ is the bit-length of q
2. Choose a random $k \leftarrow \mathbb{Z}_q^*$
3. Compute $R \leftarrow k \cdot G$. Let $R = (r_x, r_y)$.
4. Compute $r = r_x \bmod q$. If $r = 0$, go back to Step 2.
5. Compute $s \leftarrow k^{-1} \cdot (m' + r \cdot x) \bmod q$.
6. 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 $\frac{q-1}{2}$.)

The ideal commitment functionality $\mathcal{F}_{\mathsf{com}}$. In one of our subprotocols, we assume an ideal commitment functionality $\mathcal{F}_{\mathsf{com}}$, formally defined in Functionality 2.1. Any UC-secure commitment scheme fulfills $\mathcal{F}_{\mathsf{com}}$; e.g., [20,1,13]. 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 $r \leftarrow \{0,1\}^n$ is random.

FIGURE 2.1 (The Commitment Functionality $\mathcal{F}_{\mathsf{com}}$)

Functionality $\mathcal{F}_{\mathsf{com}}$ works with parties P_1 and P_2, as follows:

- Upon receiving $(\mathsf{commit}, sid, x)$ from party P_i (for $i \in \{1,2\}$), record (sid, i, x) and send $(\mathsf{receipt}, sid)$ to party P_{3-i}. If some $(\mathsf{commit}, sid, *)$ is already stored, then ignore the message.
- Upon receiving $(\mathsf{decommit}, sid)$ from party P_i, if (sid, i, x) is recorded then send $(\mathsf{decommit}, sid, x)$ to party P_{3-i}.

The ideal zero knowledge functionality $\mathcal{F}_{\mathsf{zk}}$. We use the standard ideal zero-knowledge functionality defined by $((x, w), \lambda) \rightarrow (\lambda, (x, R(x, w)))$, where λ 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 [18, Section 6.5.3]; non-interactive versions can be achieved in the random-oracle model via the Fiat-Shamir paradigm; see Functionality 2.2 for the formal definition.

FIGURE 2.2 (The Zero-Knowledge Functionality $\mathcal{F}_{\mathsf{zk}}^R$ for Relation R)

Upon receiving $(\mathsf{prove}, sid, x, w)$ from a party P_i (for $i \in \{1,2\}$): if $(x, w) \notin R$ or sid has been previously used then ignore the message. Otherwise, send (proof, sid, x) to party P_{3-i}.

The committed non-interactive zero knowledge functionality $\mathcal{F}_{\mathsf{com\text{-}zk}}$. In our protocol, we will have parties send commitments to non-interactive zero-knowledge proofs. We model this formally via a commit-zk functionality, denoted $\mathcal{F}_{\mathsf{com\text{-}zk}}$, defined in Functionality 2.3. Given non-interactive zero-knowledge proofs of knowledge, this functionality is securely realized by just having the prover commit to such a proof using the ideal commitment functionality $\mathcal{F}_{\mathsf{com}}$.

Paillier encryption. Denote the public/private key pair by (pk, sk), and denote encryption and decryption under these keys by $\mathsf{Enc}_{pk}(\cdot)$ and $\mathsf{Dec}_{sk}(\cdot)$, respectively. We denote by $c_1 \oplus c_2$ the "addition" of the plaintexts in c_1, c_2, and by $a \odot c$ the multiplication of the plaintext in c by scalar a.

FIGURE 2.3 (The Committed NIZK Functionality $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R}$ for R)
Functionality $\mathcal{F}_{\mathsf{com\text{-}zk}}$ works with parties P_1 and P_2, as follows:

- Upon receiving $(\mathsf{com\text{-}prove}, sid, x, w)$ from a party P_i (for $i \in \{1,2\}$): if $(x, w) \notin R$ or sid has been previously used then ignore the message. Otherwise, store (sid, i, x) and send $(\mathsf{proof\text{-}receipt}, sid)$ to P_{3-i}.
- Upon receiving $(\mathsf{decom\text{-}proof}, sid)$ from a party P_i (for $i \in \{1,2\}$): if (sid, i, x) has been stored then send $(\mathsf{decom\text{-}proof}, sid, x)$ to P_{3-i}.

Security, the hybrid model and composition. We prove the security of our protocol under a game-based definition with standard assumptions (in Section 4), and under the simulation-based ideal/real model definition with a non-standard ad-hoc assumption (in Section 5). In all cases, we prove our protocols secure in a hybrid model with ideal functionalities that securely compute $\mathcal{F}_{\mathsf{com}}, \mathcal{F}_{\mathsf{zk}}, \mathcal{F}_{\mathsf{com\text{-}zk}}$. 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 computationally indistinguishable to when calling a trusted party that computes the ideal functionalities.

3 Two-Party 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).

3.1 Zero-Knowledge Proofs

Our protocol is presented in the $\mathcal{F}_{\mathsf{zk}}$ and $\mathcal{F}_{\mathsf{com\text{-}zk}}$ hybrid model. We use the zero-knowledge functionalities $\mathcal{F}_{\mathsf{zk}}^{R_P}$ and $\mathcal{F}_{\mathsf{zk}}^{R_{DL}}$ for the following two relations:

1. *Proof that a Paillier public-key was generated correctly:* define the relation

$$R_P = \{(N, \phi(N)) \mid \gcd(N, \phi(N)) = 1\}$$

   of valid Paillier public keys. We remark that standard Paillier is defined for $N = p \cdot q$ with p, q prime. However, all that we require is that N defines a public key for which the additive homomorphic operations are correct, and this holds as long as $\gcd(N, \phi(N)) = 1$. This proof can be generated as described in Section 3.3 in the full version of [19]. The cost of this protocol is $3t$ Paillier exponentiations by each of the prover and verifier for statistical error 2^{-t}, as well as $3t$ GCD computations by the prover.

2. *Proof of knowledge of the discrete log of an Elliptic-curve point:* define the relation

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

of discrete log values (relative to the given group). We use the standard Schnorr proof for this [24].

In addition, we also need to prove in zero-knowledge that a value encrypted in a given Paillier ciphertext is the discrete log of a given Elliptic curve point. Our zero-knowledge proof for this statement is *not* a proof of knowledge, and we therefore define the language and not its associated relation:

$$L_{PDL} = \{(c, pk, Q_1, \mathbb{G}, G, q) \mid \exists (x_1, sk) \text{ such that } x_1 = \mathsf{Dec}_{sk}(c) \text{ and } Q_1 = x_1 \cdot G \text{ and } x_1 \in \mathbb{Z}_q\},$$

where pk is a given Paillier public key and sk is its associated private key. We will actually prove a slightly relaxed variant which is that completeness holds for $x_1 \in \mathbb{Z}_{q/3}$. This suffices for our needs. Note that since our proof for this statement is not a proof of knowledge, we cannot use the $\mathcal{F}_{\mathsf{zk}}$-hybrid model; rather we will use the zero-knowledge properties directly in our proof.

We present a highly efficient zero-knowledge proof for the language L_{PDL}; this proof by itself is a novel contribution and of interest since it bridges between two completely different worlds (Paillier encryption and Elliptic curve groups). The proof appears in Section 6.

For the sake of clarity of notation, we omit the group description $(\mathbb{G}, G, q)$ within calls to the $\mathcal{F}_{\mathsf{zk}}$ functionalities and when referring to L_{PDL}, since this is implicit. In addition, throughout, we assume that all values (Elliptic curve points) received are not equal to 0, and if zero is received then the party receiving the value aborts immediately.

3.2 Distributed Key Generation

The idea behind the distributed key generation protocol is as follows. The parties run a type of "simulatable coin tossing" in order to generate a random group element Q. This coin tossing protocol works by P_1 choosing a random x_1 and computing $Q_1 = x_1 \cdot G$, and then committing to Q_1 along with a zero-knowledge proof of knowledge of x_1, the discrete log of Q_1 (for technical reasons that will become apparent in Section 6, P_1 actually chooses $x \in \mathbb{Z}_{q/3}$, but this makes no difference). Then, P_2 chooses a random x_2 and sends $Q_2 = x_2 \cdot G$ along with a zero-knowledge proof of knowledge to P_1. Finally, P_1 decommits and P_2 verifies the proof. The output is the point $Q = x_1 \cdot Q_2 = x_2 \cdot Q_1$. This is fully simulatable due to the extractability and equivocality of the proof and commitment. In particular, assume that P_1 is corrupted. Then, a simulator receiving Q from the trusted party can cause the output of the coin-toss to equal Q. This is because it receives Q_1, x_1 from P_1 (who sends these values to the proof functionality) and can define the value sent by P_2 to be $Q_2 = (x_1)^{-1} \cdot Q$. Noting that $x_1 \cdot Q_2 = Q$, we have the desired property. Likewise, if P_2 is corrupted, then the simulator

can commit to anything and then after seeing (Q_2, x_2) as sent to the proof functionality, it can define $Q_1 = (x_2)^{-1} \cdot Q$. The fact that the P_1 is supposed to already be committed is solved by using an equivocal commitment scheme (modeled here via the $\mathcal{F}_{\mathsf{com\text{-}zk}}$ ideal functionality). Beyond generating Q, the protocol concludes with P_2 holding a Paillier encryption of x_1, where $Q_1 = x_1 \cdot G$. As described, this is used to obtain higher efficiency in the signing protocol, and is guaranteed via a zero-knowledge proof. See Protocol 3.1 for a full description.

> **PROTOCOL 3.1 (Key Generation Subprotocol $\mathsf{KeyGen}(\mathbb{G}, g, q)$)**
>
> Upon joint input $(\mathbb{G}, G, q)$ and security parameter 1^n, the parties work as follows:
>
> 1. ***P_1*'s first message:**
>   (a) P_1 chooses a random $x_1 \leftarrow \mathbb{Z}_{q/3}$, and computes $Q_1 = x_1 \cdot G$.
>   (b) P_1 sends $(\mathsf{com\text{-}prove}, 1, Q_1, x_1)$ to $\mathcal{F}^{R_{DL}}_{\mathsf{com\text{-}zk}}$ (i.e., P_1 sends a commitment to Q_1 and a proof of knowledge of its discrete log).
> 2. ***P_2*'s first message:**
>   (a) P_2 receives $(\mathsf{proof\text{-}receipt}, 1)$ from $\mathcal{F}^{R_{DL}}_{\mathsf{com\text{-}zk}}$.
>   (b) P_2 chooses a random $x_2 \leftarrow \mathbb{Z}_q$ and computes $Q_2 = x_2 \cdot G$.
>   (c) P_2 sends $(\mathsf{prove}, 2, Q_2, x_2)$ to $\mathcal{F}^{R_{DL}}_{\mathsf{zk}}$.
> 3. ***P_1*'s second message:**
>   (a) P_1 receives $(\mathsf{proof}, 2, Q_2)$ from $\mathcal{F}^{R_{DL}}_{\mathsf{zk}}$. If not, it aborts.
>   (b) P_1 sends $(\mathsf{decom\text{-}proof}, 1)$ to $\mathcal{F}^{R_{DL}}_{\mathsf{com\text{-}zk}}$.
>   (c) P_1 generates a Paillier key-pair (pk, sk) of length $\min(4\log|q| + 2, n)$ and computes $c_{key} = \mathsf{Enc}_{pk}(x_1)$.
>   (d) P_1 sends $(\mathsf{prove}, 1, N, (p_1, p_2))$ to $\mathcal{F}^{R_P}_{\mathsf{zk}}$, where $pk = N = p_1 \cdot p_2$.
> 4. **ZK proof:** P_1 proves to P_2 in zero knowledge that $(c_{key}, pk, Q_1) \in L_{PDL}$.
> 5. ***P_2*'s verification:** P_2 aborts unless all the following hold: **(a)** it received $(\mathsf{decom\text{-}proof}, 1, Q_1)$ from $\mathcal{F}^{R_{DL}}_{\mathsf{zk}}$ and $(\mathsf{proof}, 1, N)$ from $\mathcal{F}^{R_P}_{\mathsf{zk}}$, **(b)** it accepted the proof that $(c_{key}, pk, Q_1) \in L_{PDL}$, and **(c)** the key $pk = N$ is of length at least $\min(4\log|q| + 2, n)$.
> 6. **Output:**
>   (a) P_1 computes $Q = x_1 \cdot Q_2$ and stores (x_1, Q).
>   (b) P_2 computes $Q = x_2 \cdot Q_1$ and stores (x_2, Q, c_{key}).

3.3 Distributed Signing

The idea behind the signing protocol is as follows. First, the parties run a similar "coin tossing protocol" as in the key generation phase in order to obtain a random point R that will be used in generating the signature; after this, the parties P_1 and P_2 hold k_1 and k_2, respectively, where $R = k_1 \cdot k_2 \cdot G$. Then, since P_2 already holds a Paillier encryption of x_1 (under a key known only to P_1), it is possible for P_1 to singlehandedly compute r from $R = (r_x, r_y)$ and an encryption of $s' = (k_2)^{-1} \cdot m' + (k_2)^{-1} \cdot r \cdot x_2 \cdot x_1$; this can be carried out by P_2 since it knows all the values involved directly except for x_1 which is encrypted under Paillier. Observe that this is "almost" a valid signature since in a valid

signature $s = k^{-1} \cdot m' + k^{-1} \cdot r \cdot x$ (and here $x = x_1 \cdot x_2$). Indeed, P_2 can send the encryption of this value to P_1, who can then decrypt and just multiply by $(k_1)^{-1}$. Since $k = k_1 \cdot k_2$ we have that the result is a valid ECDSA signature. The only problem with this method is that the encryption of $(k_2)^{-1} \cdot m' + (k_2)^{-1} \cdot r \cdot x_2 \cdot x_1$ may reveal information to P_1 since no reduction modulo q is carried out on the values (because Paillier works over a different modulus). In order to prevent this, we have P_2 add $\rho \cdot q$ to the value inside the encryption, where ρ is random and "large enough"; in the proof, we show that if $\rho \leftarrow \mathbb{Z}_{q^2}$, then this value is statistically close to $k_1 \cdot s$, where s is the final signature. Thus, P_1 can learn nothing more than the result (and in fact its view can be simulated). Note that since $s = k_1^{-1} \cdot s'$, it holds that $s' = k_1 \cdot s$ and so s' reveals no more information to P_1 than the signature s itself (this is is due to the fact that P_1 can compute s' from the signature s and from its share k_1).

The only problem that remains is that P_2 may send an incorrect s' value to P_1. However, since we are dealing specifically with digital signatures, P_1 can verify that the result is correct before outputting it. Thus, a corrupt P_2 cannot cause P_1 to output incorrect values. However, it is conceivable that P_2 may be able to learn something from the fact that P_1 output a value or aborted. Consider, hypothetically, that P_2 could generate an encryption of a value s' so that $(k_1)^{-1} \cdot s'$ is a valid signature if $LSB(x_1) = 0$ and $(k_1)^{-1} \cdot s'$ is not a valid signature if $LSB(x_1) = 1$. In such a case, the mere fact that P_1 aborts or not can leak a single bit about P_1's private share of the key. In the proof(s) of security below, we show how we deal with this issue. See the formal definition of the signing phase in Protocol 3.2 (and a graphical representation in Figure 1).

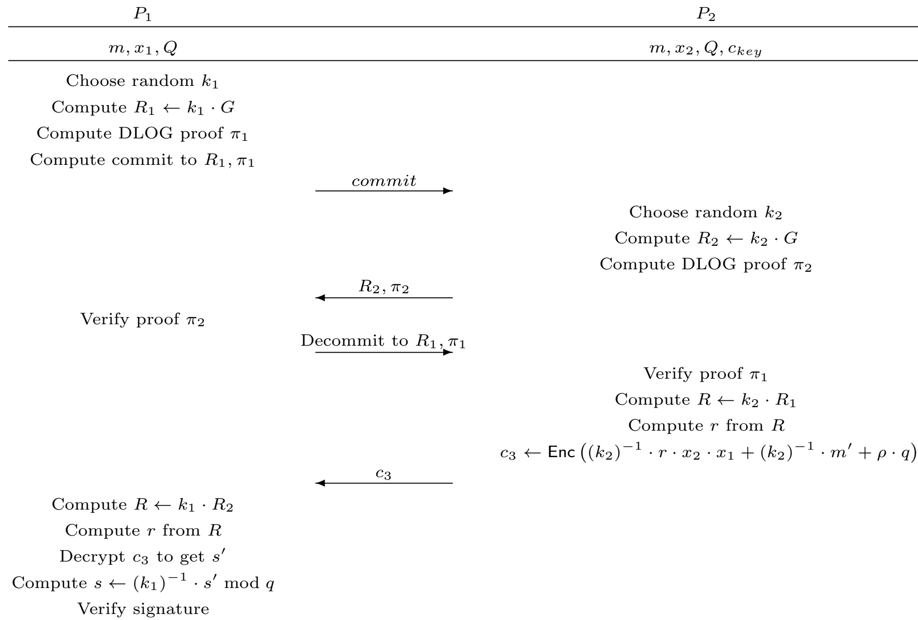

Fig. 1. The 2-Party ECDSA Signing Protocol

Offline/Online. Observe that the message to be signed is only used in P_2's second message and by P_1 to verify that the signature is valid. Thus, it is possible to run the first three steps in an offline phase. Then, when m is received, all that is required to generate a signature is for P_2 to send a single message to P_1.

Output to both parties. Observe that since the validity of the signature can be checked by P_2, it is possible for P_1 to send P_2 the signature if it verifies it and it's valid. This will not affect security at all.

Correctness. Denoting $k = k_1 \cdot k_2$ and $x = x_1 \cdot x_2$, we have that c_3 is an encryption of $s' = \rho \cdot q + (k_2)^{-1} \cdot m' + (k_2)^{-1} \cdot r \cdot x_2 \cdot x_1 = \rho \cdot q + (k_2)^{-1} \cdot (m' + r \cdot x)$ (assuming that all is done correctly). Thus, $s = (k_1)^{-1} \cdot s' = k^{-1} \cdot (m' + rx) \bmod q$.

PROTOCOL 3.2 (Signing Subprotocol $\mathsf{Sign}(sid, m)$)

A graphical representation of the protocol appears in Figure 1.

Inputs:

1. Party P_1 has (x_1, Q) as output from Protocol 3.1, the message m, and a unique session id sid.
2. Party P_2 has (x_2, Q, c_{key}) as output from Protocol 3.1, the message m and the session id sid.
3. P_1 and P_2 both locally compute $m' \leftarrow H_q(m)$ and verify that sid has not been used before (if it has been, the protocol is not executed).

The Protocol:

1. ***P_1*'s first message:**
   (a) P_1 chooses a random $k_1 \leftarrow \mathbb{Z}_q$ and computes $R_1 = k_1 \cdot G$.
   (b) P_1 sends $(\mathsf{com\text{-}prove}, sid\|1, R_1, k_1)$ to $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{DL}}$.
2. ***P_2*'s first message:**
   (a) P_2 receives $(\mathsf{proof\text{-}receipt}, sid\|1)$ from $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{DL}}$.
   (b) P_2 chooses a random $k_2 \leftarrow \mathbb{Z}_q$ and computes $R_2 = k_2 \cdot G$.
   (c) P_2 sends $(\mathsf{prove}, sid\|2, R_2, k_2)$ to $\mathcal{F}_{\mathsf{zk}}^{R_{DL}}$.
3. ***P_1*'s second message:**
   (a) P_1 receives $(\mathsf{proof}, sid\|2, R_2)$ from $\mathcal{F}_{\mathsf{zk}}^{R_{DL}}$; if not, it aborts.
   (b) P_1 sends $(\mathsf{decom\text{-}proof}, sid\|1)$ to $\mathcal{F}_{\mathsf{com\text{-}zk}}$.
4. ***P_2*'s second message:**
   (a) P_2 receives $(\mathsf{decom\text{-}proof}, sid\|1, R_1)$ from $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{DL}}$; if not, it aborts.
   (b) P_2 computes $R = k_2 \cdot R_1$. Denote $R = (r_x, r_y)$. Then, P_2 computes $r = r_x \bmod q$.
   (c) P_2 chooses a random $\rho \leftarrow \mathbb{Z}_{q^2}$ and computes $c_1 = \mathsf{Enc}_{pk}\left(\rho \cdot q + \left[(k_2)^{-1} \cdot m' \bmod q\right]\right)$. Then, P_2 computes $v = (k_2)^{-1} \cdot r \cdot x_2 \bmod q$, $c_2 = v \odot c_{key}$ and $c_3 = c_1 \oplus c_2$.
   (d) P_2 sends c_3 to P_1.
5. ***P_1* generates output:**
   (a) P_1 computes $R = k_1 \cdot R_2$. Denote $R = (r_x, r_y)$. Then, P_1 computes $r = r_x \bmod q$.
   (b) P_1 computes $s' = \mathsf{Dec}_{sk}(c_3)$ and $s'' = (k_1)^{-1} \cdot s' \bmod q$. P_1 sets $s = \min\{s'', q - s''\}$ (this ensures that the signature is always the smaller of the two possible values).
   (c) P_1 verifies that (r, s) is a valid signature with public key Q. If yes it outputs the signature (r, s); otherwise, it aborts.

If a party aborts at any point, then it does not participate in any future $\mathsf{Sign}(sid, m)$ executions.

3.4 Efficiency and Experimental Results

We now analyze the theoretical complexity of our protocol, and describe its concrete running time based on our implementation.

Theoretical complexity – key-distribution protocol. Leaving aside the ZK proofs for now, P_1 carries out 2 Elliptic curve multiplications, 1 Paillier public-key generation and 1 Paillier encryption, and P_2 carries out two Elliptic curve multiplications. In addition, the parties run two discrete log proofs (each playing as prover once and as verifier once), and P_1 proves that N is a valid Paillier public key and runs the PDL proof described in Section 3.2. The cost of these proofs is as follows:

- *Discrete log:* the standard Schnorr zero-knowledge proof of knowledge for discrete log requires a single multiplication by the prover and two by the verifier.
- *Paillier public-key validity* [19]: For a statistical error of 2^{-40} this costs 120 Paillier exponentiations by each of the prover and the verifier (but 40 of these are "short"). In addition, the prover P_1 carries out 120 GCD computations.
- *PDL proof (Section 6):* This proof, described in Protocol 6.1, also involves running one executions of a range proof, and carrying out a small constant number of operations. The cost of the proof overall is as follows:
  - The instructions within Protocol 6.1 for the prover P_1 are 1 Paillier decryption and 1 Elliptic curve multiplication. The cost for the verifier P_2 is 1 Paillier scalar multiplication and scalar addition, and 2 Elliptic curve multiplications.
  - As described in Section A, the range proof is dominated by $2t$ Paillier encryptions for a statistical soundness error of 2^{-t}. Setting $t = 40$, we have 80 Paillier encryptions each.

Theoretical complexity – signing protocol. We now count the complexity of the signing protocol. We count the number of Elliptic curve multiplications and Paillier operations since this dominates the computation. As above, the zero-knowledge proof of knowledge for discrete log requires a single multiplication by the prover and two by the verifier, and ECDSA signature verification requires two multiplications. Thus, P_1 computes 7 Elliptic curve multiplications and a single Paillier decryption. In contrast, P_2 computes 5 Elliptic curve multiplications, 1 Paillier encryptions, 1 Paillier homomorphic scalar multiplication (which is a single "short" exponentiation) and one Paillier homomorphic addition (which is a single multiplication). Observe that unlike previous work, the length of the Paillier key need only be 5 times the length of the order of the Elliptic curve group (and not 8 times). Regarding rounds of communication, the protocol has only four rounds of communication (two in each direction). Thus, the protocol is very fast even on a slow network.

Implementation and running times. We implemented our protocol in C++ and ran it on Azure between two `Standard_DS3_v2` instances. Although these instances have 4 cores each, we utilized a single core only with a single-thread implementation (note that key generation can be easily parallelized, if desired).

We ran our implementation on the standard NIST curves P-256, P-384 and P-521; the times for key generation and signing appear in Tables 1 and 2.

Curve	Mean time	Standard deviation
P-256	2435ms	142
P-384	2440ms	124
P-521	3535ms	166

Table 1. Running times for **key generation** (average over 20 executions)

Curve	Mean time	Standard deviation
P-256	36.8ms	7.30
P-384	47.11ms	1.96
P-521	78.19ms	1.45

Table 2. Running times for **signing** (average over 1,000 executions)

We remark that the size of the Paillier key has a great influence on the running time. We know this since when running our initial experiments, our analysis of the protocol required setting $N > q^5$ (instead of $N > 2q^4 + q^3$). This seemingly small difference meant that for P-521, the Paillier key needed to be of size 2560 (instead of 2086). For this mildly larger key, the running time was 110ms for signing, which is 40% longer. This is explained by the fact that Paillier operations have cubic cost, and thus the cost *doubles* when the key size increases by just 25%.

4 Proof of Security – Game-Based Definition

4.1 Definition of Security

We begin by presenting a game-based definition for the security of a digital signature scheme $\pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$. This will be used when proving the security of our protocol and thus is presented for the sake of completeness and a concrete reference.

EXPERIMENT 4.1 ($\mathsf{Expt\text{-}Sign}_{\mathcal{A},\pi}(1^n)$)

1. $(vk, sk) \leftarrow \mathsf{Gen}(1^n)$.
2. $(m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}_{sk}(\cdot)}(1^n, vk)$.
3. Let $\mathcal{Q}$ be the set of all m queried by $\mathcal{A}$ to its oracle. Then, the output of the experiment equals 1 if and only if $m^* \notin \mathcal{Q}$ and $\mathsf{Verify}_{vk}(m^*, \sigma^*) = 1$.

Standard security of digital signatures

Definition 4.2. *A signature scheme π is* existentially unforgeable under chosen-message attacks *if for every probabilistic polynomial-time oracle machine $\mathcal{A}$ there exists a negligible function μ such that for every n,*

$$\Pr[\mathsf{Expt\text{-}Sign}_{\mathcal{A},\pi}(1^n) = 1] \leq \mu(n).$$

We now proceed to define security for a distributed signing protocol. In the experiment $\mathsf{Expt\text{-}DistSign}^b_{\mathcal{A},\Pi}$, we consider $\mathcal{A}$ controlling party P_b in protocol Π for two-party signature generation. Let $\Pi_b(\cdot,\cdot)$ be a *stateful* oracle that runs the instructions of honest party P_{3-b} in protocol Π. The adversary $\mathcal{A}$ can choose which messages will be signed, and can interact with multiple instances of party P_{3-b} to concurrently generate signatures. Note that the oracle is defined so that distributed key generation is first run once, and then multiple signing protocols can be executed concurrently.

Formally, $\mathcal{A}$ receives access to an oracle that receives two inputs: the first input is a session identifier and the second is either an input or a next incoming message. The oracle works as follows:

- Upon receiving a query of the form $(0,0)$ for the first time, the oracle initializes a machine M running the instructions of party P_{3-b} in the distributed key generation part of protocol Π. If party P_{3-b} sends the first message in the key generation protocol, then this message is the oracle reply.
- Upon receiving a query of the form $(0,m)$, if the key generation phase has not been completed, then the oracle hands the machine M the message m as its next incoming message and returns M's reply. (If the key generation phase has completed, then the oracle returns $\perp$.)
- If a query of the form (sid,m) is received where $sid \neq 0$, but the key generation phase with M has not completed, then the oracle returns $\perp$.
- If a query (sid,m) is received and the key generation phase has completed and this is the first oracle query with this identifier sid, then the oracle invokes a new machine M_{sid} running the instructions of party P_{3-b} in protocol Π with session identifier sid and input message m to be signed. The machine M_{sid} is initialized with the key share and any state stored by M at the end of the key generation phase. If party P_{3-b} sends the first message in the signing protocol, then this message is the oracle reply.
- If a query (sid,m) is received and the key generation phase has completed and this is not the first oracle query with this identifier sid, then the oracle hands M_{sid} the incoming message m and returns the next message sent by M_{sid}. If M_{sid} concludes, then the output obtained by M_{sid} is returned.

The experiment for defining security is formalized by simply providing $\mathcal{A}$ who controls party P_b with oracle access to Π_b. Adversary $\mathcal{A}$ "wins" if it can forge a signature on a message not queried in the oracle queries. Observe that $\mathcal{A}$ can run multiple executions of the signing protocol concurrently. We remark that we have considered only a single signing key; the extension to multiple different signing keys is straightforward and we therefore omit it. (This is due to the fact since signing keys are independent, one case easily simulate all executions with other keys.)

EXPERIMENT 4.3 ($\mathsf{Expt\text{-}DistSign}^b_{\mathcal{A},\Pi}(1^n)$)

Let $\pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ be a digital signature scheme.

1. $(m^*, \sigma^*) \leftarrow \mathcal{A}^{\Pi_b(\cdot,\cdot)}(1^n)$.
2. Let $\mathcal{Q}$ be the set of all inputs m such that (sid, m) was queried by $\mathcal{A}$ to its oracle as the first query with identifier sid. Then, the output of the experiment equals 1 if and only if $m^* \notin \mathcal{Q}$ and $\mathsf{Verify}_{vk}(m^*, \sigma^*) = 1$, where vk is the verification key output by P_{3-b} from the key generation phase, and Verify is as specified in π.

Security experiment for secure digital signature protocol

Definition 4.4. *A protocol Π is a* secure two-party protocol for distributed signature generation for π *if for every probabilistic polynomial-time oracle machine $\mathcal{A}$ and every $b \in \{1, 2\}$, there exists a negligible function μ such that for every n, $\Pr[\mathsf{Expt\text{-}DistSign}^b_{\mathcal{A},\Pi}(1^n) = 1] \le \mu(n)$.*

4.2 Proof of Security

In this section, we prove that Π comprised of Protocols 3.1 and 3.2 for key generation and signing, respectively, constitutes a secure two-party protocol for distributed signature generation of ECDSA.

Theorem 4.5. *Assume that the Paillier encryption scheme is indistinguishable under chosen-plaintext attacks, that ECDSA is existentially-unforgeable under a chosen message attack, and that the zero-knowledge proofs and commitments are as described. Then, Protocols 3.1 and 3.2 constitute a secure two-party protocol for distributed signature generation of ECDSA.*

Proof. We prove the security of the protocol in the $\mathcal{F}_{\mathsf{com\text{-}zk}}, \mathcal{F}_{\mathsf{zk}}$ hybrid model for relations R_{DL} and R_P. Note that if the commitment and zero-knowledge protocols are UC-secure, then this means that the output in the hybrid and real protocols is computationally indistinguishable. In particular, if $\mathcal{A}$ can break the protocol with some probability ϵ in the hybrid model, then it can break the protocol with probability $\epsilon \pm \mu(n)$ for some negligible function μ. Thus, this suffices.

We separately prove security for the case of a corrupted P_1 and a corrupted P_2. Our proof works by showing that, for any adversary $\mathcal{A}$ attacking the protocol, we construct an adversary $\mathcal{S}$ who forges an ECDSA signature in Experiment 4.1 with probability that is negligibly close to the probability that $\mathcal{A}$ forges a signature in Experiment 4.3. Formally, we prove that if Paillier has indistinguishable encryptions under chosen-plaintext attacks, then for every PPT algorithm $\mathcal{A}$ and every $b \in \{1, 2\}$ there exists a PPT algorithm $\mathcal{S}$ and a negligible function μ such that for every n,

$$\left| \Pr[\mathsf{Expt\text{-}Sign}_{\mathcal{S},\pi}(1^n) = 1] - \Pr[\mathsf{Expt\text{-}DistSign}^b_{\mathcal{A},\Pi}(1^n) = 1] \right| \le \mu(n), \qquad (1)$$

where Π denotes Protocols 3.1 and 3.2, and π denotes the ECDSA signature scheme. Proving Eq. (1) suffices, since by the assumption in the theorem that ECDSA is secure, we have that there exists a negligible function μ' such that for every n, $\Pr[\mathsf{Expt\text{-}Sign}_{\mathcal{S},\pi}(1^n) = 1] \leq \mu'(n)$. Combining this with Eq. (1), we conclude that $\Pr[\mathsf{Expt\text{-}DistSign}^b_{\mathcal{A},\Pi}(1^n) = 1] \leq \mu(n) + \mu'(n)$ and thus Π is secure by Definition 4.4. We prove Eq. (1) separately for $b = 1$ and $b = 2$.

Proof of Eq. (1) for $b = 1$ – corrupted P_1: Let $\mathcal{A}$ be a probabilistic polynomial-time adversary in $\mathsf{Expt\text{-}DistSign}^1_{\mathcal{A},\Pi}(n)$; we construct a probabilistic polynomial-time adversary $\mathcal{S}$ for $\mathsf{Expt\text{-}Sign}_{\mathcal{S},\pi}(n)$. The adversary $\mathcal{S}$ essentially simulates the execution for $\mathcal{A}$, as described in the intuition behind the security of the protocol. Formally:

1. In $\mathsf{Expt\text{-}Sign}$, adversary $\mathcal{S}$ receives $(1^n, Q)$, where Q is the public verification key for ECDSA.
2. $\mathcal{S}$ invokes $\mathcal{A}$ on input 1^n and simulates oracle Π for $\mathcal{A}$ in $\mathsf{Expt\text{-}DistSign}$, answering as described in the following steps:
   (a) $\mathcal{S}$ replies $\perp$ to all queries $(sid, \cdot)$ to Π by $\mathcal{A}$ before the key-generation subprotocol is concluded. $\mathcal{S}$ replies $\perp$ to all queries from $\mathcal{A}$ before it queries $(0, 0)$.
   (b) After $\mathcal{A}$ sends $(0, 0)$ to Π, adversary $\mathcal{S}$ receives $(0, m_1)$ which is P_1's first message in the key generation subprotocol (any other query is ignored). $\mathcal{S}$ computes the oracle reply as follows:
      i. $\mathcal{S}$ parses m_1 into the form $(\mathsf{com\text{-}prove}, 1, Q_1, x_1)$ that P_1 sends to $\mathcal{F}^{R_{DL}}_{\mathsf{com\text{-}zk}}$ in the hybrid model.
      ii. $\mathcal{S}$ verifies that $Q_1 = x_1 \cdot G$. If yes, then it computes $Q_2 = (x_1)^{-1} \cdot Q$ (using the value Q received as the verification key in experiment $\mathsf{Expt\text{-}Sign}$ and the value x_1 from $\mathcal{A}$'s prove message); if no, then $\mathcal{S}$ just chooses a random Q_2.
      iii. $\mathcal{S}$ sets the oracle reply of Π to be $(\mathsf{proof}, 2, Q_2)$ and internally hands this to $\mathcal{A}$ (as if sent by $\mathcal{F}^{R_{DL}}_{\mathsf{zk}}$).
   (c) The next message of the form $(0, m_2)$ received by $\mathcal{S}$ is processed as follows:
      i. $\mathcal{S}$ parses m_2 into the following two messages: **(1)** $(\mathsf{decom\text{-}proof}, sid\|1)$ as $\mathcal{A}$ intends to send to $\mathcal{F}^{R_{DL}}_{\mathsf{com\text{-}zk}}$, and **(2)** $(\mathsf{proof}, 1, N, (p_1, p_2))$ as $\mathcal{A}$ intends to send to $\mathcal{F}^{R_P}_{\mathsf{zk}}$.
      ii. $\mathcal{S}$ verifies that $pk = N = p_1 \cdot p_2$ and that the length of $pk = N$ is as specified, and generates the oracle response to be P_2 aborting if they are not correct.
      iii. Likewise, $\mathcal{S}$ generates the oracle response to be P_2 aborting if $Q_1 \neq x_1 \cdot G$ or $c_{key} \neq \mathsf{Enc}_{pk}(x_1; r)$ or $x_1 \notin \mathbb{Z}_q$. (It can check this since it knows x_1 and can compute the Paillier private key from the prime factors p_1, p_2.)
      iv. If $\mathcal{S}$ simulates an abort, then the experiment concludes (since the honest P_2 no longer participates in the protocol and so all calls to Π_b are ignored). $\mathcal{S}$ does not output anything in this case since no verification key vk is output by P_2 in this case.

v. The next messages of the form $(0, m_i)$ received by $\mathcal{S}$ are processed as part of the zero-knowledge proof that $(c_{key}, pk, Q_1) \in L_{PDL}$. $\mathcal{S}$ verifies this proof running the honest verifier. (Any message of the form $(0, m^*)$ received following the number of messages in the zero-knowledge proof is ignored.)

vi. If $\mathcal{S}$ did not abort, then it stores (x_1, Q, c_{key}) and the distributed key generation phase is completed.

(d) Upon receiving a query of the form (sid, m) where sid is a *new* session identifier, $\mathcal{S}$ queries its signing oracle in experiment Expt-Sign with m and receives back a signature (r, s). Using the ECDSA verification procedure, $\mathcal{S}$ computes the Elliptic curve point R. (Observe that the ECDSA verification works by constructing a point R and then verifying that this defines the same r as in the signature.) Then, queries received by $\mathcal{S}$ from $\mathcal{A}$ with identifier sid are processed as follows:

i. The first message (sid, m_1) is processed by first parsing the message m_1 as $(\mathsf{com\text{-}prove}, sid\|1, R_1, k_1)$. If $R_1 = k_1 \cdot G$ then $\mathcal{S}$ sets $R_2 = (k_1)^{-1} \cdot R$; else it chooses R_2 at random. $\mathcal{S}$ sets the oracle reply to $\mathcal{A}$ to be the message $(\mathsf{proof}, sid\|2, R_2)$ that $\mathcal{A}$ expects to receive. (Note that the value R_2 is computed using R from the ECDSA signature and k_1 as sent by $\mathcal{A}$.)

ii. The second message (sid, m_2) is processed by parsing the message m_2 as $(\mathsf{decom\text{-}proof}, sid\|1)$ from $\mathcal{A}$. If $R_1 \neq k_1 \cdot G$ then $\mathcal{S}$ simulates P_2 aborting and the experiment concludes (since the honest P_2 no longer participates in *any executions* of the protocol and so all calls to Π_b are ignored).
Otherwise, $\mathcal{S}$ chooses a random $\rho \leftarrow \mathbb{Z}_{q^2}$, computes the ciphertext $c_3 \leftarrow \mathsf{Enc}_{pk}([k_1 \cdot s \bmod q] + \rho \cdot q)$, where s is the value from the signature received from $\mathcal{F}_{\text{ECDSA}}$, and sets the oracle reply to $\mathcal{A}$ to be c_3.

3. Whenever $\mathcal{A}$ halts and outputs a pair (m^*, σ^*), adversary $\mathcal{S}$ outputs (m^*, σ^*) and halts.

We proceed to prove that Eq. (1) holds. First, observe that the public-key generated by $\mathcal{S}$ in the simulation with $\mathcal{A}$ equals the public-key Q that it received in experiment Expt-Sign. This is due to the fact that $\mathcal{S}$ defines $Q_2 = (x_1)^{-1} \cdot Q$ when $\mathcal{A}$ is committed to $Q_1 = x_1 \cdot G$. Thus, the public key is defined to be $x_1 \cdot Q_2 = x_1 \cdot (x_1)^{-1} \cdot Q = Q$, as required. We now proceed to show that $\mathcal{A}$'s view in the simulation by $\mathcal{S}$ is statistically close to its view in a real execution of Protocols 3.1 and 3.2. (Note that the view is statistically close when taking $\mathcal{F}_{\mathsf{zk}}$ and $\mathcal{F}_{\mathsf{com\text{-}zk}}$ as ideal functionalities; the real protocol is computationally indistinguishable.) This suffices since it implies that $\mathcal{A}$ outputs a pair (m^*, σ^*) that is a valid signature with the same probability in the simulation and in Expt-DistSign (otherwise, the views can be distinguished by just verifying if the output signature is correct relative to the public key). Since the public key in the simulation is the same public key that $\mathcal{S}$ receives in Expt-Sign, a valid forgery generated by $\mathcal{A}$ in Expt-DistSign constitutes a valid forgery by $\mathcal{S}$ in Expt-Sign. Thus, Eq. (1) follows.

In order to see that the view of $\mathcal{A}$ in the simulation of the key generation phase is statistically close to its view in a real execution of Protocol 3.1 (as in $\mathsf{Expt\text{-}DistSign}$), note that the only difference between the simulation by $\mathcal{A}$ and a real execution with an honest P_2 is the way that Q_2 is generated: P_2 chooses a random x_2 and computes $Q_2 \leftarrow x_2 \cdot G$, whereas $\mathcal{S}$ computes $Q_2 \leftarrow (x_1)^{-1} \cdot Q$, where Q is the public verification key received by $\mathcal{S}$ in $\mathsf{Expt\text{-}Sign}$. We stress that in all other messages and checks, $\mathcal{S}$ behaves exactly as P_2 (note that the zero-knowledge proof of knowledge of the discrete log of Q_2 is simulated by $\mathcal{S}$, but in the $\mathcal{F}_{\mathsf{zk}}, \mathcal{F}_{\mathsf{com\text{-}zk}}$-hybrid model this is identical). Now, since Q is chosen randomly, it follows that the distributions over $x_2 \cdot G$ and $(x_1)^{-1} \cdot Q$ are *identical*. Observe finally that if P_2 does not abort then the public-key defined in both a real execution and the simulation by $\mathcal{S}$ equals $x_1 \cdot Q_2 = Q$. In addition, if $Q_1 \neq x_1 \cdot G$ or $c_{key} \neq \mathsf{Enc}_{pk}(x_1; r)$ or $x_1 \notin \mathbb{Z}_q$ then $\mathcal{S}$ simulates P_2 aborting. In contrast, in a real execution, P_2 aborts in such a case if the zero-knowledge proof of L_{PDL} fails. However, if $Q_1 \neq x_1 \cdot G$ or $c_{key} \neq \mathsf{Enc}_{pk}(x_1; r)$ or $x_1 \notin \mathbb{Z}_q$, then by the soundness of the proof, P_2 aborts in a real execution except with negligible probability. Thus, the view of $\mathcal{A}$ in the simulation is statistically close to a real execution, and the output public key is Q.

In order to see that the view of $\mathcal{A}$ in the simulation of the signing phase is computationally indistinguishable to its view in a real execution of Protocol 3.2 (as in $\mathsf{Expt\text{-}DistSign}$), note that the only difference between the view of $\mathcal{A}$ in a real execution and in the simulation is the way that c_3 is chosen. Specifically, R_2 is distributed identically in both cases due to the fact that R is randomly generated by $\mathcal{F}_{\mathrm{ECDSA}}$ in the signature generation and thus $(k_1)^{-1} \cdot R$ has the same distribution as $k_2 \cdot G$ (this is exactly the same as in the key generation phase with Q). The zero-knowledge proofs and verifications are also identically distributed in the $\mathcal{F}_{\mathsf{zk}}, \mathcal{F}_{\mathsf{com\text{-}zk}}$-hybrid model. Thus, the only difference is c_3: in the simulation it is an encryption of $[k_1 \cdot s \bmod q] + \rho \cdot q$, whereas in a real execution it is an encryption of $s' = (k_2)^{-1} \cdot (m' + rx) + \rho \cdot q$, where $\rho \in \mathbb{Z}_{q^2}$ is random (we stress that all additions here are over the *integers* and not mod q, except for where it is explicitly stated in the protocol description). We stress that the distribution of s' in a real execution is as above, as long as the Paillier key is valid and as long as $c_{key} = \mathsf{Enc}_{pk}(x_1)$ where $Q_1 = x_1 \cdot G$. These properties are guaranteed by the soundness of the zero-knowledge proofs in the key-generation phase, and thus the probability that this doesn't hold is negligible.

We therefore prove that $\mathcal{A}$'s view is indistinguishable by showing that despite this difference, the values are actually *statistically close*. In order to see this, first observe that by the definition of ECDSA signing, $s = k^{-1} \cdot (m' + rx) = (k_1)^{-1} \cdot (k_2)^{-1} \cdot (m + rx) \bmod q$. Thus, $(k_2)^{-1} \cdot (m' + rx) = k_1 \cdot s \bmod q$, implying that there exists some $\ell \in \mathbb{N}$ with $0 \leq \ell < q$ such that $(k_2)^{-1} \cdot (m' + rx) = k_1 \cdot s + \ell \cdot q$. The reason that ℓ is bound between 0 and q is that in the protocol the only operations without a modular reduction are the multiplication of $[(k_2)^{-1} \cdot r \cdot x_2 \bmod q]$ by x_1, and the addition of $[(k_2)^{-1} \cdot m' \bmod q]$. This cannot increase the result by more than q^2. Therefore, the difference between the real execution and simulation with $\mathcal{S}$ is:

1. *Real:* the ciphertext c_3 encrypts $[k_1 \cdot s \bmod q] + \ell \cdot q + \rho \cdot q$
2. *Simulated:* the ciphertext c_3 encrypts $[k_1 \cdot s \bmod q] + \rho \cdot q$

We show that for all k_1, s, ℓ with $k_1, s, \ell \in \mathbb{Z}_q$, the above values are statistically close (for a random choice of $\rho \in \mathbb{Z}_{q^2}$). In order to see this, fix k_1, s, ℓ, and let v be a value. If $v \neq [k_1 \cdot s \bmod q] + \zeta \cdot q$ for some ζ, then neither the real or simulated values can equal v. Else, if $v = [k_1 \cdot s \bmod q] + \zeta \cdot q$ for some ζ, then there are three cases:

1. *Case $\zeta < \ell$:* in this case, v can be obtained in the simulated execution for $\rho < \ell$, but can never be obtained in a real execution.
2. *Case $\zeta > q^2 - 1$):* in this case, v can be obtained in the real execution for $\rho \geq q^2 - 1 - \ell$, but can never be obtained in a simulated execution.
3. *Case $\ell \leq \zeta < q^2 - 1$:* in this case, v can be obtained in both the real and simulated executions, with identical probability (observe that in both the real and simulated executions, ρ is chosen uniformly in $\mathbb{Z}_{q^2}$).

Recall that the statistical distance between two distributions X and Y over a domain $\mathcal{D}$ is defined to be:

$$\Delta(X, Y) = \max_{T \subseteq D} \Big| \Pr[X \in T] - \Pr[Y \in T] \Big|$$

Let X be the values generated in a real execution of the protocol and let Y be the values generated in the simulation with $\mathcal{S}$. Then, taking T to be set of values v for which $\zeta < \ell$, we have that $\Pr[X \in T] = 0$ whereas $\Pr[Y \in T] \leq \frac{q}{q^2} = \frac{1}{q}$ (this holds since $0 \leq \ell < q$ and $\rho \in \mathbb{Z}_{q^2}$). Thus, $\Delta(X, Y) = \frac{1}{q}$, which is negligible. (Taking T to be the set of values v for which $\zeta > q^2 - 1$ would give the same result and are both the maximum since any other values add no difference.) We therefore conclude that the distributions over c_3 in the real and simulated executions are statistically close. This proves that Eq. (1) holds for the case that $b = 1$.

Proof of Eq. (1) for $b = 2$ – corrupted P_2: We follow the same strategy as for the case that P_1 is corrupted, which is to construct a simulator $\mathcal{S}$ that simulates the view of $\mathcal{A}$ while interacting in experiment Expt-Sign. This simulation is easy to construct and similar to the case that P_1 is corrupted, with one difference. Recall that the last message from P_2 to P_1 is an encryption c_3. This ciphertext may be maliciously constructed by $\mathcal{A}$, and the simulator cannot detect this. (Formally, there is no problem for $\mathcal{S}$ to decrypt, since as will be apparent below, it generates the Paillier public key. However, this strategy will fail since in order to prove computational indistinguishability it is necessary to carry out a reduction to the security of Paillier, meaning that the simulation must be designed to work *without knowing* the corresponding private key.) We solve this problem by simply having $\mathcal{S}$ simulate P_1 aborting at some random point. That is, $\mathcal{S}$ chooses a random $i \in \{1, \ldots, p(n) + 1\}$ where $p(n)$ is an upper bound on the number of queries made by $\mathcal{A}$ to Π. If $\mathcal{S}$ chose correctly, then the simulation is fine. Now, since $\mathcal{S}$'s choice of i is correct with probability $\frac{1}{p(n)+1}$, this means

that $\mathcal{S}$ simulates $\mathcal{A}$'s view with probability $\frac{1}{p(n)+1}$ (note that $\mathcal{S}$ can also choose $i = p(n)+1$, which is correct if c_3 is always constructed correctly). Thus, $\mathcal{S}$ can forge a signature in $\mathsf{Expt\text{-}Sign}$ with probability at least $\frac{1}{p(n)+1}$ times the probability that $\mathcal{A}$ forges a signature in $\mathsf{Expt\text{-}DistSign}$.

Let $\mathcal{A}$ be a probabilistic polynomial-time adversary; $\mathcal{S}$ proceeds as follows:

1. In $\mathsf{Expt\text{-}Sign}$, adversary $\mathcal{S}$ receives $(1^n, Q)$, where Q is the public verification key for ECDSA.
2. Let $p(\cdot)$ denote an upper bound on the number of queries that $\mathcal{A}$ makes to Π in experiment $\mathsf{Expt\text{-}DistSign}$. Then, $\mathcal{S}$ chooses a random $i \in \{1, \ldots, p(n)+1\}$.
3. $\mathcal{S}$ invokes $\mathcal{A}$ on input 1^n and simulates oracle Π for $\mathcal{A}$ in $\mathsf{Expt\text{-}DistSign}$, answering as described in the following steps:
   (a) $\mathcal{S}$ replies $\perp$ to all queries $(sid, \cdot)$ to Π by $\mathcal{A}$ before the key-generation subprotocol is concluded. $\mathcal{S}$ replies $\perp$ to all queries from $\mathcal{A}$ before it queries $(0,0)$.
   (b) After $\mathcal{A}$ sends $(0,0)$ to Π, adversary $\mathcal{S}$ computes the oracle reply to be $(\mathsf{proof\text{-}receipt}, 1)$ as $\mathcal{A}$ expects to receive.
   (c) The next message of the form $(0, m_1)$ received by $\mathcal{S}$ (any other query is ignored) is processed as follows:
      i. $\mathcal{S}$ parses m_1 into the form $(\mathsf{prove}, 2, Q_2, x_2)$ that P_2 sends to $\mathcal{F}^{R_{DL}}_{\mathsf{com\text{-}zk}}$ in the hybrid model.
      ii. $\mathcal{S}$ verifies that Q_2 is a non-zero point on the curve and that $Q_2 = x_2 \cdot G$; if not, it simulates P_1 aborting, and halts (there is no point outputting anything since no verification key is output by P_1 in this case and so the output of $\mathsf{Expt\text{-}DistSign}$ is always 0).
      iii. $\mathcal{S}$ generates a valid Paillier key-pair (pk, sk), computes $c_{key} = \mathsf{Enc}_{pk}(\tilde{x}_1)$ for a random $\tilde{x}_1 \in \mathbb{Z}_{q/3}$.
      iv. $\mathcal{S}$ sets the oracle response to $\mathcal{A}$ to be the messages $(\mathsf{decom\text{-}proof}, 1, Q_1)$ and $(\mathsf{proof}, 1, N)$, where $Q_1 = (x_2)^{-1} \cdot Q$ with Q as received by $\mathcal{S}$ initially.
      v. $\mathcal{S}$ runs the simulator for the zero-knowledge proof for language L_{PDL}, with the residual $\mathcal{A}$ as verifier.

      $\mathcal{S}$ stores (x_2, Q, c_{key}) and the key distribution phase is completed.
   (d) Upon receiving a query of the form (sid, m) where sid is a *new* session identifier, $\mathcal{S}$ computes the oracle reply to be $(\mathsf{proof\text{-}receipt}, sid\|1)$ as $\mathcal{A}$ expects to receive, and hands it to $\mathcal{A}$.

      Next, $\mathcal{S}$ queries its signing oracle in experiment $\mathsf{Expt\text{-}Sign}$ with m and receives back a signature (r, s). Using the ECDSA verification procedure, $\mathcal{S}$ computes the Elliptic curve point R. Then, queries received by $\mathcal{S}$ from $\mathcal{A}$ with identifier sid are processed as follows:
      i. The first message (sid, m_1) is processed by first parsing the message m_1 as $(\mathsf{prove}, sid\|2, R_2, k_2)$ that $\mathcal{A}$ sends to $\mathcal{F}^{R_{DL}}_{\mathsf{zk}}$. $\mathcal{S}$ verifies that $R_2 = k_2 \cdot G$ and that R_2 is a non-zero point on the curve; otherwise, it simulates P_1 aborting. $\mathcal{S}$ computes $R_1 = (k_2)^{-1} \cdot R$ and sets the oracle reply to be $(\mathsf{decom\text{-}proof}, sid\|, R_1)$ as if coming from $\mathcal{F}^{R_{DL}}_{\mathsf{com\text{-}zk}}$.

ii. The second message (sid, m_2) is processed by parsing m_2 as c_3. If this is the ith call by $\mathcal{A}$ to the oracle Π, then $\mathcal{S}$ simulates P_1 aborting (and not answering any further oracle calls). Otherwise, it continues.

4. Whenever $\mathcal{A}$ halts and outputs a pair (m^*, σ^*), adversary $\mathcal{S}$ outputs (m^*, σ^*) and halts.

As in the case that P_1 is corrupted, the public-key generated by $\mathcal{S}$ in the simulation with $\mathcal{A}$ equals the public-key Q that it received in experiment $\mathsf{Expt\text{-}Sign}$. Now, let j be the *first* call to oracle Π with (sid, c_3) where c_3 is such that P_1 does not obtain a valid signature (r, s) with respect to Q. Then, we argue that if $j = i$, then the only difference between the distribution over $\mathcal{A}$'s view in a real execution and in the simulated execution by $\mathcal{S}$ is the ciphertext c_{key}. Specifically, in a real execution $c_{key} = \mathsf{Enc}_{pk}(x_1)$ where $Q_1 = x_1 \cdot G$, whereas in the simulation $c_{key} = \mathsf{Enc}_{pk}(\tilde{x}_1)$ for a random $\tilde{x}_1$ and is independent of $Q_1 = x_1 \cdot G$.[2] Observe, however, that $\mathcal{S}$ does not use the private-key for Paillier at all in the simulation. Thus, indistinguishability of this simulation follows from a straightforward reduction to the indistinguishability of the encryption scheme, under chosen-plaintext attacks.

This proves that

$$\left|\Pr[\mathsf{Expt\text{-}Sign}_{\mathcal{S},\pi}(1^n) = 1 \mid i = j] - \Pr[\mathsf{Expt\text{-}DistSign}^2_{\mathcal{A},\Pi}(1^n) = 1]\right| \leq \mu(n),$$

and so

$$\begin{aligned}\Pr[\mathsf{Expt\text{-}DistSign}^2_{\mathcal{A},\Pi}(1^n) = 1] &\leq \frac{\Pr[\mathsf{Expt\text{-}Sign}_{\mathcal{S},\pi}(1^n) = 1 \wedge i = j]}{\Pr[i = j]} + \mu(n) \\ &\leq \frac{\Pr[\mathsf{Expt\text{-}Sign}_{\mathcal{S},\pi}(1^n) = 1]}{1/(p(n)+1)} + \mu(n)\end{aligned}$$

and so

$$\Pr[\mathsf{Expt\text{-}Sign}_{\mathcal{S},\pi}(1^n) = 1] \geq \frac{\Pr[\mathsf{Expt\text{-}DistSign}^2_{\mathcal{A},\Pi}(1^n) = 1]}{p(n)+1} - \mu(n).$$

This implies that if $\mathcal{A}$ forges a signature in $\mathsf{Expt\text{-}DistSign}^2_{\mathcal{A},\Pi}$ with non-negligible probability, then $\mathcal{S}$ forges a signature in $\mathsf{Expt\text{-}Sign}_{\mathcal{S},\pi}$ with non-negligible probability, in contradiction to the assumed security of ECDSA. ■

5 Simulation Proof of Security (With a New Assumption)

There are advantages to full simulation based proofs of security (via the real/ideal paradigm). Observe that we proved the security of our protocol in Section 4 by

[2] As before, this is true in the $\mathcal{F}_{\mathsf{zk}}, \mathcal{F}_{\mathsf{com\text{-}zk}}$-hybrid model; by using UC-secure protocols for $\mathcal{F}_{\mathsf{zk}}, \mathcal{F}_{\mathsf{com\text{-}zk}}$ the result is computationally indistinguishable. There is also a difference due to the fact that the zero-knowledge proof for L_{PDL} is simulated and not real; however, this is computationally indistinguishable.

simulating the view of $\mathcal{A}$ in a real execution. In fact, our simulation can be used to prove the security of our protocol under the real/ideal world paradigm except for exactly one place. Recall that when P_2 is corrupted, $\mathcal{S}$ cannot determine if c_3 is correctly constructed or not. Thus, $\mathcal{S}$ simply chooses a random point and "hopes" that the jth value c_3 generated is the first badly constructed c_3. This suffices for a game-based definition, but it does not suffice for simulation-based security definitions. Thus, in order to be able to prove our protocol using simulation, we need to be able to determine if c_3 was constructed correctly. Of course, we could add zero-knowledge proofs to the protocol, but these would be very expensive. Alternatively, we consider a rather ad-hoc but plausible assumption that suffices. The assumption is formalized below, along with a full proof of security under this assumption.

5.1 Definition of Security

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 5.1.

FIGURE 5.1 (The ECDSA Functionality $\mathcal{F}_{\text{ECDSA}}$)

Functionality $\mathcal{F}_{\text{ECDSA}}$ works with parties P_1 and P_2, as follows:

- Upon receiving $\mathsf{KeyGen}(\mathbb{G}, G, q)$ from both P_1 and P_2, 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 $Q = x \cdot G$. Choose a hash function $H_q : \{0,1\}^* \to \{0,1\}^{\lfloor \log |q| \rfloor}$, and store $(\mathbb{G}, g, q, H_q, x)$.
  2. Send Q (and H_q) to both P_1 and P_2.
  3. Ignore future calls to KeyGen.
- Upon receiving $\mathsf{Sign}(sid, m)$ from both P_1 and P_2, if KeyGen was already called and sid has not been previously used, compute an ECDSA signature (r, s) on m, and send it to both P_1 and P_2. (Specifically, choose a random $k \leftarrow \mathbb{Z}_q^*$, compute $(r_x, r_y) = k \cdot G$ and $r = r_x \bmod q$. Finally, compute $s \leftarrow k^{-1}(H_q(m) + rx)$ and output (r, s).)

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

Security in the presence of malicious adversaries. We prove security according to the standard simulation paradigm with the real/ideal model [5,17]. We prove security 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 corrupted party receives output first and then sends either $\mathsf{continue}$ or abort to the trusted party to determine whether or not the honest party also receives 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.

5.2 Background and New Assumption

In Section 4 , we proved the security of our protocol under a game-based definition. In some sense, proving security via simulation-based definitions (following the ideal/real model paradigm) is preferable. In particular, it guarantees security under composition. Following our proof in Section 4.2 closely, one may observe that $\mathcal{S}$ is essentially a simulator for an ideal functionality that securely computes ECDSA. Indeed, $\mathcal{S}$ is invoked with a public-key, and can use its oracle in $\mathsf{Expt\text{-}Sign}$ to obtain a signature on any value it wishes. This is very similar to an ideal functionality that generates a public key and can be used to generate signatures. The only problem with the simulation strategy used in Section 4.2 is that in the case that P_2 is corrupted, $\mathcal{S}$ just guesses if c_3 is correctly constructed. Needless to say, this is not allowed in a simulation-based proof. One may be tempted to solve this problem by saying that since $\mathcal{S}$ generates the Paillier key-pair (pk, sk) when playing P_1, it can decrypt c_3 and check if the value is generated as expected. However, when trying to formally prove this, one needs to show a reduction to the indistinguishability of the encryption scheme (since the simulator does not know x_1 and so cannot provide $c_{key} = \mathsf{Enc}_{pk}(x_1)$). In this reduction, the simulator is given pk externally and does not know sk (see the proof of the key generation subprotocol in Section 4.2). Thus, in this reduction, it is not possible to decrypt c_3 and the appropriate distributions cannot be generated.

We introduce a new assumption under which it is possible to prove the full simulation-based security of Protocol 3.2 without any modifications. The assumption is non-standard, but very plausible. Consider an adversary who is given a Paillier encryption of a (high-entropy) secret value w; denote $c = \mathsf{Enc}_{pk}(w)$. Then, the adversary can always randomize c to generate an encryption c' of the same w, but without anyone but itself and the secret-key owner knowing whether c and c' encrypt the same value. In addition, the adversary can always generate an encryption c' of a plaintext value that it knows but without knowing whether c and c' encrypt the same value. Now, consider a setting where an adversary is given an oracle $\mathcal{O}_c(c')$ that outputs 1 if and only if $\mathsf{Dec}_{sk}(c') = \mathsf{Dec}_{sk}(c)$, where $c = \mathsf{Enc}_{pk}(w)$ is the challenge ciphertext, and the adversary's task is to learn w. Clearly, the adversary can use this oracle to try and guess the value encrypted in c one at a time (just guess x', compute $c' = \mathsf{Enc}_{pk}(x')$ and query $\mathcal{O}_c(c')$). However, since w has high entropy, this seems to be futile. Furthermore, it seems that the oracle $\mathcal{O}_c$ cannot help in any other way.

Extending the above a further step, the adversary can generate any *affine* function of w by choosing scalars α and β and computing $c' = \alpha \odot (\mathsf{Enc}_{pk}(\beta) \oplus c)$

$= \mathsf{Enc}_{pk}(\alpha+\beta\cdot w)$. Then, as before, $\mathcal{A}$ tries to output w given an oracle $\mathcal{O}_c(c',\alpha,\beta)$ that outputs 1 if and only if $\mathsf{Dec}_{sk}(c') = \alpha + \beta \cdot \mathsf{Dec}_{sk}(c)$. The adversary can actually use this oracle, and the fact that the modulus inside the encryption is not q (for Paillier it is a much larger RSA modulus), in order to carry out a binary search on w. In particular, given w, the adversary can check if $w < \frac{q}{2}$ by setting $\beta = 1$ and $\alpha = N - \frac{q}{2}$, where N is the modulus of the encryption. In this case, if $w < \frac{q}{2}$ then the oracle will return 1 since equality holds over the integers and thus modulo q. However, if $w \geq \frac{q}{2}$, then $\alpha + w \geq N$ and so the encrypted value will be reduced modulo N and equality will *not* hold modulo q. Thus, the oracle will return 0. The adversary can proceed in a similar way and learn w efficiently. However, observe that in the signing protocol, as soon as an abort occurs, the parties do not participate further. Thus, we only need to be able to simulate up to the first time that there is inequality in the call to the oracle. Thus, we modify the oracle so that the first time that there is inequality, it returns 0 and then does not respond to any more queries. This prevents the adversary from learning anything significant from the fact that there are different moduli, and in fact can only learn as long as each "guess" always gives equality.

In order to formally define a security experiment including such an oracle, one must consider the task of the adversary. Since w must be a high-entropy random value one cannot consider the standard indistinguishability game. Rather, one could formalize a simple task where some w is randomly chosen and the adversary is given $(pk, \mathsf{Enc}_{pk}(w))$ and oracle access to $\mathcal{O}$ above (recall that $\mathcal{O}$ answers only until it returns 0 for the first time, and it then halts), and its task is to output w (in entirety). This is very plausible since without the oracle it is clearly hard, and the oracle only answers queries (c',α,β) by determining if "c' encrypts $\alpha+\beta\cdot x$", which essentially gives a *single guess* on the value of w. However, requiring that the adversary output the entire w turns out to not be very helpful for us. This is due to the fact that w must maintain some property of secrecy. We therefore extend this experiment by giving the adversary either $(pk, f(w_0), \mathsf{Enc}_{pk}(w_0))$ or $(pk, f(w_0), \mathsf{Enc}_{pk}(w_1))$, where w_0, w_1 are random and f is a *one-way function*. The adversary's task is to guess which input type it received (with the input to the one-way function equal to what is encrypted or independent of it), and it is given the oracle $\mathcal{O}$ above to help it. Note that f may reveal some information about w_0 (since it is only a one-way function), but if f is somehow *unrelated* of the encryption scheme, then it still seems that this should not help very much.

For our actual experiment, we will define the one-way function to be the computation $w_0 \cdot G$ in a group where the discrete log is hard. Observe that here the one-way function is related to the discrete log problem over Elliptic curve groups, whereas the encryption is Paillier and thus seems completely unrelated. Thus, we conjecture that this problem is hard. Since we consider a group, the equality that is actually checked by the oracle is modulo q, where q is the order of the group.

Formal assumption definition. The above description leads to the following experiment. Let G be a generator of a group $\mathbb{G}$ of order q. Consider the following experiment with an adversary $\mathcal{A}$, denoted $\mathsf{Expt}_{\mathcal{A}}(1^n)$:

1. Generate a Paillier key pair (pk, sk).
2. Choose random $w_0, w_1 \in \mathbb{Z}_q$ and compute $Q = w_0 \cdot G$.
3. Choose a random bit $b \in \{0,1\}$ and compute $c = Enc_{pk}(w_b)$.
4. Let $b' = \mathcal{A}^{\mathcal{O}_c(\cdot,\cdot,\cdot)}(pk, c, Q)$, where $\mathcal{O}_c(c', \alpha, \beta) = 1$ if and only if $Dec_{sk}(c') = \alpha + \beta \cdot w_b \bmod q$ and $\mathcal{O}$ halts after the first time it returns 0.
5. The output of the experiment is 1 if and only if $b' = b$.

We define the following:

Definition 5.2. *We say that the* Paillier-EC assumption *is hard if for every probabilistic polynomial-time adversary $\mathcal{A}$ there exists a negligible function μ such that* $\Pr[\mathsf{Expt}_{\mathcal{A}}(1^n) = 1] \leq \frac{1}{2} + \mu(n)$.

The assumption in Definition 5.2 is rather ad-hoc and tailored to the problem at hand. However, it is very plausible and enables us prove full simulation without modifying the protocol.

5.3 Proof of Security

Under the above assumption, we are able to prove full simulation-based security of our protocol. We show this now. We assume only that the Paillier-EC assumption is hard, since this trivially implies that the Paillier encryption scheme is indistinguishable under chosen-plaintext attacks.

Theorem 5.3. *Assume that the* Paillier-EC assumption is hard. *Then, Protocol 3.2 securely computes $\mathcal{F}_{\rm ECDSA}$ in the $(\mathcal{F}_{\sf zk}, \mathcal{F}_{\sf com\text{-}zk})$-hybrid model in the presence of a malicious static adversary (under the full ideal/real definition).*

Proof. We separately prove security for the case of a corrupted P_1 and a corrupted P_2. Let $\mathcal{A}$ be an adversary who has corrupted P_1; we construct a simulator $\mathcal{S}$. We separately show how to simulate the key generation and sign subprotocols.

Simulating key generation – corrupted P_1: The intuition behind the simulation of the key generation was already provided above; we therefore proceed directly to the details.

1. Upon input $\mathsf{KeyGen}(\mathbb{G}, G, q)$, simulator $\mathcal{S}$ sends $\mathsf{KeyGen}(\mathbb{G}, G, q)$ to $\mathcal{F}_{\rm ECDSA}$ and receives back Q.
2. $\mathcal{S}$ invokes $\mathcal{A}$ upon input $\mathsf{KeyGen}(\mathbb{G}, G, q)$ and receives $(\mathsf{com\text{-}prove}, 1, Q_1, x_1)$ as $\mathcal{A}$ intends to send to $\mathcal{F}_{\sf zk}^{R_{DL}}$.
3. $\mathcal{S}$ verifies that $Q_1 = x_1 \cdot G$. If yes, then it computes $Q_2 = (x_1)^{-1} \cdot Q$ (using the value Q received from $\mathcal{F}_{\rm ECDSA}$ and x_1 from $\mathcal{A}$'s prove message); if no, then $\mathcal{S}$ just chooses a random Q_2.
4. $\mathcal{S}$ internally hands $(\mathsf{proof}, 2, Q_2)$ to $\mathcal{A}$ as if sent by $\mathcal{F}_{\sf zk}^{R_{DL}}$.
5. $\mathcal{S}$ receives $(\mathsf{decom\text{-}proof}, sid\|1)$ as $\mathcal{A}$ intends to send to $\mathcal{F}_{\sf com\text{-}zk}^{R_{DL}}$, receives $(\mathsf{proof}, 1, N, (p_1, p_2))$ as $\mathcal{A}$ intends to send to $\mathcal{F}_{\sf zk}^{R_P}$, and honestly verifies the zero-knowledge proof of language L_{PDL}.

6. $\mathcal{S}$ verifies that $pk = N = p_1 \cdot p_2$ and the length of $pk = N$, and simulates P_2 aborting if they are not correct.
7. $\mathcal{S}$ simulates P_2 aborting if $Q_1 \neq x_1 \cdot G$ or $c_{key} \neq \mathsf{Enc}_{pk}(x_1; r)$ or $x_1 \notin \mathbb{Z}_q$. ($\mathcal{S}$ can check this since it knows x_1 and it can compute the Paillier private key from the prime factors p_1, p_2.)
8. $\mathcal{S}$ sends continue to $\mathcal{F}_{\text{ECDSA}}$ for P_2 to receive output, and stores x_1, Q, c_{key}.

We prove that the joint distribution of $\mathcal{A}$'s view and P_2's output in the ideal simulation is identically distributed to in a real protocol execution. The main difference between the simulation by $\mathcal{A}$ and a real execution with an honest P_2 is the way that Q_2 is generated: P_2 chooses a random x_2 and computes $Q_2 \leftarrow x_2 \cdot G$, whereas $\mathcal{S}$ computes $Q_2 \leftarrow (x_1)^{-1} \cdot Q$. However, as we have already seen, since Q is chosen randomly, the distributions over $x_2 \cdot G$ and $(x_1)^{-1} \cdot Q$ are *identical*. Another difference is that $\mathcal{S}$ simulates P_2 aborting if $Q_1 \neq x_1 \cdot G$ or $c_{key} \neq \mathsf{Enc}_{pk}(x_1; r)$ or $x_1 \notin \mathbb{Z}_q$, whereas in a real execution P_2 aborts in this case only if the zero-knowledge proof of L_{PDL} is not accepted. By the soundess of this proof, this difference is at most negligible. We stress that in all other messages and checks, $\mathcal{S}$ behaves in the same way as P_2 (note that the zero-knowledge proofs of Q_2 is simulated by $\mathcal{S}$, but in the $\mathcal{F}_{\mathsf{zk}}, \mathcal{F}_{\mathsf{com\text{-}zk}}$-hybrid model these is identical). Now, Observe finally that if P_2 does not abort then the public-key defined in both the ideal and real executions equals $x_1 \cdot Q_2 = Q$. Thus, the joint distributions over $\mathcal{A}$'s view and P_2's output are statistically close.

We remark that c_{key} is guaranteed to be an encryption of x_1 where $Q_1 = x_1 \cdot G$. This is guaranteed by the zero-knowledge proof for the language L_{PDL}; we will use this fact below.

Simulating signing – corrupted P_1: The idea behind the security of the signing subprotocol is that a corrupted P_1 cannot do anything since all it does is participate in a "coin tossing" protocol to generate R and receives a ciphertext c_3 from P_2. Since the coin-tossing subprotocol is simulatable, a simulator can make the result equal the R using in a signature received from the trusted party computing $\mathcal{F}_{\text{ECDSA}}$. Thus, the main challenge is in proving that a simulator can generate the corrupted P_1's view of the decryption of c_3, given only the signature (r, s) from $\mathcal{F}_{\text{ECDSA}}$.

1. Upon input $\mathsf{Sign}(sid, m)$, simulator $\mathcal{S}$ sends $\mathsf{Sign}(sid, m)$ to $\mathcal{F}_{\text{ECDSA}}$ and receives back a signature (r, s).
2. Using the ECDSA verification procedure, $\mathcal{S}$ computes the point R.
3. $\mathcal{S}$ invokes $\mathcal{A}$ with input $\mathsf{Sign}(sid, m)$ and simulates the first three messages so that the result is R. This follows the *exact* strategy as used in the simulation of the key generation phase, as follows (in brief):
   (a) $\mathcal{S}$ receives $(\mathsf{com\text{-}prove}, sid\|1, R_1, k_1)$ from $\mathcal{A}$.
   (b) If $R_1 = k_1 \cdot G$ then $\mathcal{S}$ sets $R_2 = (k_1)^{-1} \cdot R$; else it chooses R_2 at random. $\mathcal{S}$ hands $\mathcal{A}$ the message $(\mathsf{proof}, sid\|2, R_2)$.
   (c) $\mathcal{S}$ receives $(\mathsf{decom\text{-}proof}, sid\|1)$ from $\mathcal{A}$. If $R_1 \neq k_1 \cdot G$ then $\mathcal{A}$ simulates P_2 aborting and sends abort to the trusted party computing $\mathcal{F}_{\text{ECDSA}}$. Otherwise, it continues.

4. $\mathcal{S}$ chooses a random $\rho \leftarrow \mathbb{Z}_{q^2}$, computes $c_3 \leftarrow \mathsf{Enc}_{pk}([k_1 \cdot s \bmod q] + \rho \cdot q)$, where s is the value from the signature received from $\mathcal{F}_{\text{ECDSA}}$, and internally hands c_3 to $\mathcal{A}$.

The only difference between the view of $\mathcal{A}$ in a real execution and in the simulation is the way that c_3 is chosen. Specifically, R_2 is distributed identically in both cases due to the fact that R is randomly generated by $\mathcal{F}_{\text{ECDSA}}$ in the signature generation and thus $(k_1)^{-1} \cdot R$ has the same distribution as $k_2 \cdot G$. The zero-knowledge proofs and verifications are also identically distributed in the $\mathcal{F}_{\mathsf{zk}}, \mathcal{F}_{\mathsf{com\text{-}zk}}$-hybrid model. Thus, the only difference is c_3: in the simulation it is an encryption of $[k_1 \cdot s \bmod q] + \rho \cdot q$, whereas in a real execution it is an encryption of $s' = (k_2)^{-1} \cdot (m' + rx) + \rho \cdot q$, where $\rho \in \mathbb{Z}_{q^2}$ is random (we stress that all additions here are over the *integers* and not mod$q$, except for where it is explicitly stated in the protocol description). The fact that this is statistically close has already been shown in the proof of Theorem 4.5. This completes the proof of this simulation case.

Simulating key generation – corrupted P_2: We now consider the case of a malicious S_2.

1. Upon input $\mathsf{KeyGen}(\mathbb{G}, G, q)$, simulator $\mathcal{S}$ sends $\mathsf{KeyGen}(\mathbb{G}, G, q)$ to $\mathcal{F}_{\text{ECDSA}}$ and receives back Q.
2. $\mathcal{S}$ generates a valid Paillier key-pair (pk, sk), computes $c_{key} = \mathsf{Enc}_{pk}(\tilde{x}_1)$ for a random $\tilde{x}_1 \in \mathbb{Z}_q$, and internally hands $\mathcal{A}$ the message $(\mathsf{proof\text{-}receipt}, 1)$ as if sent by $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{DL}}$, and the pair (pk, c_{key}) as if sent by P_1.
3. $\mathcal{S}$ receives Q_2 as $\mathcal{A}$ intends to send to P_1, and $(\mathsf{prove}, 2, Q_2, x_2)$ as $\mathcal{A}$ intends to send to $\mathcal{F}_{\mathsf{zk}}^{R_{DL}}$.
4. $\mathcal{S}$ verifies that Q_2 is a non-zero point on the curve and that $Q_2 = x_2 \cdot G$; if not, it simulates P_1 aborting and halts.
5. $\mathcal{S}$ computes $Q_1 = (x_2)^{-1} \cdot Q$ and hands $\mathcal{A}$ the message $(\mathsf{decom\text{-}proof}, 1, Q_1)$ as if sent by $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{DL}}$.
6. $\mathcal{S}$ runs the simulator for the zero-knowledge proof of language L_{PDL} for common statement (c_{key}, pk, Q_1) and with the residual $\mathcal{A}$ as verifier.
7. $\mathcal{S}$ sends $\mathsf{continue}$ to $\mathcal{F}_{\text{ECDSA}}$ for P_1 to receive output, and stores Q.

It is immediate that the distributions of $\mathcal{A}$'s view in a real and ideal execution are identical, except for c_{key} which equals $\mathsf{Enc}_{pk}(x_1)$ where $Q_1 = x_1 \cdot G$ in a real execution but equals $\mathsf{Enc}_{pk}(\tilde{x}_1)$ for a random $\tilde{x}_1$ in the ideal simulation, and except for the simulation of the zero-knowledge proof of L_{PDL}. The latter is indistinguishable from the zero-knowledge property of the proof. Regarding the former, observe that $\mathcal{S}$ does not use the private-key at all. Thus, indistinguishability of this simulation follows from a straightforward reduction to the indistinguishability of the encryption scheme, under chosen-plaintext attacks. The fact that the *joint view* of the adversary $\mathcal{A}$ and the honest party P_1 is indistinguishable follows from the fact that the honest party always outputs $Q = x_1 \cdot Q_2 = x_2 \cdot Q_1$ in a real protocol execution, where $Q_1 = x_1 \cdot G$. In the simulation, we have that $Q_1 = (x_2)^{-1} \cdot Q$ and thus $x_2 \cdot Q_1 = x_2 \cdot (x_2)^{-1} \cdot Q = Q$, exactly as in the real protocol execution.

Simulating signing – corrupted P_2: The simulator for the signing phase works as follows:

1. Upon input $\mathsf{Sign}(sid, m)$, simulator $\mathcal{S}$ sends $\mathsf{Sign}(sid, m)$ to $\mathcal{F}_{\text{ECDSA}}$ and receives back a signature (r, s).
2. Using the ECDSA verification procedure, $\mathcal{S}$ computes the point R.
3. $\mathcal{S}$ invokes $\mathcal{A}$ with input $\mathsf{Sign}(sid, m)$ and internally hands $\mathcal{A}$ the message $(\mathsf{proof\text{-}receipt}, sid\|1)$ as if sent by $\mathcal{F}^{R_{DL}}_{\mathsf{com\text{-}zk}}$.
4. $\mathcal{S}$ receives R_2 as $\mathcal{A}$ intends to send to P_1, and $(\mathsf{prove}, sid||2, R_2, k_2)$ as $\mathcal{A}$ intends to send to $\mathcal{F}^{R_{DL}}_{\mathsf{zk}}$.
5. $\mathcal{S}$ verifies that $R_2 = k_2 \cdot G$ and that R_2 is a non-zero point on the curve; otherwise, it simulates P_1 aborting.
6. $\mathcal{S}$ computes $R_1 \leftarrow (k_2)^{-1} \cdot R$ and internally hands $(\mathsf{decom\text{-}proof}, sid\|1, R_1)$ to $\mathcal{A}$ as if coming from $\mathcal{F}^{R_{DL}}_{\mathsf{com\text{-}zk}}$.
7. $\mathcal{S}$ receives c_3 from P_1, decrypts it using sk and reduces the result modulo q. $\mathcal{S}$ checks if the result equals $\big((k_2)^{-1} \cdot m'\big) + \big((k_2)^{-1} \cdot r \cdot x_2\big) \cdot \tilde{x}_1 \bmod q$, where $c_{key} = \mathsf{Enc}_{pk}(\tilde{x}_1)$ was as generated by P_1 in the key-generation simulation. If the result is equal, then $\mathcal{S}$ instructs the trusted party to provide the output to the honest party (by sending $\mathsf{continue}$). Otherwise, it instructs it to abort (by sending abort).

It is clear that the distribution over the messages seen by P_2 is identical, except for the encryption of c_{key} which is computationally indistinguishable. Furthermore, there is exactly one value modulo q that P_2 can use to generate c_3, and this is validated by $\mathcal{S}$.[3] Formally, we need to show that the output distributions in the ideal model of *both* the key generation and signing phases are computationally indistinguishable from a real execution. In order to do this, we need to reduce the security to that of Paillier encryption since this is the only difference. However, in the simulation, $\mathcal{S}$ *must have the private key sk* in order to decrypt c_3 and verify that $\mathcal{A}$ (controlling P_2) computed the correct value. Thus, it is not possible to prove this via a standard reduction to the indistinguishability of the encryption scheme. We therefore prove this under the Paillier-EC assumption.

We modify $\mathcal{S}$ to a simulator $\mathcal{S}'$ who is given an oracle $\mathcal{O}_c(c', \alpha, \beta)$ that outputs 1 if and only if $Dec_{sk}(c', \alpha, \beta) = \alpha + \beta \cdot \tilde{x}_1 \bmod q$. Observe that $\mathcal{S}'$ can complete the simulation exactly as $\mathcal{S}$ as follows:

1. Compute $\alpha = (k_2)^{-1} \cdot m' \bmod q$.
2. Compute $\beta = (k_2)^{-1} \cdot r \cdot x_2 \bmod q$.
3. Query $\mathcal{O}_c(c_3, \alpha, \beta)$ and denote the response by b.
4. If $b = 1$ then $\mathcal{S}'$ continues like $\mathcal{S}$ when $\mathsf{Dec}_{sk}(c_3) = \big((k_2)^{-1} \cdot m'\big) + \big((k_2)^{-1} \cdot r \cdot x_2\big) \cdot \tilde{x}_1 \bmod q$.

It is immediate that these checks by $\mathcal{S}$ and $\mathcal{S}'$ are equivalent. In order to see this, observe that $\mathsf{Dec}_{sk}(c_3) = \big((k_2)^{-1} \cdot m'\big) + \big((k_2)^{-1} \cdot r \cdot x_2\big) \cdot \tilde{x}_1 \bmod q$ is equivalent to $\mathsf{Dec}_{sk}(c_3) = \alpha + \beta \cdot \tilde{x}_1 \bmod q$ which is equivalent to $\mathcal{O}_c(c_3, \alpha, \beta) = 1$. Thus, $\mathcal{S}$ accepts if and only if $\mathcal{S}'$ accepts.

[3] Note that for every valid ECDSA signature (r, s), the pair $(r, -s)$ is also a valid signature. Nevertheless, since the "smaller" of $s, -s$ is always taken, the value is unique.

We now construct a distinguisher D for the Paillier-EC experiment Expt_D, such that if $b = 0$ then the distribution generated by D is exactly that generated in a real execution whereas if $b = 1$ then the distribution is that generated by $\mathcal{S}'$. D receives (pk, c, Q) and runs the simulation of the key generation (as described above) with the given pk and Q. In addition, D sets $c_{key} = c$ as received. Recall that the simulation of this phase doesn't require sk and so this works. Next, D proceeds to simulate the signing phase, following the instructions of $\mathcal{S}'$. In particular, it uses its oracle $\mathcal{O}$ in order to determine whether to send $\mathsf{continue}$ or abort for P_1 to receive output. Note that D never needs to query $\mathcal{O}$ after an abort occurs; this is needed since $\mathcal{O}$ halts as soon as it returns 0 (inequality).

Observe that if $b = 0$ in the experiment then $c_{key} = \mathsf{Enc}_{pk}(w_0)$ and $Q = w_0 \cdot G$. Setting $x_1 = w_0$, these values are distributed exactly as in a real execution. Furthermore, P_1 outputs a signature if and only if c_3 encrypts $s' = (k_2)^{-1} \cdot (m' + r \cdot x_1) \bmod q$ which is equivalent to (r, s) being a valid signature where $s = (k_1)^{-1} \cdot s' \bmod q$. Thus, this is exactly a real execution. In contrast, if $b = 1$ in the experiment then $c_{key} = \mathsf{Enc}_{pk}(w_1)$ and $Q = w_0 \cdot G$. Setting $x_1 = w_0$ and $\tilde{x}_1 = w_1$, we have that this is exactly the distribution generated by $\mathcal{S}'$. Thus, by the Paillier-EC assumption, we have that the output distribution generated by $\mathcal{S}'$ in the ideal model is computationally indistinguishable from the output distribution in a real execution.

Since the output distributions of $\mathcal{S}$ and $\mathcal{S}'$ in the ideal model are identical, as described, we conclude that the output distribution generated by $\mathcal{S}$ in the ideal model is computationally indistinguishable from the output distribution in a real execution, thus concluding the proof. ■

6 Zero-Knowledge Proof for the Language L_{PDL}

In this section, we present an efficient construction of a zero-knowledge proof for the language L_{PDL}, defined by:

$$L_{PDL} = \{(c, pk, Q_1, \mathbb{G}, G, q) \mid \exists (x_1, r)) : c = \mathsf{Enc}_{pk}(x_1; r) \text{ and } Q_1 = x_1 \cdot G \text{ and } x_1 \in \mathbb{Z}_q\}.$$

Intuitively, this relation means that c is a valid Paillier encryption of the discrete log of Q_1. The idea behind the proof is as follows. First, the prover P proves that the value encrypted inside c is in $\mathbb{Z}_q$; this is a "range proof". For simplicity of implementation, we use a proof that guarantees that $\mathsf{Dec}_{sk}(c) \in \mathbb{Z}_q$, but is only complete if $\mathsf{Dec}_{sk}(c) \in \mathbb{Z}_{q/3}$. This suffices since P_1 can choose $x_1 \in \mathbb{Z}_{q/3}$ and this does not affect security, as discussed in Section 3.2. In addition, V chooses random values a and b, and computes an encryption of $\alpha = a \cdot x_1 + b$ using the homomorphic properties of Paillier encryption on the input ciphertext c, and sends it to P. In addition, V locally computes $Q' = a \cdot Q_1 + b \cdot G$. The key observation is that if $Q_1 = x_1 \cdot G$ where $x_1 = \mathsf{Dec}_{sk}(c)$, then $Q' = (a \cdot x_1 + b) \cdot G = \alpha \cdot G$, and so P can decrypt the Paillier encryption sent by V to get α and can compute $Q' = \alpha \cdot G$. However, if c is *not* an encryption of x_1 where $x_1 = \mathsf{Dec}_{sk}(c)$, then P will obtain $\alpha = a \cdot \tilde{x} + b$ for some $\tilde{x} \neq x_1$, since that will be the value encrypted in the Paillier encryption that V sends to P. In this case, V will only

accept if P can send the correct $Q' = a \cdot x_1 + b$. However, $f(x) = a \cdot x + b$ is an information-theoretic MAC, and thus P can only guess $a \cdot x_1 + b$ given $a \cdot \tilde{x} + b$ with negligible probability. (Note that the computation of f is over the integers, unlike standard information-theoretic MACs which are computed over a finite field. However, by taking large enough a, b, this is good enough. For this reason, we take $a \in \mathbb{Z}_q$ and $b \in \mathbb{Z}_{q^2}$.) Zero knowledge is achieved by having V first commit to a, b, enabling the simulator to extract these values before sending Q'. Clearly, given a, b, the simulator can generate the correct Q', even without knowing x_1 or the Paillier secret key.

PROTOCOL 6.1 (Zero-Knowledge Proof for the Language L_{PDL})

Inputs: The joint statement is $(c, pk, Q_1, \mathbb{G}, G, q)$, and the prover has a witness (x_1, sk) with $x_1 \in \mathbb{Z}_{q/3}$. (Recall that the proof is that $x_1 = \mathsf{Dec}_{sk}(c)$ and $Q_1 = x_1 \cdot G$ and $x_1 \in \mathbb{Z}_q$.)

The Protocol:

1. V chooses a random $a \leftarrow \mathbb{Z}_q$ and $b \leftarrow \mathbb{Z}_{q^2}$ and computes $c' = (a \odot c) \oplus b$ and $c'' = \mathsf{commit}(a, b)$. V sends (c', c'') to P. Meanwhile, V computes $Q' = a \cdot Q_1 + b \cdot G$.
2. P receives (c', c'') from V, decrypts it to obtain $\alpha = \mathsf{Dec}_{sk}(c')$, and computes $\hat{Q} = \alpha \cdot G$. P sends $\hat{c} = \mathsf{commit}(\hat{Q})$ to V.
3. V decommits c'', revealing (a, b).
4. P checks that $\alpha = a \cdot x_1 + b$ (over the integers). If not, it aborts. Else, it decommits $\hat{c}$ revealing $\hat{Q}$.
5. *Range-ZK proof:* In parallel to the above, P proves in zero knowledge that $x_1 \in \mathbb{Z}_q$, using the proof described in Appendix A.

V's output: V accepts if and only if it accepts the range proof and $\hat{Q} = Q'$.

Theorem 6.2. *Let $N > 2q^2+q$. Then, Protocol 6.1 is a zero-knowledge proof for the language L_{PDL} in the $\mathcal{F}_{\mathsf{com}}$-hybrid model, with completeness 1 for $x_1 \in \mathbb{Z}_{q/3}$ and with soundness error $2/q + 2^{-t}$.*

Proof. We prove completeness, soundness and zero knowledge. Completeness follows from the fact that when $N > 2q^2 + q$ there is no reduction modulo N in the Paillier computation and thus P obtains the correct value when decrypting c'. Furthermore, the range zero-knowledge proof has completeness 1 as long as $x_1 \in \mathbb{Z}_{q/3}$. We now proceed to the other properties.

Soundness. Let $x_1 = \mathsf{Dec}_{sk}(c)$. We consider two cases:

1. *Case 1 – $x_1 \notin \mathbb{Z}_q$:* The soundness of the range proof of Step 5 guarantees that V will reject in this case except with probability 2^{-t}.
2. *Case 2 – $x_1 \in \mathbb{Z}_q$ but $Q_1 \neq x_1 \cdot G$:* We claim that even an all-powerful cheating P^* cannot cause V to accept with probability greater than $2/q$, in

the $\mathcal{F}_{\mathsf{com}}$-hybrid model. In order to see this, observe that V accepts only if P^* commits to $\hat{Q} = a \cdot Q_1 + b \cdot G$ in Step 2.
P^* receives c' and can decrypt to obtain $\alpha = a \cdot x_1 + b$. Let $y \in \mathbb{Z}_q$ be such that $Q_1 = y \cdot G$; for this case, $y \neq x_1$. Then, V only accepts if P^* can compute $\beta = a \cdot y + b \bmod q$; to be more exact, P must have committed to $\hat{Q} = a \cdot Q_1 + b \cdot G = a \cdot (y \cdot G) + b \cdot G = [a \cdot y + b \bmod q] \cdot G$. (Note that although P commits to $\hat{Q}$, since it is all-powerful it can compute its discrete log. Thus, if $\hat{Q} = Q'$ then P can obtain $\beta = a \cdot y + b \bmod q$.)
Intuitively, P^* cannot succeed since V computes a type of information-theoretic MAC; it is not standard since the computation is over the integers. Formally, assume that P^* succeeds. This implies that it obtains $\alpha = a \cdot x_1 + b$ and $\beta = a \cdot y + b \bmod q$. Now, P can compute $a = \frac{\alpha - \beta}{x_1 - y} \bmod q$ in order to obtain a. Since $a \in \mathbb{Z}_q$, we have that this is the same value as a over the integers. Next, P^* can compute $b = \alpha - a \cdot x_1$. Thus, if P^* succeeds, then it obtains $(a, b) \in \mathbb{Z}_q \times \mathbb{Z}_{q^2}$.
Consider the following experiment, denoted Expt^1:

(a) P^* outputs x_1, y
(b) Values $a \leftarrow \mathbb{Z}_q$ and $b \leftarrow \mathbb{Z}_{q^2}$ are chosen uniformly, and $\alpha = a \cdot x_1 + b$ is computed.
(c) P^* is given α and outputs (a', b').
(d) P^* succeeds if and only if $a' = a$ and $b' = b$.

By what we have seen above, if P^* succeeds in computing β that causes V to accept, then P^* succeeds in this experiment. Thus, it suffices to show that P^* can succeed in this experiment with probability at most $2/q$. We denote $\mathsf{Expt1}_{P^*} = 1$ if P^* succeeds; using this notation, we wish to prove that

$$\Pr\left[\mathsf{Expt}^1_{P^*} = 1\right] \leq \frac{2}{q}.$$

We now modify the experiment to the following one, denoted Expt^2:

(a) P^* outputs x_1, y
(b) Values $a \leftarrow \mathbb{Z}_q$ and $b \leftarrow \mathbb{Z}_{q^2}$ are chosen uniformly, and a value $\alpha \leftarrow \mathbb{Z}_{2q^2}$ is chosen uniformly.
(c) P^* is given α and outputs (a', b').
(d) P^* succeeds if and only if $a' = a$ and $b' = b$.

Observe that in Expt^1, it holds that $\alpha \in \mathbb{Z}_{2q^2}$, because $a, x_1 \in \mathbb{Z}_q$ and $b \in \mathbb{Z}_{q^2}$ (we know that $x_1 \in \mathbb{Z}_q$ by this case). Thus, the values α in both experiments are from the same range.
Now, we claim that

$$\Pr\left[\mathsf{Expt}^2_{P^*} = 1\right] \geq \frac{1}{2q^2} \cdot \Pr\left[\mathsf{Expt}^1_{P^*} = 1\right].$$

This holds because with probability $1/2q^2$ the value α received by P^* in Expt^2 is such that $\alpha = a \cdot x_1 + b$. Noting now that

$$\Pr\left[\mathsf{Expt}^2_{P^*} = 1\right] \leq \frac{1}{q^3}$$

because P^* receives no information whatsoever on (a, b) in Expt^2. Combining the above, we have

$$\Pr\left[\mathsf{Expt}^1_{P^*} = 1\right] \leq 2q^2 \cdot \Pr\left[\mathsf{Expt}^2_{P^*} = 1\right] \leq \frac{2q^2}{q^3} = \frac{2}{q},$$

as required.

Zero knowledge. We construct a simulator $\mathcal{S}$ for a cheating verifier V^* in the $\mathcal{F}_{\mathsf{com}}$-hybrid model. $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ invokes V^* and obtains (c', c'').
2. $\mathcal{S}$ sends a simulated commitment value $\hat{c}$ to V^* (a receipt value from $\mathcal{F}_{\mathsf{com}}$).
3. $\mathcal{S}$ receives the decommitment (a, b) from V^*. $\mathcal{S}$ verifies that $c' = (a \odot c) \oplus b$. If no, then it aborts. If yes, then it sends a decommitment of $\hat{c}$ to $\hat{Q} = a \cdot Q_1 + b \cdot G$.
4. $\mathcal{S}$ simulates the range zero-knowledge proof.

V^*'s view is identical to a real protocol execution, in the $\mathcal{F}_{\mathsf{com}}$-hybrid model. This is because if $c' = (a \odot c) \oplus b$ then P would send the same $\hat{Q}$ as sent by $\mathcal{S}$. This is the only difference between a real execution and the simulated one.

Observe that we only need the commitment to be equivocal; extraction is actually not needed. ■

Acknowledgements

We would like to than Valery Osheter from Dyadic Security for the implementation of ECDSA protocol and for running the experiments, and Claudio Orlandi for pointing out some minor errors. We also thank Rosario Gennaro for pointing out that the oracle in Section 5.2 needs to be defined so that it halts the first time that it returns 0.

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. E. Brickell, D. Chaum, I. Damgård,J. Van de Graaf. Gradual and Verifiable Release of a Secret. In *CRYPTO87*, Springer (LNCS 293), pages 156-166, 1988.
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. In *EUROCRYPT 1998*, Springer (LNCS 1403), pages 561–575, 1998.
8. R.A. Croft and S.P. Harris. Public-Key Cryptography and Reusable Shared Secrets. In *Cryptography and Coding*, pages 189–201, 1989.
9. I. Damgård and M. Jurik. A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System. In *Public Key Cryptography 2001*, Springer (LNCS 1992), pages 119–136, 2001.
10. Y. Desmedt. Society and Group Oriented Cryptography: A New Concept. In *CRYPTO'87*, Springer (LNCS 293), pages 120–127, 1988.
11. Y. Desmedt and Y. Frankel. Threshold Cryptosystems. In *CRYPTO'89*, Springer (LNCS 435), pages 307–315, 1990.
12. 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.
13. E. Fujisaki. Improving Practical UC-Secure Commitments Based on the DDH Assumption. In *SCN 2016*, Springer (LNCS 9841), pages 257–272, 2016.
14. R. Gennaro, S. Jarecki, H. Krawczyk and T. Rabin. Robust Threshold DSS Signatures. In *EUROCRYPT96*, Springer (LNCS 1070), pages 354-371, 1996.
15. 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.
16. S. Goldfeder. Personal communication, December 2016.
17. O. Goldreich. *Foundations of Cryptography: Volume 2 – Basic Applications*. Cambridge University Press, 2004.
18. C. Hazay and Y. Lindell. *Efficient Secure Two-Party Protocols: Techniques and Constructions*. Springer, November 2010.
19. C. Hazay, G.L. Mikkelsen, T. Rabin and T. Toft. Efficient RSA Key Generation and Threshold Paillier in the Two-Party Setting. In *CT-RSA 2012*, Springer (LNCS 7178), pages 313–331, 2012. See `http://eprint.iacr.org/2011/494` for the full version.
20. Y. Lindell: Highly-Efficient Universally-Composable Commitments Based on the DDH Assumption. In *EUROCRYPT 2011*, Springer (LNCS 6632), pages 446–466, 2011.
21. H. Lipmaa. On Diophantine Complexity and Statistical Zero-Knowledge Arguments. In *ASIACRYPT 2003*, Springer (LNCS 2894), pages 398–415, 2003.
22. 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*.
23. P. Paillier. Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT99*, Springer (LNCS 1592), pages 223–238, 1999.
24. C.P. Schnorr. Efficient Identification and Signatures for Smart Cards. In *CRYPTO 1989*, Springer (LNCS 435), pages 239–252, 1990.
25. V. Shoup. Practical Threshold Signatures. In *EUROCRYPT 2000*, Springer (LNCS 1807), pages 207–220, 2000.
26. V. Shoup and R. Gennaro. Securing Threshold Cryptosystems against Chosen Ciphertext Attack. In *EUROCRYPT 1998*, Springer (LNCS 1403), pages 1–16, 1998.
27. Porticor, `www.porticor.com`.
28. Dyadic Security, `www.dyadicsec.com`.
29. Sepior, `www.sepior.com`.

A Zero-Knowledge Range Proof

For the sake of completenesss, in this appendix we present the ZK-proof that $x \in \mathbb{Z}_{q/3}$ where $c = \mathsf{Enc}_{pk}(x)$. The value sid is a unique session identifier obtained from the application. Our proof is based on the proof described [2, Section 1.2.2], with adaptations as required for our setting here. We will prove for $x \in \{0, \ldots, \lfloor \frac{q}{3} \rfloor\}$ that it is in the range $[0, q)$. Let $\ell = \lfloor \frac{q}{3} \rfloor$. Stated differently, the input is $x \in \{0, \ldots, \ell\}$ and the proof guarantees that $x \in \mathbb{Z}_q$.

- **Input:** The prover P has input (c, x, r) where $c = \mathsf{Enc}_{pk}(x; r)$ and the Paillier key-pair $(N, \phi(N))$; the verifier V has input c and the Paillier public key N. Both parties have q and $\ell = \lfloor \frac{q}{3} \rfloor$.
  Both parties have a parameter $t = 40$.
- **The protocol:**
  1. ***V*'s first message:** V chooses a random $e \leftarrow \{0,1\}^t$, computes $com = \mathsf{commit}(e, sid)$ and sends com to P. Denote $e = e_1, \ldots, e_t$.
  2. ***P*'s first message:**
     (a) P chooses random $w_1^1, \ldots, w_1^t \leftarrow \{\ell, \ldots, 2\ell\}$ and computes $w_2^i = w_1^i - \ell$ for every $i = 1, \ldots, t$.
     (b) For every $i = 1, \ldots, t$, P switches the values of w_1^i and w_2^i with probability $1/2$ (independently for each i).
     (c) For every $i = 1, \ldots, t$, P computes $c_1^i = \mathsf{Enc}_{pk}(w_1^i; r_1^i)$ and $c_2^i = \mathsf{Enc}_{pk}(w_2^i; r_2^i)$, where $r_i^1, r_i^2 \leftarrow \mathbb{Z}_N$ are the randomness used in Paillier encryption.
     (d) P sends $c_1^1, c_2^1 \ldots, c_1^t, c_2^t$ to V.
  3. ***V*'s second message:** Upon receiving $c_1^1, c_2^1 \ldots, c_1^t, c_2^t$, V decommits to com, revealing (e, sid) to P.
  4. ***P*'s second message:** For $i = 1, \ldots, t$:
     (a) If $e_i = 0$ then P sets $z_i = (w_1^i, r_1^i, w_2^i, r_2^i)$.
     (b) If $e_i = 1$ then P sets z_i as follows. Let $j \in \{1, 2\}$ be the unique value of j such that $x + w_j^i \in \{\ell, \ldots, 2\ell\}$. Then, S_1 sets $z_i = (j, x + w_j^i, r \cdot r_j^i \bmod N)$.
     (c) P sends $z_1, \ldots, z_t$ to V.
- ***V*'s output:** V parses z_i appropriately according to the value of e_i. Then: For $i = 1, \ldots, t$:
  1. If $e_i = 0$ then V checks that $c_1^i = \mathsf{Enc}_{pk}(w_1^i; r_1^i)$ and $c_2^i = \mathsf{Enc}_{pk}(w_2^i; r_2^i)$ and that one of $\hat{w}_1^i, \hat{w}_2^i \in \{\ell, \ldots, 2\ell\}$ while the other is in $\{0, \ldots, \ell\}$, where $z_i = (w_1^i, r_1^i, w_2^i, r_2^i)$.
  2. If $e_i = 1$ then V checks that $c \oplus c_j^i = \mathsf{Enc}_{pk}(w_i; r_i)$ and $w_i \in \{\ell, \ldots, 2\ell\}$, where $z_i = (j, w_i, r_i)$.

  V outputs 1 if and only if all of the checks pass.

Security. We sketch the proof here:

- **Completeness:** As long as there exists a $j \in \{1, 2\}$ such that $x + w_j^i \in \{\ell, \ldots, 2\ell\}$, for every i, it is clear that V will accept. In order to see why this

holds, observe that by the way w_i^1 and w_i^2 are chosen we have $w_i^1 \in \{\ell, \ldots, 2\ell\}$ and $w_i^2 \in \{0, \ldots, \ell\}$.

There are two cases. If $x + w_i^1 < 2\ell$ then since $x + w_i^1 \geq w_i^1 \geq \ell$ we have $x + w_i^1 \in \{\ell, \ldots, 2\ell\}$. In contrast, if $x + w_i^1 \geq 2\ell$, then $w_i^2 = w_i^1 - \ell \geq \ell$. Since $x + w_i^2 \leq 2\ell$ (since both $0 \leq x \leq \ell$ and $0 \leq w_i^2 \leq \ell$), it follows that $x + w_i^2 \in \{\ell, \ldots, 2\ell\}$, as required.

– **Soundness:** Let $c = \mathsf{Enc}_{pk}(x)$ and assume that $x \notin \mathbb{Z}_q$ and so in particular $x \geq q$ (note that if x is negative then modulo q this is the same as $x \geq q$). We need to prove that V accepts with probability at most 2^{-t}. Let P^* be the cheating prover. We show that if P^* can provide an accepting answer for both $e_i = 0$ and $e_i = 1$ for the ith ciphertext, then $x \in \mathbb{Z}_q$. This suffices since it implies that P^* can answer at most one of the e_i queries for each i, and thus the probability that it answers all is at most 2^{-t}.

  Fix i and assume that P^* can provide an accepting answer for both $e_i = 0$ and $e_i = 1$. Since P^* can answer for $e_i = 0$, this implies that $c_1^i = \mathsf{Enc}pk(w_1^i)$ and $c_2^i = \mathsf{Enc}_{pk}(w_2^i)$ and $w_1^i \in \{\ell, \ldots, 2\ell\}$ and $w_2^i \in \{0, \ldots, \ell\}$. Furthermore, since P^* can answer for $e_i = 1$ this implies that for some $j \in \{1, 2\}$ we have that $c \oplus c_j^i = \mathsf{Enc}_{pk}(w_i)$ for some $w_i \in \{\ell, \ldots, 2\ell\}$. Note that by the perfect decryption correctness of Paillier (under the assumption that the Paillier key is valid, which is proven during key generation), it holds that $w_i \in \{w_1^i, w_2^i\}$.

  We consider two cases:

  1. *Case 1 – $j = 1$:* In this case, we have that $x + w_1^i = w_i$ where $w_1^i \in \{\ell, \ldots, 2\ell\}$ and $w_i \in \{\ell, \ldots, 2\ell\}$. Since the minimal value of w_1^i is ℓ and the maximal value of w_i is 2ℓ, it follows that $x \leq \ell$.
  2. *Case 2 – $j = 2$:* In this case, we have that $x + w_2^i = w_i$ where $w_2^i \in \{0, \ldots, \ell\}$ and $w_i \in \{\ell, \ldots, 2\ell\}$. Since the minimal value of w_2^i is 0 and the maximal value of w_i is 2ℓ, it follows that $x \leq 2\ell$.

– **Zero knowledge:** The simulator $\mathcal{S}$ extracts e from the commitment provided by the potentially cheating verifier V^*. Then, for every i, if $e_i = 0$ then $\mathcal{S}$ generates c_1^i, c_2^i like the honest prover does. In contrast, if $e_i = 1$, then $\mathcal{S}$ chooses a random $j \in \{1, 2\}$, a random $w_i \in \{\ell, \ldots, 2\ell\}$ and a random $r_i \in \mathbb{Z}_N$. Then, $\mathcal{S}$ sets $c_j^i = \mathsf{Enc}_{pk}(w_i; r_i) \ominus c$ and sets c_{3-j}^i to be an encryption of 0. Finally, $\mathcal{S}$ hands V^* all of the encryptions, receives back the decommitment, and provides the answers appropriately.

  We argue that the view generated by $\mathcal{S}$ is computationally indistinguishable from the view of V^* in a real proof. In order to see this, first observe that the ciphertexts are given in random order in a real proof. Next, observe that for every i for which $e_i = 1$, the ciphertext opened is an encryption of a value that is uniformly distributed in $\{\ell, \ldots, 2\ell\}$. This holds because w_1^i is uniformly distributed in $\{\ell, \ldots, 2\ell\}$ and w_2^i is uniformly distributed in $\{0, \ldots, \ell\}$. This implies that $x + w_1^i$ is uniformly distributed in $\{x + \ell, \ldots, x + 2\ell\}$ and $x + w_2^i$ is uniformly distributed in $\{x, \ldots, x + \ell\}$. Thus, the distribution over the subset of values between ℓ and 2ℓ is uniform.