Low-Bandwidth Threshold ECDSA via Pseudorandom Correlation Generators

Damiano Abram[1], Ariel Nof[2], Claudio Orlandi[1], Peter Scholl[1], and Omer Shlomovits[3]

[1] Aarhus University, Aarhus, Denmark
{damiano.abram, orlandi, peter.scholl}@cs.au.dk
[2] Technion, Haifa, Israel
ariel.nof@cs.technion.ac.il
[3] ZenGo X, Tel Aviv, Israel
omer.shlomovits@gmail.com

Abstract. Digital signature schemes are a fundamental component of secure distributed systems, and the theft of a signing-key might have huge real-world repercussions e.g., in applications such as cryptocurrencies. Threshold signature schemes mitigate this problem by distributing shares of the secret key on several servers and requiring that enough of them interact to be able to compute a signature. In this paper, we provide a novel threshold protocol for ECDSA, arguably the most relevant signature scheme in practice. Our protocol is the first one where the communication complexity of the preprocessing phase is only logarithmic in the number of ECDSA signatures to be produced later, and it achieves therefore a so-called *silent preprocessing*. Our protocol achieves active security against any number of arbitrarily corrupted parties.

1 Introduction

Threshold signatures allow a set of n servers to produce digital signatures, while ensuring that no subset of up to $t < n$ colluding servers can forge valid signatures on their own. Threshold signatures can be used to provide an additional layer of security in a cryptographic application, by removing the single point of failure that comes from storing the signing key in one place. There has recently been a renewed interest in building threshold signatures for financial applications, particularly for the case of protecting the secret key in a cryptocurrency wallet used to authorise transactions.

The elliptic curve digital signature algorithm, or ECDSA, is one of the most popular signature schemes used today. This is in part due its use in the Bitcoin protocol, where it is deployed with the `secp256k1` curve. Unlike with Schnorr signatures, ECDSA also happens to be a more challenging scheme to make work in the threshold setting, so there have been a number of works in recent years designing new and improved protocols for threshold ECDSA such as [Lin17, GG18, DKLs18, LNR18, CCL$^+$19, DKLs19, ST19, CCL$^+$20, DOK$^+$20, CGG$^+$20, DJN$^+$20, KMOS21]. Most of these works have been in the dishonest majority setting and with active security, where up to $t = n - 1$ of the servers may be maliciously corrupted, and this is the setting we focus on in this paper.

Moreover, several of these protocols work in the preprocessing model, consisting of a *preprocessing phase*, which is a protocol used to generate correlated randomness needed for the *signing phase*. In ECDSA, we sometimes also consider a *pre-signing phase*, where the message-independent component of a signature is produced, possibly before the message is known. Each invocation of the (pre)-signing phase requires fresh preprocessing material, so in large-scale applications, a significant quantity of correlated randomness must be first generated, and then stored for later use. Generating this correlated randomness is typically quite expensive, involving heavy cryptographic machinery like homomorphic encryption and zero-knowledge proofs; however, this cost can be mitigated by executing it ahead of time and in large batches. In contrast, the signing phase is often relatively simple, with a cost not much larger than that of signing in the clear.

1.1 Our Contributions

In this work, we present a new protocol for threshold ECDSA with *silent preprocessing*, by building on recent work on pseudorandom correlation generators (PCGs) [BCG+19b]. In a silent preprocessing phase, the parties first interact to obtain a small amount of correlated randomness, called the *PCG seeds*, which can later be locally expanded to produce a much larger amount of correlated randomness of the right form. Compared with other approaches, using PCGs allows for a preprocessing phase with greatly reduced communication and storage costs among the servers, when amortised over a large number of signatures.

Furthermore, we can obtain a *non-interactive* signing phase, where, after receiving the message to be signed in the online phase, each server only needs to send a single message to obtain the digital signature. Similar to previous works [DOK+20, CGG+20], we do this by revealing the message-independent part of the signature in the pre-signing phase. This requires assuming the enhanced unforgeability property of ECDSA, namely, that unforgeability still holds when these nonces are seen in advance (see the discussion in Section 3.3).

We have implemented our protocol, for the simplified setting where the PCG seeds are distributed to the servers by a dealer in a trusted setup phase. This model is meaningful in practical applications in which, for instance, a client generates its own ECDSA secret key and then distributes it to a number of servers. In this case it is meaningful to ask the client to (also) generate the (short) PCG seeds that will be used in the protocol.

We also describe a cryptographic protocol that can be used to replace the dealer, with security against a static, malicious adversary. but have not implemented this.

1.2 Technical Overview

Background: ECDSA algorithm Let $\mathbb{G}$ be an elliptic curve group with order q and generator G. Recall that an ECDSA signature on a message m is a pair (r, s), where r is the x-coordinate of $x \cdot G$, for a random nonce $x \in \mathbb{F}_q$, and $s = x^{-1} \cdot (\mathsf{H}(m) + r \cdot \mathsf{sk})$. Here, H is a cryptographic hash function into $\mathbb{F}_q$, and $\mathsf{sk} \in \mathbb{F}_q$ is the secret key. As noted in previous work, the reason why turning ECDSA into a threshold algorithm is challenging is mainly the fact that the signing algorithm requires computing the inverse of x, which is a highly non-linear operation.

Threshold ECDSA with ECDSA Tuples We start by describing a simple, passively secure version of our ECDSA protocol and the preprocessing material it uses.

Suppose the parties start with an extended variant of additively secret-shared multiplication triple, that is, each party P_i has random shares $x_i, y_i, z_i, d_i \in \mathbb{F}_q$, satisfying

$$\sum_{i=1}^{n} z_i = x \cdot y, \sum_{i=1}^{n} d_i = \mathsf{sk} \cdot y \quad \text{where } x = \sum_i x_i, y = \sum_i y_i$$

and where sk is the ECDSA signing key. We denote these sharings by $[x], [y], [z], [d]$. (Note, the secret value x will take the role of the random nonce in the signature.)

We refer to this correlated randomness setup as an *ECDSA tuple*. Such a tuple can be used very easily for threshold ECDSA, as follows. In the first round, the parties can reconstruct $x \cdot G$ by simply broadcasting the shares $x_i \cdot G$, and adding these. Then, given the x-coordinate r of $x \cdot G$, each P_i can compute shares of

$$[\tau] := \mathsf{H}(m) \cdot [y] + r \cdot [d]$$

Notice that, since $z = xy$ and $d = \mathsf{sk} \cdot y$, computing $\tau \cdot z^{-1}$ gives the correct s component of the signature. Therefore, to obtain the signature, it is enough for each party P_i to broadcast the shares z_i and τ_i in the second round.

The above approach roughly follows the method of Smart and Talibi [ST19] and Dalskov et al. [DOK+20], who showed how to perform the signing operation using generic MPC operations over the field $\mathbb{F}_q$. However,

by using specialised ECDSA tuples instead of regular multiplication triples, our protocol reduces the number of rounds in the preprocessing phase. We can then obtain different variants for our protocol: we can get a non-interactive signing phase, with a one-round pre-signing stage (to open r), or a non-interactive pre-signing stage (with silent preprocessing) and a two-round online phase. Note moreover that this can be turned into an *amortised* one-round protocol by performing the pre-signing step for signature $i+1$ (i.e., opening r) together with the i-th signing phase (i.e., the opening of s).

Active Security In the above sketch, a corrupt party may lie about its shares $x_i \cdot G, z_i, \tau_i$, violating correctness and potentially harming security. To mitigate this, [DOK+20] uses a generic, actively secure MPC protocol, for instance, by enhancing the additive secret sharing with information-theoretic MACs as in the BeDOZa [BDOZ11] and SPDZ [DPSZ12,DKL+13] protocols. Unfortunately, for our goal of threshold ECDSA with silent preprocessing, this approach is too expensive: while there are efficient PCGs for authenticated multiplication triples [BCG+20], the best construction only works in the two-party setting, and although this was recently extended to the multi-party setting, the concrete costs are much higher [AS21].

Instead of authenticating the entire shared ECDSA tuple, we observe that this is actually unnecessary, and it is enough to *only authenticate the shares of* x. The MACs on x are used to verify the reconstruction of $x \cdot G$, while for z and τ, we allow corrupt parties to introduce errors. We show that any error in either of these values will always lead to an abort, which can be reliably detected by verifying the final signature with the ECDSA public key. This simplification of the preprocessing phase allows us to take advantage of efficient PCG constructions to realise the preprocessing phase with very low communication costs.

Silent Preprocessing When including the MACs on x, the complete correlated randomness we need for our ECDSA tuple is a set of additive sharings over $\mathbb{F}_q$:

$$[x], [y], [x \cdot y], \quad [y \cdot \mathsf{sk}], (M_{i,j}, K_{j,i})_{i,j \in [n]}$$

where sk is the ECDSA signing key and $(M_{i,j}, K_{i,j})$ is a BeDOZa-like MAC on x_i w.r.t. P_j's MAC key α_j[4].

To realise the preprocessing phase to produce ECDSA tuples with low communication, we adopt techniques based on the LPN and ring-LPN assumptions (and a random oracle to generate public LPN matrix) which were used previously to construction PCGs for vector-OLE [BCGI18] and OLE [BCG+20], respectively. The main technique is a method of *sparse vector compression*, which, roughly speaking, allows for succinctly compressing two-party additive shares of a sparse vector, using distributed point functions (DPFs). For a vector of length m with t non-zero coordinates, the size of the compressed shares is roughly $t\lambda \log m$ bits.

Our approach to compressing ECDSA tuples is to sample long, sparse vectors $\boldsymbol{u}_i, \boldsymbol{v}_i$, for each party P_i, which will later be used as a seed to expand into pseudorandom vectors $\boldsymbol{x}_i, \boldsymbol{y}_i \in \mathbb{F}_q^N$ via the LPN assumption, which will make up the shares of the x and y values. Note that if $\boldsymbol{u}_i, \boldsymbol{v}_j$ have t non-zero coordinates, then the tensor product $\boldsymbol{u}_i \otimes \boldsymbol{v}_j$ has at most t^2, and can also be shared between (P_i, P_j) with the sparse vector compression technique. This then allows the two parties to locally obtain shares of the component-wise products of $\boldsymbol{x}_i$ and $\boldsymbol{y}_j$, by applying linear maps based on the LPN assumption. By repeating this between all pairs of parties, we can obtain secret shares of the $x \cdot y$ values in the ECDSA tuples. The drawback of this approach is that computing the full shared tensor product $\boldsymbol{u}_i \otimes \boldsymbol{v}_j$ has $O(N^2)$ complexity, which is prohibitively expensive when N is large. To avoid this cost, we instead use the ring-LPN based approach from [BCG+20]; this replaces the tensor product with a constant number of polynomial multiplications over $\mathbb{F}_q$, an $O(N \log N)$ operation, while relying on a variant of LPN over polynomial rings.

The remaining terms are the shares of $y \cdot \mathsf{sk}$ and $(x_i \cdot \alpha_j)_{i,j \in [n]}$. Since sk and α_j are fixed for each tuple, these can be distributed by applying sparse vector compression on the pairs of products $\boldsymbol{u}_i \cdot \alpha_j, \boldsymbol{v}_i \cdot \mathsf{sk}_j$, each of which is a t-sparse vector. Note that this last technique is essentially a multi-party application of vector-OLE from LPN [BCGI18, BCG+19b]. However, since we are combining this with the ring-LPN based compression for $x \cdot y$, here we also need to rely on the ring-LPN assumption.

[4] $M_{i,j} = K_{i,j} + \alpha_j \cdot x_i$, where $M_{i,j}$ and x_i belong to P_i and $K_{i,j}$ and α_j are known only to P_j.

Overall, the PCG for ECDSA tuples requires compressing $O(n^2)$ sets of sparse vectors, where to achieve λ-bit computational security against the best known attacks on ring-LPN, the sparsity is up to $O(\lambda^2)$. This gives a seed size of $O(n\lambda^3 \log N)$ bits per party, for compressing N ECDSA tuples. When N is large enough, this is a significant saving on the naive storage cost of $O(\lambda N)$ bits. Importantly, since the product $[x \cdot y]$ in an ECDSA tuple does not need to be authenticated, we avoid having to use 3-party DPFs to compress the sparse vectors, as is needed when compressing fully authenticated multiplication triples between > 2 parties [BCG+20, AS21]; 3-party DPFs have a seed size scaling with $\sqrt{N}$ instead of $\log N$, so this is a significant saving in practice.

Open Problems In some applications of threshold cryptography the honest-majority setting is more appealing than the full-threshold setting of this paper. In the full-threshold case if a share is lost, there is no way of recovering the secret key which might have serious repercussions on availability of the system. Our approach could be generalised to the honest-majority setting with small number of parties (e.g., 3 parties tolerating 1 corruption) using replicated secret-sharing. However, doing so for large number of parties appears to be an interesting and challenging open problem.

1.3 Related Work

The first threshold ECDSA protocol in the dishonest majority setting was presented by Mackenzie and Reiter [MR01], but for the 2-party case only. Their protocol was later improved in [GGN16] and [Lin17]. A different approach to the 2-party case was taken by [DKLs18], which presented a less computationally expensive protocol at the expense of increasing the bandwidth and round complexity. The first attempt to generalise these solutions to *any* number of parties was taken by [GGN16]. However, this protocol relied on distributed Paillier key generation, which is not known to be practical for more than 2 parties. The first multiparty ECDSA protocols with both practical signing and key generation were introduced independently by Lindell and Nof [LN18], Gennaro and Goldfeder [GG18] and Doerner et al. [DKLs19]. These were followed by [CCL+20] and [CGG+20], which provide significant improvements to the communication and round complexity. Moreover, Canetti et al. [CGG+20] also showed how to extend their protocol to identifiable abort. Smart and Talibi [ST19] and Dalskov et al. [DOK+20] developed frameworks for threshold ECDSA based on any MPC protocol over the field $\mathbb{F}_q$. For further related work, we refer to the survey of [AHS20].

In Table 1, we present a comparison with all recent threshold ECDSA constructions tolerating any number of parties n and any dishonest majority: for each protocol, the table reports the estimate of the communication complexity amortised *between each pair of parties* and *per signature*. We believe the amortised setting is valuable in practice, for instance, in cryptocurrency custody applications where several powerful, independently located servers may be used to perform threshold signing for a large number of clients (when the clients cannot run full MPC nodes themselves). This is indeed the setting in which several companies working with threshold signatures operate.

For all protocols other than ours, we use the estimates from [CGG+20]. Consistent with them, we assume all group elements are represented using 256 bits and we ignore the communication complexity for key-generation, that in some cases also includes other one-time setup costs. The communication for our protocol considers also the amortised cost for the generation of one ECDSA tuple, when produced in batches of $N = 94019$[5] units. Note that in our construction, the communication between each pair of parties scales linearly with the total number of players. However, as it can be seen by the table, our solution greatly outperforms existing work for any reasonable value of n.

[5] We chose this batch size since it divides $p - 1$, where p is the `secp256k1` group order, which is necessary for FFT algorithms.

	Assumptions	***Communication***
Gennaro and Goldfeder [GG18]	Strong RSA, DCR, DDH	7 KiB
Lindell *et al.* [LN18] (Paillier)	DCR, DDH	7.5 KiB
Lindell *et al.* [LN18] (OT)	DDH	190 KiB
Doerner *et al.* [DKLs19]	DH	90 KiB
Castagnos *et al.* [CCL$^+$20]	Strong Root, Low Order, HSM, DDH	4.5 KiB
Canetti *et al.* [CGG$^+$20]	Strong RSA, DCR, DDH	15 KiB
This work	**Ring-LPN**	**0.017n+0.18 KiB**

Table 1: Comparison to previous works: Hardness assumptions and communication complexity of the protocol, amortised both over the number of signatures and the number of parties n. See the text for more details.

2 Preliminaries

2.1 Notation

Let $\mathbb{G}$ be a group of prime order q, and let $\mathbb{F}_q$ be the finite field with q elements. We use additive notation for $\mathbb{G}$.

We consider n parties $P_0, P_1, \ldots, P_{n-1}$. Let $\mathcal{C}$ denote the set of (indices of) corrupted parties, and $\mathcal{H}$ denote the complement set of honest parties.

We write $[m]$ to mean the set $\{0, 1, \ldots, m-1\}$. Vectors are represented using bold letters and we indicate the j-th entry of $\boldsymbol{v}$ either as a subscript, e.g. v_j, or between square brackets, e.g. $v[j]$. For two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$, we write $\langle \boldsymbol{u}, \boldsymbol{v} \rangle$ to mean the inner product, and denote the outer product and outer sum by $\boldsymbol{u} \otimes \boldsymbol{v}$ and $\boldsymbol{u} \boxplus \boldsymbol{v}$ respectively. Recall that if $\boldsymbol{u}$ and $\boldsymbol{v}$ have dimensions m and l, the outer product and outer sum are the ml-dimensional vectors whose $(im + j)$-th entry is $u_i \cdot v_j$ and $u_i + v_j$ respectively. We use $\mathbb{P}$ to denote a probability measure and λ for the security parameter.

2.2 The ECDSA Signing Algorithm

Let $(\mathbb{G}, G, q)$ be a tuple of an Elliptic curve group, generator and the group order. Assume that q is prime. Each element $a \in \mathbb{G}$ is represented by a pair (a_x, a_y) where a_x is the projection of a on the x-axis and a_y is the projection of a on the y-axis. Let $\pi : \mathbb{G} \to \mathbb{F}_q$ be the function that maps a point $a \in \mathbb{G}$ into $a_x \bmod q$, moreover, let $\mathsf{H} : \{0,1\}^* \to \mathbb{F}_q$ be a hash function. The ECDSA scheme consists of the following algorithms:

- $\mathsf{KeyGen}(\mathbb{G}, G, q)$: choose a random $\mathsf{sk} \in \mathbb{F}_q$ and set $\mathsf{PK} \leftarrow \mathsf{sk} \cdot G$. Output is $(\mathsf{sk}, \mathsf{PK})$.
- $\mathsf{Sign}_{\mathsf{sk}}(m)$: set $m' \leftarrow \mathsf{H}(m)$ and choose a random $x \in \mathbb{F}_q$. Then, compute $r \leftarrow \pi(x \cdot G)$ and $s \leftarrow x^{-1} \cdot (m' + r \cdot \mathsf{sk})$. Output (r, s).
- $\mathsf{Verify}_{\mathsf{PK}}\big(m, (r, s)\big)$: set $m' \leftarrow H(m)$, output 1 if $r = \pi\big((m' \cdot G + r \cdot \mathsf{PK}) \cdot s^{-1}\big)$, 0 otherwise.

Observe that if (r, s) is a valid ECDSA signature for the message m, also the pair $(r, q-s)$ is a valid signature for m. That implies that the ECDSA version above is not strongly unforgeable, i.e. given a valid signature for a message m, the adversary is able to generate another valid signature for m without knowing the secret key. Interestingly, [FKP16] proves that this is the only malleability attack against ECDSA and it is therefore possible to make ECDSA strongly unforgeable by only generating signatures (r, s) with $s < q/2$. Clearly, the verification algorithm needs to be modified accordingly, so that (r, s) is rejected whenever $s \geq q/2$. Our protocol is agnostic of whether we use the strongly unforgeable version of ECDSA or not, as discussed in the next subsection.

Elliptic Curves with Small Cofactor In our security proof, we need to efficiently sample a point $R \in \langle G \rangle$ such that $\pi(R) = r$, when given only the r value of a valid signature (r, s). When the cofactor of $\langle G \rangle$ in $\mathbb{G}$ is small (polynomial in λ), this is always possible. Indeed, since q is prime and $q^2 \nmid |\mathbb{G}|$ (otherwise, the cofactor would not be polynomial), it is easy to check whether a point R belongs to $\langle \mathbb{G} \rangle$ by verifying that $qR = \infty$. Moreover, by Hasse's theorem, $|\mathbb{G}| \leq 2p$, where $\mathbb{F}_p$ is the prime field on which the curve is defined. As a consequence, p/q is dominated by a polynomial, so there are only a polynomial number of values $x \in \mathbb{F}_p$ for which $x \bmod q = r$. Since the signature (r, s) is valid, we know that one of these values is the x-coordinate of a point in $\langle G \rangle$. Computing an elliptic curve point of a given x-coordinate can always be done efficiently.

In ECDSA instantiations, the cofactor of $\langle G \rangle$ is almost always small. This is in particular true for the Bitcoin curve *secp256k1* (the cofactor is 1).

2.3 Threshold ECDSA - Security Definition

We define security similarly to [LN18]. In Figure 1 we present the functionality $\mathcal{F}_{\mathsf{ECDSA}}$. The functionality is defined with three commands: key generation, pre-signing and signing. Key generation is called once, and then any number of signing operations can be carried out with the generated key. The signing process is split into pre-signing (which computes and reveals a nonce r) and the actual signing (that computes s). Our protocol achieves security with abort, reflected in the functionality by letting the ideal-world adversary choose whether to send the output to the honest parties or not. We stress that $\mathcal{F}_{\mathsf{ECDSA}}$ is defined in a generic way that can be used with both versions of ECDSA, with or without the the strongly-unforgeable property. Therefore the functionality allows the ideal adversary to see the generated signature but then force the honest parties to output another signature (r, s') as long as it verifies w.r.t. the message m. Clearly, when combining our functionality with the strongly-unforgeable version of ECDSA the adversary cannot come up with a different signature for m.

The Ideal Functionality $\mathcal{F}_{\mathsf{ECDSA}}$

$\mathcal{F}_{\mathsf{ECDSA}}$ works with parties $P_0, \ldots, P_{n-1}$ and an ideal world adversary $\mathcal{A}$, controlling a subset $\mathcal{C} \subsetneq [n]$ of parties.

KeyGen. Upon receiving KeyGen from all parties $P_0, \ldots, P_{n-1}$.

1. Generate an ECDSA key pair $(\mathsf{PK}, \mathsf{sk})$ by choosing a random $\mathsf{sk} \in \mathbb{F}_q$ and let $\mathsf{PK} \leftarrow sk \cdot G$.
2. Send PK to all parties.
3. Ignore future calls to KeyGen.

PreSign. Upon receiving $\mathsf{Presign}(sid)$ from all parties, if KeyGen was already called and sid has not been previously used:

1. Choose a random $x \in \mathbb{F}_q$, compute $r \leftarrow \pi(x \cdot G)$ and send r to $\mathcal{A}$.
2. Upon receiving $(\mathsf{continue}, sid)$ from $\mathcal{A}$, store (sid, r, x).

Sign. Upon receiving $\mathsf{Sign}(sid, m)$ from all $P_0, \ldots, P_{n-1}$, if (sid, r, x) is stored and has not been previously used:

1. Compute $s \leftarrow x^{-1} \cdot (H(m) + r \cdot sk)$.
2. Send s to $\mathcal{A}$ and wait for a pair (r, s') as a reply. If $\mathsf{Verify}_{\mathsf{PK}}\big(m, (r, s')\big) = 1$, output (r, s') to all honest parties. Otherwise abort.

Fig. 1: The ECDSA ideal functionality $\mathcal{F}_{\mathsf{ECDSA}}$

2.4 Module-LPN with Static Leakage

Our protocol is based on the Module-LPN assumption with static leakage, which was defined and analysed for the first time by Boyle et al. in [BCG+20]. This is a variant of the LPN assumption over polynomial

rings (analogously to how ring-LWE extends LWE), with the addition of some leakage. The leakage (which arises in the protocols from [BCG+20]) allows the adversary to try to guess error coordinates, but aborts if any guess is incorrect; this only reveals an average of one bit on the module-LPN secret overall.

Definition 1 (Module-LPN with static leakage). *Consider the ring $R_\lambda := \mathbb{F}_q[X]/\big(F(X)\big)$, where q is a prime and $F(X)$ is a polynomial of degree N. Let t_λ and $c_\lambda \geq 2$ be two positive integers and define the distribution $\mathcal{HW}_t$ that independently samples t noise positions $\big(\omega[i]\big)_{i\in[t]}$ uniformly in $[N]$ and t payloads $\big(\beta[i]\big)_{i\in[t]}$ uniformly in $\mathbb{F}_q$, outputting the ring element*

$$e(X) := \sum_{i\in[t]} \beta[i] \cdot X^{\omega[i]}$$

Consider the game $\mathcal{G}^{Module\text{-}LPN}_{R,t,c,\mathcal{A}}$ described in Figure 2. We say that the R^c-LPN_t problem with static leakage is hard if, for every PPT adversary $\mathcal{A}$, the advantage

$$Adv^{Module\text{-}LPN}_{R,t,c,\mathcal{A}}(\lambda) := \left|\mathbb{P}\Big(\mathcal{G}^{Module\text{-}LPN}_{R,t,c,\mathcal{A}}(\lambda) = 1\Big) - \frac{1}{2}\right|$$

is negligible in the security parameter λ.

The Game $\mathcal{G}^{\mathsf{Module\text{-}LPN}}_{R,t,c,\mathcal{A}}(\lambda)$

Initialisation. The challenger activates the adversary $\mathcal{A}$ with $\mathbb{1}^\lambda$ and samples a random bit $b \xleftarrow{\$} \{0,1\}$. Then, it samples c elements of the ring $e_0, e_1, \ldots, e_{c-1} \xleftarrow{\$} \mathcal{HW}_t$. Let the j-th noise positions of e_i be $\omega_i[j]$.
Query. The adversary is allowed to adaptively issue queries of the form (i, j, I) where $i \in [c]$, $j \in [t]$ and $I \subseteq [N]$. If $\omega_i[j] \in I$, the challenger answers with Success, otherwise, it sends Abort and halts.
Challenge. For every $i \in [c-1]$, the challenger samples $a_i \xleftarrow{\$} R$ and sets $a_{c-1} \leftarrow 1$. Then, it computes

$$u_1 \leftarrow \sum_{i=0}^{c-1} a_i \cdot e_i = \sum_{i=0}^{c-2} a_i \cdot e_i + e_{c-1}.$$

Moreover, it samples $u_0 \xleftarrow{\$} R$. Finally, it gives $(a_0, a_1, \ldots, a_{c-2}, u_b)$ to $\mathcal{A}$. The adversary replies with a bit b'. The final output of the game is 1 if and only if $b = b'$.

Fig. 2: The Module-LPN game.

It is easy to see that the bigger c and t become, the harder it is for the adversary to win the game. Observe that when $F(X)$ splits into N distinct linear factors over $\mathbb{F}_q$ corresponding to N distinct roots $\xi_0, \xi_1, \ldots, \xi_{N-1}$, the ring R is isomorphic to $\mathbb{F}_q^N$ by the Chinese Remainder Theorem[6]. The isomorphism map ϕ sends a polynomial $p(X) \in R$ to $\big(p(\xi_0), p(\xi_1), \ldots, p(\xi_{N-1})\big) \in \mathbb{F}_q^N$. Observe that the map is well defined as the ideal $\big(F(X)\big)$ vanishes over $\xi_0, \xi_1, \ldots, \xi_{N-1}$. If we additionally assume that $F(X)$ is a cyclotomic polynomial, the isomorphism can be efficiently computed using the Fast Fourier Transform (FFT) [BCG+20]. The security analysis of [BCG+20] argues that this does not introduce any significant vulnerability, so our protocol will take advantage of cyclotomic polynomials.

[6] $\mathbb{F}_q^N$ is a ring with relation to pointwise addition and multiplication.

2.5 Pseudorandom Correlation Generators

To achieve low-bandwidth in the preprocessing phase, we use *Pseudorandom Correlation Generators (PCGs)* [BCG+19a, BCG+19b, BCG+20]. Informally speaking, a PCG is a distributed form of pseudorandom generator (PRG), where each party has a different seed that can be expanded into a long stream of bits. Now, while a PRG produces a stream of uniformly random bits, a PCG lets each party expand their seed into different streams that satisfy some joint correlation. In more detail, a PCG is a pair of algorithms $(\mathsf{PCG.Gen}, \mathsf{PCG.Eval})$, the first one of which generates n short correlated seeds, one for each player. The evaluation algorithm is used to expand each seed into a large amount of correlated randomness without any interaction. Furthermore, the security of the construction guarantees that corrupted parties learn nothing about the outputs of the honest parties as long as their seeds remain secret.

This framework has a double advantage. First of all, in order to generate large amounts of correlated material, we can just focus our attention on designing secure setup protocols for the generation and distributions of the small seeds. The second and most important advantage is that, due to their small size, it possible to generate the seeds using setup protocols with very low communication complexity compared to the size of the expanded seeds.

To formalise this, we first recall the notion of a reverse samplable correlation generator [BCG+19b], which captures the class of correlations that PCGs may support.

Definition 2 (Reverse Samplable Correlation Generator). *An n-party correlation generator is a PPT algorithm* CorGen *taking as input the security parameter $1\!\!1^\lambda$ and outputting n correlated outputs $R_0, R_1, \ldots, R_{n-1}$, one for each party.*

We say that CorGen *is reverse samplable if there exists a PPT algorithm* $\mathsf{RSample}$ *such that, for every set of corrupted parties $\mathcal{C} \subsetneq [n]$ the following distribution*

$$\left\{ (R'_i)_{i\in[n]} \;\middle|\; \begin{array}{l} (R_0, R_1, \ldots, R_{n-1}) \xleftarrow{\$} \mathsf{CorGen}(1\!\!1^\lambda) \\ \forall i \in \mathcal{C}: \quad R'_i \leftarrow R_i \\ (R'_i)_{i\in\mathcal{H}} \xleftarrow{\$} \mathsf{RSample}\big(1\!\!1^\lambda, \mathcal{C}, (R'_i)_{i\in\mathcal{C}}\big) \end{array} \right\}$$

is computationally indistinguishable from $\mathsf{CorGen}(1\!\!1^\lambda)$.

The definition says that given a subset of outputs of the correlation generator CorGen, we are able to simulate the remaining outputs. In general, there exist correlation generators that are not reverse samplable. However, finding meaningful definitions for PCG becomes hard in such cases [BCG+19b]. Moreover, since the type of correlation used in this paper is reverse samplable, we do not need to worry about this issue.

We now finally formalise the definition of PCG.

Definition 3 (Pseudorandom Correlation Generator). *Let* CorGen *be an n-party reverse samplable correlation generator. A PCG for* CorGen *is a pair of PPT algorithms* $(\mathsf{PCG.Gen}, \mathsf{PCG.Eval})$ *with the following syntax.*

- $\mathsf{PCG.Gen}$ *takes as input the security parameter $1\!\!1^\lambda$ and outputs n small correlated seeds $\kappa_0, \kappa_1, \ldots, \kappa_{n-1}$, one for each party.*
- $\mathsf{PCG.Eval}$ *takes as input a PCG seed κ_i and the associated index $i \in [n]$. The output is an element R_i, ideally corresponding to the i-th output of* $\mathsf{CorGen}(1\!\!1^\lambda)$.

We require that the construction satisfies the following properties.

- ***Correctness.*** *The following distribution is computationally indistinguishable from* $\mathsf{CorGen}(1\!\!1^\lambda)$.

$$\left\{ (R_i)_{i\in[n]} \;\middle|\; \begin{array}{l} (\kappa_0, \kappa_1, \ldots, \kappa_{n-1}) \xleftarrow{\$} \mathsf{PCG.Gen}(1\!\!1^\lambda) \\ \forall i \in [n]: \quad R_i \leftarrow \mathsf{PCG.Eval}(\kappa_i, i) \end{array} \right\}$$

– ***Security.*** *For every subset of corrupted parties $\mathcal{C} \subsetneq [n]$, the following two distributions are computationally indistinguishable.*

$$\left\{ \begin{matrix} (\kappa_i)_{i\in\mathcal{C}} \\ (R_i)_{i\in\mathcal{H}} \end{matrix} \middle| \begin{matrix} (\kappa_0, \kappa_1, \dots, \kappa_{n-1}) \xleftarrow{\$} \mathsf{PCG.Gen}(1\!\!1^\lambda) \\ \forall i \in [n] : \quad R_i \leftarrow \mathsf{PCG.Eval}(\kappa_i, i) \end{matrix} \right\}$$

$$\left\{ \begin{matrix} (\kappa_i)_{i\in\mathcal{C}} \\ (R_i)_{i\in\mathcal{H}} \end{matrix} \middle| \begin{matrix} (\kappa_0, \kappa_1, \dots, \kappa_{n-1}) \xleftarrow{\$} \mathsf{PCG.Gen}(1\!\!1^\lambda) \\ \forall i \in \mathcal{C} : \quad R_i \leftarrow \mathsf{PCG.Eval}(\kappa_i, i) \\ (R_i)_{i\in\mathcal{H}} \xleftarrow{\$} \mathsf{RSample}\big(1\!\!1^\lambda, \mathcal{C}, (R_i)_{i\in\mathcal{C}}\big) \end{matrix} \right\}$$

Informally speaking, correctness states that the expansions of the seeds looks like the output of $\mathsf{CorGen}(1\!\!1^\lambda)$. Security instead asserts that the information leaked by a subset of seeds about the remaining outputs is no more than what can be extracted from their expansion.

2.6 Distributed Point Functions

Let N be a positive integer and $(D, +)$ be a group. A point function is a function $f : [N] \to D$, parametrised by $\omega \in [N]$ and $\beta \in D$, such that

$$f(x) = \begin{cases} \beta & \text{if } x = \omega, \\ 0 & \text{otherwise.} \end{cases}$$

We refer to ω as the *special position* of the point function, while $\beta = f(\omega)$ is called the *non-zero element.*

A distributed point function, or DPF, is a compact way of secret-sharing a point function without leaking its special position nor the non-zero element. Given such a share of the function, each party can locally compute an additive share of $f(x)$ at any point x.

It is not difficult to notice the strong similarities between PCGs and DPFs. In both cases, we are indeed compressing large outputs in n small objects, each of them addressed to a different party. Moreover, an evaluation algorithm permits to re-expand the information without any need for communication. For this reason, PCGs are often based on DPFs. This paper is no exception from this point of view.

Below, we give the formal syntax of a DPF [GI14, BGI15].

Definition 4 (Distributed Point Function). *Let N be a positive integer and let $(D, +)$ be a finite group. An n-party DPF with domain $[N]$ and codomain $(D, +)$ is a pair of PPT algorithms $(\mathsf{DPF.Gen}, \mathsf{DPF.Eval})$ with the following syntax.*

– $\mathsf{DPF.Gen}$ *takes as input the security parameter $1\!\!1^\lambda$ and a description of the point function f, specifically, the special position $\omega \in [N]$ and the non-zero element $\beta \in D$. The output is n keys $\kappa_0, \kappa_1, \dots, \kappa_{n-1}$.*
– $\mathsf{DPF.Eval}$ *takes as input a DPF key κ_i, index $i \in [n]$ and a value $x \in [N]$, outputting an additive share v_i of $f(x)$.*

A DPF should satisfy the following properties.

– ***Correctness.*** *For every special position $\omega \in [N]$, non-zero element $\beta \in D$ and element $x \in [N]$, we have that*

$$\mathbb{P}\left(\sum_{i\in[n]} v_i = f(x) \;\middle|\; \begin{matrix} (\kappa_i)_i \xleftarrow{\$} \mathsf{DPF.Gen}(1\!\!1^\lambda, \omega, \beta) \\ \forall i : \; v_i \leftarrow \mathsf{DPF.Eval}(\kappa_i, i, x) \end{matrix} \right) = 1.$$

– ***Security.*** *There exists a PPT simulator Sim such that, for every set of corrupted parties $\mathcal{C} \subsetneq [n]$, special position $\omega \in [N]$ and non-zero element $\beta \in D$, the output of $\mathsf{Sim}(1\!\!1^\lambda, \mathcal{C})$ is computationally indistinguishable from*

$$\left\{ (\kappa_i)_{i\in C} \;\middle|\; (\kappa_0, \kappa_1, \dots, \kappa_{n-1}) \xleftarrow{\$} \mathsf{DPF.Gen}(1\!\!1^\lambda, \omega, \beta) \right\}.$$

Observe that the correctness property states that the evaluation of the keys over the input $x \in [N]$ produces a secret-sharing of β when $x = \omega$, of 0 otherwise. Security instead guarantees that the knowledge of a proper subset of the DPF keys leaks no information about the special position ω and the non-zero element β.

To simplify notation, we write $\mathsf{DPF.FullEval}(\kappa_i, i)$ to denote the evaluation of the key κ_i over the whole domain $[N]$. This results in an additive share of the full N-dimensional vector with only one non-zero entry in the ω-th position. Abusing terminology slightly, we refer to a vector of this type as a unit vector.

State-of-the-Art We use DPFs for 2 parties only. In this setting, the most efficient construction is due to [BGI16], has $O(\lambda \cdot \log N)$ key size and supports any abelian group as the codomain D. Regarding computational efficiency, the dominant cost of a full-domain evaluation $\mathsf{DPF.FullEval}$ is around $2N$ evaluations of a length-doubling PRG.

Distributed Sum of Point Functions We will use an extension of DPFs, called distributed *sums of point functions*, as used previously [BCG+20]. A distributed sum of t point functions, or DSPF^t, is a way to secret-share a function obtained by adding t point functions. As with DPF, we express a DSPF^t by a pair of PPT algorithms $(\mathsf{DSPF}^t.\mathsf{Gen}, \mathsf{DSPF}^t.\mathsf{Eval})$, the first one of which takes as input the t special positions and the t non-zero elements describing the function f, outputting n small keys, one for each party. The latter can then be locally evaluated by the parties over an additional input $x \in [N]$, obtaining a secret-sharing of $f(x)$. This time, the evaluation over the whole domain leads to a secret-sharing of an N-dimensional t-sparse vector. Correctness and security of DSPF^{t}s are defined as for DPFs, with minimal adaptation to the increased number of special positions and non-zero elements.

To construct a DSPF^{t}s, we simply use a DPF instance for each of the t points of the DSPF^t. Each DSPF^t key is therefore composed of t DPF keys, and evaluated on input x by computing the sum of the evaluations of the t DPF keys over x.

3 Key Generation and Signing

3.1 ECDSA Tuples and the Ideal Functionality $\mathcal{F}^R_{\mathsf{Prep}}$

Our threshold ECDSA construction is based on the offline phase-online phase paradigm. In other words, the protocol is split into two parts: an input-independent preprocessing phase called the offline phase and a light online phase where we generate ECDSA signatures using the pregenerated data.

Each party P_i is associated with some key material, in particular, an additive share of the ECDSA private key $\mathsf{sk}_i \in \mathbb{F}_q$ and a BeDOZa style MAC key $\alpha_i \in \mathbb{F}_q$ [BDOZ11]. In the offline phase, we generate ECDSA tuples, i.e. each party P_i obtains a list of $\mathbb{F}_q$ elements of the form

$$\big(x_i, (M_{i,j}, K_{j,i})_{j \neq i}, y_i, d_i, z_i\big)$$

where $M_{i,j}$ and $K_{i,j}$ are BeDOZa style MACs over x_i w.r.t. the MAC key α_j[7], i.e.

$$M_{i,j} = K_{i,j} + \alpha_j \cdot x_i,$$

and the remaining terms satisfy the following conditions

$$\sum_{i \in [n]} \mathsf{sk}_i \cdot \sum_{i \in [n]} y_i = \sum_{i \in [n]} d_i, \qquad \sum_{i \in [n]} x_i \cdot \sum_{i \in [n]} y_i = \sum_{i \in [n]} z_i.$$

The offline phase takes also care of computing the ECDSA public key

$$\mathsf{PK} := \sum_{i \in [n]} \mathsf{sk}_i \cdot G.$$

[7] Observe that P_i knows only x_i and $M_{i,j}$, whereas P_j knows α_j and $K_{i,j}$.

Ring ECDSA Tuples and $\mathcal{F}_{\mathsf{Prep}}^{R}$ In our protocol, we actually deal with a generalisation of the ECDSA tuples to rings R in which the multiplication by $\mathbb{F}_q$ elements is well-defined. Specifically, we generate material with the same structure as ECDSA tuples, however using elements belonging to R rather than $\mathbb{F}_q$. The key material sk_i and α_i will instead remain in $\mathbb{F}_q$ as before. We call such tuples *ring ECDSA tuples*. Our protocol is going to generate them over a Module-LPN ring.

The functionality $\mathcal{F}_{\mathsf{Prep}}^{R}$ implemented by our preprocessing protocol is formalised in Figure 3. Essentially, the functionality allows the adversary to choose the BeDOZa MAC keys and the material of the corrupted parties. Once it received them, $\mathcal{F}_{\mathsf{Prep}}^{R}$ completes the ECDSA tuples sampling random elements. Finally, it outputs the produced material to the honest parties.

Notice that $\mathcal{F}_{\mathsf{Prep}}^{R}$ is parametrised by the ring R over which the ECDSA tuples are generated. In order to produce ECDSA signature, we are actually interested in the case $R = \mathbb{F}_q^N$ where the latter is equipped with the pointwise addition and multiplication. An ECDSA tuple over such ring corresponds indeed to N ECDSA tuples over $\mathbb{F}_q$. As we discussed in Section 2.4, $\mathbb{F}_q^N$ is isomorphic to a Module-LPN ring allowing us to base the preprocessing protocol on the corresponding assumption.

The Ideal Functionality $\mathcal{F}_{\mathsf{Prep}}^{R}$

Initialisation. On input Init from every party, the functionality samples the ECDSA private key $\mathsf{sk} \xleftarrow{\$} \mathbb{F}_q$ and waits for $(\alpha_i)_{i\in\mathcal{C}}$ from the corrupted parties. Then, it samples random $\alpha_i \xleftarrow{\$} \mathbb{F}$ for every $i \in \mathcal{H}$. The functionality sends $\mathsf{PK} \leftarrow \mathsf{sk} \cdot G$ to the adversary. Finally, upon receiving an OK from the adversary, it outputs (α_i, PK) to every honest party P_i.

ECDSA Tuple. On input Tuple from every party, the functionality samples $x, y \xleftarrow{\$} R$ and sets $z \leftarrow x \cdot y$. Then, it waits for a tuple $\left(\widehat{x}_i, (M_{i,j}, K_{j,i})_{j\in\mathcal{H}}, \widehat{y}_i, \widehat{d}_i, \widehat{z}_i\right)_{i\in\mathcal{C}}$ over R from the adversary. Finally, it samples a random tuple $(x_i, (M_{i,j}, K_{j,i})_{j\neq i}, y_i, d_i, z_i)$ over R for every $i \in \mathcal{H}$ subject to

$$\sum_{i\in\mathcal{H}} x_i + \sum_{i\in\mathcal{C}} \widehat{x}_i = x, \qquad \sum_{i\in\mathcal{H}} y_i + \sum_{i\in\mathcal{C}} \widehat{y}_i = y, \qquad \sum_{i\in\mathcal{H}} z_i + \sum_{i\in\mathcal{C}} \widehat{z}_i = z,$$
$$M_{i,j} = K_{i,j} + \alpha_j \cdot x_i, \qquad \sum_{i\in\mathcal{H}} d_i + \sum_{i\in\mathcal{C}} \widehat{d}_i = \mathsf{sk} \cdot y.$$

At the end, it outputs $(x_i, (M_{i,j}, K_{j,i})_{j\neq i}, y_i, d_i, z_i)$ to every honest party P_i.

Abort. On input Abort from the adversary, the functionality outputs $\perp$ to every honest party and halts.

Fig. 3: The offline phase functionality $\mathcal{F}_{\mathsf{Prep}}^{R}$

3.2 Distributed Key Generation and Signing in the $\mathcal{F}_{\mathsf{Prep}}^{R}$-Hybrid Model

Given our functionality $\mathcal{F}_{\mathsf{Prep}}^{R}$ for $R = \mathbb{F}_q^N$, the multiparty key generation is straight-forward. Specifically, the parties send the command Init to the ideal functionality $\mathcal{F}_{\mathsf{Prep}}^{R}$, to receive back the public key PK (and a MAC key α_i).

We proceed to present our multiparty signing protocol, where the parties jointly sign a message m. This protocol relies on an offline phase that uses $\mathcal{F}_{\mathsf{Prep}}^{R}$ and then requires the parties to have one round of interaction, in which r is revealed. Then, once the message to be signed becomes known, there is one more round of interaction where the signature is revealed.

In more details, in the offline step, the parties open $x \cdot G$ by having each party P_i sending $x_i \cdot G$ to all the other parties. To prevent cheating, we use the secret MAC key to ensure that each party used the x_i given by $\mathcal{F}_{\mathsf{Prep}}^{R}$. Specifically, each P_i sends also $M_{i,j} \cdot G$ to P_j. Holding $K_{j,i}$ and α_j, P_j can check that $M_{i,j} \cdot G = K_{i,j} \cdot G + \alpha_j \cdot (x_i \cdot G)$. If this check holds for all $j \neq i$, then party P_i can compute $x \cdot G \leftarrow \sum_{j=0}^{n-1} (x_j \cdot G)$

and take r to be the x-axis of the result. Once the parties receive a message m to sign, each party P_i sets $m' \leftarrow \mathsf{H}(m)$, computes $\tau_i \leftarrow y_i \cdot m' + r \cdot d_i$ and sends z_i and τ_i to all the other parties. Upon receiving τ_j and z_j for all $j \neq i$, each party P_i computes $\tau \leftarrow \sum_{j=0}^{n-1} \tau_j$ and $z \leftarrow \sum_{j=0}^{n-1} z_j$, and sets (r, s) where $s \leftarrow \frac{\tau}{z}$. Note that corrupted parties may send incorrect shares for z and τ. To detect this, each party runs the verification algorithm over $m, (r, s)$. If the verification succeeds, then the party outputs (r, s) as a valid signature. Otherwise, it aborts.

To see that the protocol is correct, observe that $z = x \cdot y$ and $d = \mathsf{sk} \cdot y$ and so

$$s = \tau \cdot z^{-1} = (y \cdot m' + r \cdot d) \cdot (x^{-1} \cdot y^{-1}) = x^{-1} \cdot (m' + r \cdot \mathsf{sk})$$

as required.

Communication In our protocol each party sends 2 messages in each of the two rounds to each other party. Thus, the total communication between each 2 parties is $2 \cdot (\log q + \log p + 1)$, where $\mathbb{F}_p$ is the field on which the elliptic curve is defined, plus the cost of $\mathcal{F}_{\mathsf{Prep}}^R$, which we realise in Section 4.

The Protocol Π_{ECDSA}

Let $\pi : \mathbb{G} \to \mathbb{F}_q$ be the function that maps a point $a \in \mathbb{G}$ into a_x mod q and let $\mathsf{H} : M \to \mathbb{F}_q$ be a hash function.

Distributed Key Generation The parties send the command Init to $\mathcal{F}_{\mathsf{Prep}}^R$, to receive back PK. In addition, each party P_i receives also a MAC key α_i.

Distributed Signing

- **Round 1 (Presigning):**
  1. If no ECDSA tuples are currently stored, the parties send the command Tuple to $\mathcal{F}_{\mathsf{Prep}}^R$ obtaining N fresh tuples.
  2. Each party P_i retrieves the shares of the next tuple $(x_i, (M_{i,j}, K_{j,i})_{j \neq i}, y_i, d_i, z_i)$.
  3. Each party P_i sends $x_i \cdot G$ and $M_{i,j} \cdot G$ to each P_j.
  4. Upon receiving $x_j \cdot G$ and $M_{j,i} \cdot G$ for each $j \neq i$, each party P_i checks that $M_{j,i} \cdot G = K_{j,i} \cdot G + \alpha_i \cdot (x_j \cdot G)$. If the equation holds for each $j \neq i$, then P_i computes $R \leftarrow \sum_{j=0}^{n-1} x_j \cdot G$ and sets $r \leftarrow \pi(R)$. Otherwise, it sends abort to the other parties and halts.
- **Round 2 (Signing):** Upon receiving a message m: each party P_i computes $m' \leftarrow \mathsf{H}(m)$ and $\tau_i \leftarrow y_i \cdot m' + r \cdot d_i$ and broadcasts z_i and τ_i.
- **Output:**
  1. Upon receiving τ_j and z_j for all $j \neq i$, each party P_i computes $\tau \leftarrow \sum_{j=0}^{n-1} \tau_j$ and $z \leftarrow \sum_{j=0}^{n-1} z_j$, and sets $s \leftarrow \tau / z$.
  2. Each party P_i run $\mathsf{Verify}_{\mathsf{PK}}(m, (r, s))$. If the result is 1, then P_i outputs (r, s). Otherwise, it outputs $\perp$ and halts.

Fig. 4: The distributed ECDSA protocol

3.3 Round Complexity and Relation to ECDSA Security

A natural question when using our protocol in practice, is when one should run the pre-signing phase. From one side, opening r in advance allows to have a *one-round* protocol once the message m to be signed becomes known. On the other hand, opening r before m is known does not match the standard unforgeability game for ECDSA security, and therefore one needs to be careful about opening potential vulnerabilities. The work of Canetti et al. [CGG+20] considers security of ECDSA with presignatures and shows that security of ECDSA is preserved when a constant number of nonces are available to the adversary. In particular, their reduction

shows an exponential security loss in the number of released nonces. Therefore, performing a large number of pre-signing operations before the message is known appears to be a bad idea. The work of [GS21] highlights that pre-signing requires to assume an extra property on the hash function than plain ECDSA (note that a random oracle would satisfy the assumption). They propose some countermeasures where the nonce is re-randomised after the message to be signed has been chosen. However, these countermeasures appear incompatible with any known preprocessing protocol for ECDSA. Thus, one can either use our protocol as a two-round protocol (where the pre-signing is performed after the message is chosen) if one is worried of the potential attack vector described by [GS21], or one can use our protocol as an *amortised* one-round protocol by revealing the r for signature $i+1$ in parallel with the computation of s for signature i.

3.4 Security Proof

Theorem 1. *The protocol Π_{ECDSA} UC-implements $\mathcal{F}_{\mathsf{ECDSA}}$ in the $\mathcal{F}_{\mathsf{Prep}}^{R}$-hybrid model with statistical error $\frac{1}{q}$, in the presence of malicious adversaries controlling up to $n-1$ parties.*

Proof. Let $\mathcal{S}$ be the ideal world adversary and let $\mathcal{A}$ be the real world adversary. To prove UC-security, we assume that an external environment gives $\mathcal{S}$ the commands to invoke each step of the execution and also controls $\mathcal{A}$. The simulation works as follows:

Key Generation Upon receiving the command (Init) from the external environment, $\mathcal{S}$ sends the command KeyGen to the trusted party computing $\mathcal{F}_{\mathsf{ECDSA}}$ to receive back the public key PK. Then, upon receiving $(\alpha_i)_{i\in\mathcal{C}}$ from $\mathcal{A}$, it sends PK to $\mathcal{A}$. Finally, upon receiving OK from $\mathcal{A}$, the simulator $\mathcal{S}$ stores $(\mathsf{PK}, (\alpha_i)_{i\in\mathcal{C}})$.

Presigning Upon receiving a command Presign(sid) from the external environment, S works as follows:

1. $\mathcal{S}$ sends Presign(sid) to $\mathcal{F}_{\mathsf{ECDSA}}$ to receive back r.
2. If new ECDSA tuples are needed, playing the role of $\mathcal{F}_{\mathsf{Prep}}^{R}$, $\mathcal{S}$ receives the tuples $\left(\widehat{x}_i, (M_{i,j}, K_{j,i})_{j\in\mathcal{H}}, \widehat{y}_i, \widehat{d}_i, \widehat{z}_i\right)_{i\in\mathcal{C}}$ from $\mathcal{A}$. If $\mathcal{A}$ sent abort, then $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{ECDSA}}$, simulates the honest parties aborting in the real execution and outputs whatever $\mathcal{A}$ outputs.
3. $\mathcal{S}$ chooses a point R in $\langle G\rangle$ such that $r = \pi(R)$. Note that there is more than one point on the curve that satisfies it. Finding such R is however efficient when the cofactor of $\langle G\rangle$ is small (as it usually is in ECDSA instantiations, including the Bitcoin curve).
4. $\mathcal{S}$ chooses a random $R_i \in \langle G\rangle$ for each $i \in \mathcal{H}$ (i.e., P_i is an honest party), under the constraint that $R = \sum_{i\in\mathcal{H}} R_i + \sum_{i\in\mathcal{C}} \widehat{x}_i \cdot G$. Finally, for each honest party P_j and corrupted party P_i, it computes $\overline{M}_{j,i} = K_{j,i}\cdot G + \alpha_i \cdot R_j$ (this is enabled since $\mathcal{S}$ received $K_{j,i}$ and α_i from $\mathcal{A}$).
5. $\mathcal{S}$ sends R_j and $\overline{M}_{j,i}$ for each honest P_j and corrupted P_i to $\mathcal{A}$.
6. Upon receiving $M_{i,j}\cdot G$ and $\widehat{x}_i \cdot G$ from $\mathcal{A}$ for each corrupted P_i and honest P_j, $\mathcal{S}$ checks that $\mathcal{A}$ sent the correct $M_{i,j}\cdot G$ and $\widehat{x}_i \cdot G$ to all parties. If not, it sends abort to $\mathcal{F}_{\mathsf{ECDSA}}$, simulates the honest parties aborting and outputs whatever $\mathcal{A}$ outputs. If the correct values were sent, then it sends continue to $\mathcal{F}_{\mathsf{ECDSA}}$.

Signing Upon receiving a command Sign(sid, m) from the external environment, $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ sends Sign(sid, m) to $\mathcal{F}_{\mathsf{ECDSA}}$, to receive back s.
2. $\mathcal{S}$ computes $m' \leftarrow \mathsf{H}(m)$. Then, for each corrupted P_i, it computes $\widehat{\tau}_i \leftarrow \widehat{y}_i \cdot m' + r\cdot \widehat{d}_i$. Then, $\mathcal{S}$ chooses a random $z_j \in \mathbb{F}_q$ for each $j \in \mathcal{H}$, sets $\tau \leftarrow s\cdot z$ where $z = \sum_{i\in\mathcal{H}} z_i + \sum_{i\in\mathcal{C}} \widehat{z}_i$ and chooses a random τ_j for each honest party P_j such that $\tau = \sum_{j\in\mathcal{H}} \tau_j + \sum_{i\in\mathcal{C}} \widehat{\tau}_i$.
3. $\mathcal{S}$ sends τ_j and z_j for each $j \in \mathcal{H}$ to $\mathcal{A}$, to receive back from $\mathcal{A}$ the messages τ'_i and z'_i sent by each $i \in \mathcal{C}$.
4. $\mathcal{S}$ computes
$$s' \leftarrow \frac{\sum_{j\in\mathcal{H}} \tau_j + \sum_{i\in\mathcal{C}} \tau'_i}{\sum_{j\in\mathcal{H}} z_j + \sum_{i\in\mathcal{C}} z'_i}$$
Then, $\mathcal{S}$ sends (r, s') to $\mathcal{F}_{\mathsf{ECDSA}}$.

5. Finally, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

It is straightforward that the simulation in the key generation is identical to a real execution in the key generation step. In the presigning step, the only difference between the simulation and the real execution is when the adversary sends incorrect values, but the MAC check in the exponent does not detect it. However, this event happens with probability $\frac{1}{q}$, which is allowed by the theorem.

We proceed to the signing phase. Observe that in the real execution, the view of the adversary consists of the values

$$R = x \cdot G \qquad z = x \cdot y \qquad \tau = y \cdot (m' + r \cdot \mathsf{sk})$$

where x and y are random. In contrast, in the simulation, the adversary's view is

$$R = x \cdot G \qquad z \qquad \tau = s \cdot z.$$

Where x and z are random. However, since in the real execution $s = \frac{\tau}{z}$, the distributions in the two executions are statistically close (they differ if and only if $x = 0$, this event occurs with probability at most $1/q$).

Observe that the pair (r, s') sent by the simulator to the functional is exactly what the honest parties would output assuming that the signature verifies. This concludes the proof.

□

4 Realizing $\mathcal{F}^R_{\mathsf{Prep}}$ - Silent Preprocessing

In this section, we present the major contribution of this paper, namely how to implement the functionality $\mathcal{F}^R_{\mathsf{Prep}}$ described in Figure 3, generating N ECDSA tuples with $O(\log N)$ communication complexity.

We will split the discussion into two parts. We start by presenting an efficient PCG for the generation of ECDSA tuples over a Module-LPN ring R and, secondly, we will describe the protocol Π^R_{Prep}, translating the PCG blueprint in an actual implementation of $\mathcal{F}^R_{\mathsf{Prep}}$.

We recall that $\mathbb{F}_q^N$, equipped with pointwise addition and multiplication, is isomorphic to a Module-LPN ring R (see Section 2.4). Furthermore, due to the linearity of the isomorphism ϕ, converting the ring ECDSA tuples over R to $\mathbb{F}_q^N$ does not require any communication. As a matter of fact, each party just needs to apply ϕ to its own shares. Finally, the isomorphism map is also efficiently computable using FFT. Rephrasing what we have said in simple words, we can instantiate Π^R_{Prep} over $\mathbb{F}_q^N$ without any problems.

4.1 An Efficient PCG for Ring ECDSA Tuples

In this section, we describe a PCG for ECDSA tuples over a Module-LPN ring R. Observe that the correlation we aim to produce is trivially reverse samplable as we can always efficiently complete the shares of the corrupted parties to an ECDSA tuple. The PCG we are going to present is the construction underlying the preprocessing protocol Π^R_{Prep} and the main reason why it achieves a so low communication complexity. A formal description of $\mathsf{PCG}_{\mathsf{ECDSA}}$ can be found in Figure 5. Observe that it uses a random oracle $\mathcal{O}$. We now sketch the main ideas at the base of the construction.

From the Ring R to the Vectorial Representation and its Compression In Section 2, we observed how DPFs permit to compress 2-party secret-sharings of large unit vectors. Consider now the Module-LPN assumption. Each element of the ring R can be represented as a polynomial of degree at most $N - 1$. Therefore, we can convert it into a N-dimensional vector over $\mathbb{F}_q$. When we are actually dealing with a monomial, the representation becomes a unit vector. Now, the distribution $\mathcal{HW}_t$ samples random t-sparse polynomials in R, so we can represent its outputs with sums of t unit vectors.

The Pseudorandom Correlation Generator $\mathsf{PCG}_{\mathsf{ECDSA}}$

Let $R := \mathbb{F}_q[X]/\big(F(X)\big)$, t and c be the parameters of the Module-LPN assumption. Denote the degree of $F(X)$ by N.

Gen. On input $\mathbb{1}^\lambda$, do the following:

1. Sample a BeDOZa style MAC key $\alpha_i \xleftarrow{\$} \mathbb{F}_q$ and ECDSA key shares $\mathsf{sk}_i \xleftarrow{\$} \mathbb{F}_q$, for every $i \in [n]$.
2. For every $i \in [n]$, $r \in [c]$, sample $\boldsymbol{\omega}_i^r, \boldsymbol{\eta}_i^r \xleftarrow{\$} [N]^t$ and $\boldsymbol{\beta}_i^r, \boldsymbol{\gamma}_i^r \xleftarrow{\$} \mathbb{F}_q^t$.
3. For every $i, j \in [n]$ with $i \neq j$, $r \in [c]$, compute

$$\Big(U_{i,j}^{r,0}, U_{i,j}^{r,1}\Big) \xleftarrow{\$} \mathsf{DSPF}_N^t.\mathsf{Gen}\Big(\mathbb{1}^\lambda,\ \boldsymbol{\omega}_i^r,\ \alpha_j \cdot \boldsymbol{\beta}_i^r\Big), \qquad \Big(V_{i,j}^{r,0}, V_{i,j}^{r,1}\Big) \xleftarrow{\$} \mathsf{DSPF}_N^t.\mathsf{Gen}\Big(\mathbb{1}^\lambda,\ \boldsymbol{\eta}_i^r,\ \mathsf{sk}_j \cdot \boldsymbol{\gamma}_i^r\Big).$$

4. For every $i, j \in [n]$ with $i \neq j$, $r, s \in [c]$, compute

$$\Big(C_{i,j}^{r,s,h}\Big)_{h\in[2]} \xleftarrow{\$} \mathsf{DSPF}_{2N}^{t^2}.\mathsf{Gen}\Big(\mathbb{1}^\lambda,\ \boldsymbol{\omega}_i^r \boxplus \boldsymbol{\eta}_j^s,\ \boldsymbol{\beta}_i^r \otimes \boldsymbol{\gamma}_j^s\Big).$$

5. For every $i \in [n]$, output the seed

$$\kappa_i \leftarrow \left(\alpha_i, \mathsf{sk}_i, \big(\boldsymbol{\omega}_i^r, \boldsymbol{\beta}_i^r\big)_{r\in[c]}, \big(\boldsymbol{\eta}_i^r, \boldsymbol{\gamma}_i^r\big)_{r\in[c]}, \Big(U_{i,j}^{r,0}, U_{j,i}^{r,1}\Big)_{\substack{j\neq i\\ r\in[c]}}, \Big(V_{i,j}^{r,0}, V_{j,i}^{r,1}\Big)_{\substack{j\neq i\\ r\in[c]}}, \Big(C_{i,j}^{r,s,0}, C_{j,i}^{r,s,1}\Big)_{\substack{j\neq i\\ r,s\in[c]}}\right)$$

Eval. On input the seed κ_i, do the following:

1. For every $r \in [c]$, define the two polynomials

$$u_i^r(X) := \sum_{l\in[t]} \boldsymbol{\beta}_i^r[l] \cdot X^{\boldsymbol{\omega}_i^r[l]}, \quad v_i^r(X) := \sum_{l\in[t]} \boldsymbol{\gamma}_i^r[l] \cdot X^{\boldsymbol{\eta}_i^r[l]}$$

2. For every $r \in [c]$, compute

$$\widetilde{M}_{i,j}^r \leftarrow \mathsf{DSPF}_N^t.\mathsf{FullEval}(U_{i,j}^{r,0}) \qquad \widetilde{K}_{j,i}^r \leftarrow \mathsf{DSPF}_N^t.\mathsf{FullEval}(U_{j,i}^{r,1})$$
$$\widetilde{v}_i^r \leftarrow \mathsf{sk}_i \cdot v_i^r + \sum_{j\neq i} \big(\mathsf{DSPF}_N^t.\mathsf{FullEval}(V_{i,j}^{r,0}) + \mathsf{DSPF}_N^t.\mathsf{FullEval}(V_{j,i}^{r,1})\big)$$

   (viewing outputs of $\mathsf{FullEval}$ as degree $N-1$ polynomials over $\mathbb{F}_q$)
3. For every $r, s \in [c]$, compute

$$w_i^{r,s} \leftarrow u_i^r \cdot v_i^s + \sum_{j\neq i} \Big(\mathsf{DSPF}_{2N}^{t^2}.\mathsf{FullEval}(C_{i,j}^{r,s,0}) + \mathsf{DSPF}_{2N}^{t^2}.\mathsf{FullEval}(C_{j,i}^{r,s,1})\Big)$$

4. Define the vectors of polynomials $\boldsymbol{u}_i := (u_i^0, \ldots, u_i^{c-1})$, $\boldsymbol{v}_i := (v_i^0, \ldots, v_i^{c-1})$, similarly for $\widetilde{\boldsymbol{M}}_{i,j}, \widetilde{\boldsymbol{K}}_{j,i}, \widetilde{\boldsymbol{v}}_i$. Let $\boldsymbol{w}_i := (w_i^{0,0}, \ldots, w_i^{c-1,0}, w_i^{0,1}, \ldots, w_i^{c-1,1}, \ldots, w_i^{c-1,c-1})$.
5. For a random $\boldsymbol{a} \in R^c$ with $a_{c-1} = 1$ provided by the random oracle $\mathcal{O}$, compute the final shares

$$x_i \leftarrow \langle \boldsymbol{a}, \boldsymbol{u}_i \rangle, \qquad y_i \leftarrow \langle \boldsymbol{a}, \boldsymbol{v}_i \rangle, \qquad z_i \leftarrow \langle \boldsymbol{a} \otimes \boldsymbol{a}, \boldsymbol{w}_i \rangle,$$
$$M_{i,j} \leftarrow \Big\langle \boldsymbol{a}, \widetilde{\boldsymbol{M}}_{i,j} \Big\rangle, \qquad K_{j,i} \leftarrow -\Big\langle \boldsymbol{a}, \widetilde{\boldsymbol{K}}_{j,i} \Big\rangle, \qquad d_i \leftarrow \langle \boldsymbol{a}, \widetilde{\boldsymbol{v}}_i \rangle$$

   in $\mathbb{F}_q[X]/\big(F(X)\big)$. Output $(\alpha_i, \mathsf{sk}_i, x_i, (M_{i,j}, K_{j,i})_{j\neq i}, y_i, z_i, d_i)$.

Fig. 5: The PCG for ring ECDSA tuples.

Compressing the Terms x_i and y_i When we look at a ring ECDSA tuple, we observe that the shares x_i and y_i of the i-th party are random elements in R. In order to compress them, we rely on the Module-LPN assumption: for every $r \in [c]$, each party P_i generates two t-sparse polynomials in R

$$u_i^r(X) := \sum_{l \in [t]} \beta_i^r[l] \cdot X^{\omega_i^r[l]}, \qquad v_i^r(X) := \sum_{l \in [t]} \gamma_i^r[l] \cdot X^{\eta_i^r[l]}$$

by sampling the non-zero coefficients $(\beta_i^r[l])_{l \in [t]}$ and $(\gamma_i^r[l])_{l \in [t]}$ and the degree of the associated monomials $(\omega_i^r[l])_{l \in [t]}$ and $(\eta_i^r[l])_{l \in [t]}$. During the evaluation, using a random oracle, the parties will obtain $c-1$ random elements $a_0, a_1, \ldots, a_{c-2}$ in R. The values of x_i and y_i will be computed as

$$x_i = \langle \boldsymbol{a}, \boldsymbol{u_i} \rangle = \sum_{j=0}^{c-2} a_j \cdot u_i^j + u_i^{c-1}, \qquad y_i = \langle \boldsymbol{a}, \boldsymbol{v_i} \rangle = \sum_{j=0}^{c-2} a_j \cdot v_i^j + v_i^{c-1}.$$

By the R^c-LPN$_t$ assumption, x_i and y_i are indistinguishable from random.

Compressing the BeDOZa Style MACs It remains to explain how to derive the remaining parts of the ring ECDSA tuple. We start by observing that, for every $i \neq j$, $M_{i,j}$ and $K_{i,j}$ are random elements satisfying

$$M_{i,j} - K_{i,j} = \alpha_j \cdot x_i. \tag{1}$$

We recall that $\alpha_j \in \mathbb{F}_q$, so we have that

$$\alpha_j \cdot x_i = \alpha_j \cdot \langle \boldsymbol{a}, \boldsymbol{u_i} \rangle = \langle \boldsymbol{a}, \alpha_j \cdot \boldsymbol{u_i} \rangle.$$

Now, if we secret-share $\alpha_j \cdot u_i^r = \widetilde{M}_{i,j}^r + \widetilde{K}_{i,j}^r$ between P_i and P_j, we leak no additional information to the parties, while obtaining

$$\alpha_j \cdot x_i = \langle \boldsymbol{a}, \widetilde{\boldsymbol{M}}_{\boldsymbol{i,j}} + \widetilde{\boldsymbol{K}}_{\boldsymbol{i,j}} \rangle = \langle \boldsymbol{a}, \widetilde{\boldsymbol{M}}_{\boldsymbol{i,j}} \rangle + \langle \boldsymbol{a}, \widetilde{\boldsymbol{K}}_{\boldsymbol{i,j}} \rangle.$$

In other words, the values $M_{i,j} := \langle \boldsymbol{a}, \widetilde{\boldsymbol{M}}_{\boldsymbol{i,j}} \rangle$ and $K_{i,j} = -\langle \boldsymbol{a}, \widetilde{\boldsymbol{K}}_{\boldsymbol{i,j}} \rangle$ satisfy (1). Finally, observe that, for every $r \in [c]$, $\alpha_j \cdot u_i^r$ is a t-sparse polynomial, so we can compress a 2-party secret-sharing between P_i and P_j using t DPF keys. In total, this procedure requires $c \cdot t \cdot n(n-1)$ of them.

Compressing the Term d_i Once we understood how to obtain compressed BeDOZa style MACs, it is easy to generalise the ideas for the terms $(d_i)_{i \in [n]}$. As a matter of fact, the following relation holds

$$\sum_{i \in [n]} d_i = \sum_{j \in [n]} \mathsf{sk}_j \cdot \sum_{i \in [n]} y_i = \sum_{i \in [n]} (\mathsf{sk}_i \cdot y_i) + \sum_{i \neq j} (\mathsf{sk}_j \cdot y_i).$$

Since sk_j belongs to $\mathbb{F}_q$, we can apply the techniques described in the previous paragraph to compress a secret-sharing of $\mathsf{sk}_j \cdot y_i = d_{i,j}^0 + d_{i,j}^1$ between P_i and P_j, while leaking no additional information to the parties. Specifically, we observe that $\mathsf{sk}_j \cdot y_i = \langle \boldsymbol{a}, \mathsf{sk}_j \cdot \boldsymbol{v_i} \rangle$. Moreover, $\mathsf{sk}_j \cdot v_i^r(X)$ is a t-sparse polynomial for every $r \in [c]$, allowing us to compress a two party secret-sharing between P_i and P_j using t DPF keys. Once $d_{i,j}^0$ and $d_{i,j}^1$ are available for every pair (i,j) with $i \neq j$, each party P_i can set

$$d_i \leftarrow \mathsf{sk}_i \cdot y_i + \sum_{j \neq i} (d_{i,j}^0 + d_{j,i}^1).$$

In total, this procedure requires $c \cdot t \cdot n(n-1)$ DPF keys.

Compressing the Term z_i This is probably the most complex part of the construction but the main ideas are the same as before. We observe that the terms $(z_i)_{i\in[n]}$ are random values satisfying

$$\sum_{i\in[n]} z_i = \sum_{i\in[n]} x_i \cdot \sum_{j\in[n]} y_j = \sum_{i\in[n]} (x_i \cdot y_i) + \sum_{i\neq j} (x_i \cdot y_j).$$

Again, our plan is to compress a secret-sharing of $x_i \cdot y_j$ between P_i and P_j without leaking any additional information to the parties. This time, however, the major issue is that both x_i and y_j belong to the ring R. By extending our analysis, we notice that

$$x_i \cdot y_j = \Big(\sum_{r\in[c]} a_r \cdot u_i^r\Big) \cdot \Big(\sum_{s\in[c]} a_s \cdot v_j^s\Big) = \sum_{r,s\in[c]} (a_r \cdot a_s) \cdot (u_i^r \cdot v_j^s).$$

The polynomials u_i^r and v_j^s are both t-sparse of degree at most $N-1$, so their product over $\mathbb{F}_q[X]$ consists in a t^2-sparse polynomial of degree at most $2N-2$. We can therefore compress a secret-sharing of such product between P_i and P_j using t^2 DPF keys. The linearity of the reduction modulo $F(X)$ allows then projecting such secret-sharing over R.

The generation of the terms $(z_i)_{i\in[n]}$ in the PCG follows exactly the blueprint sketched above, using outer products $\otimes$ and outer sums $\boxplus$ to compress the notation. In total, we need $c^2 \cdot t^2 \cdot n(n-1)$ DPF keys.

Theorem 2. *If the R^c-LPN$_t$ problem is hard and* DSPF *is a secure distributed sum of point functions,* $\mathsf{PCG}_{\mathsf{ECDSA}}$ *is a correct and secure PCG for ring ECDSA tuples over R in the random oracle model. Moreover, if we instantiate* DSPF *with the 2-party DPF of [BGI16], the size of the seeds is*

$$2\log q + 2c \cdot t \cdot (\log q + \log N) + 4c \cdot t \cdot (n-1) \cdot (\lambda \cdot \log N + \log q) + 2c^2 \cdot t^2 \cdot (n-1) \cdot (\lambda \cdot \log 2N + \log q).$$

Proof. Define $\mathsf{sk} := \sum_{i\in[n]} \mathsf{sk}_i$.

Claim 2.1 *The following relations hold*

$$\forall i \neq j: \quad M_{i,j} = K_{i,j} + \alpha_j \cdot x_i, \qquad \sum_{i\in[n]} d_i = \mathsf{sk} \cdot \sum_{i\in[n]} y_i.$$

Proof (of the claim). By the correctness of the DSPF, we know that for every $i \neq j$ and $r \in [c]$

$$\widetilde{M}_{i,j}^r(X) + \widetilde{K}_{i,j}^r(X) = \sum_{l\in[t]} \alpha_j \cdot \beta_i^r[l] \cdot X^{\omega_i^r[l]} = \alpha_j \cdot u_i^r(X).$$

Moreover, for every $r \in [c]$,

$$\begin{aligned}\sum_{i\in[n]} \widetilde{v}_i^r(X) &= \sum_{i\in[n]} \mathsf{sk}_i \cdot v_i^r(X) + \sum_{i\neq j} \sum_{l\in[t]} \mathsf{sk}_j \cdot \gamma_i^r[l] \cdot X^{\eta_i^r[l]} = \\ &= \sum_{i\in[n]} \mathsf{sk}_i \cdot v_i^r(X) + \sum_{i\neq j} \mathsf{sk}_j \cdot v_i^r(X) = \\ &= \sum_{i,j\in[n]} \mathsf{sk}_j \cdot v_i^r(X) = \mathsf{sk} \cdot \sum_{i\in[n]} v_i^r(X).\end{aligned}$$

As a consequence, we understand that

$$M_{i,j} = \langle \boldsymbol{a}, \widetilde{\boldsymbol{M}}_{\boldsymbol{i},\boldsymbol{j}} \rangle = \langle \boldsymbol{a}, -\widetilde{\boldsymbol{K}}_{\boldsymbol{i},\boldsymbol{j}} + \alpha_j \cdot \boldsymbol{u_i} \rangle = -\langle \boldsymbol{a}, \widetilde{\boldsymbol{K}}_{\boldsymbol{i},\boldsymbol{j}} \rangle + \alpha_j \cdot \langle \boldsymbol{a}, \boldsymbol{u_i} \rangle = K_{i,j} + \alpha_j \cdot x_i.$$

Moreover,

$$\sum_{i\in[n]} d_i = \sum_{i\in[n]} \langle \boldsymbol{a}, \widetilde{\boldsymbol{v}}_{\boldsymbol{i}} \rangle = \langle \boldsymbol{a}, \sum_{i\in[n]} \widetilde{\boldsymbol{v}}_{\boldsymbol{i}} \rangle = \langle \boldsymbol{a}, \mathsf{sk} \cdot \sum_{i\in[n]} \boldsymbol{v_i} \rangle = \mathsf{sk} \cdot \sum_{i\in[n]} \langle \boldsymbol{a}, \boldsymbol{v_i} \rangle = \mathsf{sk} \cdot \sum_{i\in[n]} y_i.$$

■

Claim 2.2 *The following relation holds*

$$\sum_{i\in[n]} z_i = \Big(\sum_{i\in[n]} x_i\Big)\cdot\Big(\sum_{j\in[n]} y_j\Big).$$

Proof (of the claim). By the correctness of the DSPF, we know that for every $r, s \in [c]$, we have

$$\begin{aligned}
\sum_{i\in[n]} w_i^{r,s} &= \sum_{i\in[n]} u_i^r(X)\cdot v_i^s(X) + \sum_{i\neq j}\sum_{l,h\in[t]} \beta_i^r[l]\cdot\gamma_j^s[h]\cdot X^{\omega_i^r[l]+\eta_j^s[h]} = \\
&= \sum_{i\in[n]} u_i^r(X)\cdot v_i^s(X) + \sum_{i\neq j}\Big(\sum_{l\in[t]} \beta_i^r[l]\cdot X^{\omega_i^r[l]}\Big)\cdot\Big(\sum_{h\in[t]} \gamma_j^s[h]\cdot X^{\eta_j^s[h]}\Big) = \\
&= \sum_{i\in[n]} u_i^r(X)\cdot v_i^s(X) + \sum_{i\neq j} u_i^r(X)\cdot v_j^s(X) = \\
&= \Big(\sum_{i\in[n]} u_i^r(X)\Big)\cdot\Big(\sum_{j\in[n]} v_j^s(X)\Big)
\end{aligned}$$

As a consequence, we have that

$$\begin{aligned}
\sum_{i\in[n]} z_i &= \sum_{i\in[n]} \langle \boldsymbol{a}\otimes\boldsymbol{a}, \boldsymbol{w_i}\rangle = \langle \boldsymbol{a}\otimes\boldsymbol{a}, \sum_{i\in[n]} \boldsymbol{w_i}\rangle = \\
&= \sum_{r,s\in[c]} a_r\cdot a_s\cdot\Big(\sum_{i\in[n]} u_i^r(X)\Big)\cdot\Big(\sum_{j\in[n]} v_j^s(X)\Big) = \\
&= \Big(\sum_{r\in[c]}\sum_{i\in[n]} a_r\cdot u_i^r(X)\Big)\cdot\Big(\sum_{s\in[c]}\sum_{j\in[n]} a_s\cdot v_j^s(X)\Big) = \\
&= \Big(\sum_{i\in[n]} \langle \boldsymbol{a}, \boldsymbol{u_i}\rangle\Big)\cdot\Big(\sum_{j\in[n]} \langle \boldsymbol{a}, \boldsymbol{v_j}\rangle\Big) = \\
&= \Big(\sum_{i\in[n]} x_i\Big)\cdot\Big(\sum_{j\in[n]} y_j\Big)
\end{aligned}$$

■

Claim 2.3 *Let $S \subseteq [n]$ be a non-empty subset of parties. No PPT adversary provided with the PCG seeds of the parties not in S can distinguish between the real $(\alpha_i, \mathsf{sk}_i, x_i, (M_{i,j}, K_{j,i})_{j\neq i}, y_i, z_i, d_i)_{i\in S}$ and the corresponding tuples in which $(M_{i,j}, K_{j,i})_{j\neq i}$, z_i and d_i are substituted with random elements $(M'_{i,j}, K'_{j,i})_{j\neq i}$, z'_i and d'_i subject to*

$$M'_{i,j} = K'_{i,j} + \alpha_j\cdot x_i, \qquad \sum_{i\in S} d'_i + \sum_{i\notin S} d_i = \mathsf{sk}\cdot\sum_{i\in[n]} y_i,$$
$$\sum_{i\in S} z'_i + \sum_{i\notin S} z_i = \Big(\sum_{i\in[n]} x_i\Big)\cdot\Big(\sum_{j\in[n]} y_j\Big).$$

Proof (of the claim). By Claims 2.1 and 2.2, we know that

$$M_{i,j} = K_{i,j} + \alpha_j\cdot x_i, \qquad \sum_{i\in[n]} d_i = \mathsf{sk}\cdot\sum_{i\in[n]} y_i,$$
$$\sum_{i\in[n]} z_i = \Big(\sum_{i\in[n]} x_i\Big)\cdot\Big(\sum_{j\in[n]} y_j\Big).$$

These relations hold in both the analysed cases. It remains to show that the the values $(M_{i,j}, K_{j,i})_{j\neq i}$, z_i and d_i for $i \in S$ are pseudorandom elements satisfying the above conditions.

In [BGI15, Theorem 9], it was shown that the outputs of $\mathsf{FullEval}$ on any single party's DPF key are pseudorandom, when the key remains hidden. As a result, it holds that for every $i, j \in S$ with $i \neq j$, $\widetilde{K}_{i,j}^{c-1}(X)$ is pseudorandom in R. Since $a_{c-1} = 1$, we conclude that $K_{i,j}$ is indistinguishable from a random element in R.

Take now $\iota \in S$. Following the same argument as above, we understand that the values $(\widetilde{v}_i^{c-1}, w_i^{c-1,c-1})_{i \in S, i \neq \iota}$ are pseudorandom in R. Again, since $a_{c-1} = 1$, we conclude that the elements $(d_i, z_i)_{i \in S, i \neq \iota}$ are indistinguishable from $2|S| - 2$ random elements in R. That ends the proof of the claim. ■

We are now ready to prove both correctness and security in one go. We do it by strengthening Claim 2.3. Specifically, for every subset of party $S \subseteq [n]$, we show that no PPT adversary provided with the PCG seeds of the parties not in S can distinguish between the real $(\alpha_i, \mathsf{sk}_i, x_i, (M_{i,j}, K_{j,i})_{j \neq i}, y_i, z_i, d_i)_{i \in S}$ and the corresponding tuples in which x_i and y_i are substituted with random elements in R and $(M_{i,j}, K_{j,i})_{j \neq i}$, z_i and d_i are substituted with random elements $(M'_{i,j}, K'_{j,i})_{j \neq i}$, z'_i and d'_i subject to

$$M'_{i,j} = K'_{i,j} + \alpha_j \cdot x_i, \tag{2}$$

$$\sum_{i \in S} d'_i + \sum_{i \notin S} d_i = \mathsf{sk} \cdot \sum_{i \in [n]} y_i, \tag{3}$$

$$\sum_{i \in S} z'_i + \sum_{i \notin S} z_i = \Big(\sum_{i \in [n]} x_i\Big) \cdot \Big(\sum_{j \in [n]} y_j\Big). \tag{4}$$

We achieve this goal by a series of $2|S|+2$ hybrids. In the initial stage, we provide the adversary with the PCG seeds of the parties not in S, the original $(\alpha_i, \mathsf{sk}_i, x_i, y_i)_{i \in S}$ and the randomly sampled $(M'_{i,j}, K'_{j,i})_{j \neq i}$, z'_i and d'_i. Observe that the adversary cannot distinguish this stage from the situation in which it receives the real outputs of the PCG. This is due to Claim 2.3.

In the subsequent stage, for every $i \notin S$ and $j \in S$, we substitute the DSPF keys $(U_{i,j}^{r,0}, U_{j,i}^{r,1}, V_{i,j}^{r,0}, V_{j,i}^{r,1})_{r \in [c]}$ and $(C_{i,j}^{r,s,0}, C_{j,i}^{r,s,1})_{r,s \in [c]}$ in the PCG seeds of the parties not in S with keys generated using Sim. This hybrid is indistinguishable from the previous one, due to the security of the 2-party DSPF.

We now describe the subsequent stages. Let j be the index of the i-th party in S. The $2i$-th hybrid is obtained from the $(2i-1)$-th one, by substituting x_j with a random element in R before sampling the fake $(M'_{i,j}, K'_{j,i})_{j \neq i}$, z'_i and d'_i. In a similar way, the $(2i+1)$-th hybrid is obtained from the $2i$-th one, by substituting y_j with a random element in R.

We show that any adversary $\mathcal{A}$ distinguishing between two consecutive hybrids can be converted into a successful Module-LPN attacker $\mathcal{A}'$. Without loss of generality, we can assume that $\mathcal{A}$ distinguishes between the $(2l-1)$-th and the $2l$-th hybrid. Let ι be the index of the l-th party in S.

Upon activation $\mathcal{A}'$ generates the PCG seeds of the parties not in S sampling random key material, random special positions and non-zero elements and simulating the DSPF keys using Sim whenever the other key is addressed to a party in S. Then, $\mathcal{A}'$ samples random sk_i and α_i in $\mathbb{F}_q$ for every $i \in S$ and random x_i and y_i in R for every $i \in S$ with $i < \iota$. Moreover, it waits for $(\boldsymbol{a}, x_\iota)$ from its challenger and, for every $r \in [c]$ and $i, j \in S$ with $i > \iota$ and $j \geq \iota$, samples random t-sparse polynomials $u_i^r(X), v_j^r(X) \xleftarrow{\$} \mathcal{HW}_t$. As a last step, it computes $x_i \leftarrow \langle \boldsymbol{a}, \boldsymbol{u_i} \rangle$ and $y_j \leftarrow \langle \boldsymbol{a}, \boldsymbol{v_j} \rangle$ for every $i, j \in S$ with $i > \iota$ and $j \geq \iota$, generates the values $(M'_{i,j}, K'_{j,i})_{j \neq i}$, z'_i and d'_i for every $i \in S$ sampling them randomly subject to (2), (3) and (4) and provides $\mathcal{A}$ with the PCG seeds of the parties not in S, $\boldsymbol{a}$ and the tuples $(\alpha_i, \mathsf{sk}_i, x_i, (M'_{i,j}, K'_{j,i})_{j \neq i}, y_i, z'_i, d'_i)_{i \in S}$. At the end, $\mathcal{A}'$ outputs the same bit as $\mathcal{A}$.

Observe that when the Module-LPN challenger generates x_ι using the uniform distribution, the view of $\mathcal{A}$ is identical to the $2l$-th hybrid. If instead x_ι is generated using t-sparse polynomials, the view of $\mathcal{A}$ is perfectly indistinguishable from the $(2l-1)$-th hybrid. Hence, if $\mathcal{A}$ wins with non-negligible advantage, $\mathcal{A}'$ breaks the Module-LPN assumption, reaching a contradiction. In a totally analogous way, we are able to show that no PPT adversary can distinguish between the $2l$-th and the $(2l+1)$-th hybrid.

Observe that in the last hybrid of the sequence, the values $(x_i, y_i)_{i \in S}$ are all sampled randomly in R, however, the DSPF keys in the seeds of the parties not in S are still generated using Sim. In the last stage, we revert to the original DSPF keys. Indistinguishability is guaranteed by the DSPF security.

The Ideal Functionality $\mathcal{F}_{\mathsf{MPC}}$

In addition to the usual operations (i.e. additions, multiplications, inputs and outputs over $\mathbb{F}_q$ and $\mathbb{F}_2$), the functionality features the following procedures.

From-$\mathbb{F}_q$-to-$\mathbb{G}$. On input $\mathsf{To\text{-}G}([[\mathsf{sk}]])$, the functionality computes $\mathsf{PK} \leftarrow \mathsf{sk} \cdot G$ and sends it to the adversary, waiting for a reply. If the answer is OK, the functionality outputs PK to every honest party, otherwise, it aborts.

2-DPF. On input $\mathsf{2\text{-}DPF}(N, [[\omega]]_2, [[\beta]], \sigma_1, \sigma_2)$ from every party, where σ_1 and σ_2 are different indexes in $[n]$, N is a power of 2, ω is the bit representation of an integer in $[N]$ and β belongs to $\mathbb{F}_q$, the functionality does the following.

- If P_{σ_1} and P_{σ_2} are both corrupted, it sends β and ω to the adversary.
- If $\beta = 0$, it sends Zero to the adversary. If the reply is OK, the functionality outputs Zero to the honest parties[a].
- If one party P_σ among P_{σ_1} and P_{σ_2} is corrupted, it waits for the adversary to send $\boldsymbol{y_\sigma}$ in $\mathbb{F}_q^N$. Moreover, it waits for a set $I \subseteq [N]$. If $\omega \notin I$, it aborts. Otherwise, denoting by θ the index of the honest party among P_{σ_1} and P_{σ_2}, it outputs to P_θ

$$\boldsymbol{y_\theta} \leftarrow (\underbrace{0, 0, \ldots, \overbrace{0}^{\omega}}_{N}, \beta, 0, 0, \ldots, 0) - \boldsymbol{y_\sigma}.$$

- If P_{σ_1} and P_{σ_2} are both honest, it samples $\boldsymbol{y_1}$ uniformly in $\mathbb{F}_q^N$ and computes

$$\boldsymbol{y_2} \leftarrow (\underbrace{0, 0, \ldots, \overbrace{0}^{\omega}}_{N}, \beta, 0, 0, \ldots, 0) - \boldsymbol{y_1}.$$

  Finally, it outputs $\boldsymbol{y_i}$ to P_{σ_i} for every $i \in \{1, 2\}$.

[a] The $\mathsf{2\text{-}DPF}$ protocol reveals the multiplication of β by a random element in $\mathbb{F}_q^\times$. The operation leaks no information except whether $\beta = 0$ or not.

Fig. 6: The MPC functionality

This ends the proof of correctness and security of the PCG. The second part of the theorem is a mere computation, which is easily checkable. □

4.2 The Preprocessing Protocol

We start by assuming the existence of a random oracle $\mathcal{O}$ and a generic MPC functionality $\mathcal{F}_{\mathsf{MPC}}$, which is formalised in Figure 6. Such resource can be implemented in several ways, for instance, using Tiny-OT [NNOB12] (for bit operations) SPDZ [BDOZ11,DPSZ12] (for field operations) and the induced multiparty computation protocol over elliptic curves [DOK+20]. A protocol implementing $\mathsf{2\text{-}DPF}$ with $O(\lambda \cdot \log N)$ communication was instead presented in [BCG+20]. Sometimes, we use $\mathcal{F}_{\mathsf{MPC}}$ to perform additions between Module-LPN special positions, although integer operations are not supported by the functionality. In such case, we assume that the special positions are stored bit by bit, and therefore, we are able to compute sums using operations over $\mathbb{F}_2$. In order to make our presentation clearer, we will use the symbols $[[\cdot]]$ and $[[\cdot]]_2$ to denote elements stored by $\mathcal{F}_{\mathsf{MPC}}$ over $\mathbb{F}_q$ and $\mathbb{F}_2$ respectively.

We are now ready to describe the preprocessing protocol $\Pi^{\mathsf{R}}_{\mathsf{Prep}}$. A formal description is available in Figure 7 and 8. The construction closely follows the blueprint outlined by $\mathsf{PCG}_{\mathsf{ECDSA}}$, performing however the operations in a distributed manner and merging the seed generation and evaluation phases indissolubly[8]. In order to ease the notation, we will denote by $\mathsf{2\text{-}DSPF}$ the multiparty procedure implementing the distributed sum of point functions, although it is not featured in $\mathcal{F}_{\mathsf{MPC}}$. The operation will take as input the dimension

[8] The reason why this is unavoidable is that the DPF protocol of [BCG+20] does not permit to perform the DPF key generation and evaluation separately.

of the resulting vector N, the t special positions, the corresponding t non-zero elements and the indexes i and j of the two parties among which the output is secret-shared. Observe that the output is not a series of DSPF keys but their full evaluation. The instruction has to be regarded as a shorthand for t executions of $\mathsf{2\text{-}DPF}$ among the same parties P_i and P_j, outputting the sum of the resulting t secret-shared unit vectors and allowing the corresponding influence and leakage. In Π^R_{Prep}, the latter will be absorbed by the hardness of the Module-LPN assumption and therefore, it will not constitute a problem for security.

We now proceed by outlining the operations of Π^R_{Prep}.

During the initialisation every party P_i samples the key material sk_i and α_i, inputting their values in $\mathcal{F}_{\mathsf{MPC}}$. Using the latter, the parties compute and output the ECDSA public key $\mathsf{PK} = \sum_{i\in[n]} \mathsf{sk}_i \cdot G$.

In order to generate a ring ECDSA tuple, each party P_i starts by sampling, for every $r \in [c]$, the t special positions and non-zero elements describing $u^r_i(X)$ and $v^r_i(X)$. The sampled values are input in the $\mathcal{F}_{\mathsf{MPC}}$ functionality. Later on, using $\mathsf{2\text{-}DSPF}$ as in $\mathsf{PCG}_{\mathsf{ECDSA}}$, each party P_i computes $\widetilde{\boldsymbol{v}}_{\boldsymbol{i}}$, $\widetilde{\boldsymbol{w}}_{\boldsymbol{i}}$ and, for every $j \neq i$, $\widetilde{\boldsymbol{M}}_{\boldsymbol{i},\boldsymbol{j}}$ and $\widetilde{\boldsymbol{K}}_{\boldsymbol{j},\boldsymbol{i}}$. Finally, after sampling $\boldsymbol{a}$ using the random oracle $\mathcal{O}$, each party can terminate the evaluation of the PCG seeds obtaining their share of the ring ECDSA tuple.

Theorem 3. *Let $F(X)$ be a degree-N polynomial over the prime field $\mathbb{F}_q$ and let $t, c \in \mathbb{N}$. Define the ring $R := \mathbb{F}_q[X]/\big(F(X)\big)$. If the R^c-LPN$_t$ problem with static leakage is hard, then the protocol Π^R_{Prep} UC-implements $\mathcal{F}^R_{\mathsf{Prep}}$ in the $\mathcal{F}_{\mathsf{MPC}}$-hybrid model with random oracle. Moreover, if all the parties are honest, the protocol aborts with negligible probability.*

The security proof of the preprocessing protocol Π^R_{Prep} strictly resembles the proof of Theorem 2. The only major difference is the leakage about the special positions allowed by the DPSF procedure in $\mathcal{F}_{\mathsf{MPC}}$. In any case, this fact will not constitute a security issue as the leakage will be absorbed the the hardness of Module-LPN.

The complete proof follows.

Proof. This is an adaptation of the proof of [AS21, Theorem 5] to Π^R_{Prep}.

Consider the simulator $\mathcal{S}^R_{\mathsf{Prep}}$ described in Figure 9. We prove that no PPT adversary is able to distinguish between the protocol Π^R_{Prep} and the composition of $\mathcal{F}^R_{\mathsf{Prep}}$ with $\mathcal{S}^R_{\mathsf{Prep}}$. Clearly, the simulation of the initialisation is perfectly secure. We therefore focus our attention the the generation of ECDSA tuples.

It is immediate to see that Π^R_{Prep} aborts with negligible probability if all the parties are honest. Indeed, an abort happens only if the non-zero element of a DPF execution is actually equal to zero, and therefore if and only if $\alpha_j = 0$ or $\mathsf{sk}_j = 0$ for any $j \in [n]$, or there exist $i \in [n]$, $r \in [c]$ and $l \in [t]$ such that $\beta^r_i[l] = 0$ or $\gamma^r_i[l] = 0$. All these elements are uniformly distributed over $\mathbb{F}_q$, therefore, since $q \sim 2^\lambda$, the probability of such event is negligible.

The simulation is unconditionally secure until the outputs are revealed. Moreover, if the adversary chose $\alpha_j = 0$, or $\mathsf{sk}_j = 0$, or $\beta^r_j[l] = 0$ or $\gamma^r_j[l] = 0$ for some $r \in [c]$ and $l \in [t]$ and corrupted party P_j, both the protocol and the simulation abort during an execution of $\mathsf{2\text{-}DSPF}$.

Before proceeding with our analysis, we define the random variables $\mathsf{sk} := \sum_{i\in[n]} \mathsf{sk}_i$, $x := \sum_{i\in[n]} x_i$ and $y := \sum_{i\in[n]} y_i$.

Claim 3.1 *In the protocol, we have the following relations*

$$\sum_{i\in[n]} d_i := \mathsf{sk} \cdot y, \qquad \sum_{i\in[n]} z_i = x \cdot y, \qquad M_{i,j} = K_{i,j} + \alpha_j \cdot x_i.$$

Proof (of the claim). The proof strictly resembles the one of Claims 2.1 and 2.2. ■

Claim 3.2 *Consider the protocol and let P_ι be an honest party. The values $(d_j, z_j)_{j\in\mathcal{H}\setminus\{\iota\}}$ are uniformly distributed in R and independent of the remaining outputs and the view of the adversary. Moreover, if P_i and P_j are both honest, $K_{i,j}$ and $M_{i,j}$ are random subject to $M_{i,j} = K_{i,j} + \alpha_j \cdot x_i$.*

The Protocol Π^R_{Prep}

Let N be a power of 2. Take a degree N polynomial $F(X)$ over the prime field $\mathbb{F}_q$. Define the ring $R := \mathbb{F}_q[X]/\big(F(X)\big)$ and consider Module-LPN parameters $t, c \in \mathbb{N}$.

INITIALISATION Each party P_i samples $\alpha_i, \mathsf{sk}_i \xleftarrow{\$} \mathbb{F}_q$ and inputs the values in $\mathcal{F}_{\mathsf{MPC}}$ to obtain $[[\alpha_i]]$ and $[[\mathsf{sk}_i]]$. Using $\mathcal{F}_{\mathsf{MPC}}$, the parties compute and open $\mathsf{PK} \leftarrow \sum_{i=1}^{n} [[\mathsf{sk}_i]]G$. Then, every party P_i outputs (α_i, PK).

ECDSA TUPLE

1. For all $i \in [n]$ and $r \in [c]$, P_i samples $\boldsymbol{\beta_i^r} \xleftarrow{\$} \mathbb{F}_q^t$, $\boldsymbol{\omega_i^r} \xleftarrow{\$} [N]^t$, $\boldsymbol{\gamma_i^r} \xleftarrow{\$} \mathbb{F}_q^t$ and $\boldsymbol{\eta_i^r} \xleftarrow{\$} [N]^t$. Then, it computes the polynomials
$$u_i^r(X) \leftarrow \sum_{l\in[t]} \beta_i^r[l] \cdot X^{\omega_i^r[l]}, \qquad v_i^r(X) \leftarrow \sum_{l\in[t]} \gamma_i^r[l] \cdot X^{\eta_i^r[l]}$$
2. For every $i \in [n]$ and $r \in [c]$, the parties compute the following operations using $\mathcal{F}_{\mathrm{MPC}}$.
$$[[\boldsymbol{\beta_i^r}]] \leftarrow \mathsf{Input}(P_i, \boldsymbol{\beta_i^r}), \qquad [[\boldsymbol{\gamma_i^r}]] \leftarrow \mathsf{Input}(P_i, \boldsymbol{\gamma_i^r}),$$
$$[[\boldsymbol{\omega_i^r}]]_2 \leftarrow \mathsf{Input}(P_i, \boldsymbol{\omega_i^r}), \qquad [[\boldsymbol{\eta_i^r}]]_2 \leftarrow \mathsf{Input}(P_i, \boldsymbol{\eta_i^r}).$$
3. For every $i, j \in [n]$ with $i \neq j$ and $r \in [c]$, the parties compute, using $\mathcal{F}_{\mathsf{MPC}}$,
$$[[\boldsymbol{\mu_{i,j}^r}]] \leftarrow [[\alpha_i]] \cdot [[\boldsymbol{\beta_j^r}]], \qquad [[\boldsymbol{\nu_{i,j}^r}]] \leftarrow [[\mathsf{sk}_i]] \cdot [[\boldsymbol{\gamma_j^r}]].$$
4. For every $i, j \in [n]$ with $i \neq j$ and $r \in [c]$, the parties call $\mathcal{F}_{\mathsf{MPC}}$ to compute
$$(\widetilde{\boldsymbol{K}}_{\boldsymbol{j,i}}^{\boldsymbol{r}}, \widetilde{\boldsymbol{M}}_{\boldsymbol{j,i}}^{\boldsymbol{r}}) \leftarrow \text{2-DSPF}(N, [[\boldsymbol{\omega_j^r}]]_2, [[\boldsymbol{\mu_{i,j}^r}]], i, j)$$
and obtain a 2-party secret-sharing among P_i and P_j of the N-dimensional t-point vector with special positions $[[\boldsymbol{\omega_j^r}]]_2$ and non-zero elements $[[\boldsymbol{\mu_{i,j}^r}]]$. If $\mathcal{F}_{\mathsf{MPC}}$ outputs Zero, the protocol aborts. Let $\widetilde{\boldsymbol{K}}_{\boldsymbol{j,i}}^{\boldsymbol{r}}$ and $\widetilde{\boldsymbol{M}}_{\boldsymbol{j,i}}^{\boldsymbol{r}}$ denote the shares obtained by P_i and P_j respectively. We regard them as degree-$(N-1)$ polynomials $\widetilde{K}_{j,i}^r(X)$ and $\widetilde{M}_{j,i}^r(X)$.
5. For every $i, j \in [n]$ with $i \neq j$ and $r \in [c]$, the parties call $\mathcal{F}_{\mathsf{MPC}}$ to compute
$$(\widetilde{\boldsymbol{v}}_{\boldsymbol{i,j}}^{\boldsymbol{r,0}}, \widetilde{\boldsymbol{v}}_{\boldsymbol{i,j}}^{\boldsymbol{r,1}}) \leftarrow \text{2-DSPF}(N, [[\boldsymbol{\eta_j^r}]]_2, [[\boldsymbol{\nu_{i,j}^r}]], i, j)$$
and obtain a 2-party secret-sharing among P_i and P_j of the N-dimensional t-point vector with special positions $[[\boldsymbol{\eta_j^r}]]_2$ and non-zero elements $[[\boldsymbol{\nu_{i,j}^r}]]$. If $\mathcal{F}_{\mathsf{MPC}}$ outputs Zero, the protocol aborts. Let $\widetilde{\boldsymbol{v}}_{\boldsymbol{i,j}}^{\boldsymbol{r,0}}$ and $\widetilde{\boldsymbol{v}}_{\boldsymbol{i,j}}^{\boldsymbol{r,1}}$ denote the shares obtained by P_i and P_j respectively. We regard them as degree-$(N-1)$ polynomials $\widetilde{v}_{i,j}^{r,0}(X)$ and $\widetilde{v}_{i,j}^{r,1}(X)$.
6. For every $i \neq j$ and $r, s \in [c]$, the parties compute
$$[[\boldsymbol{\rho_{i,j}^{r,s}}]] \leftarrow [[\boldsymbol{\beta_i^r}]] \otimes [[\boldsymbol{\gamma_j^s}]], \qquad [[\boldsymbol{\zeta_{i,j}^{r,s}}]] \leftarrow [[\boldsymbol{\omega_i^r}]]_2 \boxplus [[\boldsymbol{\eta_j^s}]]_2.$$
7. For every $i, j \in [n]$ with $i \neq j$ and $r, s \in [c]$, the parties call $\mathcal{F}_{\mathsf{MPC}}$ to compute
$$(\boldsymbol{w}_{\boldsymbol{i,j}}^{\boldsymbol{r,s,0}}, \boldsymbol{w}_{\boldsymbol{i,j}}^{\boldsymbol{r,s,1}}) \leftarrow \text{2-DSPF}(2N, [[\boldsymbol{\zeta_{i,j}^{r,s}}]]_2, [[\boldsymbol{\rho_{i,j}^{r,s}}]], i, j)$$
and obtain a 2-party secret-sharing among P_i and P_j of the $2N$-dimensional t^2-point vector with special positions $[[\boldsymbol{\zeta_{i,j}^{r,s}}]]_2$ and non-zero elements $[[\boldsymbol{\rho_{i,j}^{r,s}}]]$. Let $\boldsymbol{w}_{\boldsymbol{i,j}}^{\boldsymbol{r,s,0}}$ and $\boldsymbol{w}_{\boldsymbol{i,j}}^{\boldsymbol{r,s,1}}$ denote the shares obtained by P_i and P_j respectively. We regard them as degree-$(2N-1)$ polynomials $w_{i,j}^{r,s,0}(X)$ and $w_{i,j}^{r,s,1}(X)$.
8. For every $r \in [c]$, each party P_i computes
$$\widetilde{v}_i^r(X) \leftarrow \mathsf{sk}_i \cdot v_i^r(X) + \sum_{j\neq i} \Big(\widetilde{v}_{i,j}^{r,0}(X) + \widetilde{v}_{j,i}^{r,1}(X) \Big).$$

Fig. 7: The preprocessing protocol - Part 1

9. For every $r, s \in [c]$, each party P_i computes over R

$$w_i^{rc+s}(X) \leftarrow u_i^r(X) \cdot v_i^s(X) + \sum_{j \neq i} \Big(w_{i,j}^{r,s,0}(X) + w_{j,i}^{r,s,1}(X) \Big).$$

10. The parties call $\mathcal{O}$ to obtain $a_i \xleftarrow{\$} R$ for every $i \in [c-1]$. Let $\boldsymbol{a} \leftarrow (a_0, a_1, \ldots, a_{c-2}, 1)$.
11. Each party P_i outputs

$$x_i \leftarrow \langle \boldsymbol{a}, \boldsymbol{u_i} \rangle, \qquad y_i \leftarrow \langle \boldsymbol{a}, \boldsymbol{v_i} \rangle, \qquad z_i \leftarrow \langle \boldsymbol{a} \otimes \boldsymbol{a}, \boldsymbol{w_i} \rangle,$$
$$M_{i,j} \leftarrow \langle \boldsymbol{a}, \widetilde{\boldsymbol{M}}_{\boldsymbol{i,j}} \rangle, \qquad K_{j,i} \leftarrow -\langle \boldsymbol{a}, \widetilde{\boldsymbol{K}}_{\boldsymbol{j,i}} \rangle \qquad d_i \leftarrow \langle \boldsymbol{a}, \widetilde{\boldsymbol{v}}_{\boldsymbol{i}} \rangle.$$

Fig. 8: The preprocessing protocol - Part 2

Proof (of the claim). Observe that for every $j \in \mathcal{H}$ with $j \neq \iota$, the polynomials $\widetilde{v}_{j,\iota}^{c-1,0}(X)$ and $w_{j,\iota}^{c-1,c-1,0}(X)$ are both random in R and independent of the view of the adversary and of the honest parties $(P_i)_{i \in \mathcal{H} \setminus \{j,\iota\}}$. This is due to the behaviour of 2-DPF when the targeted parties are both honest. Actually, we are implicitly relying on the fact that the polynomials in $\mathbb{F}_q[X]$ of degree less than $2N$ form a vector space and the reduction modulo $F(X)$ is a $\mathbb{F}_q$-linear operation from this space to R. As a consequence, every element in R has the same number of preimages, therefore, the reduction modulo $F(X)$ maps the uniform distribution over the polynomials of degree less than $2N$ into the uniform distribution over R.

Since $\widetilde{v}_{j,\iota}^{c-1,0}(X)$ and $w_{j,\iota}^{c-1,c-1,0}(X)$ are terms of $\widetilde{v}_j^{c-1}(X)$ and $w_j^{c^2-1}(X)$ respectively, we understand that the latter are random in R and independent.

To conclude the first part of the claim, observe that $\widetilde{v}_j^{c-1}(X)$ is multiplied by $a_{c-1} = 1$ when computing $d_j = \langle \boldsymbol{a}, \widetilde{\boldsymbol{v}}_{\boldsymbol{j}} \rangle$. Moreover, $w_j^{c^2-1}(X)$ is multiplied by $a_{c-1} \cdot a_{c-1} = 1$ when computing $z_j = \langle \boldsymbol{a} \otimes \boldsymbol{a}, \boldsymbol{w_j} \rangle$.

For the second part of the claim, notice that when P_i and P_j are both honest, $\widetilde{K}_{i,j}^{c-1}$ is random, due to the behaviour of 2-DPF. Since $\widetilde{K}_{i,j}^{c-1}$ is multiplied by $a_{c-1} = 1$ when computing $K_{i,j} = -\langle \boldsymbol{a}, \widetilde{\boldsymbol{K}}_{\boldsymbol{i,j}} \rangle$, $K_{i,j}$ is random. The fact that $M_{i,j} = K_{i,j} + \alpha_j \cdot x_i$ is a consequence of Claim 3.1. ■

Claim 3.3 *By the R^c-LPN$_t$ assumption with static leakage, no PPT adversary can distinguish between the procedure* Tuple *and its simulation.*

Proof (of the claim). Let the random variables $\widehat{d}$ and $\widehat{z}$ represent the sum of the shares of the corrupted parties, defined as in $\mathcal{S}_{\mathsf{Prep}}^R$.

We proceed by a series of $2|\mathcal{H}|$ hybrids. In the initial stage, we consider the protocol execution in which, for every $i \in \mathcal{H}$, the values d_i and z_i are substituted with random elements in R, subject to

$$\sum_{i \in \mathcal{H}} d_i + \widehat{d} = \mathsf{sk} \cdot y, \qquad \sum_{i \in \mathcal{H}} z_i + \widehat{z} = x \cdot y$$

and $(M_{i,j}, K_{j,i})_{j \neq i}$ are sampled randomly under the condition $M_{i,j} = K_{i,j} + \alpha_j \cdot x_i$. Observe that by Claims 3.1 and 3.2, the initial stage is perfectly indistinguishable from Π_{Prep}^R.

Consider now an integer $i \leq |\mathcal{H}|$ and let j be the index of the i-th honest party. In the $2i$-th hybrid, we will substitute the final output x_j with a random element in R, keeping all the rest as in the previous stage. In the $(2i+1)$-th hybrid, we will do the same with y_j. Observe that in the last stage, the execution is identical to the simulation.

We show that any PPT adversary $\mathcal{A}$ distinguishing between two consequent hybrids can be converted into an efficient attacker $\mathcal{A}'$ against the R^c-LPN$_t$ assumption.

Suppose that $\mathcal{A}$ distinguishes between the $(2i-1)$-th and the $2i$-th hybrid. Let P_j be the i-th honest party. Observe that the only difference between the two stages is that x_j is computed as in the protocol in the former and randomly sampled in the latter. We construct the Module-LPN attacker $\mathcal{A}'$ as follows. Upon

The Simulator $\mathcal{S}^R_{\mathsf{Prep}}$

INITIALISATION The simulator waits for $(\alpha_i)_{i\in\mathcal{C}}$ from the adversary and forwards them to the functionality. Upon receiving PK as a reply, the simulator forwards it to the adversary. At the end, $\mathcal{S}^R_{\mathsf{Prep}}$ samples $\alpha_i, \mathsf{sk}_i \xleftarrow{\$} \mathbb{F}_q$ for every $i \in \mathcal{H}$.

ECDSA TUPLE The simulator runs the protocol with the adversary simulating the honest parties:

1. If the simulation aborts, the simulator sends Abort to the functionality.
2. At the end, the simulator reconstructs the sum of the outputs of the corrupted parties

$$\widehat{x} \leftarrow \sum_{i\in\mathcal{C}} x_i, \qquad \widehat{y} \leftarrow \sum_{i\in\mathcal{C}} y_i, \qquad \widehat{z} \leftarrow \sum_{i\in\mathcal{C}} z_i, \qquad \widehat{d} \leftarrow \sum_{i\in\mathcal{C}} d_i.$$

   Furthermore, it computes the pair $(M_{i,j}, K_{j,i})$ for every $i \in \mathcal{C}$ and $j \in \mathcal{H}$. Observe that $\mathcal{S}^R_{\mathsf{Prep}}$ can perform this operation. Indeed, in the initialisation, it learnt $(\alpha_i)_{i\in\mathcal{C}}$. Moreover, at the beginning of the procedure, it received $\boldsymbol{\beta}^{\boldsymbol{r}}_{\boldsymbol{i}}, \boldsymbol{\gamma}^{\boldsymbol{r}}_{\boldsymbol{i}}, \boldsymbol{\omega}^{\boldsymbol{r}}_{\boldsymbol{i}}, \boldsymbol{\eta}^{\boldsymbol{r}}_{\boldsymbol{i}}$ for every $i \in \mathcal{C}$ and $r \in [c]$. Finally, in every execution of $\mathsf{2\text{-}DSPF}$ involving corrupted parties, the adversary sends to the simulator the shares that the corrupted parties selected for the output.
3. Let ι be the index of a corrupted party. The simulator sets $\widehat{x}_\iota \leftarrow \widehat{x}$, $\widehat{y}_\iota \leftarrow \widehat{y}$, $\widehat{z}_\iota \leftarrow \widehat{z}$ and $\widehat{d}_\iota \leftarrow \widehat{d}$. Moreover, for every $i \in \mathcal{C} \setminus \{\iota\}$, $\mathcal{S}^R_{\mathsf{Prep}}$ sets $\widehat{x}_i \leftarrow 0$, $\widehat{y}_i \leftarrow 0$, $\widehat{z}_i \leftarrow 0$ and $\widehat{d}_i \leftarrow 0$. Finally, the simulator sends $(\widehat{x}_i, (M_{i,j}, K_{j,i})_{j\in\mathcal{H}}, \widehat{y}_i, \widehat{d}_i, \widehat{z}_i)_{i\in\mathcal{C}}$ to the functionality.

Fig. 9: The simulator $\mathcal{S}^R_{\mathsf{Prep}}$

activation $\mathcal{A}'$ initialises an internal copy of $\mathcal{A}$ and runs the protocol simulating the honest parties. During the generation of ECDSA tuples, $\mathcal{A}'$ lets its challenger select the non-zero values $\boldsymbol{\beta}^{\boldsymbol{r}}_{\boldsymbol{j}}$ and special positions $\boldsymbol{\omega}^{\boldsymbol{r}}_{\boldsymbol{j}}$ for every $r \in [c]$. When $\mathcal{A}$ tries to guess a special position $\omega^r_j[l]$ for some $r \in [c]$ and $l \in [t]$ by specifying a set $I \subseteq [N]$ during $\mathsf{2\text{-}DSPF}$, $\mathcal{A}'$ issues a query (r, l, I) to its challenger and forwards the reply to $\mathcal{A}$. Moreover, when $\mathcal{A}$ tries to guess a special position $\zeta^{r,s}_{j,k}[lt + h] = \omega^r_j[l] + \eta^s_k[h]$ for some $k \in [n]$, $r, s \in [c]$ and $l, h \in [t]$ by specifying a set $I' \subseteq [2N]$ during $\mathsf{2\text{-}DSPF}$, $\mathcal{A}'$ computes the set

$$I'' \leftarrow \{\chi - \eta^s_k[h] \mid \chi \in I'\},$$

issues the query (r, l, I'') to its challenger and forwards the reply to $\mathcal{A}$. Observe that $\mathcal{A}'$ knows $\eta^s_k[h]$ so I'' can always be computed.

Finally, $\mathcal{A}'$ waits for $(\boldsymbol{a}, x_j)$ from its challenger. We recall that x_j is computed as in the protocol with probability $1/2$. In the other cases, it is uniformly sampled in R. $\mathcal{A}'$ models $\mathcal{O}$ by sending $\boldsymbol{a}$ to $\mathcal{A}$. At the end, the attacker $\mathcal{A}'$ computes $(M_{i,j}, K_{j,i})_{i\in\mathcal{C}, j\in\mathcal{H}}$, $\widehat{d}$, $\widehat{z}$ and the outputs of the honest parties. For every $k \in \mathcal{H}$ with $k < j$, it substitutes x_k and y_k with random elements in R. Finally, it generates $(M_{i,j}, K_{j,i})_{j\neq i}, d_i$ and z_i for every $i \in \mathcal{H}$ as in the simulation.

Observe that when the challenger replies with random elements in R, the view of $\mathcal{A}$ is indistinguishable from the view in the $2i$-th hybrid. If instead the challenger computes x_j using sparse polynomials in R, the view of $\mathcal{A}$ is indistinguishable from the view in the $(2i-1)$-th hybrid.

So, if $\mathcal{A}$ distinguished between the $(2i-1)$-th and the $2i$-th hybrid with non-negligible advantage, then $\mathcal{A}'$ would break the R^c-LPN$_t$ hardness. In a totally analogous way, we can prove that the same holds if $\mathcal{A}$ distinguished between the $2i$-th and the $(2i+1)$-th hybrid. ■

□

Efficiency Our protocol is particularly efficient from a communication point of view. The cost of the triple generation procedure per party is indeed

- $2(n-1)\cdot c\cdot t$ times the total complexity of 2-DPF with output length N,
- $(n-1)\cdot c^2\cdot t^2$ times the total complexity of 2-DPF with output length $2N$,
- $2c\cdot t\cdot\big(\log q+\log N\big)$ bits of communication for the inputs,
- $4n(n-1)\cdot c\cdot t\cdot\log q$ bits for the multiplications in step 3,
- $2n(n-1)\cdot c^2\cdot t^2\cdot\log q$ bits for the outer product,
- $2n(n-1)\cdot c^2\cdot t^2\cdot\log N$ bits for the outer sum,
- $O(\lambda\cdot n+\log q)$ complexity for the MAC checks[9].

Considered the complexity analysis of 2-DPF [BCG+20], we conclude that the communication complexity of the procedure is dominated by $13n^2\cdot c^2\cdot t^2\cdot(\log N+\log q)+4n\cdot c^2\cdot t^2\cdot\log N\cdot\lambda$. We recall that every execution of Tuple permits to produce N fresh ECDSA tuples.

Observe that the implementation of $\mathcal{F}_{\mathsf{MPC}}$ requires some additional preprocessing material, the generation of which does not affect the overall asymptotic complexity (even if it needs to be produced by some other preprocessing protocol). Specifically, we need

- $2n\cdot c\cdot t$ input masks over $\mathbb{F}_q$ (step 2),
- $2n\cdot c\cdot t\cdot\log(N)$ input masks over $\mathbb{F}_2$ (step 2),
- $2n(n-1)\cdot c\cdot t+n(n-1)\cdot c^2\cdot t^2$ multiplication triples over $\mathbb{F}_q$ (step 3 and outer product),
- $n(n-1)\cdot c^2\cdot t^2\cdot\log(N)$ AND triples over $\mathbb{F}_2$ (outer sum).

5 Implementation and Experimental Results

We implemented our protocol and run experiments to test its practicality. [10]

Our code is implemented for the *Secp256k1* elliptic curve used in Bitcoin. This is done to demonstrate the applicability of our threshold ECDSA to blockchain wallets. For the DPF, we choose to implement the optimised protocol from [BGI16, Figure 4]. The DSPF is optimised using multi-threading.

Setup We chose to implement the simplified version of the protocol in the setting where a trusted dealer distributes the PCG seeds, and then the servers perform the local seed expansion before interacting for the distributed computation of the signing phase. This model is meaningful in practical applications in which, for instance, a client generates its own ECDSA secret key and then distributes it to a number of servers. In this case it is meaningful to ask the client to (also) generate the (short) PCG seeds that will be used in the protocol. This is a setting which makes sense e.g. in applications to threshold wallets [AF21].

Instantiating Module-LPN For Module-LPN, we use a cyclotomic ring as defined in [BCG+20] where the prime q is the order of the elliptic curve. Note that q is not well suited for radix 2 FFT, because the maximum power of two dividing $q-1$ is 2^6. Our FFT, implementing the Cooley-Tuckey algorithm, is optimised for the factors of $q-1$. This is why the parameter N, which eventually accounts for the number of offline signatures, is not a power of 2 and is taken from a given set of optimised values. We considered different configurations of t, c that achieve the 128-bit security level, and finally picked the one for which we got the best performances. In particular, we chose $(c,t)=(4,16)$, which we found performed better than $(c,t)=(2,76)$ and $(c,t)=(8,5)$. These values are taken from [BCG+20] for dimension $N=2^{20}$; however, as noted by the analysis in that paper, the hardness of ring-LPN with cyclotomic polynomials essentially only depends on (c,t) and not N, due to a dimension-reduction attack.

[9] It is fundamental to run a check on the inputs of 2-DPF before executing the procedures. Clearly, the MAC checks can be batched.

[10] `https://github.com/ZenGo-X/silent-ecdsa`

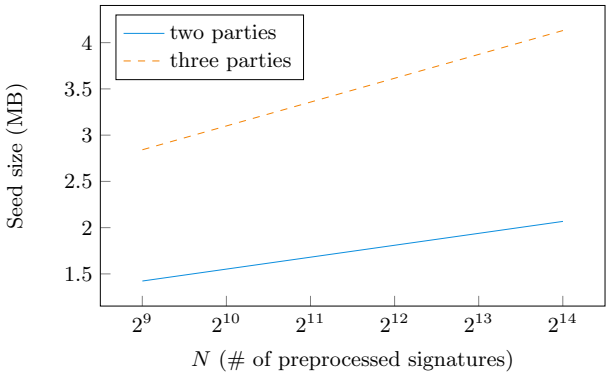

Fig. 10: PCG seed size in N, the number of ECDSA tuples we generate offline. The x-axis uses a logarithmic scale.

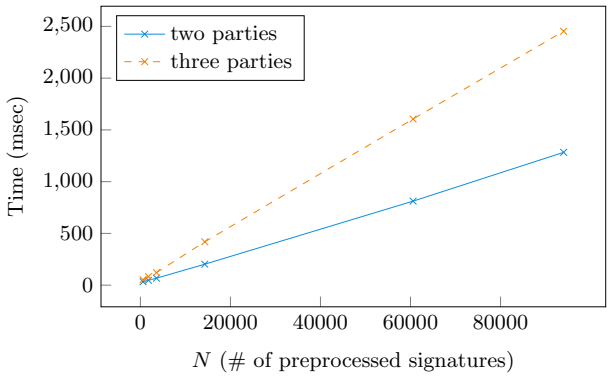

Fig. 11: Offline computation time per signature (amortised).

Measurements We measured the performance of our protocol by looking at two measures: (i) offline time per signature (amortised); and (ii) online time to generate a single signature. Our measurements were done on a machine with Intel i7 2.6GHz 6 core CPU, and 16GB memory. For benchmarks, we used an AWS t3.large machine with 8GB of memory.

One unique feature of our construction is that the storage costs for the PCG preprocessing are only logarithmic in N, the number of preprocessed signatures, unlike previous constructions. We show this property in Figure 10 for 2 and 3-party computation (the graph is drawn using theoretical values). Note that this feature is only suitable for the amortised setting, when preprocessing many signatures in advance. This can be useful, for instance, in applications where several powerful, independently located servers may be used to perform threshold signing on behalf of a large number of clients (who cannot run full MPC nodes themselves).

For the running costs, we measured the time it takes for the parties to produce one signature. The online signing phase in our protocol (Round 2 in Figure 4) is simple: it involves one message, after which each party can run a local linear computation, which is dependent in the number of parties, to compute the full signature and verify it. In our implementation, the operation takes on average 5ms in the 2-party case and 11ms for three parties.

Finally, we measured the local computation time for the PCG seed expansion in the offline protocol (Round 1 in Figure 4). Since the seed expansion is fully non-interactive, local computation time is the main bottleneck. One additional cost is that after seed expansion, the ECDSA tuples for N signatures must be stored in memory. However, for 3 parties and $N = 94019$, this is only around 24MB, so insignificant.

Figure 11 shows the amortised runtime of the seed expansion, per signature. From profiling the pre-signing stage, we can say that on average, 98% of the time goes on step 2 – retrieving shares of the next ECDSA tuple, which includes expanding the stored data. This is also where the dependence in N comes into play, since the FFT algorithm is super-linear. While we did not implement the protocol for distributing the PCG seeds, we believe this would not significantly change the overall costs, since seed expansion is the bottleneck.

Future work The main bottleneck in our code is the execution of an NTT/FFT over Secp256k1, which is needed for polynomial arithmetic in the ring-LPN assumption, as well as polynomial evaluation when converting the ring-ECDSA tuple into N tuples over $\mathbb{F}_p$. This is particularly challenging due to the order of Secp256k1, which has no large enough power of 2 factor needed for typical FFT algorithms. Improvements to the FFT algorithm for such curves may dramatically reduce the computation time of the preprocessing [BCKL21]. In addition, another possibility is to use ring-LPN with a more structured, regular error distribution [BCG+20], which would reduce the runtime of DPF evaluation (however, this is not currently the bottleneck). Moreover, our preprocessing protocol to setup the PCG seeds uses generic MPC primitives, and its implementation is left as future work.

Comparison with previous approaches In Figure 4, we gave benchmarks for the non-interactive expansion phase of the PCG which our protocol uses. While we did not implement the full protocol to setup the seeds, we believe this would not significantly change computational costs, since seed expansion is the bottleneck. Without properly implementing and benchmarking all protocols on the same machines, it is hard to do an accurate comparison with previous works. However, we can get a rough idea by comparing numbers from existing work for the 1-out-of-2 case: [LN18] takes 100–300ms per signature, [GG18] take 30–90ms and [CCL+20] takes 400–700ms. For comparison, our amortized computational cost of PCG expansion is around 1–2s per signature, however, our bandwidth complexity is 1–2 orders of magnitude smaller than these (see Table 1), because of our PCG-based approach.

Acknowledgments. We would like to thank Matan Hamilis for helping out with the implementation of the protocol. Work supported partially by: the Concordium Blockhain Research Center, Aarhus University, Denmark; the Carlsberg Foundation under the Semper Ardens Research Project CF18-112 (BCM); the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme under grant agreement No 803096 (SPEC); ERC Project NTSC (742754); the Aarhus University Research Foundation (AUFF); and the Independent Research Fund Denmark (DFF) under project number 0165-00107B. Claudio Orlandi is a co-founder of Partisia Infrastructure and has been advising Concordium and ZenGo.

References

AF21. Robert Annessi and Ethan Fast. Improving security for users of decentralized exchanges through multi-party computation. *CoRR*, abs/2106.10972, 2021.

AHS20. Jean-Philippe Aumasson, Adrian Hamelink, and Omer Shlomovits. A survey of ECDSA threshold signing. Cryptology ePrint Archive, Report 2020/1390, 2020. `https://eprint.iacr.org/2020/1390`.

AS21. Damiano Abram and Peter Scholl. Low-Communication Multiparty Triple Generation for SPDZ from Ring-LPN. Cryptology ePrint Archive, 2021, 2021.

BCG+19a. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure computation. In *ACM CCS 2019*. ACM Press, November 2019.

BCG+19b. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In *CRYPTO 2019, Part III*, LNCS. Springer, Heidelberg, August 2019.

BCG+20. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators from ring-LPN. In *CRYPTO 2020, Part II*, LNCS. Springer, Heidelberg, August 2020.

BCGI18. Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. In *ACM CCS 2018*. ACM Press, October 2018.

BCKL21. Eli Ben-Sasson, Dan Carmon, Swastik Kopparty, and David Levit. Elliptic curve fast fourier transform (ECFFT) part I: fast polynomial algorithms over all finite fields. *CoRR*, abs/2107.08473, 2021.

BDOZ11. Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT 2011*, LNCS. Springer, Heidelberg, May 2011.

BGI15. Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *EUROCRYPT 2015, Part II*, LNCS. Springer, Heidelberg, April 2015.

BGI16. Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *ACM CCS 2016*. ACM Press, October 2016.

CCL+19. Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. Two-party ECDSA from hash proof systems and efficient instantiations. In *CRYPTO 2019, Part III*, LNCS. Springer, Heidelberg, August 2019.

CCL+20. Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. Bandwidth-efficient threshold EC-DSA. In *PKC 2020, Part II*, LNCS. Springer, Heidelberg, May 2020.

CGG+20. Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In *ACM CCS 20*. ACM Press, November 2020.

DJN+20. Ivan Damgård, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Jakob Illeborg Pagter, and Michael Bæksvang Østergaard. Fast threshold ECDSA with honest majority. In *SCN 20*, LNCS. Springer, Heidelberg, September 2020.

DKL+13. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS 2013*, LNCS. Springer, Heidelberg, September 2013.

DKLs18. Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. Secure two-party threshold ECDSA from ECDSA assumptions. In *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2018.

DKLs19. Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. Threshold ECDSA from ECDSA assumptions: The multiparty case. In *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2019.

DOK+20. Anders P. K. Dalskov, Claudio Orlandi, Marcel Keller, Kris Shrishak, and Haya Shulman. Securing DNSSEC keys via threshold ECDSA from generic MPC. In *ESORICS 2020, Part II*, LNCS. Springer, Heidelberg, September 2020.

DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO 2012*, LNCS. Springer, Heidelberg, August 2012.

FKP16. Manuel Fersch, Eike Kiltz, and Bertram Poettering. On the provable security of (EC)DSA signatures. In *ACM CCS 2016*. ACM Press, October 2016.

GG18. Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. In *ACM CCS 2018*. ACM Press, October 2018.

GGN16. Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security. In *ACNS 16*, LNCS. Springer, Heidelberg, June 2016.

GI14. Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *EUROCRYPT 2014*, LNCS. Springer, Heidelberg, May 2014.

GS21. Jens Groth and Victor Shoup. On the security of ecdsa with additive key derivation and presignatures. Cryptology ePrint Archive, Report 2021/1330, 2021. `https://ia.cr/2021/1330`.

KMOS21. Yashvanth Kondi, Bernardo Magri, Claudio Orlandi, and Omer Shlomovits. 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*, pages 608–625. IEEE, 2021.

Lin17. Yehuda Lindell. Fast secure two-party ECDSA signing. In *CRYPTO 2017, Part II*, LNCS. Springer, Heidelberg, August 2017.

LN18. Yehuda Lindell and Ariel Nof. Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In *ACM CCS 2018*. ACM Press, October 2018.

LNR18. Yehuda Lindell, Ariel Nof, and Samuel Ranellucci. Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. Cryptology ePrint Archive, Report 2018/987, 2018. `https://eprint.iacr.org/2018/987`.

MR01. Philip D. MacKenzie and Michael K. Reiter. Two-party generation of DSA signatures. In *CRYPTO 2001*, LNCS. Springer, Heidelberg, August 2001.

NNOB12. Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *CRYPTO 2012*, LNCS. Springer, Heidelberg, August 2012.

ST19. Nigel P. Smart and Younes Talibi Alaoui. Distributing any elliptic curve based protocol. In *17th IMA International Conference on Cryptography and Coding*, LNCS. Springer, Heidelberg, December 2019.