Efficient Secure Two Party ECDSA

Sermin Kocaman[1[0000−0001−8334−8587]] ⋆ and Younes Talibi Alaoui[2[0000−0002−7947−9450]] ⋆⋆

[1] Department of Cryptology, Institute of Applied Mathematics, METU, Ankara, Turkey
[2] Fabric Cryptography
sermin.cakin@gmail.com, talibialaouiyounes@gmail.com

Abstract. Distributing the Elliptic Curve Digital Signature Algorithm (ECDSA) has received increased attention in past years due to the wide range of applications that can benefit from this, particularly after the popularity that the blockchain technology has gained. Many schemes have been proposed in the literature to improve the efficiency of multi-party ECDSA. Most of these schemes either require heavy homomorphic encryption computation or multiple executions of a functionality that transforms Multiplicative shares to Additive shares (MtA). Xue et al. (CCS 2021) proposed a 2-party ECDSA protocol secure against malicious adversaries and only requires one execution of MtA, with an online phase that consists of only one party sending one field element to the other party with a computational overhead dominated by the verification step of the signature scheme. We propose a novel protocol, based on the assumption that the Computational Diffie-Hellman problem is hard, that offers the same online phase performance as the protocol of Xue et al., but improves the offline phase by reducing the computational cost by one elliptic curve multiplication and the communication cost by two field elements. To the best of our knowledge, our protocol offers the most efficient offline phase for a two-party ECDSA protocol with such an efficient online phase.

Keywords: ECDSA · Two-party Protocols · Threshold Signatures.

1 Introduction

Multi-party computation (MPC) is a technique from cryptography that enables multiple parties to conduct computation on their secrets while preserving them private. MPC was formally introduced with Yao's 2-party protocol for the Millionaires' problem [26]. Today, it became a pioneering solution for a wide variety of real-world problems, such as cryptographic key protection, privacy-preserving data analytics, and so forth [15].

With the rise of the blockchain technology and cryptocurrencies, multi-party signing [7] and, in particular, threshold signing has gained significant attention

⋆ Corresponding author
⋆⋆ Most of the work done while at imec-COSIC, KU Leuven, Belgium

in the past decade. Namely, a (t, n) signature scheme enables n parties to distribute the signing power in such a way that signing a message m requires the collaboration of at least $t+1$ of them. This is accomplished by having the n parties participate in the key generation phase to produce a private key unknown to them. At the end of this phase, each party will hold a share of the private key, together with the public key. Then the signing phase is executed as an interactive protocol as well, where at least $t+1$ parties participate with their shares so as to produce the signature, which is then checked with the verification algorithm of the signature scheme being distributed. This benefits cryptocurrencies as transactions are sent by producing a signature using the sender's private key. Thus to prevent a single point of failure while maintaining the key, one can share it among different parties placed in different locations, who need to collaborate to sign.

In this regard, thresholdizing the ECDSA algorithm has drawn most of the attention, as it is the signing algorithm used in Bitcoin. We can find in the literature many works that addressed this [14,12,8,25,13,2,9,5,23,6] where various schemes were constructed, either addressing the 2-party case [14,8,25,13], or more generally, the n-party case [12,2,9,5,23,6]; using generic MPC protocols [5,23], or special purpose protocols targeting ECDSA [14,12,8,25,13,2,9,6]. Those schemes differ particularly in the way of sharing values, namely additively or multiplicatively. That is, at the heart of the ECDSA algorithm, one needs to calculate $s = k^{-1}(H(m) + x \cdot r) \mod q$. In a threshold version of ECDSA, both the private key x and the random nonce k used for signing the message m are secretly shared among parties. In fact, to provide a threshold version of ECDSA, the main challenge consists of choosing an adequate way to secretly share k and x so that s can be computed efficiently. Note that this calculation contains inverting a secret, and multiplying it with another value obtained by evaluating linear operations over another secret (addition and multiplication with opened values).

For instance, for the 2-party case, additively secret sharing k is problematic for inversion, as in this case, party $\mathcal{P}_1$ holds k_1 and party $\mathcal{P}_2$ holds k_2 subject to $k_1 + k_2 = k \mod q$, and from this, the two parties need to calculate k^{-1}. Alternatively, one can secretly share k in a multiplicative way to overcome this obstacle, as in this case, inverting becomes a local operation; however, the resulting value still needs to be multiplied by $H(m) + x \cdot r$, which still introduces obstacles either x was additively or multiplicatively secret shared.

As a solution to these challenges, several authors in the field proposed using homomorphic encryption. This approach allows one party to transmit a secret to the other party in encrypted form so that they can execute the challenging computation and decrypt it afterward. The homomorphic encryption schemes that were used are partially homomorphic, as performing one type of operation over the ciphertexts was sufficient for the computation needed.

For the most part, homomorphic encryption was introduced to realize a Multiplicative-to-Additive (MtA) functionality which enables parties to obtain an additive version of the shares of a secret from a multiplicative one, adopt-

ing ideas from [17]. Based on this functionality, parties who hold multiplicative shares α and β, respectively, can get the corresponding additive shares a and b where $a + b = \alpha.\beta$ by querying this functionality. The need for querying this protocol arises when an additive sharing is preferable than a multiplicative one from a performance point of view. Of course, this functionality does not come for free, and it introduces a cost to the protocol whenever it is called; however, there exist many instantiations of it, such as Paillier encryption [20]-based MtA [12], El Gamal encryption[10]-based MtA [16], and Castagnos-Laguillaumie (CL) encryption[4]-based MtA [3]. Besides, one can also construct Oblivious Transfer (OT) [21]-based MtA [8], which has the advantage of decreasing the computational complexity by eliminating the need for homomorphic encryption at the expense of incurring a relatively high bandwidth. As a result, one has multiple options for MtA instantiations, each of which offers a different tradeoff between the computation and communication costs, thanks to which one can select the one that best fits the constraints faced. Also, it should be noted that Fireblocks' teams showed a Paillier key vulnerability in [12] leaking secret key or inverse nonce information [18]. These attacks occur when the MtA functionality is used without range proofs that ensure the inputs of MtA are chosen from the required domain. Thus, it is recommended to use suitable range proofs to detect maliciously formed input in Paillier-based MtA functionality.

For the 2-party case of threshold ECDSA, two works are most related to ours, namely, the one of Lindell [14] and Xue et al. [25]. Lindell has proposed a simple and efficient 2-party protocol against malicious adversaries. To briefly go over this protocol, both x and k are secretly shared in a multiplicative way, where each party $\mathcal{P}_i$ generates x_i in the key generation phase so that the private key x is equal to $x = x_1 \cdot x_2$. Party $\mathcal{P}_1$ also encrypts x_1 so as to send it to $\mathcal{P}_2$, then in the signing phase, the two parties generate their share of the nonce k, then $\mathcal{P}_2$ computes its share of s and sends it to $\mathcal{P}_1$, which involves encrypting and performing homomorphic encryption operations. Finally, $\mathcal{P}_1$ calculates the signature s, which involves decryption before the verification step.

On the other hand, Xue et al. proposed an online-friendly algorithm against malicious adversaries. That is, this protocol has a nearly optimal online phase, in the sense that the heaviest part of it consists of the verification step of the signature, which in turn consists of calculating two scalar multiplications M of elliptic curve points (scalar multiplications will be denoted as M from now on). The communication cost is also efficient, as only a single field element needs to be sent. This is opposed to [14] as one needs to send and operate over ciphertexts during the online phase. However, providing such an efficient online phase came with the cost of offloading all the heavy computation in the offline phase of the signing step. That is, while the key generation does not involve any encryption, an MtA is being executed for every signature during the signing phase, which is still a good compromise as it reduces the number of calls to the MtA functionality compared to other schemes. Thus the resulting protocol offers an efficient online phase with a good overall cost. However, this scheme can be further optimized, as we will see in the next section.

1.1 Our Contribution

We present a protocol against malicious adversaries with a nearly optimal online phase as in [25], but with reduced computation and communication costs for the offline phase. That is, our key generation is the same as in [25], where we produce additive secret sharings of x ($\mathcal{P}_i$ generates $Q_i \leftarrow [x_i] \cdot P$, where P is a generator of the curve, and the public key is $Q \leftarrow Q_1 + Q_2$), and our online phase requires two scalar multiplication M as in [25]. However, our offline phase reduces the number of EC multiplications by one and the size of data communicated by two field elements.

The cost reduction is achieved by eliminating the additional step of re-sharing the secret x in [25], and basing the security of our protocol on the 1-Weak Diffie-Hellman problem, which is equivalent to the Computational Diffie-Hellman problem. That is, at the heart of the signing phase of the protocol of [25], x was re-shared between the two parties (following obvious notation) as $x = x'_1.(k_2 + r_1) + x'_2$, where the nonce k is shared as $k = k_1(r_1 + k_2)$, then the shares x'_1 and k_2 are the values forwarded to the MtA functionality.

Instead, we simplified the protocol by adopting a multiplicative sharing of k where it is unnecessary to perform a re-sharing step ($\mathcal{P}_i$ generates $R_i \leftarrow [k_i] \cdot P$ for P the generator of the curve, and the point R from which we take the x-coordinate r is $R \leftarrow [k_1 \cdot k_2] \cdot P$). We query the MtA only once on the most convenient inputs for our choices. Namely, querying the MtA on x_1 as the input of $\mathcal{P}_1$, and k_2^{-1} as the input of $\mathcal{P}_2$. This was a logical choice as holding an additive sharing as

$$x_1 \cdot k_2^{-1} = a + b \mod q$$

by the players allows them to do the online phase in only one pass, as the signature s can be written as

$$s = k_1^{-1} \cdot (k_2^{-1} \cdot (H(m) + x_2 \cdot r) + x_1 \cdot k_2^{-1} \cdot r) \mod q$$

In this case, $\mathcal{P}_2$ computes locally its signature share as

$$s_2 \leftarrow k_2^{-1} \cdot (H(m) + x_2 \cdot r) + b \cdot r \mod q$$

and sends it to $\mathcal{P}_1$ to construct the signature

$$s \leftarrow k_1^{-1} \cdot (s_2 + a \cdot r) \mod q$$

However, it is crucial to note that the protocol requires $\mathcal{P}_1$ to input x_1 for MtA. If there are no checks on this input to MtA, a malicious $\mathcal{P}_1$ can corrupt the system since $\mathcal{P}_1$ takes the partial signature s_2 and then generates the full signature s. For example, a malicious $\mathcal{P}_1$ can forge a signature on a different message m' of his choice by crafting the value to be sent to MtA as $x'_1 \leftarrow -r^{-1} \cdot (H(m') - H(m) + x_1 \cdot r)$, then $\mathcal{P}_1$ will compute the full signature s as $k_1^{-1} \cdot (s_2 + a \cdot r) = k_1^{-1} \cdot (k_2^{-1} \cdot (H(m) + x_2.r) + (a+b) \cdot r) = k^{-1} \cdot (H(m') + x \cdot r)$ which is a valid signature on m' that is chosen by $\mathcal{P}_1$.

In order to prevent $\mathcal{P}_1$ from mounting such an attack and manipulating the distribution of s_2, we add a check operation on the correctness of the MtA input of $\mathcal{P}_1$. Namely after calling MtA and receiving its outputs, $\mathcal{P}_1$ computes $[a] \cdot P$ and sends it to $\mathcal{P}_2$, who computes $k_2 \cdot ([a] \cdot P + [b] \cdot P)$ and checks whether it is equal to Q_1 or not. The correctness of this equality ensures that $\mathcal{P}_1$ used the correct x_1 value as MtA input, and as we will see, $\mathcal{P}_1$ will not be able to bypass it, unless he breaks the standard assumption that the Computational Diffie-Hellman problem is hard. It is worth noting that the check we add is not concerned with the security of the underlying MtA, but rather to ensure that the parties involved are invoking the MtA with the appropriate inputs. This of course adds a round of communication to the protocol, however, it is a critical step in ensuring the protocol's security, which is analogous to the consistency check executed immediately following the MtA call in [25].

In sum, the protocol we end up with utilized an additive sharing of x and a multiplicative sharing of k, which is a similar setting of [8] for the (1, n)-ECDSA case (i.e., any two parties among the n parties can construct a valid signature). However, we only call the MtA functionality once while it is being called three times in [8]. Besides, we only perform 13M, while 16M are needed for [8].

This improvement has an impact depending on the instantiation of MtA. For instance, in the case of an OT-based MTA, where such a choice is usually made to have a low computation cost, reducing the number of EC multiplications by one will decrease the computation cost of the offline phase of [25] (Table 4 of [25]) by 5.4 percent. On the other hand, in the case of a CL-based MtA, which introduces a low communication cost, reducing the size of transmitted data by two field elements decreases the communication cost of the offline phase of [25] (Table 5 of [25]) for the case of the secp256k1 curve by 3.7 percent. While these percentages may seem modest, the potential gains are substantial, given the vast scale at which ECDSA signatures are executed, and all the applications that can benefit from a distributed version of it.

We also implemented our protocol and obtained an online phase of 0.1 ms, which is half the time required for the online phase of [25]; however, given the similarity of the online phase between the two protocols, this difference in time is most likely due to our implementation's use of the highly optimized C library secp256k1 for the operations over the curve.

1.2 Paper Organization

This paper is organized as follows: Section 2 provides the necessary background over the hardness assumption upon which we are basing the security of our protocol, the ECDSA scheme, and the ideal functionalities we used. Section 3 presents the proposed protocol, along with the cost analysis, comparison with related work, and its running time based on our implementation. Section 4 concludes the paper. Then in the Appendix, security proofs are given.

2 Preliminaries

2.1 Hardness Assumptions

The security of our protocol is based on the 1-Weak Diffie-Hellman problem [19], also referred to as the Inverse Diffie-Hellman problem [1]. That is, this problem is a special case of the k-Weak Diffie-Hellman problem (and can be proven to be equivalent to it), where the adversary is given a set of points $\{P,\ [x]{\cdot}P,\ [x^2]{\cdot}P,\ \ldots,\ [x^k]{\cdot}P\}$ for a randomly chosen x, and asked to find $[x^{-1}]{\cdot}P$.

Definition 1. *(Computational Diffie-Hellman Assumption.) Let $\mathbb{G}$ be a cyclic group of a large prime order, and P a generator of $\mathbb{G}$. Given a tuple $(P,\ [a]{\cdot}P,\ [b]{\cdot}P)$ for a randomly chosen a and b, it is computationally hard to compute $[a{\cdot}b]{\cdot}P$.*

Definition 2. *(1-Weak Diffie-Hellman Assumption.) Let $\mathbb{G}$ be a cyclic group of a large prime order, and P a generator of $\mathbb{G}$. Given a tuple $(P,\ [x]{\cdot}P)$ for a randomly chosen x, it is computationally hard to compute $[x^{-1}]{\cdot}P$.*

Theorem 1. *The 1-Weak Diffie-Hellman and the Computational Diffie-Hellman assumptions are equivalent.*

The proof of theorem 1 is given Appendix A.

2.2 The ECDSA Scheme

The ECDSA scheme is a signature algorithm that involves key generation, signing, and verification. Let $\mathbb{G}$ be an elliptic curve group of order q of size λ bits, with a generator P, and the neutral element being denoted as $\mathcal{O}$. The ECDSA scheme works as follows:

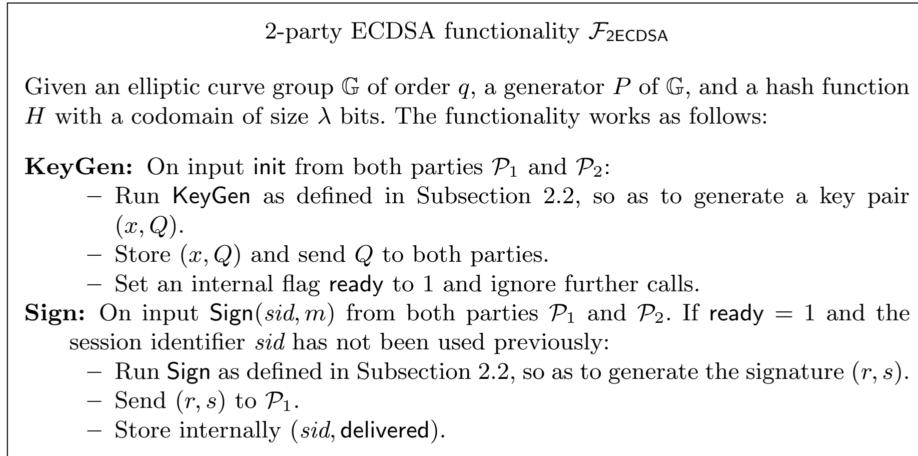
- $\mathsf{KeyGen}(1^\lambda) \rightarrow (x, Q)$: set a random private key $x \leftarrow Z_q$ and compute the corresponding public key $Q = [x] \cdot P$.
- $\mathsf{Sign}(x, m) \rightarrow (r, s)$: generate the signature (r, s) using private key x, message m, and hash function H with codomain of size λ bits. That is:
  - Set a random nonce $k \leftarrow Z_q^*$ and compute $R \leftarrow [k] \cdot P = (r_x, r_y)$, then set $r \leftarrow r_x \mod q$.
  - Compute $s \leftarrow k^{-1} \cdot (H(m) + r \cdot x) \mod q$ and output (r, s).
- $\mathsf{Verify}(m; (r, s)) \rightarrow b \in \{0, 1\}$: equals 1 if the signature is valid; 0 otherwise. That is:
  - Compute $R \leftarrow s^{-1} \cdot ([H(m)] \cdot P + [r] \cdot Q) = (r_x, r_y)$.
  - If $r = r_x \mod q$, output 1; otherwise output 0.

Due to the structure of elliptic curves, if (r, s) is a valid signature, then its complement $(r,\ -s)$ is also a valid signature. Thus, this gives rise to the malleability problem of the ECDSA scheme. To overcome this problem, one can follow the low-s rule, where the low-s is the value between 0 and $\frac{q-1}{2}$. Therefore, we assume that the output of the signing procedure is always the lower s value.

2.3 Ideal Functionalities

We describe below the ideal $\mathcal{F}_{\mathsf{2ECDSA}}$ functionality that our protocol realizes, as well as the ideal functionalities queried by our protocol, namely, an ideal zero-knowledge proof (ZKP) functionality $\mathcal{F}_{\mathsf{ZKP}}$ and an ideal committed non-interactive zero-knowledge functionality $\mathcal{F}_{\mathsf{Commit\text{-}ZK}}$ which are similar to the ones used in [14], as well as an ideal Multiplicative-to-Additive (MtA) functionality $\mathcal{F}_{\mathsf{MtA}}$. In this content, we assume that each functionality provides a fresh session identifier (sid) for each invocation of it. This can be achieved by having the parties exchange random strings between each other, which will be further concatenated then hashed so as to produce the session identifiers.

$\mathcal{F}_{\mathsf{2ECDSA}}$ Functionality. The $\mathcal{F}_{\mathsf{2ECDSA}}$ functionality is composed of a key generation phase and a signing phase. In the key generation phase, the key pair (x, Q) is generated, where x is stored internally, and Q is given to the parties. In the signing phase, the signature on the given message is constructed and given to $\mathcal{P}_1$. The functionality is introduced in Figure 1.



2-party ECDSA functionality $\mathcal{F}_{\mathsf{2ECDSA}}$

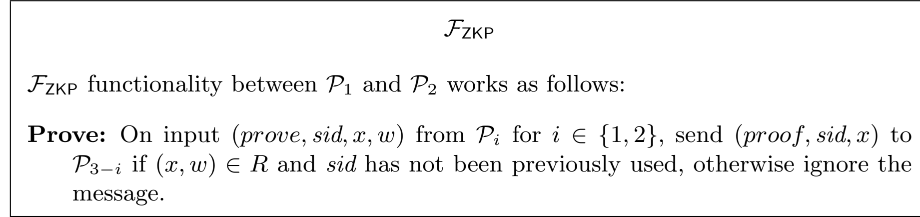
Given an elliptic curve group $\mathbb{G}$ of order q, a generator P of $\mathbb{G}$, and a hash function H with a codomain of size λ bits. The functionality works as follows:

KeyGen: On input init from both parties $\mathcal{P}_1$ and $\mathcal{P}_2$:
- Run KeyGen as defined in Subsection 2.2, so as to generate a key pair (x, Q).
- Store (x, Q) and send Q to both parties.
- Set an internal flag ready to 1 and ignore further calls.

Sign: On input $\mathsf{Sign}(sid, m)$ from both parties $\mathcal{P}_1$ and $\mathcal{P}_2$. If $\mathsf{ready} = 1$ and the session identifier sid has not been used previously:
- Run Sign as defined in Subsection 2.2, so as to generate the signature (r, s).
- Send (r, s) to $\mathcal{P}_1$.
- Store internally $(sid, \mathsf{delivered})$.

Fig. 1. 2-party ECDSA functionality $\mathcal{F}_{\mathsf{2ECDSA}}$

$\mathcal{F}_{\mathsf{ZKP}}$ Functionality. The $\mathcal{F}_{\mathsf{ZKP}}$ functionality is depicted in Figure 2. With this functionality, one party can prove the knowledge of a witness w for an element y, such that the pair (y, w) satisfies the relation $\mathcal{R}$. For our protocol, this relation is $\mathcal{R} \leftarrow \{(Q, x) \in \mathbb{G} \times Z_q | Q = [x] \cdot P\}$ for public parameters $\mathbb{G}$ and its generator P, which allows to prove knowledge of the discrete log of an elliptic curve point. The sigma protocol of Schnorr [22] can be used to instantiate this functionality, which can be made non-interactive using the Fiat-Shamir paradigm in the random-oracle model [11].



$\mathcal{F}_{\mathsf{ZKP}}$

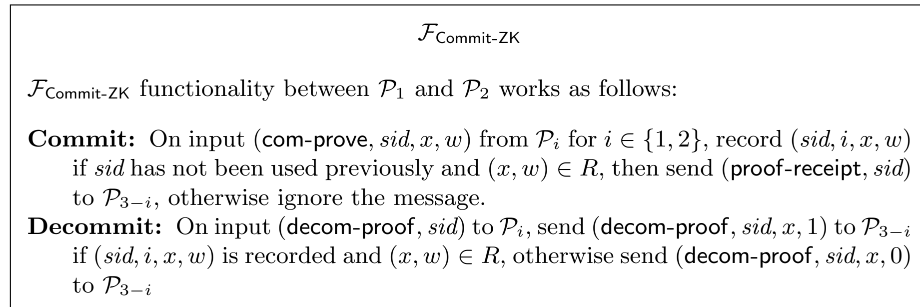
$\mathcal{F}_{\mathsf{ZKP}}$ functionality between $\mathcal{P}_1$ and $\mathcal{P}_2$ works as follows:

Prove: On input $(prove, sid, x, w)$ from $\mathcal{P}_i$ for $i \in \{1, 2\}$, send $(proof, sid, x)$ to $\mathcal{P}_{3-i}$ if $(x, w) \in R$ and sid has not been previously used, otherwise ignore the message.

Fig. 2. $\mathcal{F}_{\mathsf{ZKP}}$

ZKPs are expected to satisfy three key properties, completeness, soundness, and zero knowledge. Completeness means that given a witness w for a statement $x \in \mathcal{R}$, there is an efficient algorithm that provides a convincing $proof$, i.e. it ensures that if two parties follow the protocol, the verifier accepts the proof. Soundness means a malicious prover cannot construct a convincing $proof$ for $x \notin \mathcal{R}$, i.e. soundness prevents the verifier from accepting a false proof of the statement. Also, zero-knowledge means that $proof$ does not reveal the used witness w, i.e. it states that $proof$ does not leak any information except for the truth of the statement.

$\mathcal{F}_{\mathsf{Commit\text{-}ZK}}$ Functionality. The $\mathcal{F}_{\mathsf{Commit\text{-}ZK}}$ functionality is depicted in Figure 3. Through this functionality, a party will be able to commit to its Non-interactive ZKP (NIZKP) and open it afterwards. As mentioned in [14], this functionality can be realized in the random oracle model by having the parties hash their NIZKP concatenated with a randomness r, which will be both opened in the decommitment phase.



$\mathcal{F}_{\mathsf{Commit\text{-}ZK}}$

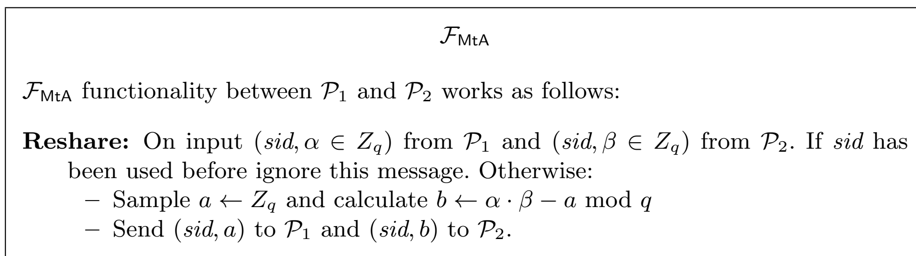
$\mathcal{F}_{\mathsf{Commit\text{-}ZK}}$ functionality between $\mathcal{P}_1$ and $\mathcal{P}_2$ works as follows:

Commit: On input $(\mathsf{com\text{-}prove}, sid, x, w)$ from $\mathcal{P}_i$ for $i \in \{1, 2\}$, record (sid, i, x, w) if sid has not been used previously and $(x, w) \in R$, then send $(\mathsf{proof\text{-}receipt}, sid)$ to $\mathcal{P}_{3-i}$, otherwise ignore the message.

Decommit: On input $(\mathsf{decom\text{-}proof}, sid)$ to $\mathcal{P}_i$, send $(\mathsf{decom\text{-}proof}, sid, x, 1)$ to $\mathcal{P}_{3-i}$ if (sid, i, x, w) is recorded and $(x, w) \in R$, otherwise send $(\mathsf{decom\text{-}proof}, sid, x, 0)$ to $\mathcal{P}_{3-i}$

Fig. 3. $\mathcal{F}_{\mathsf{Commit\text{-}ZK}}$

$\mathcal{F}_{\mathsf{MtA}}$ Functionality. The $\mathcal{F}_{\mathsf{MtA}}$ functionality is depicted in Figure 4. This functionality takes as an input the two values α and β coming from $\mathcal{P}_1$ and $\mathcal{P}_2$ respectively, and forwards to them respectively two random values a and b, subject to the relation $a + b = \alpha \cdot \beta \mod q$, i.e., it transforms a multiplicative sharing of a secret to an additive sharing. As stated earlier, one can instantiate MtA from many constructions, such as the Paillier encryption scheme [20] or El Gamal [10], Castagnos-Laguillaumie (CL) [4] or OT [21].



$\mathcal{F}_{\mathsf{MtA}}$

$\mathcal{F}_{\mathsf{MtA}}$ functionality between $\mathcal{P}_1$ and $\mathcal{P}_2$ works as follows:

Reshare: On input $(sid, \alpha \in Z_q)$ from $\mathcal{P}_1$ and $(sid, \beta \in Z_q)$ from $\mathcal{P}_2$. If sid has been used before ignore this message. Otherwise:
- Sample $a \leftarrow Z_q$ and calculate $b \leftarrow \alpha \cdot \beta - a \bmod q$
- Send (sid, a) to $\mathcal{P}_1$ and (sid, b) to $\mathcal{P}_2$.

Fig. 4. $\mathcal{F}_{\mathsf{MtA}}$

3 Protocol

Our two party ECDSA protocol is composed of two phases; one phase for a distributed key generation that runs once, at the end of which the parties will hold an additive sharing of the secret x as $x = x_1 + x_2$, then the second phase is for signing, which consists of:

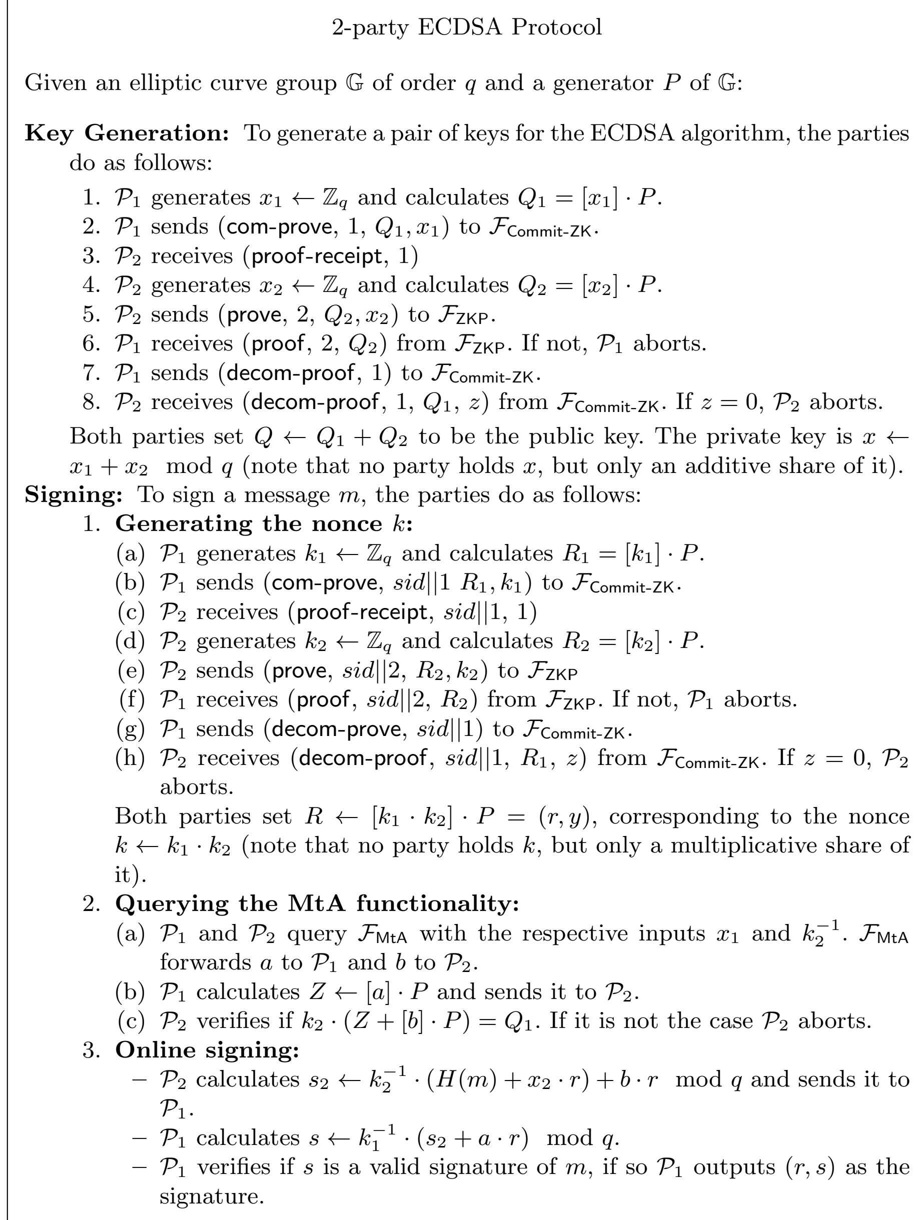
- Generating the nonce k, which will be multiplicatively shared between the parties as $k = k_1 \cdot k_2$.
- Querying the MtA functionality, so as to convert the product of $\mathcal{P}_1$'s secret key x_1 and $\mathcal{P}_2$'s nonce k_2^{-1} to an additive sharing $a + b$, namely, $\mathcal{P}_1$ and $\mathcal{P}_2$ receive a and b respectively such that $a+b = x_1.k_2^{-1} \mod q$. After the query, $\mathcal{P}_1$ computes $Z \leftarrow [a] \cdot P$ and sends it to $\mathcal{P}_2$, who computes $(Z + [b] \cdot P) \cdot k_2$ and checks if it is equal to Q_1, so as to control the correctness of the MtA input against a malicious $\mathcal{P}_1$.
- Online signing, that starts by $\mathcal{P}_2$ generating locally its share of the signature after the MtA invocation, namely $s_2 = k_2^{-1}(H(m) + r.x_2) + b \cdot r \mod q$, then sends it to $\mathcal{P}_1$ who will generate the signature by calculating locally $s = k_1^{-1}(s_2 + a \cdot r) \mod q$ and verifying whether this signature is valid. Note that the nonce generation and the MtA invocation are message-independent, thus we can refer to these two steps as the offline signing.

The complete process is illustrated in Figure 5. Also the graphical representation of the key distribution and signing phase are given in Figure 6 and Figure 7, respectively.

Security of our protocol is simulation based, following the real/ideal paradigm [24]. The type of adversary we considered is a malicious one with static corruption. This implies that the adversary $\mathcal{A}$ can deviate from the protocol, but the party he corrupts (either $\mathcal{P}_1$ or $\mathcal{P}_2$) is set prior to the protocol execution.

Theorem 2. *The protocol of Figure 5 securely implements the functionality of Figure 1 in the* $(\mathcal{F}_{\mathsf{ZKP}}, \mathcal{F}_{\mathsf{Commit\text{-}ZK}}, \mathcal{F}_{\mathsf{MtA}})$*-hybrid model in the presence of a malicious static adversary under the ideal/real definition of [24], assuming the Computational Diffie-Hellman problem is hard.*

The proof of theorem 2 is given in Appendix B.



2-party ECDSA Protocol

Given an elliptic curve group $\mathbb{G}$ of order q and a generator P of $\mathbb{G}$:

Key Generation: To generate a pair of keys for the ECDSA algorithm, the parties do as follows:

1. $\mathcal{P}_1$ generates $x_1 \leftarrow \mathbb{Z}_q$ and calculates $Q_1 = [x_1] \cdot P$.
2. $\mathcal{P}_1$ sends ($\mathsf{com\text{-}prove}$, 1, Q_1, x_1) to $\mathcal{F}_{\mathsf{Commit\text{-}ZK}}$.
3. $\mathcal{P}_2$ receives ($\mathsf{proof\text{-}receipt}$, 1)
4. $\mathcal{P}_2$ generates $x_2 \leftarrow \mathbb{Z}_q$ and calculates $Q_2 = [x_2] \cdot P$.
5. $\mathcal{P}_2$ sends (prove, 2, Q_2, x_2) to $\mathcal{F}_{\mathsf{ZKP}}$.
6. $\mathcal{P}_1$ receives (proof, 2, Q_2) from $\mathcal{F}_{\mathsf{ZKP}}$. If not, $\mathcal{P}_1$ aborts.
7. $\mathcal{P}_1$ sends ($\mathsf{decom\text{-}proof}$, 1) to $\mathcal{F}_{\mathsf{Commit\text{-}ZK}}$.
8. $\mathcal{P}_2$ receives ($\mathsf{decom\text{-}proof}$, 1, Q_1, z) from $\mathcal{F}_{\mathsf{Commit\text{-}ZK}}$. If $z = 0$, $\mathcal{P}_2$ aborts.

Both parties set $Q \leftarrow Q_1 + Q_2$ to be the public key. The private key is $x \leftarrow x_1 + x_2 \mod q$ (note that no party holds x, but only an additive share of it).

Signing: To sign a message m, the parties do as follows:

1. **Generating the nonce k:**
   (a) $\mathcal{P}_1$ generates $k_1 \leftarrow \mathbb{Z}_q$ and calculates $R_1 = [k_1] \cdot P$.
   (b) $\mathcal{P}_1$ sends ($\mathsf{com\text{-}prove}$, $sid||1$ R_1, k_1) to $\mathcal{F}_{\mathsf{Commit\text{-}ZK}}$.
   (c) $\mathcal{P}_2$ receives ($\mathsf{proof\text{-}receipt}$, $sid||1$, 1)
   (d) $\mathcal{P}_2$ generates $k_2 \leftarrow \mathbb{Z}_q$ and calculates $R_2 = [k_2] \cdot P$.
   (e) $\mathcal{P}_2$ sends (prove, $sid||2$, R_2, k_2) to $\mathcal{F}_{\mathsf{ZKP}}$
   (f) $\mathcal{P}_1$ receives (proof, $sid||2$, R_2) from $\mathcal{F}_{\mathsf{ZKP}}$. If not, $\mathcal{P}_1$ aborts.
   (g) $\mathcal{P}_1$ sends ($\mathsf{decom\text{-}prove}$, $sid||1$) to $\mathcal{F}_{\mathsf{Commit\text{-}ZK}}$.
   (h) $\mathcal{P}_2$ receives ($\mathsf{decom\text{-}proof}$, $sid||1$, R_1, z) from $\mathcal{F}_{\mathsf{Commit\text{-}ZK}}$. If $z = 0$, $\mathcal{P}_2$ aborts.

   Both parties set $R \leftarrow [k_1 \cdot k_2] \cdot P = (r, y)$, corresponding to the nonce $k \leftarrow k_1 \cdot k_2$ (note that no party holds k, but only a multiplicative share of it).
2. **Querying the MtA functionality:**
   (a) $\mathcal{P}_1$ and $\mathcal{P}_2$ query $\mathcal{F}_{\mathsf{MtA}}$ with the respective inputs x_1 and k_2^{-1}. $\mathcal{F}_{\mathsf{MtA}}$ forwards a to $\mathcal{P}_1$ and b to $\mathcal{P}_2$.
   (b) $\mathcal{P}_1$ calculates $Z \leftarrow [a] \cdot P$ and sends it to $\mathcal{P}_2$.
   (c) $\mathcal{P}_2$ verifies if $k_2 \cdot (Z + [b] \cdot P) = Q_1$. If it is not the case $\mathcal{P}_2$ aborts.
3. **Online signing:**
   – $\mathcal{P}_2$ calculates $s_2 \leftarrow k_2^{-1} \cdot (H(m) + x_2 \cdot r) + b \cdot r \mod q$ and sends it to $\mathcal{P}_1$.
   – $\mathcal{P}_1$ calculates $s \leftarrow k_1^{-1} \cdot (s_2 + a \cdot r) \mod q$.
   – $\mathcal{P}_1$ verifies if s is a valid signature of m, if so $\mathcal{P}_1$ outputs (r, s) as the signature.

Fig. 5. 2-party ECDSA Protocol

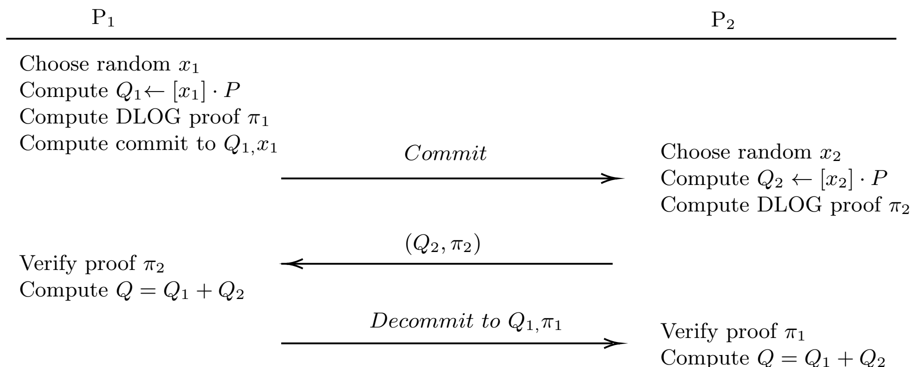


Fig. 6. The 2-Party ECDSA Key Distribution Protocol

3.1 Cost Analysis

We analyze below the theoretical complexity of our two party ECDSA protocol, and compare it with the one of [25] and [14].

Theoretical complexity - key distribution. The distributed key generation consists of generating keys and zero-knowledge proofs. The computation cost can be examined in terms of EC multiplications as this is the heaviest operation performed. For the keys, each party carries out 1M to produce its share of the public key. On the other hand, two zero-knowledge proofs of knowledge of discrete log need to be produced. Using the standard Schnorr proofs in non-interactive from, each party carries out 1M as a prover and 2M as a verifier. Thus the key distribution requires 8M in total. For the communication cost, each party needs to send its share of the public key and the corresponding NIZKP, and $\mathcal{P}_1$, needs to send as well a commitment to its share at the beginning of the protocol, which consists of an output of the hash function H being used (of size λ bits). Assuming one EC point can be represented in λ bits, and a NIZKP consists of two field elements and one EC point, the size of data communicated between the parties is $9 \cdot \lambda$. Note that the cost of our key distribution is the same as [25], which is a negligible cost compared to the one of Lindell [14], as the latter is dominated by the usage of homomorphic encryption.

Theoretical complexity - signing. The computation cost of the signing protocol can be examined in terms of EC multiplications and MtA invocations. That is, the first step of the offline phase is similar to the key generation, except that the nonce is multiplicatively shared, thus each party needs to perform an extra EC multiplication so as to obtain R. Also, the calculation needed to check $\mathcal{P}_1$'s input to MtA requires 3 EC multiplications. Thus, it results in a computation cost of 13M, and a communication cost of $10 \cdot \lambda$. To obtain the total cost of the offline phase, one needs to add these costs to the executing of 1 MtA. The cost of this depends on the instantiation used, which can yield different results. For instance, MtA can be instantiated from the Paillier encryption scheme, i.e.,

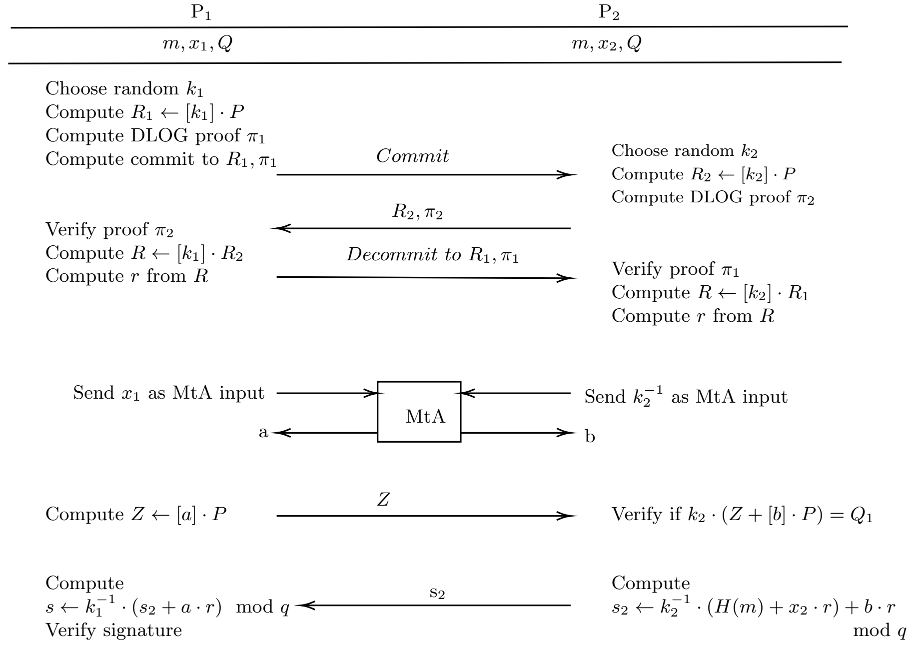


Fig. 7. The 2-Party ECDSA Signing Protocol

the building block upon which [14] is based. This would result in a protocol where homomorphic encryption is used in the offline phase, with an inferior performance to that of [14], however with an improved online phase performance than [14]

In fact, the online phase consists of performing operations over a field by both parties, and a verification phase of the signature, which requires from the verifier (in our case $\mathcal{P}_1$) to carry out 2M. Thus neglecting the cost of operations over a field, the computation cost of the online phase if 2M. As for the communication cost, $\mathcal{P}_2$ needs to send one field element to $\mathcal{P}_1$, thus λ bits of data need to be communicated between the parties.

Table 1 compares these costs with the ones of [25] and [14]. For [14], the cost of the homomorphic operations is dominated by exponentiations modulo N^2 by numbers from Z_N. We refer to these exponentiations as E. The value N refers to the public key of Paillier, which determines the size of a Paillier encryption, which is a number from Z_{N^2}. MtA refers to the cost of invoking an instantiation of the MtA functionality.

As one can notice, the computation and communication cost of our online phase is the same as [25], which outperforms the one of [14], for which the online phase requires performing an extra exponentiation, and sending an encryption of Paillier (N is typically of size 2048 bits) instead of a field element. However,

our offline phase outperforms the one of [25], as in our case the computation and communication required are reduced respectively by 1M, and $2 \cdot \lambda$.

Table 1. Cost Analysis of Signing

Protocol	Computation		Communication	
	Offline	Online	Offline	Online
Lindell [14]	10M+2E	2M+1E	$9 \cdot \lambda$	$2 \cdot log_2(N)$
Xue et al. [25]	14M+1MtA	2M	$12 \cdot \lambda$+1MtA	λ
Ours	13M+1MtA	2M	$10 \cdot \lambda$+1MtA	λ

3.2 Implementation

We implemented our protocol in C++, over the secp256k1 curve standardized by NIST, which is the one used by Bitcoin. The hash function we used is Sha256, and for the curve operation we used the Secp256k1[3] C library. The implementation can be found in `https://github.com/YounesTal1/2ecdsa`

We took runtimes with an Amazon instance of "t2.xlarge" (16 GiB of memory and 4 vCPU), running with "Ubuntu 18.04.6 LTS", this instance was located in "us-east-1" (Virginia). The runtimes we obtained are given in Table 2. Note that our implementation used a single thread, and that the runtimes reflect only the computation cost of our protocol. These runtimes were obtained by calculating the average time needed for a 1000 key generation, where each key was used to sign 100 messages. Note also that the MtA implemented is a dummy one (one party receives the multiplicative share of the other party, and produces the additive shares for both parties), hence Table 2 contains the term MtA, where one can plug in the time needed to execute the MtA of their choice to obtain the overall runtime of the offline signing. As can be observed, our protocol is efficient in terms of the computation cost, for both key generation and signing. That is, the key generation only requires 1.05ms and the offline phase (excluding the MtA call) requires 1.26ms. The difference in runtimes is mainly due to the five extra EC multiplications, namely two extra EC multiplications that need to be performed for calculating R, as the nonce is shared multiplicatively, and three extra EC multiplications that need to be performed for checking the correctness of $\mathcal{P}_1$'s input to MtA. The online phase only requires 0.1ms, as this is dominated by two EC multiplications for the signature verification.

Table 2. Runtimes in milliseconds of our protocol. These runtimes correspond to the time needed for one key generation, one execution of the offline phase, and one execution of the online phase.

Key generation	Offline signing	Online signing
1.05	1.26 + MtA	0.10

[3] https://github.com/bitcoin-core/secp256k1

To understand the impact of the MtA functionality on the runtimes, so as to give a comprehensive evaluation of our protocol, let us consider two cases, an OT based MtA, and a CL based MtA. For this we will base our analysis on the runtimes of [25]. That is, [25] implemented their protocol with different MtA instanciations. For the case of OT, the offline phase took 2.6ms and required 90.9 KBytes of data to be communicated (Table 4 of [25]). For the case of CL, the offline phase took 1386ms and required 1.7KB of data to be communicated (Table 5 of [25]). As for the online phase, it took 0.2ms, which is dominated by 2M operations.

As the offline phase of [25] consists of 14M+1MtA, and requires $12 \cdot \lambda$ +1MtA (see Table 1), for the case of OT, one would estimate the MtA runtime to be around 1.2ms, and the communication cost of the MtA to be 90.52 KBytes, thus based on this, our offline phase would take around 2.46ms and require 9.84 KBytes, hence a gain of 5.4 percent on the running time. For the case of CL, one would estimate the MtA runtime to be around 1384.6ms, and the communication cost of the MtA to be 1.32 KBytes, thus based on this, our offline phase would take around 1385.9ms and require 1.636 KBytes, hence a gain of 3.7 percent of the size of communicated data.

4 Conclusion

We proposed an efficient two-party ECDSA protocol secure against malicious adversaries. Our protocol has a light online phase, dominated by the verification step of ECDSA, and requires only sending one field element from one party to the other. Our offline phase uses a single call of the MtA functionality, and to the best of our knowledge, it offers the most efficient offline phase in terms of the computational and communication cost for such an online phase.

It is worth noting that the asymmetry introduced to the protocol, between what the two parties do, particularly the inputs they send to the MtA functionality, poses a challenge to generalize the protocol to the multiparty case with a low number of invocation to the MtA functionality (say at most equal to the number of parties). We leave further exploration as future work.

Acknowledgments

Authors would like to thank the anonymous reviewers for their valuable comments, as well as Muhammed Ali Bingol and Daniele Cozzo for the valuable discussions over the protocol security. This work has been supported by TUBITAK under 2244 project, and by CyberSecurity Research Flanders with reference number VR20192203. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of any of the funders.

References

1. Bao, F., Deng, R.H., Zhu, H.: Variations of diffie-hellman problem. In: International conference on information and communications security. pp. 301–312. Springer (2003)
2. Canetti, R., Gennaro, R., Goldfeder, S., Makriyannis, N., Peled, U.: UC non-interactive, proactive, threshold ECDSA with identifiable aborts. IACR Cryptol. ePrint Arch. p. 60 (2021), https://eprint.iacr.org/2021/060
3. Castagnos, G., Catalano, D., Laguillaumie, F., Savasta, F., Tucker, I.: Two-party ECDSA from hash proof systems and efficient instantiations. In: Boldyreva, A., Micciancio, D. (eds.) Advances in Cryptology – CRYPTO 2019. pp. 191–221. Springer International Publishing, Cham (2019)
4. Castagnos, G., Laguillaumie, F.: Linearly homomorphic encryption from. In: Cryptographersâ Track at the RSA Conference. pp. 487–505. Springer (2015)
5. Dalskov, A.P.K., Orlandi, C., Keller, M., Shrishak, K., Shulman, H.: Securing DNSSEC keys via threshold ECDSA from generic MPC. In: Chen, L., Li, N., Liang, K., Schneider, S.A. (eds.) Computer Security - ESORICS 2020 - 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14-18, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12309, pp. 654–673. Springer (2020). https://doi.org/10.1007/978-3-030-59013-0_32, https://doi.org/10.1007/978-3-030-59013-0_32
6. Damgård, I., Jakobsen, T., Nielsen, J., Pagter, J., Østergaard, M.: Fast threshold ECDSA with honest majority. In: Galdi, C., Kolesnikov, V. (eds.) Security and Cryptography for Networks. pp. 382–400. Lecture Notes in Computer Science, Springer, Netherlands (2020). https://doi.org/10.1007/978-3-030-57990-6_19, 12th International Conference on Security and Cryptography for Networks, SCN 2020 ; Conference date: 14-09-2020 Through 16-09-2020
7. Desmedt, Y.: Society and group oriented cryptography: a new concept. In: Pomerance, C. (ed.) Advances in Cryptology — CRYPTO '87. pp. 120–127. Springer Berlin Heidelberg, Berlin, Heidelberg (1988)
8. Doerner, J., Kondi, Y., Lee, E., Shelat, A.: Secure two-party threshold ECDSA from ECDSA assumptions. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 980–997. IEEE (2018)
9. Doerner, J., Kondi, Y., Lee, E., Shelat, A.: Threshold ECDSA from ECDSA assumptions: The multiparty case. 2019 IEEE Symposium on Security and Privacy (SP) pp. 1051–1066 (2019)
10. ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE transactions on information theory **31**(4), 469–472 (1985)
11. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Odlyzko, A.M. (ed.) Advances in Cryptology — CRYPTO' 86. pp. 186–194. Springer Berlin Heidelberg, Berlin, Heidelberg (1987)
12. Gennaro, R., Goldfeder, S.: Fast multiparty threshold ECDSA with fast trustless setup. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 1179–1194 (2018)
13. Kondi, Y., Magri, B., Orlandi, C., Shlomovits, O.: Refresh when you wake up: Proactive threshold wallets with offline devices. In: 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021. pp. 608–625. IEEE (2021). https://doi.org/10.1109/SP40001.2021.00067, https://doi.org/10.1109/SP40001.2021.00067

14. Lindell, Y.: Fast secure two-party ECDSA signing. In: Katz, J., Shacham, H. (eds.) Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10402, pp. 613–644. Springer (2017). https://doi.org/10.1007/978-3-319-63715-0_21, https://doi.org/10.1007/978-3-319-63715-0_21
15. Lindell, Y.: Secure multiparty computation. Communications of the ACM **64**(1), 86–96 (2020)
16. Lindell, Y., Nof, A.: Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 1837–1854 (2018)
17. MacKenzie, P., Reiter, M.K.: Two-party generation of DSA signatures. In: Annual International Cryptology Conference. pp. 137–154. Springer (2001)
18. Makriyannis, N., Peled, U.: A note on the security of GG18 (2021)
19. Mitsunari, S., Sakai, R., Kasahara, M.: A new traitor tracing. IEICE transactions on fundamentals of electronics, communications and computer sciences **85**(2), 481–484 (2002)
20. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: International conference on the theory and applications of cryptographic techniques. pp. 223–238. Springer (1999)
21. Rabin, M.O.: How to exchange secrets with oblivious transfer. Technical Report TR-81, Aiken Computation Lab (1981), https://www.iacr.org/museum/rabin-obt/obtrans-eprint187.pdf
22. Schnorr, C.: Efficient signature generation by smart cards. In: Advances in Cryptology — CRYPTO '87. pp. 161–174 (1991)
23. Smart, N.P., Talibi Alaoui, Y.: Distributing any elliptic curve based protocol. In: Albrecht, M. (ed.) Cryptography and Coding. pp. 342–366. Springer International Publishing, Cham (2019)
24. Wigderson, A., Goldreich, O., Micali, S.: How to play any mental game. In: Proceedings the 19th Annual ACM Symposium on the Theory of Computing. pp. 218–229 (1987)
25. Xue, H., Au, M.H., Xie, X., Yuen, T.H., Cui, H.: Efficient online-friendly two-party ECDSA signature. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. pp. 558–573 (2021)
26. Yao, A.C.: Protocols for secure computations. In: 23rd annual symposium on foundations of computer science (sfcs 1982). pp. 160–164. IEEE (1982)

A Proof of Theorem 1

The proof below is taken from [19].

Let us assume we can solve the Diffie-Hellman problem, then given the tuple $(P,[x]\cdot P)$ one can obtain $[x^{-1}]\cdot P = [x^{-1}\cdot x^{-1}]\cdot([x]\cdot P)$ from solving the Diffie-Hellman problem over the tuple $([x]\cdot P,\ P = x^{-1}\cdot([x]\cdot P),\ P = x^{-1}\cdot([x]\cdot P))$.

Conversely, let us assume we can solve the 1-Weak Diffie-Hellman problem, then given the tuple $(P,\ [a]\cdot P,\ [b]\cdot P)$, one can obtain $([a^2]\cdot P)$ from solving the 1-Weak Diffie-Hellman problem over the tuple $([a]\cdot P,\ P = a^{-1}\cdot([a]\cdot P))$. Similarly,

one can obtain ($[b^2]\cdot P$) and ($[(a+b)^2]\cdot P$) from the tuples ($[b]\cdot P$, P) and ($[a+b]\cdot P$, P) respectively. Next, one can obtain ($[a\cdot b]\cdot P$) by calculating $2^{-1}\cdot[((a+b)^2 - (a)^2 - (b)^2)]\cdot P$.

B Proof of Theorem 2

In Figure 8, we build a simulator $\mathcal{S}$, to simulate $\mathcal{P}_1$ when $\mathcal{P}_2$ is corrupt, and to simulate $\mathcal{P}_2$ when $\mathcal{P}_1$ is corrupt. Below, we sketch a proof to demonstrate why the views in a real and a simulated execution will be indistinguishable for an adversary $\mathcal{A}$.

B.1 Corrupted $\mathcal{P}_1$

Key generation phase. The difference between the real execution and the simulated execution is the generation of Q_2. In the case of a real execution, Q_2 is computed as $[x_2]\cdot P$ where x_2 is randomly generated, while in the case of a simulated run, Q_2 is computed by calculating $Q_2 \leftarrow Q - Q_1$. Since Q is randomly generated by the $\mathcal{F}_{\mathsf{2ECDSA}}$ functionality of Figure 1 ($Q \leftarrow [x]\cdot P$ for a randomly generated x), then the distributions from which Q_2 is generated in the real and simulated executions are indistinguishable.

Signing phase. In the nonce generation, a similar argument can be given to show that the views are indistinguishable. That is in a real execution, R_2 is computed as $[k_2]\cdot P$ where k_2 is randomly generated, while in the case of a simulated run, R_2 is computed by calculating $R_2 \leftarrow [k_1^{-1}]\cdot R$. Since R is randomly generated by $\mathcal{F}_{\mathsf{2ECDSA}}$ ($R \leftarrow [k]\cdot P$ for a randomly generated k), then the distributions from which R_2 is generated in the real and simulated executions are indistinguishable.

In the MtA call, both in the real and simulated executions, $\mathcal{P}_1$ is intended to receive a randomly generated a, thus the views are indistinguishable. Afterwards, $\mathcal{P}_1$ sends Z to $\mathcal{P}_2$. In a simulated execution, $\mathcal{P}_2$ aborts if $\mathcal{P}_1$ has provided to the MtA functionality a different input than x_1, or if he sends a different value than $[a]\cdot P$. This behaviour is equivalent to what happens in a real execution, where $\mathcal{P}_2$ checks whether $k_2\cdot(Z+[b]\cdot P) = Q_1$. That is, let us denote by ϵ_1, the additive error that $\mathcal{P}_1$ can introduce to x, namely, $\mathcal{P}_1$ sends to MtA the value $x' \leftarrow x+\epsilon \mod q$, and by E, the additive error that $\mathcal{P}_1$ can introduce to Z, namely, $\mathcal{P}_1$ sends $\mathcal{P}_2$ the value $Z' \leftarrow Z+E$. To pass the check of $\mathcal{P}_2$, the following equation needs to be satisfied:

$$\begin{aligned}
Q_1 &= k_2\cdot(Z' + [b]\cdot P)\\
&= k_2\cdot(E + Z + [b]\cdot P)\\
&= k_2\cdot(E + [a]\cdot P + [b]\cdot P)\\
&= k_2\cdot(E + (x_1+\epsilon_1)\cdot k_2^{-1}\cdot P)\\
&= k_2\cdot(E + x_1\cdot k_2^{-1}\cdot P + \epsilon_1\cdot k_2^{-1}\cdot P)\\
&= Q_1 + k_2\cdot(E + \epsilon_1\cdot k_2^{-1}\cdot P)
\end{aligned}$$

2-party ECDSA Simulator

The simulator $\mathcal{S}$ does as follows:

Corrupt $\mathcal{P}_1$ (i.e. simulating $\mathcal{P}_2$):

1. **Key Generation:**
   - $\mathcal{S}$ queries $\mathcal{F}_{\mathsf{2ECDSA}}$ to obtain the public key Q.
   - $\mathcal{S}$ receives (com-prove, 1, Q_1, x_1) from $\mathcal{A}$ intended to be sent to $\mathcal{F}_{\mathsf{Commit\text{-}ZK}}$.
   - $\mathcal{S}$ checks whether $Q_1 = [x_1] \cdot P$, if it is the case, $\mathcal{S}$ calculates $Q_2 = Q - Q_1$, and sends (proof,2 Q_2) to $\mathcal{A}$, as if $\mathcal{F}_{\mathsf{ZKP}}$ sent it. If Q_1 is different than $[x_1] \cdot P$, $\mathcal{S}$ does the same with a randomly generated Q_2.
   - $\mathcal{S}$ receives (decom-proof, 1, Q_1, z) from $\mathcal{F}_{\mathsf{Commit\text{-}ZK}}$. If $z = 1$, the simulator stores (x_1, Q) for further use, otherwise, $\mathcal{S}$ simulates $\mathcal{P}_2$ aborting.
2. **Signing:**
   (a) **Nonce generation:**
   - $\mathcal{S}$ queries $\mathcal{F}_{\mathsf{2ECDSA}}$ to obtain the signature (r, s), then calculates $R \leftarrow [s^{-1} \cdot H(m)] \cdot P + [s^{-1} \cdot r] \cdot Q$ as in the verification procedure.
   - $\mathcal{S}$ receives (com-prove, $sid||1$, R_1, k_1) from $\mathcal{A}$ intended to be sent to $\mathcal{F}_{\mathsf{Commit\text{-}ZK}}$.
   - $\mathcal{S}$ checks whether $R_1 = [k_1] \cdot P$, if it is the case, $\mathcal{S}$ calculates $R_2 = k_1^{-1} \cdot R$, and sends (proof, $sid||2$, R_2) to $\mathcal{A}$, as if $\mathcal{F}_{\mathsf{ZKP}}$ sent it. If R_1 is different than $[k_1] \cdot P$, $\mathcal{S}$ does the same with a randomly generated R_2.
   - $\mathcal{S}$ receives (decom-proof, 1, R_1, z) from $\mathcal{F}_{\mathsf{Commit\text{-}ZK}}$. If $z = 1$, the simulator stores (k_1, R) for further use, otherwise, $\mathcal{S}$ simulates $\mathcal{P}_2$ aborting.

   (b) **MtA**:
   - The simulator here receives x_1 from $\mathcal{A}$ intended to be sent to $\mathcal{F}_{\mathsf{MtA}}$, then forwards a randomly generated number a to $\mathcal{P}_1$. If the x_1 received here is different from the share of the secret key of $\mathcal{P}_1$, the simulator sets an internal flag $\mathsf{cheat}_{sid||1}$ to be 1.
   - The simulator receives Z from $\mathcal{P}_1$. If $\mathsf{cheat}_{sid||1}$ is equal to 1, or Z is different than $[a] \cdot P$, $\mathcal{S}$ simulates $\mathcal{P}_2$ aborting.

   (c) **Online signing**:
   - $\mathcal{S}$ calculates $s_2 \leftarrow s \cdot k_1 - a \cdot r \mod q$ and sends it to $\mathcal{P}_1$.

Corrupt $\mathcal{P}_2$ (i.e. simulating $\mathcal{P}_1$):

1. **Key Generation:**
   - $\mathcal{S}$ queries $\mathcal{F}_{\mathsf{2ECDSA}}$ to obtain the public key Q.
   - $\mathcal{S}$ sends (receipt, 1) to $\mathcal{A}$ as if it was sent by $\mathcal{F}_{\mathsf{Commit\text{-}ZK}}$.
   - $\mathcal{S}$ receives (prove, 2, Q_2 , x_2) from $\mathcal{P}_2$ intended to be sent to $\mathcal{F}_{\mathsf{ZKP}}$.
   - $\mathcal{S}$ checks if $Q_2 = [x_2] \cdot P$. If it is not the case, $\mathcal{S}$ simulates $\mathcal{P}_1$ aborting.
   - $\mathcal{S}$ calculates $Q_1 = Q - Q_2$, and sends (decom-proof, 1, Q_1, 1) as if $\mathcal{F}_{\mathsf{Commit\text{-}ZK}}$ sent it. $\mathcal{S}$ stores (x_2, Q) for further use.
2. **Signing:**
   (a) **Nonce generation:**
   - $\mathcal{S}$ queries $\mathcal{F}_{\mathsf{2ECDSA}}$ to obtain the signature (r, s), then calculates $R \leftarrow [s^{-1} \cdot H(m)] \cdot P + [s^{-1} \cdot r] \cdot Q$ as in the verification procedure.
   - $\mathcal{S}$ sends (receipt, $sid||1$, 1) to $\mathcal{A}$ as if it was sent by $\mathcal{F}_{\mathsf{Commit\text{-}ZK}}$.
   - $\mathcal{S}$ receives (prove, $sid||2$, R_2 , k_2) from $\mathcal{P}_2$ intended to be sent to $\mathcal{F}_{\mathsf{ZKP}}$.
   - $\mathcal{S}$ checks if $R_2 = [k_2] \cdot P$. If it is not the case $\mathcal{S}$ simulates $\mathcal{P}_1$ aborting.
   - $\mathcal{S}$ calculates $R_1 = k_2^{-1} \cdot R$, and sends (decom-proof, 1, R_1, 1) as if $\mathcal{F}_{\mathsf{Commit\text{-}ZK}}$ sent it. $\mathcal{S}$ stores (k_2, R) for further use.

   (b) **MtA:**
   - The simulator here receives k_2^{-1} from $\mathcal{A}$ intended to be sent to $\mathcal{F}_{\mathsf{MtA}}$, then forwards a randomly generated number b to $\mathcal{P}_2$. If the k_2^{-1} received here is different from the one stored in the nonce generation, the simulator sets an internal flag $\mathsf{cheat}_{sid||2}$ to be 1.
   - $\mathcal{S}$ sends $k_2^{-1} \cdot Q_1 - [b] \cdot P$ to $\mathcal{A}$. The k_2 used by the simulator here and in the next step is the one he received in the MtA call.

   (c) **Online signing:**
   - $\mathcal{S}$ receives s_2 from $\mathcal{A}$. If $\mathsf{cheat}_{sid||2} = 1$ or s_2 is different from $k_2^{-1} \cdot (H(m) + x_2 \cdot r) + b \cdot r \mod q$, $\mathcal{S}$ simulates $\mathcal{P}_1$ aborting. Otherwise, $\mathcal{S}$ outputs (r, s) as the signature.

Fig. 8. 2-party ECDSA Simulator

which implies that $k_2 \cdot E + \epsilon_1 \cdot P = 0$. If $E = \mathcal{O}$, then $\epsilon_1 = 0 \mod q$. Also if $\epsilon_1 = 0 \mod q$, then $E = \mathcal{O}$ as $k_2 \neq 0$ mod q. Thus $E = \mathcal{O}$ or $\epsilon_1 = 0 \mod q$ implies that the adversary has not cheated, as we end up with a case where he does not modify the values he is supposed to send.

Let us look at the case where $E \neq \mathcal{O}$ and $\epsilon_1 \neq 0 \mod q$. The equation holds if the adversary chooses ϵ_1 in such a way that $E = \epsilon_1 \cdot [k_2^{-1}] \cdot P = \mathcal{O}$. While $R_2 = [k_2] \cdot P$ is known to the adversary, obtaining $[k_2^{-1}] \cdot P$ from it would mean breaking the 1-Weak Diffie-Hellman problem, which as we have seen is equivalent to the Computational Diffie-Hellman problem which is believed to be hard.

Thus to summarize, the adversary will not be able to make the check pass if he cheats, either in the MtA call or the step afterward.

In the online signing:

If the parties reach this stage, $\mathcal{P}_1$ will be receiving in the simulated execution $s_2 = s \cdot k_1 - a \cdot r \mod q$, which is equal to

$$\begin{aligned}
s_2 &= s \cdot k_1 - a \cdot r \\
&= k^{-1} \cdot (H(m) + r \cdot x) \cdot k_1 - a \cdot r \\
&= k_2^{-1} \cdot (H(m) + r \cdot x) - a \cdot r) \\
&= k_2^{-1} \cdot (H(m) + r \cdot x_1 + r \cdot x_2) - a \cdot r \\
&= k_2^{-1} \cdot (H(m) + r \cdot x_2) + k_2^{-1} \cdot r \cdot x_1 - a \cdot r \\
&= k_2^{-1} \cdot (H(m) + r \cdot x_2) + r \cdot (a + b) - a \cdot r \\
&= k_2^{-1} \cdot (H(m) + r \cdot x_2) + r \cdot b
\end{aligned}$$

which is what $\mathcal{P}_1$ receives in a real execution.

B.2 Corrupted $\mathcal{P}_2$

Key generation phase. Similarly to the case of a corrupted $\mathcal{P}_1$, the difference between the real execution and the simulated execution is the generation of Q_1. In the case of a real execution, Q_1 is computed as $[x_1] \cdot P$ where x_1 is randomly generated, while in the case of a simulated run, Q_1 is computed by calculating $Q_1 \leftarrow Q - Q_2$. Since Q is randomly generated by the $\mathcal{F}_{\mathsf{2ECDSA}}$ functionality of Figure 1 ($Q \leftarrow [x] \cdot P$ for a randomly generated x), then the distributions from which Q_1 is generated in the real and simulated executions are indistinguishable.

Signing phase. Similarly to the case of a corrupted $\mathcal{P}_1$, in the nonce generation, a similar argument can be given to show that the views are indistinguishable. That is in a real execution, R_1 is computed as $[k_1] \cdot P$ where k_1 is randomly generated, while in the case of a simulated run, R_1 is computed by calculating $R_1 \leftarrow [k_2^{-1}] \cdot R$. Since R is randomly generated by $\mathcal{F}_{\mathsf{2ECDSA}}$ ($R \leftarrow [k] \cdot P$ for a randomly generated k), then the distributions from which R_1 is generated in the real and simulated executions are indistinguishable.

In the MtA call, both in a real and simulated executions, $\mathcal{P}_2$ is intended to receive a randomly generated b (In the simulated execution $b = x_1 \cdot k_2^{-1} - a \mod q$ for a randomly generated a. Note that the Simulator uses here and in what follows the k_2 he received at the MtA call, and not the one received during the nonce generation), thus the views are indistinguishable. In the step afterwards, in the simulated execution, $\mathcal{P}_2$ receives $[k_2^{-1}] \cdot Q_1 - [b] \cdot P$, which is the same as what he receives in a real execution, as $[k_2^{-1}] \cdot Q_1 - [b] \cdot P = [k_2^{-1} \cdot x_1] \cdot P - [b] \cdot P = [a] \cdot P$. Thus the views are indistinguishable.

In the online signing:

- if $\mathcal{P}_2$ does not cheat at all during the protocol, he will be able to calculate $s_2 = k_2^{-1} \cdot (H(m) + x_2 \cdot r) + b \cdot r \mod q$ and send it to $\mathcal{P}_1$. In the real execution $\mathcal{P}_1$ will add it to its share s_1, and the sum will yield a valid signature which will be published by $\mathcal{P}_1$. In the simulated execution, s_2 will pass the check of the simulator and therefore he will publish the signature.
- if $\mathcal{P}_2$ cheated at the MtA call, or does not send the correct s_2, in the real execution, $\mathcal{P}_1$ will not find a valid signature after summing up its share with the one of $\mathcal{P}_2$, thus $\mathcal{P}_1$ will send the abort signal. In the simulated execution, either cheat flag will be equal to 1 at this stage, or s_2 will not pass the check of the simulator. In both cases the simulator will abort. That is, the only case where the views will be distinguishable, is when the adversary cheats on the MtA call, and yet manages to send the correct s_2. Let us denote by ϵ, the additive error that the adversary introduces to his input to MtA, namely he sends $k_2^{-1} + \epsilon$ instead of k_2^{-1}. In this case $a + b = x_1 \cdot (k_2^{-1} + \epsilon)$. In order to pass the check, $\mathcal{P}_2$ needs to send s_2 such that $s \cdot k_1 = s_2 + a \cdot r \mod q$. This implies that:

$$\begin{aligned} s_2 &= s \cdot k_1 - a \cdot r \\ &= k^{-1} \cdot (H(m) + r \cdot x) \cdot k_1 - a \cdot r \\ &= k_2^{-1} \cdot (H(m) + r \cdot x_1 + r \cdot x_2) - a \cdot r \\ &= k_2^{-1} \cdot (H(m) + r \cdot x_2) + k_2^{-1} \cdot r \cdot x_1 - a \cdot r \\ &= k_2^{-1} \cdot (H(m) + r \cdot x_2) + r \cdot b - x_1 \cdot r \cdot \epsilon \end{aligned}$$

  As x_1 is unknown to the adversary, he can satisfy this equation only if $\epsilon = 0$, i.e., the case where he does not cheat in the MtA call. Thus the behaviour of the simulator will make the real execution and the simulated one indistinguishable.