IACR Communications in Cryptology
Vol. 0, No. 0, 54 pages.

Robust ECDSA Threshold Signing at Scale

Carsten Baum[a,1] and Anders Dalskov[2]

[1] DTU, Denmark
[2] Partisia, Denmark

Abstract. This work presents an efficient, scalable, and robust threshold ECDSA signing protocol for the $t \leq (n-1)/3$ scenario, making it suitable for e.g., bridging consensus from one Blockchain to another. Our construction allows to batch-compute pre-processing material for efficient online signing. While our protocol is based on established ideas for (Verifiable) Secret Sharing, we optimize it to scale efficiently both in terms of computation and communication with a large number of signers. We therefore combine efficient batching techniques for sharing (Fitzi & Hirt, PODC 2006), randomness extraction (Beerliova-Trubiniova & Hirt, TCC 2008) and share opening (Damgård & Nielsen, CRYPTO 2007) with recent advances in efficient Zero-Knowledge proofs (Attema & Cramer, CRYPTO 2020). We implement and evaluate the protocol experimentally and show that it is practical for large number of parties n.

Keywords: ECDSA · Threshold Signing · Blockchain Bridge

1 Introduction

A t-out-of-n threshold signature scheme allows n parties $P_1, \ldots, P_n$ to sign a message m, provided at least $t+1 < n$ of the parties participate. Conversely, any subset of t or less parties will be unable to sign anything.[1] Threshold signature schemes are attractive because they extend an ability normally reserved for a single party—namely, the party holding the signing key sk—to a group of parties [Des88]. Moreover, and because a threshold signature protocol is usually a threshold version of a traditional digital signature scheme, the resulting signature can be consumed by any verifier holding the right public key. A threshold signature protocol therefore makes it possible to increase security of processes that rely on digital signatures. Threshold signatures have been studied extensively as a tool for securing cryptocurrency wallets [ANO+22, KMOS21, GG18, AHS20, DKLs18, DKLs19, DKLs24, BMP22, DJN+20]. In the cryptocurrency wallet use-case, a user's account key (a signing key) is split between multiple different entities, typically controlled by the original owner. The use of threshold signatures in the cryptocurrency context means that a substantial amount of effort has been put into creating efficient threshold versions of the *Elliptic Curve Digital Signature Algorithm*, or ECDSA [JMV01].

Beyond securing cryptocurrency wallets, threshold ECDSA can also be used to bridge information (such as assets) between two blockchains. Roughly speaking, bridging is the task of moving consensus information from one blockchain A to another blockchain B. If both blockchains A and B use the same block validation algorithm then bridging might be easy since all the machinery used to validate a block on B can be repurposed to validate a

E-mail: cabau@dtu.dk (Carsten Baum), anderspkd@partisia.com (Anders Dalskov)
[a] Part of this work was supported by Partisia.
[1] In threshold signing literature, a t-out-of-n scheme often means that t shares are needed to produce a signature. This is in contrast to secure computation literature where t-out-of-n means that t shares reveal no information (and so in the signing context, cannot be used for signing). We use the latter convention in this work.

This work is licensed under a "CC BY 4.0" license.

block coming from A. On the other hand, if A and B run different consensus algorithms then the task is much more difficult. A simplistic approach could be to encode the full validation algorithm in a smart contract on B, thus effectively turning blockchain A into a layer-2 blockchain with respect to B. However, block validation is often a complex process, so such an approach is not likely to scale.

A different, and more efficient, solution can be obtained from a threshold signature scheme, and in particular a threshold ECDSA scheme if ECDSA signatures can be efficiently verified by the target blockchain. Bridging data from chain A to B using a threshold signature scheme works as follows: On chain A the validators of the chain generate a secret-shared signing key $\langle \mathsf{sk} \rangle$ and public key pk. This is achieved by letting the validators run an interactive protocol, where the result is that each validator holds a "share" sk_i of a signing key. If a t-out-of-n secret-sharing scheme is used, then t shares reveal nothing about sk, whereas $t+1$ or more reveal the whole of sk. And so, by setting t to be a sufficiently large fraction of the validators (e.g., $t \approx n/3$) we may reasonably assume that sk is hidden. To bridge some information m from A, the validators use the shared signing key to produce a regular ECDSA signature σ on m. This signature is then moved to a smart contract on B, whose only job is to verify σ with respect to the validators public key pk. Suppose this system uses the same threshold t for the security of the signature scheme as it does for the security of consensus on A. Then, a valid signature for a piece of information m will be received by B if and only if m was part of the chain state on A. We have to be careful here: It is clearly the case that, if B receives a valid σ on some information m, then m was placed on-chain on A. However, if m was placed on-chain on A, then the creation of σ depends on the robustness properties of the threshold signature scheme. In particular, the threshold signature scheme must guarantee output, regardless of the behavior of up to t parties. Typical blockchains use a committee based approach for validators, and each committee typically contains around 100 validators.

Our contributions. In this work, we present a threshold ECDSA signature scheme that is optimized towards the blockchain setting. More precisely, we present and benchmark a threshold ECDSA signature scheme suitable for n parties of which $t \leq (n-1)/3$ are assumed to be corrupt. Our protocol achieves guaranteed output delivery against a malicious adversary, corrupting t parties and is demonstrably efficient for $n \approx 100$. In addition, as our protocol is to be used in the blockchain setting, it makes minimal use of a broadcast channel. This is because broadcasting large messages is expensive, particularly in the blockchain setting.

Our work uses Secure Multiparty Computation (MPC) techniques designed for protocols with a large number of parties that should communicate as little as possible. Our main technical contribution is an efficient protocol with guaranteed output delivery that can create batches of verifiable secret sharings, where the same random value is simultaneously shared with two corruption thresholds (so-called *double-sharings*). We use this to batch-generate pre-signatures for ECDSA, which allow the n parties to sign a message in a single round of interaction. To create these pre-signatures, our protocol requires parties to show that a certain message is well-formed. We achieve this by adapting existing zero-knowledge proof techniques to balance prover and verifier runtime with communication overhead. Finally, we implemented all algorithms necessary for signing that lie on the "happy path" (i.e. player elimination is not implemented) and benchmark against an existing ECDSA protocol implementation that runs in the same environment as ours. Like prior works, ours focused primarily on implementing the happy path for benchmarking

More concretely, our threshold ECDSA signing protocol combines and optimizes the following building blocks:

- We construct a protocol (following the ideas of [FH06]) that allows parties to generate consistent verifiable secret sharings (of degree t or $2t$) of random values. This protocol

is optimized for small communication among parties and low utilization of broadcast resources, while guaranteeing output delivery.

- Next, we use hyper-invertible matrices as in [BTH08] to extract many random tuples from a single valid double sharing of each party. This allows us to generate $\Theta(t)$ random double sharings from $O(1)$ random sharings per party, while only incurring a small computational overhead.

- To perform reliable and communication-efficient openings of degree $2t$ sharings, we construct a batch-opening protocol akin to [DN07]. We enhance their original protocol by additionally using Zero-Knowledge (ZK) proofs to ensure correctness when opening the sharings. In order to keep the low communication per party, we use so-called Compressed Σ-protocols [AC20] to implement the ZK protocol.

- When implementing batch-opening with ZK, we observe that computation quickly becomes the bottleneck: each party sends $n-1$ shares privately, so it must make $n-1$ ZK proofs of correctness towards every party. In order to bring down the computational complexity, we observe that large parts of the involved ZK proofs are identical for all potential receivers. This allows a sender to "recycle" the most expensive proof towards all receivers, at the expense of additional communication that allows a cheap, receiver-dependent proof which leads to lower concrete prover time.

- We implemented all parts of the signing protocol (on the "happy path"), including the complex pre-processing and its ZK proofs. We provide benchmarks for up to 88 parties in a simulated WAN.

Paper outline. The rest of the introduction will be spent introducing our contributions and techniques at a high level (Section 1.1). We will also provide a short survey of recent as well as historic threshold ECDSA systems for the purpose of contextualizing this work (Section 1.2). In Section 2 we introduce the cryptographic primitives we will be using. Then, in Section 3 we present the non-interactive signing protocol, which can be used to sign messages once some suitable pre-processing material has been created. In Section 4 we show how to efficiently and robustly generate the pre-processing material our signing protocol requires. In Section 5 we present the efficient Zero-Knowledge proofs our pre-processing protocols make use of. A C++ implementation of our protocol is presented in Section 6, where we also present benchmarks showcasing the protocols effectiveness for n up to 88. Finally, our conclusion is provided in Section 7.

1.1 Overview of Techniques.

Let $\mathtt{G}$ be a generator of a group $\mathbb{G}$ of prime order q and let $k, \mathsf{sk} \in \mathbb{Z}_q$, where sk is the secret signing key, while k denotes a nonce. Given a message m, an ECDSA signature on m is defined as the pair (r, s) with $s = (H(m) + r \cdot \mathsf{sk})/k$ and $r = F(k\mathtt{G})$ for a function F.[2]

In a *Threshold* ECDSA scheme, the signing key sk is secret-shared as $\langle \mathsf{sk} \rangle = (\mathsf{sk}_1, \ldots, \mathsf{sk}_n)$, where sk_i is held by party P_i. Signing is performed as an interactive protocol which receives $\langle \mathsf{sk} \rangle$, $\langle k \rangle$, and m as input, and which outputs a regular ECDSA signature $\sigma = (r, s)$. Threshold signing can be sped up significantly using *pre-signatures* [DOK+20, CGG+20, DJN+20]. In a nutshell, a pre-signature is a piece of message independent (but signing key-dependent) material which can be created ahead of time.

[2] Given $(r_x, r_y) = \mathtt{R} \in \mathbb{G}$, the function F is often defined as $F(\mathtt{R}) \mapsto r_x \mod q$.

Using a pre-signature, the signing of a message m can then be performed non-interactively: Let $\sigma = (r, s)$ be an ECDSA signature. By definition

$$r = F(k\mathtt{G}), \qquad s = \frac{H(m) + r \cdot \mathsf{sk}}{k} = \frac{H(m)}{k} + r\frac{\mathsf{sk}}{k}.$$

If the parties pre-process $\langle \alpha \rangle = \langle k^{-1} \rangle$, $\langle \beta \rangle = \langle \mathsf{sk} \cdot k^{-1} \rangle$ and $\mathtt{R} = k\mathtt{G}$, then signing m does not involve any multiplications of secrets and can thus be performed non-interactively. Indeed, all parties simply compute $r = F(\mathtt{R})$, and output their share of $\langle s \rangle = H(m)\langle \alpha \rangle + r\langle \beta \rangle$. Creating a pre-signature only requires standard techniques: $\langle k \rangle$ can be computed using a protocol for generating random sharings; $\langle \alpha \rangle = \langle k^{-1} \rangle$ can be computed using the protocol of [BIB89]; computing $\langle \beta \rangle = \langle \mathsf{sk} \cdot k^{-1} \rangle$ can be computed using a secure multiplication protocol (of which there are numerous instantiations). Finally, $\mathtt{R}$ can be obtained by locally computing $\langle k \rangle \cdot \mathtt{G}$ and opening the result (see e.g., [ADEO21] for a formalization of this action).

ECDSA With Re-Randomizable Pre-Signatures. Rather than use a pre-signature of the form $(\mathtt{R}, \langle \alpha \rangle, \langle \beta \rangle)$ as above, we will use a slightly larger form $(\mathtt{R}, \langle h \rangle, \langle \gamma \rangle, \langle \rho \rangle)$ with h random, $\gamma = h \cdot k$ and $\rho = h \cdot \mathsf{sk}$. The main idea remains the same as above—using $(\mathtt{R}, \langle h \rangle, \langle \gamma \rangle, \langle \rho \rangle)$ a signature on m can be computed non-interactively by computing $r = F(\mathtt{R})$ and s as

1. $\langle s' \rangle = H(m)\langle h \rangle + r \cdot \langle \rho \rangle$.
2. Reveal s' and γ.
3. Output $s = s' \cdot \gamma^{-1}$.

The advantage of this slightly larger form is that it allows *re-randomization*, which was shown by [GS22b] to be necessary in order "preserve" the security of the threshold signing protocol.[3] Given a pre-signature $(\mathtt{R}, \langle h \rangle, \langle \gamma \rangle, \langle \rho \rangle)$ and a random value $\delta \in \mathbb{Z}_q$, then

$$(\mathtt{R}', \langle h \rangle, \langle \overline{\gamma} \rangle, \langle \rho \rangle) = (\mathtt{R} + \delta\mathtt{G}, \langle h \rangle, \langle \gamma \rangle + \delta\langle h \rangle, \langle \rho \rangle),$$

is another valid pre-signature. Thus, signing proceeds as follows:

1. Parties agree on pre-signature $(\mathtt{R}, \langle h \rangle, \langle \gamma \rangle, \langle \rho \rangle)$ and message m. We assume that such an agreement is possible, but also remark that it is necessary for security as pointed out by [DJN+20]. As our protocols rely on a broadcast channel, we allow ourselves to take this agreement for granted.
2. Parties generate a random offset $\delta \in \mathbb{Z}_q$, and use it to re-randomize the pre-signature. There are several valid ways in which δ could be obtained—the only requirement is that δ is not easily guessable in advance. Our protocol will utilize a random oracle.
3. The parties compute $r = F(\mathtt{R}')$ and $\langle s' \rangle = H(m)\langle h \rangle + r\langle \rho \rangle$. The parties then exchange shares of $\langle s' \rangle$. Finally, all parties then recover s' and output $\sigma = (r, s' \cdot (\overline{\gamma})^{-1})$.

As might be evident from the steps above, signing is a relatively straightforward process *provided* a valid pre-signature is available. Unfortunately, creating such a pre-signature—efficiently and robustly when the number of parties is large—is not as straightforward. Moreover, observe that the secret-sharing scheme we utilize must be *locally verifiable*. If this is not the case, then the parties only have one approach to robustly obtain the final signature σ: reconstruct σ, and verify it against the signed message. While this works, the

[3] The simpler pre-signature form $(\mathtt{R}, \langle \alpha \rangle, \langle \beta \rangle)$ allows the adversary to pick m based on the nonce that will be used. Although this does not lead to a direct break, it does reduce security.

computational complexity scales exponentially, making it a non-viable strategy for even moderate n. For this reason, we will make use of a *verifiable secret-sharing* (VSS) scheme.

In the following, we will use $\langle\cdot\rangle$ to denote a Shamir secret-sharing, and $[\![\cdot]\!]$ to denote a verifiable secret-sharing a'la Feldman [Fel87].[4] That is, $[\![v]\!] = (\langle v\rangle, \vec{\mathtt{C}})$, where $\langle v\rangle$ is a regular Shamir sharing of the value v, and $\vec{\mathtt{C}}$ is a vector of $t+1$ elements such that $\mathtt{C}_i = v_i\mathtt{G}$ is a commitment to party P_i's share and $\mathtt{C}_0 = v\mathtt{G}$. At a high level, there are two technical challenges we must overcome:

1. First, we need a robust and efficient protocol for creating uniformly random verifiable secret-sharings $[\![r]\!]$. Such a protocol will form the basis for everything else.
2. Second, we need a robust and efficient protocol for multiplying random verifiable secret-sharings together. That is, a protocol which takes as input $[\![a]\!]$, $[\![b]\!]$ and outputs $[\![c = ab]\!]$.

Robust Generation of Random Shares. Our pre-signature generation protocol uses [BTH08] as a starting point. More precisely, each party P_i locally picks $s^{(i)}$ at random and distributes verifiable secret-shares $[\![s^{(i)}]\!]$ to all the other parties. Each party then locally applies a $n \times (n-t)$ hyper-invertible matrix $\overrightarrow{M}$ on the received vector $([\![s^{(1)}]\!], \ldots, [\![s^{(n)}]\!])$ of shares. The result $([\![r^{(1)}]\!], \ldots, [\![r^{(n-t)}]\!])$ is a vector of *uniformly* random secret-shares. The tricky part of this protocol is achieving agreement among the honest parties on the input vector $([\![s^{(1)}]\!], \ldots, [\![s^{(n)}]\!])$. To achieve this agreement, we use a protocol of five phases, described here from the perspective of a single dealer P_i.

1. **Share**: The dealer creates a random secret-sharing $\langle s^{(i)}\rangle = (s_1^{(i)}, \ldots, s_n^{(i)})$ and commitments $\vec{\mathtt{C}}^{(i)}$ (that is, P_i creates a verifiable secret-sharing $[\![s^{(i)}]\!]$ of some random value $s^{(i)}$). Next, the dealer computes two hashes: a hash $h_1^{(i)}$ of $\vec{\mathtt{C}}^{(i)}$, and a Merkle tree hash $h_2^{(i)}$ of a rate 1/3 error-correcting encoding of $\vec{\mathtt{C}}^{(i)}$. Finally, the dealer privately sends $s_j^{(i)}$ and $\vec{\mathtt{C}}^{(i)}$ to each P_j, and broadcasts $h_1^{(i)}, h_2^{(i)}$ to all parties.
2. **Complain**: Each receiver P_j broadcasts a bit $v_i^{(j)}$ indicating if the data received from P_i was invalid. I.e., $v_i^{(j)} = 0$ if the $\vec{\mathtt{C}}^{(i)}$ received was consistent with $h_1^{(i)}$ and $h_2^{(i)}$, and if $s_j^{(i)}$ is consistent with $\vec{\mathtt{C}}^{(i)}$. Otherwise $v_i^{(j)} = 1$.
3. **Fill**: All parties then use the bits $\vec{v}_i = (v_i^{(1)}, \ldots, v_i^{(n)})$ to achieve consensus on the behavior of P_i: If $\vec{v}_i$ has weight $> t$ then the shares of P_i are discarded and P_i is marked as corrupt. If the weight of $\vec{v}_i$ is 0, then P_i's shares are accepted as-is. Otherwise, for every j such that $v_i^{(j)} = 1$, the dealer P_i sends $s_j^{(i)}$ to all parties P_k (recall that $v_i^{(j)} = 1$ means that P_j did not receive a valid share and/or commitments from P_i). Each P_k checks the received share after which it is forwarded to P_j together with an error-correcting code word of $\vec{\mathtt{C}}^{(i)}$. At the same time, P_k broadcasts a bit $w_i^{(k)}$ indicating if the share received from P_i was valid.
4. **Finalize**: Each P_j for which $v_i^{(j)} = 1$ either receives enough valid shares and code words of $\vec{\mathtt{C}}^{(i)}$ in order to obtain their share of $[\![s^{(i)}]\!]$, or it will be the case that $\vec{w}_i = (w_i^{(1)}, \ldots, w_i^{(n)})$ has weight $> t$. At this point, the parties will have achieved consensus on $[\![s^{(i)}]\!]$.
5. **Extract**: The vector $([\![s^{(1)}]\!], \ldots, [\![s^{(n)}]\!])$ is now well-defined and so the parties can locally apply a hyper-invertible matrix $\overrightarrow{M}$ on the shares to obtain a batch $([\![r^{(1)}]\!], \ldots, [\![r^{(n-t)}]\!])$ of $n-t$ uniformly random shares and their commitments.

[4] The use of Feldman's VSS scheme [Fel87] here is purely for the sake of simplifying the presentation. Our actual protocol uses Pedersen's VSS [Ped92].

The above outlines a protocol that allows the parties to batch-generate uniformly random shares in a way that is quite efficient in the "sunshine scenario"—i.e., when all parties act honest (or when malicious behavior manifests in the form of crashes). This is most often the case in a real life deployment. However, even the "bad" path in this protocol is still quite efficient. The use of error-correcting codes for re-sending $\vec{\mathtt{C}}$ ensures communication stays low.

Robust Generation of Random Double Shares. For our robust multiplication protocol, we will use the double-share technique of [DN07], which requires pairs of secret-sharings $(\langle r\rangle_t, \langle r\rangle_{2t})$, where the subscript denotes the degree of the Shamir sharing. The protocol for robust generating of random values that was described above can be used for generating double shares as well. Only minor changes are required to the **Share** step: Each party picks random s as before, but creates two sharings $\langle s\rangle_t$ and $\langle s\rangle_{2t}$. The rest of the protocol is the same (except that each step is run for both the degree t and $2t$ sharing). Because $\mathtt{C}_0$ is used in the commitments for both the degree t and degree $2t$ sharing, and because this commitment encodes the shared secret, the protocol guarantees that accepted shares encode the same secret. This also avoids the need to open (and discard) shares after the final step, e.g., as is done in [BTH08].

Robust Multiplication of Secret Shares. To multiply two degree-t secret-shared values $\langle a\rangle_t$, $\langle b\rangle_t$, we will use the technique from [DN07]:

1. Let $(\langle r\rangle_t, \langle r\rangle_{2t})$ be a double-share, obtained using the protocol above. (For now we ignore the commitments).
2. Compute and open $\langle w\rangle_{2t} = \langle a\rangle_t\langle b\rangle_t - \langle r\rangle_{2t}$.
3. Output $\langle a \cdot b\rangle_t = \langle c\rangle_t = w + \langle r\rangle_t$.

However, this approach is not directly applicable: First, although a $\langle\cdot\rangle$-sharing is 1-multiplicative, the $[\![\cdot]\!]$-sharing is not. In other words, the [DN07] trick can only be used to go from two $[\![\cdot]\!]$ shares to a $\langle\cdot\rangle$ share of their product. Second, we will want to perform multiplications in batches, which requires batching opening of w as well, as opening w one-by-one would be prohibitive for large n.

Efficient Batch Opening. Multiplications are performed in parallel as much as possible. This allows us to use a batch opening protocol in order to keep communication low. We utilize the batched opening protocol of [DN07]:

1. Fix $\theta_1, \ldots, \theta_n \in \mathbb{Z}_q$ ahead of time, and let $\langle w^{(1)}\rangle_{2t}, \ldots, \langle w^{(t)}\rangle_{2t}$ be t values that we wish to open (these are the intermediary values in the multiplication, hence they are degree $2t$).
2. Each party P_i locally computes $\langle u^{(k)}\rangle_{2t} = \sum_{j=1}^{t} \theta_k^j \langle w^{(j)}\rangle_{2t}$ and sends their share of $\langle u^{(k)}\rangle_{2t}$ to P_k together with a Zero-Knowledge proof of correctness (more on this below).
3. P_k receives at least $2t+1$ correct shares and so can recover $u^{(k)}$, since it is a degree $2t$ sharing. P_k then privately sends $u^{(k)}$ to all other parties.
4. In the end, each party holds $u^{(1)}, \ldots, u^{(n)}$, some of which may be faulty. However, as they correspond to degree t Reed-Solomon code-words, error correction can be used to recover the coefficients $w^{(1)}, \ldots, w^{(t)}$.

The missing piece is now the Zero-Knowledge proof in step two. Recall that the value that P_i sends to P_k is the share $u_i^{(k)} = \sum_{j=1}^{t} \theta_k^j (a_i^{(j)} b_i^{(j)} - r_i^{(j)})$, where $a_i^{(j)}$, $b_i^{(j)}$ and $r_i^{(j)}$ are all *verifiably* secret-shared (concretely, they will be outputs of the random share generation protocol). In particular, this implies that P_i also holds commitments $\mathtt{A}_i^{(j)} = a_i^{(j)}\mathtt{G}$, $\mathtt{B}_i^{(j)} = b_i^{(j)}\mathtt{G}$ and $\mathtt{R}_i^{(j)} = r_i^{(j)}\mathtt{G}$. P_i can therefore create a Zero-Knowledge proof for the relation:

$$\mathcal{R} = \left\{ \begin{array}{c} ((\mathtt{A}_i^{(j)}, \mathtt{B}_i^{(j)}, \mathtt{R}_i^{(j)})_{j=1}^{t}, \theta_k, u_i^{(k)}) \\ (a_i^{(j)}, b_i^{(j)}, r_i^{(j)})_{j=1}^{t} \end{array} \middle| \begin{array}{c} (\forall j \in \{1, \ldots, t\} : \mathtt{A}_i^{(j)} = a_i^{(j)}\mathtt{G}, \mathtt{B}_i^{(j)} = b_i^{(j)}\mathtt{G}, \mathtt{R}_i^{(j)} = r_i^{(j)}\mathtt{G}) \\ \text{and } u_i^{(k)} = \sum_{j=1}^{t} \theta_k^j (a_i^{(j)} b_i^{(j)} - r_i^{(j)}) \end{array} \right\}$$

A straightforward Zero-Knowledge proof of this relation would require $O(n)$ $\mathbb{G}$ and $\mathbb{Z}_q$ elements, and thus be too inefficient. We will therefore use Compressed Σ-protocol theory [AC20] to compress the proof. Through experimental validation we found that a Compressed Σ-protocol proof of the above relation is too computationally expensive. In a nutshell, the issue is that the prover (P_i in the above presentation) has to do an expensive computation *for each* receiver, as θ_k depends on the receiver P_k's identity. We thus carefully split the proof that P_i creates into two parts: An expensive, but reusable proof π_i, and a cheap per-party proof $\pi_{i,k}$.

Provable Security vs. Implementation All our protocols necessary for key generation and signing are defined and proven secure in the Universal Composability (UC) model [Can01]. However, we divert from UC when implementing the ZK proofs as Non-Interactive Zero-Knowledge proofs (NIZKs): while it is known that the Fiat-Shamir transform does not create UC-secure NIZKs, we nevertheless use it in our implementation. This is mainly because it is an assumption that is often made. E.g., [DKLs24], [CGG+20], [WMYC23] all claim UC security, but make use of Fiat-Shamir in their constructions. In addition, for e.g. the compressed Σ-protocol proof, no provably UC-secure non-interactive version exists (to the best of our knowledge). The only viable alternative for us would have been to prove security in a different security model in which the Fiat-Shamir transform would yield a secure NIZK. Given the prevalence of UC in the cryptographic protocols community, both to model protocols and define security, we believe our approach is simpler and makes the paper more accessible to a larger audience, at the expense of a heuristic assumption.

1.2 Related work

Threshold ECDSA with guaranteed output is not as thoroughly studied as the two-party or dishonest majority settings that are more suitable for cryptocurrency wallets [LN18, GG18, DKLs18, DKLs19, XAX+21, ANO+22, FMM+24, DGNP25] (and others). Although these works are highly efficient they do not achieve guaranteed output, making them unsuited for our area of application.

An early work on *robust* threshold ECDSA is [GJKR96, GJKR01], which presents (among other constructions) a computationally secure robust threshold signing protocol for the $t \leq (n-1)/3$ regime. In a nutshell, this classical protocol combines a verifiable secret-sharing scheme with a Zero-Knowledge proof in order to achieve robustness. [Pet21] presents an improved version of [GJKR96, GJKR01], tolerating a higher corruption threshold ($t < n/2$), albeit at the cost of only being able to achieve identifiable abort.

The work of [GS22a] presents another robust threshold ECDSA protocol. Like our work, [GS22a] guarantees output delivery but assumes an asynchronous network. In comparison to [GS22a] we rely on a synchronous network (as we assume broadcast), and avoid broadcasting ciphertexts. Instead, we only broadcast hashes of the commitment vectors $\mathtt{C}$ which is nevertheless sufficient when using a complaint-based mechanism for

distributing VSS commitments `C` and shares of each dealer over authenticated channels. [GS22a] also does not consider batched pre-signature generation.

Another work [WMYC23] builds a threshold signing protocol that has robustness for an honest majority and identifiable abort for a dishonest majority. Their construction sidesteps pair-wise multiplication of full-threshold sharings, thus allowing to exclude corrupted contributions after verifying the outcome of the multiplication protocol. In comparison to our work, they have to identify erroneous shares after signing fails, while our pre-processing phase always outputs well-formed pre-signatures. In follow-up work, [WMC24] improve communication efficiency and the delayed cheater identification after multiplications using a threshold version of the CL [CL15] cryptosystem. In comparison, our work does not require additional cryptosystem to perform the pre-signature generation. The recent work of [TX25] further improves on the CL based approach, by presenting a robust threshold ECDSA protocol requiring only three rounds. This work moreover presents an attack on [WMYC23] showing that [WMYC23] is insecure if the protocol continues after cheating identification. In particular, this implies that [WMYC23] cannot obtain robustness and cheater identification simultaneously. Another recent work utilizing the CL cryptosystem to obtain a threshold ECDSA protocol is that of [DGNP25]. Although their protocol is shown to be performant for large n, it only achieves identifiable abort.

Batched pre-signature generation has only received limited consideration in prior work. The work of [KU25] considers batch key generation for *key-independent* pre-signatures. This makes their work suitable for key management networks, where servers aid in signing for many different users, with different signing keys. However, [KU25] does not provide guaranteed output, and moreover relies on pseudorandom secret-sharing which is only viable for small n. Another work which considers batching is [GS24], albeit for Schnorr signatures. Threshold Schnorr signatures are in some ways an easier task, as they can be generated without multiplications of secrets.

The recent work [FMM+24] introduces a notion of 2PC-MPC and shows it to be scalable to a large number of parties. Robustness was shown to be possible as well in a follow-up work [FMM+27]. In a nutshell, 2PC-MPC is a two-party secure computation where a client acts as one party, and a large committee of MPC nodes acts as the other party. It is not clear how, or if, it is possible to use their techniques in order to get a robust threshold signing protocol for the $t \leq (n-1)/3$ setting.

Finally, see also [AHS20] for an excellent (although by now slightly outdated) overview over the state-of-the-art for ECDSA signing.

Our work compared to deployed threshold ECDSA solutions. Our main motivation is using threshold ECDSA to perform decentralized bridging. We therefore provide a short discussion on how our work compares to the protocols utilized by the Internet Computer [Sho25], THORChain [THO20a], and Partisia Blockchain [Blo24] as all of these companies utilize threshold ECDSA in some form for decentralized bridging.

THORChain [THO20a] use [GG18] as their protocol, making it susceptible to denial of service from malicious or non-responding participants. On the other hand, since [GG18] is secure against a dishonest majority, their protocol enjoys a great deal of flexibility in terms of the threshold. THORChain have published benchmarks [THO20b] showing that their implementation of [GG18] is performant for up to 30 parties placed in different geographic regions.

DFINITY's Internet Computer use [GS22a] as their threshold signature protocol [Sho25]. Benchmarks have been published on DFINITY's forums [eic25] indicating a setup comprising 34 nodes.

Finally, Partisia Blockchain also makes use of threshold ECDSA for bridging. In particular, a threshold signature protocol is used as part of their token bridge [Blo24].

The protocol they use is the computationally robust variant of [GJKR96][5]. Although no official benchmarks have been published, we can infer the number of supported nodes from historical committee sizes (because all committee nodes participate in the protocol). E.g., committee nr. 17 had 101 members.[6] As far as we can tell, the threshold ECDSA protocol currently deployed which supports the largest number of parties is that of Partisia Blockchain. In Section 6 we chose to compare our protocol against their implementation.

2 Preliminaries

We denote vectors $\vec{x}$ in lower-case and matrices $\vec{M}$ in upper-case letters. $\vec{M}_i$ denotes its i'th row, and $\vec{M}^j$ its j'th column. A particular entry of a matrix $\vec{M}$ is written as M_i^j. For a vector $\vec{x}$, x_i denotes the ith coordinate of said vector. Let $\mathbb{G}$ be a group of order q, then we use $\mathtt{G} \in \mathbb{G}$ to denote a group element. Given $\vec{\mathtt{G}} \in \mathbb{G}^n$, and $\vec{v} \in \mathbb{Z}_q^n$ then $\vec{v} \star \vec{\mathtt{G}} = v_1\mathtt{G}_1 + \cdots + v_n\mathtt{G}_n$ denotes an inner product. Parties are denoted as $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ for an $n \in \mathbb{N}$ and we assume that at most $t \leq (n-1)/3$ are malicious. When S is a finite set, we write $s \xleftarrow{\$} S$ to say that s is chosen from S uniformly at random. We use κ to denote the computational security parameter. For two strings $x, y \in \{0,1\}^*$ we write $x \,||\, y$ to denote their concatenation. We assume a synchronous communication pattern and let Δ denote the length of a round.

2.1 ECDSA

Let $E(\mathbb{Z}_p)$ be an elliptic curve over $\mathbb{Z}_p$ and $\mathbb{G} \subseteq E(\mathbb{Z}_p)$ be a subgroup of order q generated by the point $\mathtt{G} \in \mathbb{G}$. We assume that q is exponential in κ. Let $\mathsf{H}_{\mathsf{ECDSA}} : \{0,1\}^* \to \mathbb{Z}_q$ be a hash function and $F : \mathbb{G} \to \mathbb{Z}_q$ a "conversion function" defined as $F(\mathtt{R}) = r_x \mod q$ where $\mathtt{R} = (r_x, r_y) \in \mathbb{G}$. We will ignore the key generation algorithm, as a large part of our contribution focuses on developing an efficient variant for signing. The ECDSA signature scheme [JMV01] is a function $\mathsf{ECDSA.sign}$ for signing messages, and $\mathsf{ECDSA.verf}$ for verifying them, defined below. Correctness of the scheme can easily be verified. The security of ECDSA has been verified only in certain idealized models [FKP16, FKP17] and the results in [HK23] imply that this is unavoidable.

$\mathsf{ECDSA.sign}(m, \mathsf{sk})$: On input $m \in \{0,1\}^*$, $\mathsf{sk} \in \mathbb{Z}_q$, output signature $\sigma = (r, s)$ where:

- $r = F(\mathtt{R})$ with $k \xleftarrow{\$} \mathbb{Z}_q$ and $\mathtt{R} = k \cdot \mathtt{G}$.
- $s = (\mathsf{H}_{\mathsf{ECDSA}}(m) + r \cdot \mathsf{sk})/k$

$\mathsf{ECDSA.verf}(\sigma, m, \mathsf{pk})$: On input $\sigma = (r, s)$, $m \in \{0,1\}^*$ and $\mathsf{pk} = \mathsf{sk} \cdot \mathtt{G}$:

1. Compute $\mathtt{R} = (\mathsf{H}_{\mathsf{ECDSA}}(m)s^{-1}) \cdot \mathtt{G} + (rs^{-1}) \cdot \mathsf{pk}$
2. Output 1 or accept if $\mathtt{R} \neq \infty$ and $F(\mathtt{R}) = r$ and 0 or reject otherwise.

2.2 Non-Interactive Zero Knowledge Proofs

Zero-Knowledge proofs were first introduced by [GMR85]. Let $\mathcal{R}$ be an NP-relation for a language $\mathcal{L}$. A statement x is in the language $\mathcal{L}$ if there exists witness w such that $(x, w) \in \mathcal{R}$. In the definitions below, P, P^*, V and V^* all denote polynomial time algorithms.

[5] We have also confirmed this through private communication with Partisia Blockchain's developers.

[6] https://browser.partisiablockchain.com/contracts/04203b77743ad0ca831df9430a6be515195733ad91?tab=state

Definition 1 (Interactive Proofs)**.** An *interactive proof* Π concerns the interaction of two parties: A prover P, and verifier V. Both parties receive x as an input, and the prover in addition receives w as an input. Let T denote the transcript of the protocol, and $\langle P(x,w), V(x)\rangle \in \{0,1\}$ denote the output of V when interacting with P.

An Interactive Proof is called a *Honest-Verifier Zero-Knowledge proof* if it satisfies the following properties:

- **Completeness:** If $(x,w) \in \mathcal{R}$, then $\langle P(x,w), V(x)\rangle = 1$.
- **Knowledge Soundness:** If $\langle P^*(x), V(x)\rangle = 1$ with high probability for a statement x, and polynomial time P^*, then there exists another polynomial time algorithm E that can extract a w satisfying $(x,w) \in \mathcal{R}$, using black-box access to P^*.
- **Honest-Verifier Zero-Knowledge:** There exists a polynomial time *simulator* S that on input the statement x, can produce a transcript indistinguishable from real protocol transcripts.

In addition, the proof system is called public-coin if all messages sent by V are uniformly random and sampled by V, and if the output of $\langle P(x,w), V(x)\rangle$ only depends on the messages sent between P, V as well as x. An interactive honest-verifier public-coin Zero-Knowledge proof protocol can be made non-interactive Zero-Knowledge (NIZK) protocol using the Fiat-Shamir transform [FS87].

2.3 Verifiable Secret Sharing

A *Verifiable Secret Sharing* (VSS) scheme extends a secret-sharing scheme with additional information such that a share can be verified locally. A VSS scheme is a set of three procedures: A procedure for creating a secret sharing and the auxiliary information; a procedure for reconstructing a secret; and a procedure for verifying a secret share against the auxiliary information.

We consider Pedersen VSS [Ped92] specifically in this work. Secrets live in the group $\mathbb{Z}_q$, thus matching the signing key space of the ECDSA scheme. The auxiliary information are elements of the order q subgroup generated by a generator $\mathtt{G}$. More precisely, fix two generators $\mathtt{G}, \mathtt{H} \in \mathbb{G}$ such that the discrete logarithm $\log_{\mathtt{G}}(\mathtt{H})$ is not known. Pedersen's VSS works as follows:

$\mathsf{PSS.share}(s, d, n; r)$ where $s \in \mathbb{Z}_q$ is the secret to share and $n, d \in \mathbb{N}$. $r \in \mathbb{Z}_q$ is an optional parameter. If r is not passed, then choose it uniformly at random in $\mathbb{Z}_q$. Pick $f, g \in \mathbb{Z}_q[X]$ at random such that $\deg f = d$ and $\deg g = d$ and $f(0) = s$ and $g(0) = r$. Compute $\langle s\rangle = (s_1, s_2, \ldots, s_n), \langle r\rangle = (r_1, \ldots, r_n)$ and $\vec{\mathtt{C}} = (s\mathtt{G}+r\mathtt{H}, s_1\mathtt{G}+r_1\mathtt{H}, s_2\mathtt{G}+r_2\mathtt{H}, \ldots, s_d\mathtt{G}+r_d\mathtt{H})$ where $s_i = f(i)$ and $r_i = g(i)$ for $i = 1, \ldots, n$. The shares held by party P_i are s_i, r_i, while the public auxiliary information is $\vec{\mathtt{C}}$. We denote the output of $\mathsf{PSS.share}$ as $[\![s]\!]$ if $d = t$ and $[\![s]\!]_{2t}$ if $d = 2t$.

$\mathsf{PSS.recon}(S, d)$ where $S = (s_i, r_i)_{i\in I}$ with $I \subseteq \{1, \ldots, n\}$ and $|I| \geq d+1$ and $d \in \mathbb{N}$. Interpolate the polynomials f, g running through the points (i, s_i) and (i, r_i) for all $i \in I$, and output $f(0)$ if both are uniquely defined.

$\mathsf{PSS.verf}(s, r, \vec{\mathtt{C}}, i, d)$ where s, r are the shares to verify, $\vec{\mathtt{C}}$ the auxiliary information, $i \in [n]$ and $d \in \mathbb{N}$. Compute the Lagrange coefficients $\vec{\lambda}_i = (\lambda_0, \lambda_1, \ldots, \lambda_d) \in \mathbb{Z}_q^{d+1}$ for interpolating $f(i), g(i)$ of a degree d polynomial, using evaluation points $(0, 1, \ldots, d)$. Output 1 if $\vec{\lambda}_i \star \vec{\mathtt{C}} = s\mathtt{G} + r\mathtt{H}$ and 0 otherwise.

NIZK for Pedersen VSS. We will use a NIZK that allows P_i to reveal the $\mathtt{H}$-component $r_i\mathtt{H}$ of a commitment, and then prove that the remaining secret s_i it holds is consistent with $\vec{\mathtt{C}}$, and that it only appears in the $\mathtt{G}$-component.

Assume a party P (the prover) reveals $\mathtt{R} = r\mathtt{H}$ for a commitment $\mathtt{C} = x\mathtt{G} + r\mathtt{H}$ without revealing x or r. That is, P wants to prove knowledge of a witness for the relation:

$$\mathcal{R}_{\mathtt{POE}} = \{((\mathtt{G}, \mathtt{H}, \mathtt{C}, \mathtt{R}), (x, r)) \mid \mathtt{C} = x\mathtt{G} + r\mathtt{H} \wedge \mathtt{R} = r\mathtt{H}\}.$$

A proof for $\mathcal{R}_{\mathtt{POE}}$ can be constructed by performing two proofs for the standard Schnorr relation $\mathcal{R}_{\mathtt{Schnorr}}$ given below.

$$\mathcal{R}_{\mathtt{Schnorr}} = \{((\mathtt{G}, \mathtt{X}), x) \mid \mathtt{X} = x\mathtt{G}\}.$$

The first proof is done for $((\mathtt{G}, \mathtt{C} - \mathtt{R}), x)$; the second proof is done for $((\mathtt{H}, \mathtt{R}), r)$. This is sufficient to prove $\mathcal{R}_{\mathtt{POE}}$.

Batching proofs. Let $x_1, \ldots, x_\ell$ be P's witnesses and $\mathtt{G}$, $\mathtt{D}_1, \ldots, \mathtt{D}_\ell$ with $\mathtt{D}_i = x_i\mathtt{G}$ for some $\ell > 1$. The prover P can construct ℓ proofs for $\mathcal{R}_{\mathtt{Schnorr}}$ at once with the following protocol:

1. P chooses $a \xleftarrow{\$} \mathbb{Z}_q$, computes $\mathtt{A} = a\mathtt{G}$ and sends $\mathtt{A}$ to verifier V.
2. V chooses $\mathsf{ch} \xleftarrow{\$} \mathbb{Z}_q$ uniformly at random and sends it to P.
3. Both parties then define $\mathtt{D} = \sum_{i=1}^{\ell} \mathsf{ch}^i \mathtt{D}_i$, while P sets $x := \sum_{i=1}^{\ell} \mathsf{ch}^i x_i$.
4. P computes $z = a + x$ and sends z to V.
5. V accepts if $z\mathtt{G} = \mathtt{A} + \mathtt{D}$, otherwise it rejects.

The prover only sends one $\mathbb{G}$-element and one $\mathbb{Z}_q$-element. The proof can be made non-interactive using the standard Fiat-Shamir transform. One can easily show that this protocol is $\ell + 1$-special sound, based on the invertibility of Vandermonde matrices as long as $q > \ell$: assume you have $\ell + 1$ accepting transcripts $(\mathtt{A}, \mathsf{ch}_i, z_i)$ for $i \in [\ell + 1]$ with distinct challenges. Then it holds that

$$z_i\mathtt{G} = \mathtt{A} + \sum_{j=1}^{\ell} \mathsf{ch}_i^j \mathtt{D}_j = \mathsf{ch}_i^0 \mathtt{A} + \sum_{j=1}^{\ell} \mathsf{ch}_i^j \mathtt{D}_j.$$

Thus one can write $[z_1, \ldots, z_{\ell+1}]\mathtt{G} = \vec{V}[\mathtt{A}, \mathtt{D}_1, \ldots, \mathtt{D}_\ell]$ where $\vec{V}$ is the Vandermonde matrix for $\mathsf{ch}_1, \ldots, \mathsf{ch}_{\ell+1}$. Then, by invertibility of $\vec{V}$, multiplying all with $\vec{V}^{-1}$ always yields a solution.

The specific relation we will make use of in our work, is a combination of $\mathcal{R}_{\mathtt{POE}}$ and the batching technique above. We formalize this relation below:[7]

$$\mathcal{R}_{\mathtt{APOE}} = \{((\mathtt{G}, \mathtt{H}, (\mathtt{C}_i, \mathtt{R}_i)_{i\in[\ell]}), (x_i, r_i)_{i\in[\ell]}) \mid \forall i \in [\ell] : \ \mathtt{C}_i = x_i\mathtt{G} + r_i\mathtt{H} \wedge \mathtt{R}_i = r_i\mathtt{H}\}. \quad (1)$$

2.4 Hyper-invertible Matrices

A hyper-invertible matrix is a matrix such that every square sub-matrix is invertible.

[7] **A**mortized **P**roof **O**f (knowledge of) **E**xponent.

Definition 2 ([BTH08])**.** An r-by-c matrix $\overrightarrow{M}$ is *hyper-invertible* if for any index sets $R \subseteq [r], C \subseteq [c]$ with $|R| = |C| > 0$ the matrix $\overrightarrow{M}_R^C$ is invertible, where $\overrightarrow{M}_R$ denotes the matrix consisting of the rows $i \in R$, $\overrightarrow{M}^C$ denotes the matrix consisting of the columns $j \in C$, and $\overrightarrow{M}_R^C = (\overrightarrow{M}_R)^C$.

The authors of [BTH08] show that the following construction is a hyper-invertible matrix over a field $\mathbb{Z}_q$ as long as $q \geq 2n$:

Lemma 1. *Let $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n$ denote distinct elements in $\mathbb{Z}_q$ and consider the function $f : \mathbb{Z}_q^n \to \mathbb{Z}_q^n$ mapping $(x_1, \ldots, x_n)$ to $(y_1, \ldots, y_n)$ such that the points $(\beta_1, y_1), \ldots, (\beta_n, y_n)$ lie on a polynomial $g(\cdot)$ of degree $n-1$ defined by the points $(\alpha_1, x_1), \ldots, (\alpha_n, x_n)$. Due to linearity of Lagrange interpolation, f is linear and can be expressed as a matrix $\overrightarrow{M} = \{\lambda_{i,j}\}_{i,j\in[n]}$ where $\lambda_{i,j} = \prod_{k=1,k\neq j}^{n} \frac{\beta_i - \alpha_k}{\alpha_j - \alpha_k}$. Then $\overrightarrow{M}$ is a hyper-invertible matrix.*

Clearly, by choosing an $n \times (n-t)$ submatrix of the aforementioned construction, we still obtain a hyper-invertible matrix.

2.5 Merkle Trees

Let $\mathsf{H} : \{0,1\}^m \to \{0,1\}^{2\kappa}$ be a collision-resistant hash function, and let $\overrightarrow{s} = (s_1, \ldots, s_k)$ be a list of bit strings. A Merkle Tree [Mer88] on $\overrightarrow{s}$ is a tree where the i'th leaf is $\mathsf{H}(s_i)$ for $i = 1, \ldots, k$, and where an internal node is the hash of its two child nodes. For example, if $k = 4$, then the Merkle Tree hash of $\overrightarrow{s} = (s_1, s_2, s_3, s_4)$ is the hash

$$h = \mathsf{H}(\mathsf{H}[\mathsf{H}(s_1) \mid\mid \mathsf{H}(s_2)] \mid\mid \mathsf{H}[\mathsf{H}(s_3) \mid\mid \mathsf{H}(s_4)]).$$

We call h the Merkle root of the tree.

It is possible to efficiently prove that a certain leaf s' exists in the tree implied by h by providing a path from s' to h. E.g., to prove that s_3 exists in the tree rooted in h, one can supply the proof $\pi = (\mathsf{H}(s_4), \mathsf{H}(\mathsf{H}(s_1) \mid\mid \mathsf{H}(s_2)))$. In general, a proof contains $\log(k)$ elements from $\{0,1\}^{2\kappa}$, allowing a "compact" proof of membership.

The following three procedures will be utilized in our protocol:

$\underline{\mathsf{Merkle.hash}(\overrightarrow{s})}$ computes the Merkle Tree on $\overrightarrow{s} = (s_1, \ldots, s_k)$ and outputs the root h.

$\underline{\mathsf{Merkle.prov}(h, i, \overrightarrow{s})}$ given a root h, index i and values $\overrightarrow{s} = (s_1, \ldots, s_k)$, compute a proof π that $s_i \in \overrightarrow{s}$ is part of h.

$\underline{\mathsf{Merkle.verf}(h, i, \pi, s_i)}$ given a root h, index i, proof π and purported leaf node s_i, verifies that s_i is part of h at index i by recomputing the root using the proof π and checking that it is identical to h.

2.6 Universal Composability

We use a version of the universal composability (UC) framework [Can01] that respects global subroutines [BCH+20] for analyzing security of our protocols.

In UC, protocols are run by interactive Turing Machines (iTMs) called *parties* which we denote with a P. The *adversary* $\mathcal{A}$, which is also an iTM, can actively corrupt any of these parties and then gains full control over it, seeing messages that it obtains and choosing any messages that the corrupted party or parties send. In this work we only consider static corruptions, meaning that the choice of corruption is fixed from the beginning of the protocol.

The parties can communicate with each other directly and with so-called *ideal functionalities* (which themselves are iTMs). These ideal functionalities are denoted by $\mathcal{F}$ and abstractly describe cryptographic protocols with certain guarantees. As usual, we define

security with respect to an iTM $\mathcal{Z}$ called the *environment*. The environment provides inputs to the parties, receives outputs from them and controls $\mathcal{A}$.

To define security, let $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}}$ be the distribution of the output of an arbitrary $\mathcal{Z}$ when interacting with $\mathcal{A}$ in a real protocol instance Π. Furthermore, let $\mathcal{S}$ denote an *ideal world adversary* and $\text{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ be the distribution of the output of $\mathcal{Z}$ when interacting with parties which run with $\mathcal{F}$ instead of Π and where $\mathcal{S}$ takes care of adversarial behavior.

Definition 3. We say that Π UC-securely implements $\mathcal{F}$ if for any $\mathcal{A}$ there exists an iTM $\mathcal{S}$ such that for every environment $\mathcal{Z}$ the outputs of an execution $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}}$ are identical to an execution $\text{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ except with negligible probability.

UC with global subroutines. While UC allows to analyze security of protocols under arbitrary composition, it falls short of modeling functionalities that maintain a shared state for multiple sessions - such as random oracles. Therefore, we analyze our work in the *Universal Composition with Global Subroutines* (UCGS) model, proposed in [BCH+20].

Let Π be a protocol that makes use of a global subroutine $\mathcal{G}$ (or more such subroutines). UCGS describes how to transform $\Pi, \mathcal{G}$ into a standard UC-protocol $\mathsf{M}[\Pi, \mathcal{G}]$, in which Π, $\mathcal{F}$ and $\mathcal{Z}$ all have access to $\mathcal{G}$. Security then means that $\mathsf{M}[\Pi, \mathcal{G}]$ UC-emulates $\mathsf{M}[\mathcal{F}, \mathcal{G}]$. The global subroutine $\mathcal{G}$ has to fulfill the *regularity* condition: a subroutine is *regular* if only previously registered machines can speak with it and if it does not activate other machines outside of Π. Moreover, the subroutine is not allowed to spawn any new instances of the protocol calling it. Note that those conditions are trivially satisfied for a random oracle.

Definition 4. We say that Π UC-securely implements $\mathcal{F}$ in the presence of a global subroutine $\mathcal{G}$ if for any $\mathcal{A}$ there exists an iTM $\mathcal{S}$ such that for every environment $\mathcal{Z}$ the outputs of an execution $\text{EXEC}^{\mathcal{G}}_{\Pi,\mathcal{A},\mathcal{Z}}$ are identical to an execution $\text{EXEC}^{\mathcal{G}}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ except with negligible probability.

More details, as well as a composition theorem for UCGS and further examples, can be found in [BCH+20].

2.7 Ideal Functionalities

We assume the existence of a broadcast channel modeled by an ideal functionality $\mathcal{F}_{\mathtt{BC}}$. Due to its simplicity, we do not include it here.

Random Oracles. We use the formulation of [CC25] to model our global random oracle $\mathcal{F}_{\mathtt{RO}}$. In the approach of [CC25], only functionalities program the oracle. The calling functionality can initially be assigned control of the oracle and is then referred to as the controller of the session sid. Only registered controllers of sid can program points that have not been queried before. In this formulation of the global random oracle the programming access is controlled by the calling functionalities.

Functionality 1: $\mathcal{F}_{\mathtt{RO}}$ (global)

$\mathcal{F}_{\mathtt{RO}}$ interacts with a set of parties. For each session $\mathcal{F}_{\mathtt{RO}}$ can be controlled by a party with $\mathsf{sid} = \bot$ (functionalities) which has previously been assigned control. Let $\mathtt{map}_{\text{ctrl}}$ be an initially empty map storing controllers of each session and $\mathtt{li}^{\mathsf{sid}}_{\mathrm{H}}$ be an initially empty list for each sid. $\mathcal{F}_{\mathtt{RO}}$ samples from a finite space $\mathbb{Z}_q$.

Assign control Upon receiving $(\textsc{Control}, \mathsf{sid})$ from a functionality $\mathcal{F}$, add $(\mathsf{sid}, \mathcal{F})$ to $\mathtt{map}_{\text{ctrl}}$ and return $(\textsc{Ok}, \mathtt{li}^{\mathsf{sid}}_{\mathrm{H}})$.

Query On input $(\textsc{Query}, \mathsf{sid}, m)$ from a party P_i:

1. If P' is registered as the controller of this sid in $\mathtt{map}_{\text{ctrl}}$, output

(ProgramReq, sid, m) to P'.

2. Upon response (Program, sid, m, h) and if there is no record of the form $(m, \cdot) \in \mathtt{li}_{\mathrm{H}}^{\mathsf{sid}}$, then add (m, h) to $\mathtt{li}_{\mathrm{H}}^{\mathsf{sid}}$.
3. Find $(m, h) \in \mathtt{li}_{\mathrm{H}}^{\mathsf{sid}}$. If it is absent, generate $h \xleftarrow{\$} \mathbb{Z}_q$ and add the tuple to $\mathtt{li}_{\mathrm{H}}^{\mathsf{sid}}$. Then send (Query, sid, h) to the caller.

To not further complicate our functionalities we model access to $\mathcal{F}_{\mathtt{RO}}$ using a macro MacroProg. That is, if $\mathcal{F}$ is a functionality which uses $\mathcal{F}_{\mathtt{RO}}$, then the code in MacroProg is part of $\mathcal{F}$'s code. Every functionality that makes use of $\mathcal{F}_{\mathtt{RO}}$ will first execute **Init Control**, and programming is performed using the **Program** action.

Macro 1: MacroProg

If any of the processes return $\perp$ abort the execution inside the calling functionality. Every functionality keeps track of initially empty list $\mathtt{li}_{\mathrm{H}}^{\mathsf{sid}}$. We only allow programming from the ideal process adversary ($\mathcal{S}$) with same sid as the calling functionality.

Init control send (Control, sid) to $\mathcal{F}_{\mathtt{RO}}$ and wait for its response m. If $m = (\mathrm{Ok}, \mathtt{li}_{\mathrm{H}}^{\mathsf{sid}})$ and $\mathtt{li}_{\mathrm{H}}^{\mathsf{sid}} \neq \emptyset$ output $\perp$.

Program On input (ProgramReq, sid, m) from $\mathcal{F}_{\mathtt{RO}}$. Send (Program, sid, m) to $\mathcal{S}$ and wait for its response h. Forward (Program, sid, m, h) to $\mathcal{F}_{\mathtt{RO}}$.

NIZK Proofs. In this work, we use the UC NIZK functionality $\mathcal{F}_{\mathtt{NIZK}}$ from [CSW22] for arbitrary NP relations $\mathcal{R}$.

Functionality 2: $\mathcal{F}_{\mathtt{NIZK}}$

$\mathcal{F}_{\mathtt{NIZK}}$ is parameterized by an NP relation $\mathcal{R}$ and interacts with parties P, V and the simulator $\mathcal{S}$. $\mathcal{S}$ may corrupt either of the parties.

Proof On input (Prove, sid, x, w) from a party P and if $(x, w) \in \mathcal{R}$:

1. Send (Prove, sid, P, x) to $\mathcal{S}$.
2. Upon receiving (Proof, sid, π) from $\mathcal{S}$, store (sid, x, w, π).
3. Send (Proof, sid, π) to P.

Verification On input (Verify, sid, x, π) from a party V:

- If (sid, x, w, π) is stored, then return (Verification, sid, x, π, d) to V, where $d = 1$ if $(x, w) \in \mathcal{R}$ and $d = 0$ otherwise.
- Else, send (Verify, sid, V, x, π) to $\mathcal{S}$. Upon receiving (Witness, sid, w) from $\mathcal{S}$ store (sid, x, w, π) and return (Verification, sid, x, π, d) to V, where $d = 1$ if $(x, w) \in \mathcal{R}$ and $d = 0$ otherwise.

Throughout this work, we use $\mathcal{F}_{\mathtt{NIZK}}$ as a building block in multiple protocols. We do this as we prove our protocols UC-secure, given that it is the best framework for security proofs for our constructions. However, our implementations of $\mathcal{F}_{\mathtt{NIZK}}$ use interactive proofs that are being made non-interactive via the Fiat-Shamir transform. We make the heuristic assumption that the security does not degrade when instantiating $\mathcal{F}_{\mathtt{NIZK}}$ with Fiat Shamir-based NIZKs, but are well aware that a provably secure instantiation would have to use e.g. the Fischlin transform [Fis05, CL24]. To the best of our knowledge, no existing works extend [Fis05] to Compressed Σ-protocols, the technique that we use for proofs of correct multiplication.

3 Threshold Signing

Let psig be a *pre-signature* of the form $\mathsf{psig} = (\mathtt{R}, [\![h]\!], [\![\gamma]\!], [\![\rho]\!])$ where $\mathtt{R} = k\mathtt{G}, \gamma = h \cdot k$ and $\rho = h \cdot \mathsf{sk}$. As mentioned in the introduction, psig is all that is needed to produce a signature on a message m that is valid under the public key $\mathsf{pk} = \mathsf{sk}\mathtt{G}$. In the following section we focus on the online (or signing) phase of our protocol.

3.1 Ideal ECDSA Threshold Signing

We begin by presenting an ideal functionality $\mathcal{F}_{\mathtt{TSS}}$ for threshold signing with ECDSA in Functionality 3. The functionality is similar in style to [ANO+22], with some differences. Firstly, our functionality contains only the two commands for the protocol parties: `KeyGen` and `Sign`. This both simplifies the presentation while at the same time remaining sufficient for our use-case. Note that `KeyGen` is responsible for both generating the signing key (in particular the public key) as well as t pre-signatures. The second difference is that we restrict the actions allowed by the ideal adversary $\mathcal{S}$. In particular, we will not allow the adversary to abort the computation or attempt forgeries. This is because all our protocols will be secure with guaranteed output, and so protocol aborts are not possible.

Functionality 3: $\mathcal{F}_{\mathtt{TSS}}$

Let $\mathcal{P} = (P_1, \ldots, P_n)$ be the set of parties and $\mathcal{S}$ the ideal adversary corrupting $t \leq (n-1)/3$ of the parties in $\mathcal{P}$. Let q be a prime, $\mathbb{G}$ be a group and $\mathtt{G} \in \mathbb{G}$ the generator of an order q subgroup. Let $\mathsf{H}_{\mathsf{ECDSA}}$ be a cryptographic hash function as used in ECDSA. Let List be an initially empty internal list. The functionality uses Macro MacroProg.

KeyGen

On input (KeyGen, sid) from at least $n - t$ parties in $\mathcal{P}$ and where sid has not been used yet:

1. Pick $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_q$ and compute $\mathsf{pk} \leftarrow \mathsf{sk}\mathtt{G}$.
2. Pick $\vec{r} = (r_1, \ldots, r_t) \xleftarrow{\$} \mathbb{Z}_q^t$ and compute $\vec{\mathtt{B}} = (r_1\mathtt{G}, \ldots, r_t\mathtt{G})$.
3. Send $\mathsf{pk}, \vec{\mathtt{B}}$ to $\mathcal{S}$ and all parties in $\mathcal{P}$.
4. Store $(\mathsf{sid}, \mathsf{sk}, \vec{r}, 1, t)$

Sign

On input (Sign, sid, m, ζ) from $t + 2$ or more parties where $(\mathsf{sid}, \mathsf{sk}, \vec{r}, \zeta, t)$ was previously stored and $\zeta \leq t$:

1. Set $k \leftarrow \mathsf{sampleK}(\mathsf{sid}, m, \zeta)$ and compute $\mathtt{R} = k\mathtt{G}$.
2. Compute $r = F(\mathtt{R})$ and $s = (\mathsf{H}_{\mathsf{ECDSA}}(m) + r \cdot \mathsf{sk})/k$.
3. Replace $(\mathsf{sid}, \mathsf{sk}, \vec{r}, \zeta, t)$ with $(\mathsf{sid}, \mathsf{sk}, \vec{r}, \zeta + 1, t)$
4. Output (r, s).

Re-randomization

On input (Probe, sid, m, ζ) by $\mathcal{S}$ and if **KeyGen** has been run for sid:

1. Let $k \leftarrow \mathsf{sampleK}(\mathsf{sid}, m, \zeta)$.

2. Let $\delta \leftarrow k - r_\zeta \bmod q$.
3. Return $(\textsc{Probe}, \mathsf{sid}, \delta)$.

Define the following procedure $\mathsf{sampleK}(\mathsf{sid}, m, \zeta)$:

1. If $(\mathsf{sid}, m, \zeta, \cdot) \notin \mathsf{List}$:
   (a) Sample $h \in \mathbb{Z}_q$ uniformly at random.
   (b) Add $(\mathsf{sid}, m, \zeta, h)$ to List.
2. Find $(\mathsf{sid}, m, \zeta, h) \in \mathsf{List}$ and return h.

An additional difference to standard $\mathcal{F}_{\mathtt{TSS}}$ functionalities is that ours does not have a uniformly random nonce k. Instead, the adversary first sees $\vec{\mathtt{B}}$ and can then shift the underlying randomness r_ζ by an additive (but unpredictable) term δ, unique for every (m, ζ). Choosing m freely based on $\vec{\mathtt{B}}$ leads to attacks as demonstrated in [GS22b]. Making $\mathtt{R}$ fully independent of m is only possible for non-interactive ECDSA signing if there would exist an external perfect source of randomness to re-randomize the pre-signature $\mathtt{R}$. We opt for the second-best approach and re-randomize $\vec{\mathtt{B}}$ based on an output of a Random Oracle call which produces the unpredictable but public additive term δ, unique for every (m, ζ). [GS22b, Section 8] recommends this strategy as a remedy to their attacks.

3.2 Threshold Signing Protocol

We present now a protocol that securely realizes $\mathcal{F}_{\mathtt{TSS}}$ in a $\mathcal{F}_{\mathtt{key\text{-}gen}}$-hybrid model. Concretely, we assume parties can call an ideal key/pre-signature generation functionality, an instantiation of which we provide in the next section.

The ideal key generation functionality is shown below. On invocation from a sufficient number of parties, it generates a signing/verification key pair where sk is never revealed. It also generates a fixed number of pre-signatures and then returns all of these. The functionality allows the adversary to pick its own shares using the Adjust procedure in the MacroAdjust macro below. Note, however, that neither pk nor $k^{(\zeta)}$ can be biased by the adversary. The functionality also guarantees output delivery.

Macro 2: MacroAdjust—Adversarial share influence

Let $\mathcal{S}$ be the ideal adversary having corrupted a set I of size $t \leq (n-1)/3$ parties in $\mathcal{P}$.

Procedure $\mathsf{Adjust}(\llbracket r^{(1)} \rrbracket_d, \ldots, \llbracket r^{(\ell)} \rrbracket_d, \mathtt{fl}_{\mathrm{zero}})$:

1. For $\iota = 1, \ldots, \ell$ write $\llbracket r^{(\iota)} \rrbracket_d = (\langle r^{(\iota)} \rangle_d, \langle a^{(\iota)} \rangle_d, \vec{\mathtt{C}}^{(\iota)})$.
2. Send $(\textsc{Shares}, \mathsf{sid}, (\vec{\mathtt{C}}^{(\iota)}, (r_i^{(\iota)}, a_i^{(\iota)})_{i \in I})_{\iota \in [\ell]})$ to $\mathcal{S}$.
3. Obtain $(\textsc{Diff}, \mathsf{sid}, (f^{(\iota)}, g^{(\iota)})_{\iota \in [\ell]})$ from $\mathcal{S}$ where $f^{(\iota)}, g^{(\iota)}$ are polynomials of degree at most d. If $\mathtt{fl}_{\mathrm{zero}} = 1$ then we require that $f^{(\iota)}(0) = 0$. If $\mathtt{fl}_{\mathrm{zero}} = 2$ then we require that $f^{(\iota)}(0) = g^{(\iota)}(0) = 0$.
4. For $\iota \in [\ell]$:
   - Set $\mathtt{D}_i^{(\iota)} := \mathtt{C}_i^{(\iota)} + f^{(\iota)}(i)\mathtt{G} + g^{(\iota)}(i)\mathtt{H}$ for $i \in \{0, \ldots, d\}$.
   - Set $s_i^{(\iota)} := r_i^{(\iota)} + f^{(\iota)}(i)$ and $b_i^{(\iota)} := a_i^{(\iota)} + g^{(\iota)}(i)$ for $i \in [n]$.
   - Write $\llbracket s^{(\iota)} \rrbracket_d := (\langle s^{(\iota)} \rangle_d, \langle b^{(\iota)} \rangle_d, \vec{\mathtt{D}}^{(\iota)})$.
5. Return $(\llbracket s^{(1)} \rrbracket_d, \ldots, \llbracket s^{(\ell)} \rrbracket_d)$.

Functionality 4: $\mathcal{F}_{\texttt{key-gen}}$

Let $\mathcal{P} = (P_1, \ldots, P_n)$ be the set of parties and $\mathcal{S}$ the ideal adversary corrupting $t \leq (n-1)/3$ parties in $\mathcal{P}$. Let q be a prime, $\mathbb{G}$ be a group and $\mathtt{G}, \mathtt{H} \in \mathbb{G}$ be generators of an order q subgroup. Uses MacroAdjust.

Key and Pre-signature generation

On input (KeyGen, sid) from $n - t$ parties:

1. Pick $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_q$ and compute $\mathsf{pk} \leftarrow \mathsf{sk}\mathtt{G}$.
2. Output (Public Key, sid, pk) to all parties.
3. For $\zeta = 1, \ldots, t$, do
   (a) Pick $k^{(\zeta)}, \overline{h}^{(\zeta)} \xleftarrow{\$} \mathbb{Z}_q^2$.
   (b) Output (Presigs, sid, $k^{(\zeta)}\mathtt{G}$) to $\mathcal{S}$.
   (c) Compute $[\![\overline{h}^{(\zeta)}]\!]_t \leftarrow \mathsf{PSS.share}(\overline{h}^{(\zeta)}, t, n)$ and run $([\![h^{(\zeta)}]\!]_t) \leftarrow \mathsf{Adjust}([\![\overline{h}^{(\zeta)}]\!]_t, 0)$
   (d) Compute
   $$[\![\gamma^{(\zeta)}]\!]_t \leftarrow \mathsf{PSS.share}(h^{(\zeta)} \cdot k^{(\zeta)}, t, n)$$
   $$[\![\rho^{(\zeta)}]\!]_t \leftarrow \mathsf{PSS.share}(h^{(\zeta)} \cdot \mathsf{sk}, t, n).$$
   (e) Run $([\![\gamma^{(\zeta)}]\!]_t, [\![\rho^{(\zeta)}]\!]_t) \leftarrow \mathsf{Adjust}([\![\gamma^{(\zeta)}]\!]_t, [\![\rho^{(\zeta)}]\!]_t, 1)$.
   (f) Set $\mathsf{psig}^{(\zeta)} \leftarrow (k^{(\zeta)}\mathtt{G}, [\![h^{(\zeta)}]\!], [\![\gamma^{(\zeta)}]\!], [\![\rho^{(\zeta)}]\!])$.
4. Output (Presigs, sid, $(\mathsf{psig}^{(\zeta)})_{\zeta=1}^{t}$)) to all parties.

Our signing protocol $\Pi_{\texttt{sign}}$, which as mentioned works in the $\mathcal{F}_{\texttt{key-gen}}$-hybrid is fairly simple. It outlines the procedure for signing based on pre-signatures, and derives the unpredictable δ in the process. In addition, during the openings of the sharings, parties also check if the openings were made correctly. For the security proof to work, parties can abort in the protocol but honest parties will never do it.

Protocol 1: $\Pi_{\texttt{sign}}$

This protocol runs with the parties $\mathcal{P} = (P_1, \ldots, P_n)$, and the $\mathcal{F}_{\texttt{key-gen}}, \mathcal{F}_{\texttt{RO}}$ functionalities and a broadcast channel. Let q be a prime, $\mathbb{G}$ be a group and $\mathtt{G} \in \mathbb{G}$ the generator of an order q subgroup. A party P_i may at any point broadcast (Leave, sid, i) upon which all receiving parties mark P_i as corrupt and replace its input shares with default dummy values.

KeyGen

On input (KeyGen, sid), each P_i sends (KeyGen, sid) to $\mathcal{F}_{\texttt{key-gen}}$ to obtain the public key pk and t pre-signatures $(\mathsf{psig}^{(\zeta)})_{\zeta=1}^{t}$.

Sign

On input (Sign, sid, m, ζ) for a message m where either $\zeta = 1$ or the last input Sign had $\zeta - 1$ as pre-signature identifier and $\zeta \leq t$, each party P_i does the following:

1. Let $\mathsf{psig}^{(\zeta)} = (\mathtt{R}, [\![h]\!], [\![\gamma]\!], [\![\rho]\!])$ be the current pre-signature.

2. Send $(\text{QUERY}, \mathsf{sid}, \mathsf{pk} \mid\mid m \mid\mid \mathtt{R})$ to $\mathcal{F}_{\mathtt{RO}}$ to obtain $\delta \in \mathbb{Z}_q$.
3. Compute $r \leftarrow F(\mathtt{R} + \delta\mathtt{G})$, $[\![s']\!] \leftarrow \mathsf{H}_{\mathsf{ECDSA}}(m)[\![h]\!] + r[\![\rho]\!]$, and $[\![\overline{\gamma}]\!] \leftarrow [\![\gamma]\!] + \delta[\![h]\!]$.
4. P_i sends its shares of $[\![s']\!], [\![\overline{\gamma}]\!]$ to each other party P_j. Denote these shares as $s'_i, r_i^{(s')}$ for $[\![s']\!]$ and $\overline{\gamma}_i, r_i^{(\overline{\gamma})}$ for $[\![\overline{\gamma}]\!]$.
5. On receiving $s'_j, r_j^{(s')}, \overline{\gamma}_j, r_j^{(\overline{\gamma})}$ from P_j, party P_i runs $\mathsf{PSS.verf}(s'_j, r_j^{(s')}, \vec{\mathsf{S}}', j, t)$ and $\mathsf{PSS.verf}(\overline{\gamma}_j, r_j^{(\overline{\gamma})}, \vec{\Gamma}, j, t)$ where $\vec{\Gamma}$ is the commitment vector for $[\![\overline{\gamma}]\!]$, and $\vec{\mathsf{S}}'$ is the commitment vector for $[\![s']\!]$.
6. If both checks return 1, then P_i accepts the received shares s'_j, $\overline{\gamma}_j$, otherwise it rejects them.
7. When P_i has received $t+2$ valid shares s'_j, $\overline{\gamma}_j$, it recovers s', $\overline{\gamma}$ using $\mathsf{PSS.recon}$ and outputs $(r, s) := (r, s'\overline{\gamma}^{-1})$.

3.3 Analysis

Complexity. The $\Pi_{\mathtt{sign}}$ protocol requires a total of 4 field elements per party per signature to be sent to each other party, requiring a total of $4n^2$ communication (or $4n$ if only one party receives the signature). Only one round of communication is required, making signing non-interactive.

A moderate requirement exists on local computation. In particular, each party have to perform at least $2(t+2)(t+1)$ scalar multiplications. This is because each party has to verify at least $2(t+2)$ commitments, and each verification requires an interpolation in the exponent, which can be done using $t+1$ scalar multiplications and t curve additions. One can alternatively use Berlekamp-Welch to directly decode the secret from all shares instead.

Security. We will now prove the following Theorem:

Theorem 1. *The protocol $\Pi_{\mathtt{sign}}$ UC-securely implements $\mathcal{F}_{\mathtt{TSS}}$ in the $\mathcal{F}_{\mathtt{key\text{-}gen}}, \mathcal{F}_{\mathtt{RO}}$-hybrid model and using broadcast with security against any polynomial-time attacker statically and actively corrupting at most $t \leq (n-1)/3$ parties, assuming hardness of the Discrete Logarithm problem in $\mathbb{G}$.*

In the proof (and all following proofs in the next sections), we construct a simulator $\mathcal{S}$ that itself emulates instances of $\mathcal{F}_{\mathtt{key\text{-}gen}}, \mathcal{F}_{\mathtt{RO}}$ and interacts with $\mathcal{F}_{\mathtt{TSS}}$ while "simulating" honest parties towards the corrupted parties controlled by an adversary $\mathcal{A}$. We simulate the responses of random oracle queries in the protocol using $\mathcal{F}_{\mathtt{TSS}}$ and let the hybrid functionality $\mathcal{F}_{\mathtt{key\text{-}gen}}$ output sharings for key and pre-signatures that are consistent with $\mathcal{F}_{\mathtt{TSS}}$. The signature can then be made consistent by letting the simulator choose s' at random, setting $\overline{\gamma} = s'/s$ and generating consistent shares using knowledge of $\log_{\mathtt{G}}(\mathtt{H})$ which the simulator has.

Proof. To prove Theorem 1, we will construct a simulator $\mathcal{S}$ which will, with access to $\mathcal{F}_{\mathtt{TSS}}$, generate a transcript that is indistinguishable from $\Pi_{\mathtt{sign}}$ in the presence of a real adversary $\mathcal{A}$. We will also assume that $\mathcal{S}$ knows the discrete logarithm $\log_{\mathtt{G}}(\mathtt{H})$ in the group $\mathbb{G}$.

Simulator: Upon being notified by $\mathcal{A}$ which parties I it will corrupt, I is forwarded to $\mathcal{F}_{\mathtt{TSS}}$ by $\mathcal{S}$. It sets up an instance of $\mathcal{F}_{\mathtt{key\text{-}gen}}$ which $\mathcal{S}$ simulates. $\mathcal{S}$ thereafter essentially runs the protocol $\Pi_{\mathtt{sign}}$ on dummy inputs, with the following changes:

- For each honest party, simulate sending $(\text{KEYGEN}, \mathsf{sid})$ to $\mathcal{F}_{\texttt{key-gen}}$. Upon input $(\text{KEYGEN}, \mathsf{sid})$ from any adversarially controlled party to $\mathcal{F}_{\texttt{key-gen}}$, send $(\text{KEYGEN}, \mathsf{sid})$ in the name of this party to $\mathcal{F}_{\texttt{TSS}}$. Upon obtaining $\mathsf{pk}, \vec{\mathtt{B}}$ from $\mathcal{F}_{\texttt{TSS}}$ $\mathcal{S}$ makes $\mathcal{F}_{\texttt{key-gen}}$ output pk to every dishonest party. The shares of each psig are simulated for random input values, while each $\mathtt{R}^{(\zeta)}$ in $\mathsf{psig}^{(\zeta)}$ is set to be $\mathtt{B}_\zeta$ from $\mathcal{F}_{\texttt{TSS}}$. Moreover, we set the commitments in $[\![h]\!]_t, [\![\gamma]\!]_t, [\![\rho]\!]_t$ to be random group elements in $\mathbb{G}$.

- While simulating Step 2 of **Sign** we obtain the $\mathcal{F}_{\texttt{RO}}$ programming query and forward it to $\mathcal{F}_{\texttt{TSS}}$, i.e. on input $\mathsf{pk} \mid\mid m \mid\mid \mathtt{R}$ to $\mathcal{F}_{\texttt{RO}}$ query $(\text{PROBE}, \mathsf{sid}, m, \zeta)$ to $\mathcal{F}_{\texttt{TSS}}$ and respond with δ obtained from $\mathcal{F}_{\texttt{TSS}}$, which programs $\mathcal{F}_{\texttt{RO}}$ for this input to give the output. If any such query to $\mathsf{pk} \mid\mid m \mid\mid \mathtt{R}$ was made to $\mathcal{F}_{\texttt{RO}}$ before pk is available, respond with a random output. If any such query was made for pk which coincides with pk generated by $\mathcal{F}_{\texttt{TSS}}$, then abort the simulation.

- Upon start of **Sign** send $(\text{SIGN}, \mathsf{sid}, m, \zeta)$ to $\mathcal{F}_{\texttt{TSS}}$ for any dishonest party and obtain (r, s).

- $\mathcal{S}$ chooses a value s' for reconstruction uniformly at random, and $\overline{\gamma}$ such that $s = s'/\overline{\gamma}$. For the shares of $[\![s']\!]$, $[\![\overline{\gamma}]\!]$ that we send in the protocol in Step 4 for the honest parties, observe that $\mathcal{S}$ has all adversarial shares, and that we can hence fix (at least) one honest simulated share such that the reconstructed values will coincide with $s', \overline{\gamma}$. Moreover, since $\mathcal{S}$ knows $\log_{\mathtt{G}}(\mathtt{H})$ it can also generate consistent $r_j^{(s')}, r_j^{(\overline{\gamma})}$ such that the Pedersen commitments $\vec{\mathtt{S}}', \vec{\Gamma}$ of the verifiable secret sharings $[\![s']\!]$, $[\![\overline{\gamma}]\!]$ are opened correctly.

- For each adversarial share of $[\![s']\!]$, $[\![\overline{\gamma}]\!]$, let $s'_j, \overline{\gamma}_j, r_j^{(s')}, r_j^{(\overline{\gamma})}$ be the values that $\mathcal{A}$ is supposed to send. Assume that $\mathcal{A}$ sends, for P_j, values that disagree with those expected shares but that still make $\mathsf{PSS.verf}$ for $[\![s']\!]$, $[\![\overline{\gamma}]\!]$ output 1. Moreover, let these be among the first $t+2$ shares that a simulated honest party receives. Then $\mathcal{S}$ makes the simulation abort.

Clearly, the simulator $\mathcal{S}$ runs in polynomial time as it essentially runs a version of the protocol. We will now argue that its behavior is indistinguishable from $\Pi_{\texttt{sign}}$ with $\mathcal{A}$.

First, observe that the secret shares that the adversary sees during the protocol, the pre-signatures as well as the openings of $[\![s']\!]$, $[\![\overline{\gamma}]\!]$ are identically distributed as in the protocol. This is because Pedersen commitments are perfectly hiding, hence they are also uniform elements in the group generated by $\mathtt{G}$ in the protocol. Moreover, the openings sent by the simulator correctly open the commitments and allow reconstruction of the signature.

Let τ_I be the distribution of the outputs of the honest parties as well as of all values sent to the dishonest parties or otherwise observed by $\mathcal{A}$ when run with $\mathcal{S}$. We similarly define τ_R for when $\mathcal{A}$ runs with the actual protocol $\Pi_{\texttt{sign}}$.

Let τ_1 be like τ_I, except that $\mathcal{S}$ does not abort when a query to $\mathcal{F}_{\texttt{RO}}$ was made that contained pk before pk was defined. In this case, the adversary can query $\mathcal{F}_{\texttt{RO}}$ on the same input later and observe the difference. Since pk is chosen by $\mathcal{F}_{\texttt{key-gen}}$ uniformly in $\langle \mathtt{G} \rangle$, which has size q that is exponential in the security parameter, the difference between τ_I and τ_1 must be negligible.

Let τ_2 be the same distribution as τ_1, but where instead of running $\mathcal{S}$ we now run the actual $\Pi_{\texttt{sign}}$ with $\mathcal{A}$ (including using $\mathcal{F}_{\texttt{RO}}$ outputs not coming from $\mathcal{F}_{\texttt{TSS}}$), but with the following change: If any honest party obtains $t+2$ valid openings which only reconstruct to a degree-$t+1$ polynomial but not a degree-t polynomial, then the honest party aborts. Observe that τ_1 and τ_2 abort for the same event, while all other values observed by $\mathcal{A}$ as well as honest outputs still have the same distribution. But importantly, τ_2 does not require knowledge of $\log_{\mathtt{G}}(\mathtt{H})$.

Consider the difference between τ_R and τ_2. This is exactly the situation where τ_2 aborts but honest parties in τ_R simply output a value that might be incorrect. Consider any distinguisher between τ_R and τ_2, then it must make τ_2 abort. Then, we can use $\mathcal{A}$ used by the distinguisher which makes τ_2 abort to solve a Discrete Logarithm challenge for $\mathbb{G}$ by setting $\mathtt{G}, \mathtt{H}$ for the protocol according to the DLog challenge. Then, before abort, we can use all $t+2$ openings together with Lagrange interpolation to find 2 openings for one of the Pedersen commitments. This then allows to reconstruct the DLog challenge. □

4 Pre-processing pre-signatures

This section presents our pre-processing protocol in multiple stages. We will begin by presenting the instantiation of $\mathcal{F}_{\mathtt{key\text{-}gen}}$ from the previous section. This protocol, dubbed $\Pi_{\mathtt{key\text{-}gen}}$, assumes access to a functionality $\mathcal{F}_{\mathtt{mul}}$, shown below. This functionality itself builds on another functionality called $\mathcal{F}_{\mathtt{rand}}$. We provide instantiations of $\mathcal{F}_{\mathtt{mul}}$ and $\mathcal{F}_{\mathtt{rand}}$ to complete our protocol descriptions.

$\mathcal{F}_{\mathtt{rand}}$ obtains a parameter ℓ as input. It can then either output ℓ consistent Pedersen sharings of random values of degree t, or ℓ so-called double sharings. The latter are pairs of Pedersen sharings with the same secret, but shared with degree t as well as $2t$. The functionality is described in Functionality 5.

Functionality 5: $\mathcal{F}_{\mathtt{rand}}$

Let $\mathcal{P} = (P_1, \ldots, P_n)$ be the set of parties and $\mathcal{S}$ the ideal adversary corrupting a set I of size $t \leq (n-1)/3$ parties in $\mathcal{P}$. For a sharing $\langle x \rangle$ we let x_i denote the share of party P_i. The functionality also uses the procedure Adjust introduced in $\mathcal{F}_{\mathtt{key\text{-}gen}}$. Uses MacroAdjust.

Generate random committed shares

On input $(\textsc{Rand}, \mathsf{sid}, \ell)$ from all honest parties with $\ell \in \mathbb{N}^+$.

1. For $\iota = 1, \ldots, \ell$ pick $r^{(\iota)} \xleftarrow{\$} \mathbb{Z}_q$ and run $[\![r^{(\iota)}]\!]_t \leftarrow \mathsf{PSS.share}(r^{(\iota)}, t, n)$.
2. Set $([\![r^{(1)}]\!]_t, \ldots, [\![r^{(\ell)}]\!]_t) \leftarrow \mathsf{Adjust}([\![r^{(1)}]\!]_t, \ldots, [\![r^{(\ell)}]\!]_t, 0)$.
3. Write $[\![r^{(\iota)}]\!]_t = (\langle r^{(\iota)} \rangle_t, \langle a^{(\iota)} \rangle_t, \vec{\mathtt{C}}^{(\iota)})$.
4. Output $(\textsc{Share}, \mathsf{sid}, (r_i^{(\iota)}, a_i^{(\iota)}, \vec{\mathtt{C}}^{(\iota)})_{\iota \in [\ell]})$ to each honest P_i.

Generate random committed double shares

On input $(\textsc{DblRand}, \mathsf{sid}, \ell)$ from all honest parties with $\ell \in \mathbb{N}^+$

1. For $\iota = 1, \ldots, \ell$ pick $r^{(\iota)} \xleftarrow{\$} \mathbb{Z}_q$ and set $[\![r^{(\iota)}]\!]_t \leftarrow \mathsf{PSS.share}(r^{(\iota)}, t, n)$.
2. Set $([\![s^{(1)}]\!]_t, \ldots, [\![s^{(\ell)}]\!]_t) \leftarrow \mathsf{Adjust}([\![r^{(1)}]\!]_t, \ldots, [\![r^{(\ell)}]\!]_t, 0)$. Denote the secret of $[\![s^{(\iota)}]\!]_t$ as $s^{(\iota)}$ and its randomness as $\tilde{r}^{(\iota)}$.
3. For $\iota = 1, \ldots, \ell$ set $[\![s^{(\iota)}]\!]_{2t} \leftarrow \mathsf{PSS.share}(s^{(\iota)}, 2t, n; \tilde{r}^{(\iota)})$.
4. Set $([\![s^{(1)}]\!]_{2t}, \ldots, [\![s^{(\ell)}]\!]_{2t}) \leftarrow \mathsf{Adjust}([\![s^{(1)}]\!]_{2t}, \ldots, [\![s^{(\ell)}]\!]_{2t}, 2)$
5. Write $[\![s^{(\iota)}]\!]_t = (\langle s^{(\iota)} \rangle_t, \langle a^{(\iota)} \rangle_t, \vec{\mathtt{C}}^{(\iota)})$ and $[\![s^{(\iota)}]\!]_{2t} = (\langle \widehat{s}^{(\iota)} \rangle_{2t}, \langle \widehat{a}^{(\iota)} \rangle_{2t}, \vec{\mathtt{D}}^{(\iota)})$
6. Output $(\textsc{Share}, \mathsf{sid}, (s_i^{(\iota)}, \widehat{s}_i^{(\iota)}, a_i^{(\iota)}, \widehat{a}_i^{(\iota)}, \vec{\mathtt{C}}^{(\iota)}, \vec{\mathtt{D}}^{(\iota)})_{\iota \in [\ell]})$ to each honest P_i.

As in $\mathcal{F}_{\mathtt{key\text{-}gen}}$, we allow the adversary in $\mathcal{F}_{\mathtt{rand}}$ to choose the shares of dishonest parties. We again model this via the Adjust procedure, but this time allow attackers to also

additively bias the secret (as long as this does not break the correlation for random double shares). This does not pose a problem, as the generated secret will remain uniformly random under additive changes.

The functionality $\mathcal{F}_{\texttt{mul}}$ allows the same sampling of random sharings as $\mathcal{F}_{\texttt{rand}}$. Additionally, given a set of previously generated t pairs of Pedersen-shared values as inputs, it outputs t Pedersen sharings of the products. The functionality is described in Functionality 6. We again let the adversary pick the shares of dishonest parties, while ensuring correctness of the output sharings towards the honest parties.

Functionality 6: $\mathcal{F}_{\texttt{mul}}$

Let $\mathcal{P} = (P_1, \ldots, P_n)$ be the set of parties and $\mathcal{S}$ the ideal adversary corrupting a set I of size $t \leq (n-1)/3$ parties in $\mathcal{P}$. For a sharing $\langle x \rangle$ we let x_i denote the share of party P_i. The functionality also uses the procedure Adjust introduced in $\mathcal{F}_{\texttt{key-gen}}$.
$\mathcal{F}_{\texttt{mul}}$ allows the same **Generate random committed shares** and **Generate random committed double shares** instructions as $\mathcal{F}_{\texttt{rand}}$. In addition, it also permits batch-multiplication of t sharings. Uses MacroAdjust.

Multiplication of Shares

On input $(\textsc{Mult}, \mathsf{sid}, t, ([\![a^{(\zeta)}]\!]_t, [\![b^{(\zeta)}]\!]_t)_{\zeta \in [t]})$ from all honest parties where the input sharings were generated by $\mathcal{F}_{\texttt{mul}}$ previously:

1. For $\zeta = 1, \ldots, t$ set $c^{(\zeta)} \leftarrow a^{(\zeta)} \cdot b^{(\zeta)}$ and run $[\![c^{(\zeta)}]\!]_t \leftarrow \mathsf{PSS.share}(c^{(\zeta)}, t, n)$.
2. Set $([\![c^{(1)}]\!]_t, \ldots, [\![c^{(t)}]\!]_t) \leftarrow \mathsf{Adjust}([\![c^{(1)}]\!]_t, \ldots, [\![c^{(t)}]\!]_t, 1)$.
3. Write $[\![c^{(\zeta)}]\!]_t = (\langle c^{(\zeta)} \rangle_t, \langle r^{(\zeta)} \rangle_t, \vec{\mathtt{C}}^{(\zeta)})$.
4. Output $(\textsc{Mult}, \mathsf{sid}, (c_i^{(\zeta)}, r_i^{(\zeta)}, \vec{\mathtt{C}}^{(\zeta)})_{\zeta \in [t]})$ to each honest P_i.

4.1 Instantiating $\mathcal{F}_{\texttt{key-gen}}$

We begin with a high level instantiation of $\mathcal{F}_{\texttt{key-gen}}$ which will rely on a number of helper ideal functionalities. The protocol is relatively simple, and works much the same as the triple pre-processing protocol of [DN07]: A pre-signature is essentially a type of multiplication triple and the public key a random output of $\mathcal{F}_{\texttt{rand}}$, with the randomness of the Pedersen commitment taken off. Thus, the below simple $\Pi_{\texttt{key-gen}}$ protocol arises which generates random sharings using $\mathcal{F}_{\texttt{rand}}$ and then either keeps one such sharing as pk or subsequently multiplies them using $\mathcal{F}_{\texttt{mul}}$ to generate the multiplication triples. Thereafter the randomness is revealed verifiably and with guaranteed termination. This ensures that the attacker cannot bias pk or the pre-signatures.

Protocol 2: $\Pi_{\texttt{key-gen}}$ Key Generation

This protocol runs with the parties $\mathcal{P} = (P_1, \ldots, P_n)$ and functionalities $\mathcal{F}_{\texttt{mul}}, \mathcal{F}_{\texttt{NIZK}}$. Let q be a prime and $\mathtt{G}, \mathtt{H} \in \mathbb{G}$ be generators of an order q subgroup of unknown discrete logarithm. A party P_i may at any point broadcast $(\textsc{Leave}, \mathsf{sid}, i)$ upon which all receiving parties mark P_i as corrupt and replace its input shares with default dummy values.

Protocol

On input $(\textsc{KeyGen}, \mathsf{sid})$, each P_i does the following:

1. Invoke $\mathcal{F}_{\texttt{mul}}$ with input $(\textsc{Rand}, \mathsf{sid}, 1)$ to obtain $[\![\mathsf{sk}]\!]_t = (\langle \mathsf{sk} \rangle, \langle r \rangle, \vec{\mathtt{D}})$.

2. Send $(\textsc{Rand}, \mathsf{sid}, 2t)$ to $\mathcal{F}_{\texttt{mul}}$ to obtain $[\![h^{(\zeta)}]\!]_t$ and $[\![k^{(\zeta)}]\!]_t = (\langle k^{(\zeta)} \rangle, \langle r^{(\zeta)} \rangle, \vec{\mathtt{D}}^{(\zeta)})$ for $\zeta \in [t]$.
3. Call $\mathcal{F}_{\texttt{mul}}$ with input $(\textsc{Mult}, \mathsf{sid}, t, ([\![h^{(\zeta)}]\!]_t, [\![k^{(\zeta)}]\!]_t)_{\zeta \in [t]})$ and obtain $([\![\gamma^{(\zeta)}]\!]_t)_{\zeta \in [t]}$.
4. Call $\mathcal{F}_{\texttt{mul}}$ with input $(\textsc{Mult}, \mathsf{sid}, t, ([\![h^{(\zeta)}]\!]_t, [\![\mathsf{sk}]\!]_t)_{\zeta \in [t]})$ and obtain $([\![\rho^{(\zeta)}]\!]_t)_{\zeta \in [t]}$.
5. Let sk_i, r_i be the opening that P_i has of $\mathtt{D}_i$ and $k_i^{(\zeta)}, r_i^{(\zeta)}$ its opening for $\mathtt{D}_i^{(\zeta)}$. P_i computes $\mathtt{R}_i = r_i \mathtt{H}, \mathtt{R}_i^{(\zeta)} = r_i^{(\zeta)} \mathtt{H}$ and then sends
$$(\textsc{Prove}, \mathsf{sid}, (\mathtt{G}, \mathtt{H}, (\mathtt{D}_i, \mathtt{R}_i, (\mathtt{D}_i^{(\zeta)}, \mathtt{R}_i^{(\zeta)})_{\zeta \in [t]})), (\mathsf{sk}_i, r_i, (k_i^{(\zeta)}, r_i^{(\zeta)})_{\zeta \in [t]}))$$
to $\mathcal{F}_{\texttt{NIZK}}$ for $\mathcal{R}_{\texttt{APOE}}$, obtains $(\textsc{Proof}, \mathsf{sid}, \pi_i)$ and sends $\mathtt{R}_i, \mathtt{R}_i^{(1)}, \ldots, \mathtt{R}_i^{(t)}, \pi_i$ to every other party.
6. Each party P_i receives $\mathtt{R}_j, \mathtt{R}_j^{(1)}, \ldots, \mathtt{R}_j^{(t)}, \pi_j$ from P_j. Compute the commitment $\mathtt{D}_j$ and for each $\zeta \in [t]$, also compute the commitment $\mathtt{D}_j^{(\zeta)}$ using Lagrange interpolation corresponding to party P_j. Send
$$(\textsc{Verify}, \mathsf{sid}, (\mathtt{G}, \mathtt{H}, (\mathtt{D}_j, \mathtt{R}_j, (\mathtt{D}_j^{(\zeta)}, \mathtt{R}_j^{(\zeta)})_{\zeta \in [t]})), \pi_j)$$
to $\mathcal{F}_{\texttt{NIZK}}$ for $\mathcal{R}_{\texttt{APOE}}$, consider all j where $\mathcal{F}_{\texttt{NIZK}}$ returns 1 for the proof and set $\vec{\mathtt{C}}^{(\zeta)} := \vec{\mathtt{D}}^{(\zeta)} - (\mathtt{R}_0^{(\zeta)}, \ldots, \mathtt{R}_t^{(\zeta)})$ where $\mathtt{R}_0^{(\zeta)}, \ldots, \mathtt{R}_t^{(\zeta)}$ are the evaluations of the unique polynomial of degree t of the at least $t+1$ evaluations $(j, \mathtt{R}_j^{(\zeta)})$ where the proof verified. Similarly, P_i reconstructs $\vec{\mathtt{C}}$ from $\vec{\mathtt{D}}$ by subtracting $(\mathtt{R}_0, \ldots, \mathtt{R}_t)$ reconstructed from all valid $\mathtt{R}_i$.
7. For $\zeta \in [t]$ set $\mathsf{psig}^{(\zeta)} \leftarrow (\mathtt{C}_0^{(\zeta)}, [\![h^{(\zeta)}]\!]_t, [\![\gamma^{(\zeta)}]\!]_t, [\![\rho^{(\zeta)}]\!]_t)$.
8. Define $\mathsf{pk} := \mathtt{C}_0$ and output $(\mathsf{pk}, \{\mathsf{psig}^{(\zeta)}\}_{\zeta=1}^t)$.

Complexity. The complexity is similarly straightforward to see: namely, $\Pi_{\texttt{key-gen}}$ requires $2t+1$ calls to generate random values and $2t$ calls to the multiplication interface. In addition, each party sends $t+1$ $\mathbb{G}$-elements as well as one proof π_j for $\mathcal{R}_{\texttt{APOE}}$. Therefore, the total communication consists of $2n^2$ $\mathbb{Z}_q$-elements and $(t+3)n^2$ $\mathbb{G}$-elements. The parties just have one round of interaction. Since the protocol generates $t < n/3$ pre-signatures at once, the total communication per pre-signature is approximately $6n$ $\mathbb{Z}_q$-elements and n^2 $\mathbb{G}$-elements.

Security. We will now prove the following Theorem:

Theorem 2. *The protocol $\Pi_{\texttt{key-gen}}$ UC-securely implements $\mathcal{F}_{\texttt{key-gen}}$ in the $\mathcal{F}_{\texttt{mul}}, \mathcal{F}_{\texttt{NIZK}}$-hybrid model with security against any polynomial-time attacker statically and actively corrupting at most $t \leq (n-1)/3$ parties, assuming hardness of the Discrete Logarithm problem in $\mathbb{G}$.*

In the proof, we let the simulator choose the outputs of $\mathcal{F}_{\texttt{mul}}$ such that they are identical with those of $\mathcal{F}_{\texttt{key-gen}}$, except that they are shifted by a random $\mathtt{H}$-component r known to the simulator in case of $[\![\mathsf{sk}]\!]_t$ and $[\![k^{(\zeta)}]\!]_t$. The adversary can bias the shares and commitments in $\mathcal{F}_{\texttt{mul}}$, but the simulator will remove this influence: When it later reveals $\mathtt{R}_i$ for each honest party it computes these based on $\mathsf{pk}, \mathsf{psig}^{(\zeta)}$ and the openings that the adversary will make, using Lagrange interpolation. Here, we are guaranteed that the adversary can only make one opening by the use of $\mathcal{F}_{\texttt{NIZK}}$. The simulator then has to simulate the π_j-proofs for the $\mathtt{R}_i$ that it reveals as it does not know the openings of the commitments, but this is indistinguishable as it controls $\mathcal{F}_{\texttt{NIZK}}$.

Proof. To prove the statement, we will construct a simulator $\mathcal{S}$ which will, with access to $\mathcal{F}_{\texttt{key-gen}}$, generate a transcript that is indistinguishable from $\Pi_{\texttt{key-gen}}$ in the presence of a real adversary $\mathcal{A}$.

Simulator: Upon being notified by $\mathcal{A}$ which parties I it will corrupt, I is forwarded to $\mathcal{F}_{\texttt{key-gen}}$ by $\mathcal{S}$. It sets up an instance of $\mathcal{F}_{\texttt{mul}}$ and multiple instances of $\mathcal{F}_{\texttt{NIZK}}$ which $\mathcal{S}$ simulates. $\mathcal{S}$ thereafter essentially runs the protocol $\Pi_{\texttt{key-gen}}$ on dummy inputs, with the following changes:

- Send $(\textsc{KeyGen}, \mathsf{sid})$ in the name of each dishonest party to $\mathcal{F}_{\texttt{key-gen}}$, from which it obtains $\mathsf{pk}, \mathtt{C}_0^{(\zeta)}$ as well as the adversarial shares for $[\![h^{(\zeta)}]\!]_t, [\![\gamma^{(\zeta)}]\!]_t, [\![\rho^{(\zeta)}]\!]_t$. For each honest party, simulate the protocol messages being sent by honest parties to $\mathcal{F}_{\texttt{mul}}$. $\mathcal{S}$ lets the sharings generating $[\![\mathsf{sk}]\!]_t$ and $[\![k^{(\zeta)}]\!]_t$ of $\mathcal{F}_{\texttt{mul}}$ be as follows: for $[\![\mathsf{sk}]\!]_t = (\langle \mathsf{sk} \rangle, \langle r \rangle, \mathtt{D})$ we choose $\mathtt{D}_0 = \mathtt{C}_0 + r_0\mathtt{H}$ for a uniformly random r_0. For all corrupt parties choose uniformly random $\mathtt{D}_i = \mathsf{sk}_i\mathtt{G} + r_i\mathtt{H}$ and deliver the adversarial shares of $\langle \mathsf{sk} \rangle, \langle r \rangle$ accordingly. Any additive updates due to Adjust make $\mathcal{S}$ update the dishonest parties' $\mathtt{R}_i = r_i\mathtt{H}$.

  We equivalently define each $\mathtt{D}_0^{(\zeta)}$ of $[\![k^{(\zeta)}]\!]_t$ based on the $\mathtt{C}_0^{(\zeta)}$ obtained from $\mathcal{F}_{\texttt{key-gen}}$. Finally output all these sharings to $\mathcal{A}$.

- The outputs of $\mathcal{F}_{\texttt{mul}}$ for $[\![h^{(\zeta)}]\!]$ as well as those of $\mathcal{F}_{\texttt{mul}}$ generating $[\![\gamma^{(\zeta)}]\!], [\![\rho^{(\zeta)}]\!]$ towards $\mathcal{A}$ are simply exactly those values that $\mathcal{S}$ obtained from $\mathcal{F}_{\texttt{key-gen}}$, and we forward any share adjustments that $\mathcal{A}$ makes to $\mathcal{F}_{\texttt{key-gen}}$.

- Towards revealing $\mathtt{R}_j, \mathtt{R}_j^{(1)}, \ldots, \mathtt{R}_j^{(t)}, \pi_j$ of $[\![\mathsf{sk}]\!]_t, [\![k^{(1)}]\!]_t, \ldots, [\![k^{(t)}]\!]_t$ for each simulated honest party, $\mathcal{S}$ uses the points defined by the dishonest $\mathtt{R}_i, \mathtt{R}_i^{(\zeta)}$ (plus whatever the adversary contributed in Adjust). Knowing that the adversary will reveal $\mathtt{R}_i, \mathtt{R}_i^{(\zeta)}$ leading to $\mathtt{C}_i, \mathtt{C}_i^{(\zeta)}$ where $\mathtt{C}_0 = \mathsf{pk}$ and the current $\mathtt{D}_0$ is fixed by the public transcript, we can compute $\widehat{\mathtt{R}}_0 = \mathtt{D}_0 - \mathsf{pk}$ and let all $\mathtt{R}_j$ be points on the degree-t polynomial going through the dishonest $\mathtt{R}_j$ and $\widehat{\mathtt{R}}_0$. The same can be done for the simulated honest $\mathtt{R}_j^{(\zeta)}$ based on $\mathtt{C}_0^{(\zeta)}$. The instances of $\mathcal{F}_{\texttt{NIZK}}$ simulate the proof π_j. It then sends those values to each dishonest party as in the protocol.

- For any dishonest $\mathtt{R}_j, \mathtt{R}_j^{(1)}, \ldots, \mathtt{R}_j^{(t)}, \pi_j$ $\mathcal{S}$ aborts if π_j has been correctly generated, but any $\mathtt{R}_j, \mathtt{R}_j^{(\zeta)}$ is different from those $\mathcal{S}$ stores as part of simulating $\mathcal{F}_{\texttt{mul}}$.

Clearly, the simulator $\mathcal{S}$ described above runs in polynomial time. We now argue why its output is indistinguishable from a real protocol transcript from $\Pi_{\texttt{key-gen}}$. Hybrids can simply be constructed, starting from the real protocol $\Pi_{\texttt{key-gen}}$, by first constructing a simulation where we abort if any π_j received from $\mathcal{A}$ is valid but the revealed share differs from the shares that was generated by $\mathcal{S}$ controlling $\mathcal{F}_{\texttt{mul}}$. Here, indistinguishability follows as any valid proof of a differing opening allows extracting $\log_{\mathtt{G}}(\mathtt{H})$ based on the original opening that the party obtained from $\mathcal{F}_{\texttt{mul}}$. We can then switch the $[\![\cdot]\!]$-shares from honestly generated by $\mathcal{F}_{\texttt{mul}}$ to the simulated ones as according to $\mathcal{F}_{\texttt{key-gen}}$ using that Pedersen commitments hide the secret perfectly and using that Shamir shares obtained by $\mathcal{A}$ in the protocol are uniformly random as in the simulation. By construction, the opened pk as well as the $\mathsf{psig}^{(\zeta)}$ shares of the adversary are consistent with those of $\mathcal{F}_{\texttt{key-gen}}$. □

Next, we will provide an efficient instantiation of $\mathcal{F}_{\texttt{mul}}$ that will perform batch multiplications and random element generation, in the $\mathcal{F}_{\texttt{rand}}$-hybrid model.

4.2 Instantiating $\mathcal{F}_{\texttt{mul}}$

We now provide an instantiation of $\mathcal{F}_{\texttt{mul}}$. At a high level, our batch multiplication protocol follows the protocol for multiplication with "double shares" from [DN07]. The major challenge is that the verifiable secret-sharing scheme we employ does not support the type of local multiplication that Shamir's secret-sharing does; While it is possible to locally multiply two degree t shares to obtain a degree $2t$ share of their product, this cannot be done directly with Pedersen's verifiable secret-sharing scheme.

However, all is not lost. By considering the verifiable secret-sharing $[\![a]\!]_t = (\langle a \rangle_t, \langle x \rangle_t, \vec{\mathtt{A}})$ as being a regular Shamir secret-sharing (the $\langle a \rangle_t$ part) and a set of commitments of the parties shares (the $\vec{\mathtt{A}}$ part), we can run the multiplication protocol from [DN07] using the Shamir share, and then make use of an efficient Zero-Knowledge proof protocol for proving that a share of the product sent to another party is correct with respect to the commitments. This can moreover be combined with the efficient opening protocol also from [DN07], such that we obtain a protocol for securely multiplying a batch of verifiable secret-shares.

Proof of Correct Multiplication. Let $\mathtt{G}$ be a generator of a group $\mathbb{G}$ of prime order q. We wish to obtain an efficient Zero-Knowledge proof protocol for the following relation:[8]

$$\mathcal{R}_{\texttt{APOCM}} = \left\{ \left(\left(u, \theta, \begin{pmatrix} \mathtt{A}_i, \mathtt{B}_i, \\ \mathtt{R}_i \end{pmatrix}_{i=1}^{t} \right), \begin{pmatrix} a_i, b_i, r_i, \\ x_i, y_i, z_i \end{pmatrix}_{i=1}^{t} \right) \middle| \begin{array}{l} \forall i \in \{1, \dots, t\} : \mathtt{A}_i = a_i \mathtt{G} + x_i \mathtt{H} \wedge \\ \mathtt{B}_i = b_i \mathtt{G} + y_i \mathtt{H} \wedge \mathtt{R}_i = r_i \mathtt{G} + z_i \mathtt{H} \wedge \\ u = \sum_{i=1}^{t} \theta^i (a_i b_i - r_i) \end{array} \right\}$$

Of course, a ZK proof protocol for $\mathcal{R}_{\texttt{APOCM}}$ can be constructed by simply proving each term of the sum separately, followed by a proof that the linear combination is correct. However, this will incur communication that is linear in t, which we would prefer to avoid. We therefore use Compressed Σ-protocol theory [AC20] to build a ZK proof for $\mathcal{R}_{\texttt{APOCM}}$ with sublinear communication. The proof protocol is denoted as $\Pi_{\texttt{APOCM}}$, and we will fully specify it in Section 5.

The Protocol. Our batch multiplication protocol is shown below in Protocol 3.

Protocol 3: $\Pi_{\texttt{mul}}$ Batch Multiplication

This protocol runs with the parties $\mathcal{P} = (P_1, \dots, P_n)$ and functionalities $\mathcal{F}_{\texttt{rand}}$ and $\mathcal{F}_{\texttt{NIZK}}$. Let $(\theta_1, \dots, \theta_n)$ be a set of n distinct elements from $\mathbb{Z}_q$. A party P_i may at any point broadcast $(\textsc{Leave}, \mathsf{sid}, i)$ upon which all receiving parties mark P_i as corrupt and replace its input shares with default dummy values.
Any commands $(\textsc{Rand}, \mathsf{sid}, \ell)$ or $(\textsc{DblRand}, \mathsf{sid}, \ell)$ are carried out using $\mathcal{F}_{\texttt{rand}}$.

Protocol

Let $\{[\![a^{(\zeta)}]\!]_t\}_{\zeta=1}^t$, $\{[\![b^{(\zeta)}]\!]_t\}_{\zeta=1}^t$ be the t sharings to be multiplied together, where $[\![a^{(\zeta)}]\!]_t = (\langle a^{(\zeta)} \rangle_t, \langle x^{(\zeta)} \rangle_t, \vec{\mathtt{A}}^{(\zeta)})$ and $[\![b^{(\zeta)}]\!]_t = (\langle b^{(\zeta)} \rangle_t, \langle y^{(\zeta)} \rangle_t, \vec{\mathtt{B}}^{(\zeta)})$.

1. Each party sends $(\textsc{DblRand}, \mathsf{sid}, t)$ to $\mathcal{F}_{\texttt{rand}}$ to obtain the t double shares $\{([\![r^{(\zeta)}]\!]_t, [\![r^{(\zeta)}]\!]_{2t})\}_{\zeta=1}^t$, where $[\![r^{(\zeta)}]\!]_{2t} = (\langle r^{(\zeta)} \rangle_{2t}, \langle z^{(\zeta)} \rangle_{2t}, \vec{\mathtt{S}}^{(\zeta)})$.
2. Each P_i locally computes $\langle w^{(\zeta)} \rangle_{2t} = \langle a^{(\zeta)} \rangle_t \langle b^{(\zeta)} \rangle_t - \langle r^{(\zeta)} \rangle_{2t}$ for $\zeta = 1, \dots, t$.
3. For $k = 1, \dots, n$ each party computes $\langle u^{(k)} \rangle_{2t} = \sum_{\zeta=1}^{t} \langle w^{(\zeta)} \rangle_{2t} \cdot \theta_k^\zeta$ locally.
4. For $k = 1, \dots, n$, each P_i sends $u_i^{(k)}$ to each P_k. Moreover, it sends proofs $\pi_{\mathcal{R}_{\texttt{APOCM}}, i, k}$ computed as outputs of $(\textsc{Prove}, \mathsf{sid}, (u_i^{(k)}, \theta_k, (\mathtt{A}_i^{(\zeta)}, \mathtt{B}_i^{(\zeta)}, \mathtt{S}_i^{(\zeta)})_{\zeta=1}^t), \texttt{witness})$ from $\mathcal{F}_{\texttt{NIZK}}$

[8] **A**mortized **P**roof **O**f **C**orrect **M**ultiplication.

for $\mathcal{R}_{\texttt{APOCM}}$ where $\texttt{witness} = (a_i^{(\zeta)}, b_i^{(\zeta)}, r_i^{(\zeta)}, x_i^{(\zeta)}, y_i^{(\zeta)}, z_i^{(\zeta)})_{\zeta=1}^{t}$. This shows that $u_i^{(k)}$ was computed correctly with respect to the commitments $\vec{\mathtt{A}}^{(\zeta)}$, $\vec{\mathtt{B}}^{(\zeta)}$, $\vec{\mathtt{S}}^{(\zeta)}$ and multiplier θ_k.

5. P_i then collects $u_k^{(i)}, \pi_{\mathcal{R}_{\texttt{APOCM}},k,i}$ from parties P_k where it stores $u_k^{(i)}$ if $\mathcal{F}_{\texttt{NIZK}}$ accepts on input $(\textsc{Verify}, \mathsf{sid}, (u_k^{(i)}, \theta_i, (\mathtt{A}_k^{(\zeta)}, \mathtt{B}_k^{(\zeta)}, \mathtt{S}_k^{(\zeta)})_{\zeta=1}^{t}), \pi_{\mathcal{R}_{\texttt{APOCM}},k,i})$.
6. When P_i has collected $2t+1$ or more valid shares, it reconstructs $u^{(i)}$ and sends $u^{(i)}$ to P_j for all $j = 1, \ldots, n$.
7. Once P_j received enough shares $u^{(i)}$ from other parties P_i, it recovers $\{w^{(\zeta)}\}_{\zeta=1}^{t}$ using Reed-Solomon error correction.
8. Each party outputs $[\![c^{(\zeta)}]\!]_t = w^{(\zeta)} + [\![r^{(\zeta)}]\!]_t$ for $\zeta = 1, \ldots, t$.

Firstly, note that the above protocol outputs the correct shares. Indeed, $[\![c^{(\zeta)}]\!]_t = w^{(\zeta)} + [\![r^{(\zeta)}]\!]_t = a^{(\zeta)}b^{(\zeta)} - r^{(\zeta)} + [\![r^{(\zeta)}]\!]_t = [\![a^{(\zeta)}b^{(\zeta)}]\!]_t$. The correct $w^{(\zeta)}$ values are obtained because (1) each honest P_k is guaranteed to receive at least $2t+1$ valid shares in step 6, namely the shares from the other honest parties; and (2) because each P_j in step 7 will receive enough shares to recover the coefficients, which are the $w^{(\zeta)}$s that are needed to compute the final output.

Complexity. During the protocol, the parties make one call for t double-sharings to $\mathcal{F}_{\texttt{rand}}$. Each party also sends two $\mathbb{Z}_q$ elements to each other party. In addition, each party sends one proof $\pi_{\mathcal{R}_{\texttt{APOCM}}}$ to each other party. This proof consists of $\psi + 2(\log_2(n/\psi + 1) + \log_2(8t+n+\psi+n/\psi+5))$ $\mathbb{G}$-elements and $10+n/\psi$ $\mathbb{Z}_q$-elements for a tweakable parameter ψ, as we will describe in Section 5. The protocol takes 2 rounds of interaction. Since the protocol computes t products at once, the total communication per product is a factor $n/3$ smaller.

Security. We will now prove the following Theorem:

Theorem 3. *The protocol $\Pi_{\texttt{mul}}$ UC-securely implements $\mathcal{F}_{\texttt{mul}}$ in the $\mathcal{F}_{\texttt{rand}}, \mathcal{F}_{\texttt{NIZK}}$-hybrid model with security against any polynomial-time attacker statically and actively corrupting at most $t \leq (n-1)/3$ parties, assuming hardness of the Discrete Logarithm problem in $\mathbb{G}$.*

The simulator sets up the outputs of $\mathcal{F}_{\texttt{rand}}$, specifically the commitment part of $[\![r^{(\zeta)}]\!]$ such that they differ from the outputs of $\mathcal{F}_{\texttt{mul}}$ by a random value $w^{(\zeta)}$ known to the simulator. Then, it can generate a random polynomial w consistent with $w^{(\zeta)}$, as well as polynomials that share the coefficients of the aforementioned polynomial and that are consistent with the values $u_i^{(k)}$ to be sent by the dishonest parties. This then allows the simulator to send $u_i^{(k)}$ itself such that the view is consistent. Again, the simulator simulates the proofs $\Pi_{\texttt{APOCM}}$.

Proof. To prove the statement, we will construct a simulator $\mathcal{S}$ which will, with access to $\mathcal{F}_{\texttt{mul}}$, generate a transcript that is indistinguishable from $\Pi_{\texttt{mul}}$ in the presence of a real adversary $\mathcal{A}$.

Simulator: Upon being notified by $\mathcal{A}$ which parties I it will corrupt, I is forwarded to $\mathcal{F}_{\texttt{mul}}$ by $\mathcal{S}$. It sets up an instance of $\mathcal{F}_{\texttt{rand}}$ which $\mathcal{S}$ simulates. $\mathcal{S}$ thereafter essentially runs the protocol $\Pi_{\texttt{mul}}$ on dummy inputs, with the following changes:

1. Any instance of $(\textsc{Rand}, \mathsf{sid}, \ell)$ or $(\textsc{DblRand}, \mathsf{sid}, \ell)$ which is not part of simulating $\Pi_{\texttt{mul}}$ is directly forwarded to $\mathcal{F}_{\texttt{mul}}$, and the responses are returned by the local emulated

$\mathcal{F}_{\texttt{rand}}$ towards the dishonest parties. This means that we can assume on the outset of any multiplication that $\mathcal{S}$ knows the input shares of the dishonest parties to the multiplication protocol, as these were simulated by the $\mathcal{F}_{\texttt{rand}}$-hybrid functionality towards them.

2. Send $(\textsc{Mult}, \mathsf{sid}, t, (\llbracket a^{(\zeta)} \rrbracket_t, \llbracket b^{(\zeta)} \rrbracket_t)_{\zeta \in [t]})$ in the name of each dishonest party to $\mathcal{F}_{\texttt{mul}}$. $\mathcal{S}$ then obtains $(c_i^{(\zeta)}, \hat{r}_i^{(\zeta)}, \vec{\mathtt{C}}^{(\zeta)})_{\zeta \in [t]}$ for each dishonest party P_i as part of Adjust.

3. $\mathcal{S}$ chooses $w^{(\zeta)} \in \mathbb{Z}_q$ uniformly at random and defines $\mathtt{R}_i^{(\zeta)} = \mathtt{C}_i^{(\zeta)} - w^{(\zeta)}\mathtt{G}$ for all $i \in \{0, \ldots, t\}$. For the shares of $\llbracket r^{(\zeta)} \rrbracket_t$ for the dishonest parties $\mathcal{S}$ will use the shares of $c_i^{(\zeta)}$ for each P_i, minus the constant polynomial with secret $w^{(\zeta)}$. In response, $\mathcal{A}$ returns polynomials $f^{(\zeta)}, g^{(\zeta)}$ adjusting $\llbracket r^{(\zeta)} \rrbracket_t$.

   For $\llbracket r^{(\zeta)} \rrbracket_{2t}$ the simulator uses $\mathtt{S}_0^{(\zeta)} = \mathtt{R}_0^{(\zeta)} + f^{(\zeta)}(0)\mathtt{G} + g^{(\zeta)}(0)\mathtt{H}$. It then chooses uniformly random values $r_i^{(\zeta)}, z_i^{(\zeta)}$ and then samples the rest of $\vec{\mathtt{S}}^{(\zeta)}$ to be consistent with the openings $r_i^{(\zeta)}, z_i^{(\zeta)}$ but otherwise be random group elements. Finally, it lets the simulated $\mathcal{F}_{\texttt{rand}}$ output these values to $\mathcal{A}$, and obtains polynomials $\widehat{f}^{(\zeta)}, \widehat{g}^{(\zeta)}$ for the $\llbracket \cdot \rrbracket_{2t}$-sharings as part of Adjust. It uses the functions $f^{(\zeta)}, g^{(\zeta)}, \widehat{f}^{(\zeta)}, \widehat{g}^{(\zeta)}$ to adjust the shares of $\llbracket r^{(\zeta)} \rrbracket_t$ and $\llbracket r^{(\zeta)} \rrbracket_{2t}$ accordingly. Note that each $w^{(\zeta)}$ remains uniformly random.

4. Given the $w^{(\zeta)}$, $\mathcal{S}$ can compute all $u^{(i)}$ that parties will obtain. In step 2, it will choose the $w_i^{(\zeta)}$ of each simulated honest party P_i to be consistent with a uniformly random degree-$2t$ polynomial which is $w^{(\zeta)}$ at 0 and consistent with the t shares $w_j^{(\zeta)}$ that $\mathcal{A}$ will locally compute during the protocol. It then computes the $u_i^{(k)}$ accordingly and simulates proofs $\pi_{\mathcal{R}_{\texttt{APOCM}}, i, k}$ of well-formedness of $u_i^{(k)}$ for $\mathcal{F}_{\texttt{NIZK}}$. If any simulated honest party P_i ever obtains a value $u_k^{(i)}$ from a dishonest P_k such that the proof accepts but $u_k^{(i)}$ differs from the value that P_k should send according to the protocol, then P_i aborts.

5. For all remaining steps, the simulated parties follow the protocol.

   Finally, $\mathcal{S}$ sends $\overline{f}^{(\zeta)}, g^{(\zeta)}$ adjusting $\llbracket r^{(\zeta)} \rrbracket_t$ to $\mathcal{F}_{\texttt{mul}}$, where $\overline{f}^{(\zeta)} = f^{(\zeta)} - f^{(\zeta)}(0)$, during $(\textsc{Diff}, \mathsf{sid})$ in Adjust.

The simulation of the responses to $(\textsc{Rand}, \mathsf{sid}, \ell)$ and $(\textsc{DblRand}, \mathsf{sid}, \ell)$ is perfect and we will therefore only focus on the multiplication protocol now.

We see that the output of the simulated $\mathcal{F}_{\texttt{rand}}$ towards the input query $(\textsc{DblRand}, \mathsf{sid}, t)$ during multiplication has the same distribution as in the protocol, since the commitments are uniformly random in both cases. The shares that $\mathcal{A}$ obtains are consistent with the distribution in the protocol as they are random, but open the commitments as required. Also, by construction, the output shares of the protocol will be consistent with the outputs that the honest parties obtain from $\mathcal{F}_{\texttt{mul}}$ as the simulator forwards the additive changes $\overline{f}^{(\zeta)}, g^{(\zeta)}$. The only other message that $\mathcal{A}$ obtains in the simulation are the values $u_i^{(k)}$, which are linear combinations of random values subject to reconstructing the correct secret by defining degree-$2t$ polynomials that share the $w^{(\zeta)}$. This is indistinguishable from the real protocol, where each share of an honest party in $w^{(\zeta)}$ is hidden by subtracting the uniformly random $r^{(\zeta)}$ from the product shares. Concerning the values $u_k^{(i)}$ that $\mathcal{S}$ obtains from $\mathcal{A}$, we have that these are well-formed due to $\mathcal{F}_{\texttt{NIZK}}$, or else we can extract $\log_{\mathtt{G}}(\mathtt{H})$ as argued before. Moreover, the simulation of the proof is indistinguishable from real proofs as $\mathcal{F}_{\texttt{NIZK}}$ still generates the proofs of the simulated honest parties as before. □

4.3 Instantiating $\mathcal{F}_{\texttt{rand}}$

We now show how to instantiate the second (and final) missing piece, namely $\mathcal{F}_{\texttt{rand}}$.

Our protocol can be seen as a mix of a modification of the pre-processing protocol from [DN07] that generates batches of $n-t$ random elements, but using a hyper-invertible matrix, and the broadcast protocol by [FH06] in order to minimize the amount of data sent over the broadcast channel. The main challenge is to obtain consensus on the VSS commitments in a communication efficient manner.

Our protocol $\Pi_{\mathtt{rand}}$ produces batches of random sharings of size $\ell = n - t$. Any other number of sharings can be easily obtained by either dropping shares or rerunning the protocol.

To construct the protocol, we assume a Reed-Solomon encoding algorithm $\mathsf{RS.enc}$ which we will introduce first: Let $\mathbb{F}$ be a finite field and n, k be constants. $\mathsf{RS.enc}(s, k, n)$ on input a string $s \in \mathbb{F}^k$ computes a string in $\mathbb{F}^n$ by using the Reed-Solomon encoding mechanism. We want that k out of the n symbols of the output of $\mathsf{RS.enc}$ are sufficient to reconstruct s. Since our strings s will consist of $\approx k$ group elements, this means each symbol will be of length one group element in $\mathbb{G}$ - but we are not bound to use $\mathbb{G}$ arithmetic when computing $\mathsf{RS.enc}$.

Protocol 4: $\Pi_{\mathtt{rand}}$

Let $\mathcal{P} = \{P_1, \ldots, P_n\}$ be a set of parties. Let Δ be an agreed upon timeout, $\mathsf{H} : \{0,1\}^* \to \{0,1\}^{2\kappa}$ a collision-resistant hash function. We initially set $I \leftarrow \emptyset$. The protocol runs in the $\mathcal{F}_{\mathtt{BC}}$-hybrid model, upon input $(\textsc{Rand}, \mathsf{sid}, \ell)$, and generates $\ell = n - t$ random shares of degree t.

Share

Each party P_i does the following:

1. Pick $s^{(i)} \xleftarrow{\$} \mathbb{Z}_q$ and run
$$(\langle s^{(i)} \rangle, \langle a^{(i)} \rangle, \vec{\mathtt{C}}^{(i)}) = [\![s^{(i)}]\!]_t \leftarrow \mathsf{PSS.share}(s^{(i)}, t, n).$$
2. Encode $\vec{\mathtt{C}}^{(i)}$ as $\vec{\mathtt{T}}^{(i)} \leftarrow \mathsf{RS.enc}(\vec{\mathtt{C}}^{(i)}, t+1, n)$.
3. Compute $h_1^{(i)} \leftarrow \mathsf{H}(\vec{\mathtt{C}}^{(i)})$, $h_2^{(i)} \leftarrow \mathsf{Merkle.hash}(\vec{\mathtt{T}}^{(i)})$.
4. Invoke $\mathcal{F}_{\mathtt{BC}}$ with $(\textsc{broadcast}, \mathsf{sid}, h_1^{(i)}, h_2^{(i)})$.
5. Send $s_j^{(i)}, a_j^{(i)}$ and $\vec{\mathtt{C}}^{(i)}$ to each $P_j \in \mathcal{P}$.

Agree

Any P_k that did not broadcast $h_1^{(k)}, h_2^{(k)}$ after delay Δ is added to I. Then run
$$([\![s^{(1)}]\!]_t, \ldots, [\![s^{(n)}]\!]_t) \leftarrow \Pi_{\mathtt{agree}}(I, ([\![s^{(i)}]\!]_t, h_1^{(i)}, h_2^{(i)})_{i \in [n] \setminus I})$$
in order to agree on a set of n consistent shares (with dummies assigned in $\Pi_{\mathtt{agree}}$ for parties who cheat or abort).

Extract

Let $\vec{M}$ be an n-by-$(n-t)$ hyper-invertible matrix, and $[\![s^{(1)}]\!]_t, \ldots, [\![s^{(n)}]\!]_t$ the n consistent shares output from $\Pi_{\mathtt{agree}}$. Each party locally computes and outputs
$$([\![r^{(1)}]\!]_t, \ldots, [\![r^{(n-t)}]\!]_t) = ([\![s^{(1)}]\!]_t, \ldots, [\![s^{(n)}]\!]_t)\vec{M}.$$

Protocol 5: Π_{agree}

We use the same notation as in $\Pi_{\mathtt{rand}}$.

Complain

Let m_j be the message broadcast by P_j in **Share**, and $s_i^{(j)}, a_i^{(j)}, \vec{\mathsf{C}}^{(j)}$ be the values sent from P_j to P_i. If $P_j \in I$, P_j did not broadcast or send a message to P_i, then P_i sets $m_j \leftarrow \perp$. Then P_i invokes $\mathcal{F}_{\mathtt{BC}}$ with $(\textsc{broadcast}, \mathsf{sid}, \vec{v}^{(i)})$ where:

$$v_j^{(i)} = \begin{cases} 0 & \text{if } m_j = (h_1^{(j)}, h_2^{(j)}) \text{ and consistent with } s_i^{(j)}, a_i^{(j)}, \vec{\mathsf{C}}^{(j)} \\ 1 & \text{otherwise} \end{cases}$$

where "consistent" means that

1. $h_1^{(j)} = \mathsf{H}(\vec{\mathsf{C}}^{(j)})$,
2. $h_2^{(j)} = \mathsf{Merkle.hash}(\mathsf{RS.enc}(\vec{\mathsf{C}}^{(j)}, t+1, n))$, and
3. $\mathsf{PSS.verf}(s_i^{(j)}, a_i^{(j)}, \vec{\mathsf{C}}^{(j)}, i, t) = 1$.

Fill

After delay Δ, let $\vec{H} \in \{0,1\}^{n \times n}$ be the matrix where the j'th row is $\vec{H}_j = \vec{v}^{(j)}$. Set $\vec{H}^j = \vec{1}^n$ for any P_j that did not broadcast a vector in the previous step. All parties P_i set $\vec{w}^{(i)} \leftarrow \vec{0}^n$. Next:

1. For each j such that $|\vec{H}^j| > t$ or $|\vec{H}_j| > t$, set $I \leftarrow I \cup \{P_j\}$.
2. For each i such that $0 < |\vec{H}^i| \leq t$ and j such that $H_j^i = 1$
   (a) Each P_k computes $\vec{\mathtt{T}}^{(i)} \leftarrow \mathsf{RS.enc}(\vec{\mathsf{C}}^{(i)}, t+1, n)$ and the Merkle proof $\pi_j^{(k)} \leftarrow \mathsf{Merkle.prov}(h_2^{(i)}, k, \vec{\mathtt{T}}^{(i)})$. Then it sends $\mathtt{T}_k^{(i)}$, $\pi_j^{(k)}$ to P_j.
   (b) P_i sends $s_j^{(i)}, a_j^{(i)}$ privately to all parties.
   (c) When P_k receives $s_j^{(i)}, a_j^{(i)}$ and has $\vec{\mathsf{C}}^{(i)}$, it runs $\mathsf{PSS.verf}(s_j^{(i)}, a_j^{(i)}, \vec{\mathsf{C}}^{(i)}, j, t)$. If the check fails, or $s_j^{(i)}, a_j^{(i)}$ or $\vec{\mathsf{C}}^{(i)}$ was not received, P_k sets $w_i^{(k)} \leftarrow 1$. Otherwise, P_k sends $s_j^{(i)}, a_j^{(i)}$ to P_j.
3. Each P_i invokes $\mathcal{F}_{\mathtt{BC}}$ with $(\textsc{broadcast}, \mathsf{sid}, \vec{w}^{(i)})$.

Finalize

After delay Δ, each P_i that lacked either $\vec{\mathsf{C}}^{(j)}$ or $s_i^{(j)}$:

1. Receives $\pi_i^{(k)}$ and $\mathtt{T}_k^{(j)}$ from at least $t+1$ parties P_k such that $\mathsf{Merkle.verf}(h_2^{(j)}, k, \pi_i^{(k)}, \mathtt{T}_k^{(j)}) = 1$. It then recovers $\vec{\mathsf{C}}^{(j)}$ from the received codeword parts $\mathtt{T}_k^{(j)}$.
2. Checks the received $s_i^{(j)}, a_i^{(j)}$ against the recovered $\vec{\mathsf{C}}^{(j)}$ and output the valid $s_i^{(j)}, a_i^{(j)}$. If more than 1 distinct valid share values is received, then broadcast $(\textsc{Leave}, \mathsf{sid}, i)$ and abort.
3. Define $\vec{J} \in \{0,1\}^{n \times n}$ by setting $\vec{J}_j = \vec{w}^{(j)}$ (with $\vec{J}^j = \vec{1}^n$ if P_j did not broadcast anything). Mark any parties P_k with $|\vec{J}^k| > t$ or $|\vec{J}_k| > t$ as corrupt, and set $I \leftarrow I \cup \{P_k\}$.
4. Output $([\![s^{(1)}]\!]_t, \ldots, [\![s^{(n)}]\!]_t)$ where $[\![s^{(j)}]\!]_t$ for each $P_j \in I$ is assigned a pre-defined dummy value.

Optimization. We now consider a potential computational bottleneck in the code: In step 2.*c* of **Fill**, each intermediate party has to verify potentially t shares coming from each participant to be a valid share of the VSS. We can check all of them at once by letting P_k choose a random point x and make a Schwartz-Zippel test "in the exponent". Here, we evaluate the polynomial defined by the $t+1$ VSS commitments in x. Then, for the t additional shares that we got from P_i, we pick an additional point from the VSS, consider this polynomial that we now implicitly got (as each share defines another point of the polynomial) and Lagrange-interpolate it again in x. If we get the same result, all points correct.

Complexity. In the protocol $\Pi_{\mathtt{rand}}$ each party sends two $\mathbb{Z}_q$-elements as well as one vector of $t+1$ $\mathbb{G}$-elements to every other party. This leads to a total of $2n^2$ $\mathbb{Z}_q$-elements and $(t+1)n^2$ $\mathbb{G}$-elements. In addition, each party broadcasts two hashes of total length 4κ. In total, $4n\kappa$ bits are broadcast in $\Pi_{\mathtt{rand}}$. The protocol $\Pi_{\mathtt{rand}}$ has one round of interaction, as private communication and broadcast can be performed in parallel.

In the protocol $\Pi_{\mathtt{agree}}$, each party broadcasts two vectors of length n. In the worst case, in order to ensure consistency of the commitment vectors, each party has to send $n \cdot t$ $\mathbb{G}$-elements and $n \cdot t$ membership proofs for $\mathsf{Merkle.prov}$ of length $2\kappa \log_2(n)$ each, leading to a maximal total communication of $n^2 t$ $\mathbb{G}$-elements and $n^2 t 2\kappa \log_2(n)$ bits for the membership proofs. In addition, each party may have to send 2 $\mathbb{Z}_q$ elements for each complaining party to n receivers, leading to a total communication of $2tn^2$ $\mathbb{Z}_q$-elements. The protocol runs in 3 rounds: first a round of broadcast, then a round of inter-party communication and finally an interleaved round of broadcast and inter-party communication.

Overall, both protocols together communicate $(2t+2)n^2$ $\mathbb{Z}_q$ elements, $(2t+1)n^2$ $\mathbb{G}$-elements and an additional $2tn^2\kappa \log_2(n)$ bits in 4 rounds. Since the protocol generates up to $2t$ random $[\![\cdot]\!]$-sharings at once, the overall communication per random sharing is approximately n^2 $\mathbb{Z}_q$ elements, n^2 $\mathbb{G}$-elements and $n^2\kappa \log_2(n)$ additional bits.

Security. We will now prove the following Theorem:

Theorem 4. *The protocol $\Pi_{\mathtt{rand}}$ UC-securely implements $\mathcal{F}_{\mathtt{rand}}$ in the $\mathcal{F}_{\mathtt{BC}}$-hybrid model with security against any polynomial-time attacker statically and actively corrupting at most $t \leq (n-1)/3$ parties, assuming hardness of the Discrete Logarithm problem in $\mathbb{G}$ and collision-resistance of H.*

The simulator for the proof is straightforward: the commitments of the honest parties as well as shares for the dishonest parties can be obtained from the outputs of $\mathcal{F}_{\mathtt{rand}}$ using the hyper-invertibility of $\overrightarrow{M}$. Otherwise, the simulator essentially follows the protocol but complains about dishonest parties if they cheat by breaking collision resistance of H or opening Pedersen commitments inconsistently.

Proof. To prove the statement, we will construct a simulator $\mathcal{S}$ which will, with access to $\mathcal{F}_{\mathtt{rand}}$, generate a transcript that is indistinguishable from $\Pi_{\mathtt{rand}}$ in the presence of a real adversary $\mathcal{A}$. We assume, for simplicity, that H (which is used in $\Pi_{\mathtt{rand}}$ to generate $h_1^{(i)}, h_2^{(i)}$) is used in $\mathsf{Merkle.hash}$ as well.

Simulator: Upon being notified by $\mathcal{A}$ which parties I it will corrupt, I is forwarded to $\mathcal{F}_{\mathtt{rand}}$ by $\mathcal{S}$. It sets up an instance of $\mathcal{F}_{\mathtt{BC}}$ which $\mathcal{S}$ simulates. $\mathcal{S}$ thereafter essentially runs the protocol $\Pi_{\mathtt{rand}}$ on dummy inputs, with the following changes:

1. $\mathcal{S}$ obtains the shares of all dishonest parties for $[\![r^{(\iota)}]\!]_t := (\langle \hat{r}^{(\iota)} \rangle, \langle \hat{a}^{(\iota)} \rangle, \overrightarrow{\mathsf{D}}^{(\iota)})$, as well as the commitment $\overrightarrow{\mathsf{D}}^{(\iota)}$ for each $\iota \in [\ell]$, from $\mathcal{F}_{\mathtt{rand}}$.

2. Fix the $n-t$ by $n-t$ submatrix of $\overrightarrow{M}$ whose rows are all parties not in I, and denote it as $\overrightarrow{K}$. We set the commitments $\overrightarrow{\mathtt{C}}^{(i)}$ of each simulated honest party such that they are the inverse of the matrix formed by the $\overrightarrow{\mathtt{D}}^{(1)}, \ldots, \overrightarrow{\mathtt{D}}^{(\ell)}$ under $\overrightarrow{K}^{-1}$. Observe that this inverse must exist by the definition of $\overrightarrow{M}$. Here, we compute $\overrightarrow{K}^{-1}(\mathtt{D}_0^{(1)}, \ldots, \mathtt{D}_0^{(\ell)})$ to obtain all $\mathtt{C}_0^{(i)}$ and then continue for all other indices of the vectors simultaneously.

3. During **Share**, $\mathcal{S}$ broadcasts honestly computed $h_1^{(i)}, h_2^{(i)}$ computed based on the $\overrightarrow{\mathtt{C}}^{(i)}$ obtained as outlined above.

4. The shares $s_j^{(i)}, a_j^{(i)}$ that $\mathcal{S}$ sends to the dishonest parties P_j in **Share** are computed as $\overrightarrow{K}^{-1}(\hat{r}_j^{(1)}, \ldots, \hat{r}_j^{(\ell)})$ and $\overrightarrow{K}^{-1}(\hat{a}_j^{(1)}, \ldots, \hat{a}_j^{(\ell)})$, i.e. by applying the same inverse map on the shares of the adversarial parties obtained from $\mathcal{F}_{\mathtt{rand}}$.

5. **Agree** is run as in the protocol. In addition we also mark dishonest parties as corrupt if (1) they sent different $\overrightarrow{\mathtt{C}}^{(j)}$ in private to simulated honest parties during **Share** which nevertheless hash to the same $h_1^{(j)}$, (2) if they sent $s_i^{(j)}, a_i^{(j)}$ during **Share** which do not lie on a degree-t polynomial, (3) during **Fill** a dishonest party sends shares $s_j^{(i)}, a_j^{(i)}$ which make $\mathsf{PSS.verf}$ accept but do not lie on the same polynomial, (4) during **Fill** a dishonest party sends a Merkle proof $\pi_j^{(k)}$ such that the proof is valid, but the opened $\overrightarrow{\mathtt{T}}_k^{(i)}$ is not part of $\overrightarrow{\mathtt{C}}^{(i)}$. In any of these cases, all simulated honest parties accuse the respective dishonest party.

6. $\mathcal{S}$, after having extracted all dishonest $[\![s^{(i)}]\!]_t$, forwards the additive changes to all sharings $[\![r^{(\iota)}]\!]_t$ obtained under the map $\overrightarrow{M}$ to $\mathcal{F}_{\mathtt{rand}}$ using $(\textsc{Diff}, \mathsf{sid}, \cdot)$.

It is easy to see that the simulation runs in polynomial time. Moreover, the shares output by $\mathcal{F}_{\mathtt{rand}}$ to all honest parties are consistent with the sharings of the dishonest parties and the respective commitments. The shares and commitments that $\mathcal{A}$ sees during the simulation have the same distribution as those in the real protocol, as those are uniform from the same distribution given the hyper-invertibility of $\overrightarrow{M}$. The values $h_1^{(i)}, h_2^{(i)}$ also have the same distribution.

Concerning the distribution of the broadcast $\overrightarrow{v}^{(j)}, \overrightarrow{w}^{(j)}$ we observe that honest parties are never added to I because there are only t corrupt parties and honest parties will never accuse honest parties by construction of the protocol. Hence only dishonest $[\![s^{(i)}]\!]_t$ are ever set to a default value. But accusations of dishonest parties differ in the simulation: parties are accused if they break collision resistance of H, can forge set membership proofs for the Merkle tree $\mathsf{Merkle.hash}$ or if they can reveal shares which allow to compute the discrete logarithm $\log_{\mathtt{G}}(\mathtt{H})$ by finding a different opening for a Pedersen commitment. By a sequence of hybrids, one can then show that this can only occur in the real protocol with bounded probability as the underlying hardness assumption would have to be broken.

During **Complain**, parties are only not excluded if they obtain $\leq t$ accusations, meaning $\geq t+1$ honest parties have obtained valid shares and valid commitments $\overrightarrow{\mathtt{C}}^{(i)}$. This means that after **Fill** each honest party will have obtained every such $\overrightarrow{\mathtt{C}}^{(i)}$ due to the Merkle tree construction. Moreover, the extraction of the values being sent during $(\textsc{Diff}, \mathsf{sid}, \cdot)$ is well-defined. Contributions from a dishonest party are only accepted if $\leq t$ parties have accused it as a cheater in $\overrightarrow{J}$. But this means that $\geq t+1$ simulated honest parties have obtained valid shares earlier for its $[\![r^{(i)}]\!]_t$, or have seen a valid share that they should forward to a party P_i, thus guaranteeing that all honest parties will either have obtained a commitment and shares, or all will exclude a dishonest party. □

Random Committed Double Shares. The protocol described in Protocol 4 only generates random sharings of degree t, but can easily be modified to obtain sharings of

the same value but of degree t and $2t$ simultaneously, as required for multiplications. To achieve this, the protocol $\Pi_{\mathtt{rand}}$ is modified as follows:

1. In **Share**, we make a sharing of degree t as well as $2t$ of the same value $s^{(i)}$. We also use the same secret for the randomness $a^{(i)}$. Then, the rest of **Share** is run twice for both sharings simultaneously. Of course, we still only need two hashes $h_1^{(i)}, h_2^{(i)}$. We can make sure that the same value is shared in both sharings by requiring that $\mathtt{C}_0^{(i)}$ is identical for both sharings. We run $\mathsf{RS.enc}$ with the same degree $k = t + 1$ for both sharings, which however requires longer symbols for the degree-$2t$ sharing. At the end of **Share** each P_i sends the shares of both the degree t and degree $2t$ sharings to each P_j.

2. We run $\Pi_{\mathtt{agree}}$ simultaneously for both openings, where parties identify others as cheaters whenever at least one of the openings is not valid.

The rest of the protocol is essentially identical, and the security proof is the same because, as specified in the proof, at no point does the security argument rely on the degree of the sharing. Showing that the shared secret must be the same, given the modifications, is trivial: in the simulation, the simulator will abort and blame a dishonest sender when it obtains sharings such that the degree-t and $2t$ sharings open different secrets. This is possible because the simulator always obtains $2t + 1$ sharings. Then, to bound the probability of this event happening, we observe that a different opened secret allows to obtain two different openings for the same commitment $\mathtt{C}_0^{(i)}$, which implies that the adversary would be able to compute the discrete logarithm $\log_{\mathtt{G}}(\mathtt{H})$ efficiently.

5 Batch ZK Proof of Correct Multiplication

Let $\mathtt{G}_1, \ldots, \mathtt{G}_N, \mathtt{H} \in \mathbb{G}$ be fixed group generators with unknown discrete logarithms among them for a cyclic group of order q. Here, $\mathtt{G}_1$ is identical to $\mathtt{G}$ from the previous sections, and we use the same $\mathtt{H}$.

We will write $\vec{\mathtt{G}} = (\mathtt{G}_1, \ldots, \mathtt{G}_N)$, and recall that $\vec{x} \star \vec{\mathtt{G}} = x_1\mathtt{G}_1 + \cdots + x_N\mathtt{G}_N$. Let $\mathtt{P} = \vec{x} \star \vec{\mathtt{G}} + \gamma\mathtt{H}$ and $L : \mathbb{Z}_q^N \to \mathbb{Z}_q$ be a linear form mapping $\vec{x} \in \mathbb{Z}_q^N$ to $\vec{x} \star \overrightarrow{\mathsf{query}}$ for a public $\overrightarrow{\mathsf{query}} \in \mathbb{Z}_q^N$. Assume that the prover is able to show that $y = L(\vec{x})$ for the vector $\vec{x}$ committed to in $\mathtt{P}$. More precisely, the prover can construct a proof for the language:

$$\mathcal{R}_{P\text{-}\mathtt{Lin}} = \left\{((\mathtt{P}, y), (\gamma, \vec{x})) \mid \gamma \in \mathbb{Z}_q, \vec{x} \in \mathbb{Z}_q^N \wedge \mathtt{P} = \vec{x} \star \vec{\mathtt{G}} + \gamma\mathtt{H} \wedge y = L(\vec{x})\right\}.$$

We will show that a proof protocol $\Pi_{P\text{-}\mathtt{Lin}}$ for $\mathcal{R}_{P\text{-}\mathtt{Lin}}$ is effectively all that is needed to construct an efficient proof for $\mathcal{R}_{\mathtt{APOCM}}$. The goal of this section is to construct such a proof that sends $o(N)$ $\mathbb{Z}_q$ or $\mathbb{G}$ elements, as well as whatever is needed to run the proof protocol for $\mathcal{R}_{P\text{-}\mathtt{Lin}}$.

As the actual ZK proof construction is very complex, we will explain all the involved techniques before putting them together.

1. We start by introducing a "central" commitment $\mathtt{P}$. It contains all the secrets a_i, b_i, r_i used in $\mathcal{R}_{\mathtt{APOCM}}$, and having them in one place is what allows to actually verify the statement. We will argue how we check that the values committed in $\mathtt{P}$ are consistent with the committed shares.

2. We let the sender commit to all outputs u_i of $\mathcal{R}_{\mathtt{APOCM}}$ as separate commitments $\mathtt{Q}_i$. Again, we will show how to prove consistency between $\mathtt{P}$ and the auxiliary output commitments.

3. We show how to check multiplicative relations of values committed in $\mathtt{P}$ by checking linear relations only, and how we combine this check with the multiplication with powers of θ as in $\mathcal{R}_{\mathtt{APOCM}}$.

4. We put all these steps together into one proof that makes two $\mathcal{R}_{P\text{-}\mathtt{Lin}}$ queries and uses the Fiat-Shamir transform.

5. Finally, we describe how to construct an efficient proof for $\mathcal{R}_{P\text{-}\mathtt{Lin}}$, based on compressed Σ-protocols.

In each step we will need multiple relations $\mathcal{R}_{P\text{-}\mathtt{Lin}}$ for the same $\mathtt{P}$, and therefore assume a proof system $\Pi^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ that can prove multiple linear relations simultaneously. We will later show that this is without loss of generality.

Throughout this section, we let $\mathsf{un}(i, n)$ be a function that on input $n \in \mathbb{N}, 1 \leq i \leq n$ outputs a vector of length n that is 1 at position i and 0 everywhere else. We will write $\vec{0}^{\ell}$ to denote a 0-vector of length ℓ.

5.1 Constructing a "central" commitment $\mathtt{P}$

The first step is to construct a commitment $\mathtt{P}$ which we can show as being consistent with the other commitments, namely those to the shares for $[\![a]\!]_t$, $[\![b]\!]_t$ and $[\![r]\!]_{2t}$. To be more precise, recall that $[\![a]\!]_t = (\langle a \rangle_t, \langle x \rangle_t, \vec{\mathtt{A}})$ where $\vec{\mathtt{A}}$ is a vector of commitments to the shares of $\langle a \rangle_t$, using randomness shared as $\langle x \rangle_t$. Equally we have that $[\![b]\!]_t = (\langle b \rangle_t, \langle y \rangle_t, \vec{\mathtt{B}})$ and $[\![r]\!]_{2t} = (\langle r \rangle_{2t}, \langle z \rangle_{2t}, \vec{\mathtt{R}})$.

We want a protocol that shows for a commitment $\mathtt{P}$ of (approximately) the form

$$\mathtt{P} = \gamma\mathtt{H} + \sum_{i=1}^{t} a_i\mathtt{G}_i + b_i\mathtt{G}_{t+i} + r_i\mathtt{G}_{2t+i} + x_i\mathtt{G}_{3t+i} + y_i\mathtt{G}_{4t+i} + z_i\mathtt{G}_{5t+i},$$

that $\mathtt{P}$ is consistent with $\mathtt{A}_i = a_i\mathtt{G} + x_i\mathtt{H}$, $\mathtt{B}_i = b_i\mathtt{G} + y_i\mathtt{H}$ and $\mathtt{R}_i = r_i\mathtt{G} + z_i\mathtt{H}$. These correspond to the t tuples of commitments $\mathtt{A}_i, \mathtt{B}_i, \mathtt{R}_i$, each for a different sharing but the same party. More formally, we want to design a proof for the following relation

$$\mathcal{R}_{\mathtt{ccom}} = \left\{ \begin{array}{l} \left((\mathtt{P}, (\mathtt{A}_i, \mathtt{B}_i, \mathtt{R}_i)_{i=1}^{t}), \left(\gamma, \alpha, \beta, \begin{pmatrix} a_i, b_i, r_i, \\ x_i, y_i, z_i \end{pmatrix}_{i=1}^{t} \right) \right) \Bigg| \\ \forall i \in \{1, \ldots, t\} : \\ \quad \{\mathtt{A}_i = a_i\mathtt{G} + x_i\mathtt{H} \wedge \mathtt{B}_i = b_i\mathtt{G} + y_i\mathtt{H} \wedge \mathtt{R}_i = r_i\mathtt{G} + z_i\mathtt{H}\} \\ \quad \wedge \mathtt{P} = \gamma\mathtt{H} + \alpha\mathtt{G}_{6t+1} + \beta\mathtt{G}_{6t+2} \\ \qquad\quad + \sum_{i=1}^{t}(a_i\mathtt{G}_i + b_i\mathtt{G}_{t+i} + r_i\mathtt{G}_{2t+i} + x_i\mathtt{G}_{3t+i} + y_i\mathtt{G}_{4t+i} + z_i\mathtt{G}_{5t+i}) \end{array} \right\}$$

While a protocol does not follow trivially from the commitments for the sharings $[\![a]\!]_t$, $[\![b]\!]_t$ and $[\![r]\!]_{2t}$ (as these, in comparison to $\mathtt{P}$, have identical bases), one can be constructed using a proof for $\mathcal{R}_{P\text{-}\mathtt{Lin}}$ as follows:

1. The verifier knows commitments $(\mathtt{A}_i, \mathtt{B}_i, \mathtt{R}_i)_{i \in [t]}$. The prover knows a_i, b_i, r_i, x_i, y_i and z_i for $i = 1, \ldots, t$ such that $\mathtt{A}_i = a_i\mathtt{G} + x_i\mathtt{H}$, $\mathtt{B}_i = b_i\mathtt{G} + y_i\mathtt{H}$ and $\mathtt{R}_i = r_i\mathtt{G} + z_i\mathtt{H}$.

2. Let $N = 6t + 2$. The prover samples $\alpha, \beta, \gamma \xleftarrow{\$} \mathbb{Z}_q^3$ and defines the vector

$$\vec{x} = (a_1, \ldots, a_t, b_1, \ldots, b_t, r_1, \ldots, r_t, x_1, \ldots, x_t, y_1, \ldots, y_t, z_1, \ldots, z_t, \alpha, \beta).$$

   The prover computes $\mathtt{P} = \vec{x} \star \vec{\mathtt{G}} + \gamma\mathtt{H}$ and sends $\mathtt{P}$ as well as $\mathtt{R} = \alpha\mathtt{G} + \beta\mathtt{H}$ to the verifier.

3. The verifier returns a challenge $\mathsf{ch}_{\mathrm{T}} \in \mathbb{Z}_q$ to the prover.

4. Define

$$\vec{\mathsf{ch}}_{\mathrm{T},1} = (\mathsf{ch}_{\mathrm{T}}, \ldots, \mathsf{ch}_{\mathrm{T}}^{3t}, \vec{0}^{3t}, 1, 0), \vec{\mathsf{ch}}_{\mathrm{T},2} = (\vec{0}^{3t}, \mathsf{ch}_{\mathrm{T}}, \ldots, \mathsf{ch}_{\mathrm{T}}^{3t}, 0, 1)$$

   and define L_1 as $L_1(\vec{x}) \mapsto \vec{x} \star \vec{\mathsf{ch}}_{\mathrm{T},1}$ as well as $L_2(\vec{x}) \mapsto \vec{x} \star \vec{\mathsf{ch}}_{\mathrm{T},2}$. The prover then sends $y_1 = L_1(\vec{x})$ and $y_2 = L_2(\vec{x})$ to the verifier.

5. Prover and verifier then run $\Pi^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ on $\mathtt{P}$ to show that the linear forms L_1, L_2 hold with respect to the committed secret $\vec{x}$.

6. Finally, the verifier accepts if the proof from $\Pi^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ accepts, and if

$$\mathtt{R} + \left(\sum_{i=1}^{t} \mathsf{ch}_{\mathrm{T}}^{i} \mathtt{A}_i + \mathsf{ch}_{\mathrm{T}}^{i+t} \mathtt{B}_i + \mathsf{ch}_{\mathrm{T}}^{i+2t} \mathtt{R}_i \right) = y_1 \mathtt{G} + y_2 \mathtt{H} \tag{2}$$

holds.

We will now show the following statement:

Lemma 2. *Assume that $\Pi^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ is a Zero-Knowledge Proof of Knowledge. Then the aforementioned protocol runs in 3 rounds and generates a commitment $\mathtt{P}$ fulfilling $\mathcal{R}_{\mathtt{ccom}}$ except with probability $(6t+2)/q$.*

Proof. Correctness of $\mathtt{P}$ follows from the completeness of $\Pi^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ and from applying L_1 and L_2 to the vector $\vec{x}$ and checking that for the values y_1, y_2 as well as ch_{T} the equation 2 holds. This can trivially be done by plugging in the definitions of $\vec{x}, \vec{\mathsf{ch}}_{\mathrm{T},1}, \vec{\mathsf{ch}}_{\mathrm{T},2}$.

Security against a dishonest prover holds due to a standard Schwartz-Zippel argument, except with soundness error $(6t+2)/q$. For this, consider a_i, b_i, r_i, α as well as x_i, y_i, z_i, β from $\mathtt{R}, \mathtt{A}_i, \mathtt{B}_i, \mathtt{R}_i$ as coefficients of polynomials f and g of degree $3t+1$ each. We can make this knowledge assumption because the prover reveals y_1, y_2 which open a linear combination of these commitments. If the prover can find openings with good probability which lie outside of such polynomials, then we can compute the discrete logarithm $\log_{\mathtt{G}}(\mathtt{H})$.

After fixing $\mathtt{P}$, by computing L_1, L_2 we evaluate two polynomials f', g' whose coefficients are committed to in the vector commitment $\mathtt{P}$ in the point ch_{T}, and the verifier does the same by evaluating f, g which led to y_1, y_2. If the polynomials are different (i.e., either the constant term is different or either a_i, b_i, r_i or x_i, y_i, z_i is not correctly committed to in $\mathtt{P}$) and assuming that $\Pi^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ is sound, then both polynomials can only coincide in at most $3t+1$ points without being identical. By a union bound, the security bound follows.

The protocol is secure against a dishonest verifier because the values y_1, y_2 are uniformly random due to the choice of α, β, so the protocol messages can be simulated and the messages of $\Pi^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ can be simulated using the Zero-Knowledge property of it. □

5.2 Include auxiliary opening commitments

We now add all the values $u_1, \ldots, u_n$ of $\mathcal{R}_{\mathtt{APOCM}}$ for all possible receivers P_i to $\mathtt{P}$ as well. Unfortunately, opening them directly from $\mathtt{P}$ later will turn out to be computationally expensive for the prover, as we have to do an expensive proof to reveal each u_i that is sent to a party, which quickly dominates local computation.

Instead, the prover will create ψ auxiliary commitments $\mathtt{Q}_1, \ldots, \mathtt{Q}_\psi$ (containing $u_1, \ldots, u_n$) and show that the values committed in them are also contained in $\mathtt{P}$. Here we assume that ψ divides n and define $\mu := n/\psi$. We let $\mathtt{Q}_1 := u_1\mathtt{G}_1 + \cdots + u_\mu \mathtt{G}_\mu + f_1\mathtt{H}$, while $\mathtt{Q}_2 := u_{\mu+1}\mathtt{G}_1 + \cdots + u_{2\mu}\mathtt{G}_\mu + f_2\mathtt{H}$ etc.

We now extend the protocol for constructing $\mathtt{P}$ from Section 5.1 to

- create these commitments $\mathtt{Q}_1, \ldots, \mathtt{Q}_\psi$;
- add $u_1, \ldots, u_n, f_1, \ldots, f_\psi$ to $\mathtt{P}$; and
- show consistency between the commitments.

Formally, the proof will show the following relation:

$$\mathcal{R}_{\texttt{auxcom}} = \left\{ \begin{array}{l} \left(\begin{pmatrix} \mathtt{P}, (\mathtt{Q}_1, \ldots, \mathtt{Q}_\psi), \\ (\mathtt{A}_i, \mathtt{B}_i, \mathtt{R}_i)_{i=1}^t \end{pmatrix}, \left(\gamma, \beta, (f_i)_{i=1}^\psi, (\alpha_i)_{i=1}^\mu, (u_i)_{i=1}^n, \begin{pmatrix} a_i, b_i, \\ r_i, x_i, \\ y_i, z_i, \end{pmatrix}_{i=1}^t \right) \right) \Bigg| \\ \forall i \in \{1, \ldots, t\} : \{\mathtt{A}_i = a_i\mathtt{G} + x_i\mathtt{H} \wedge \mathtt{B}_i = b_i\mathtt{G} + y_i\mathtt{H} \wedge \mathtt{R}_i = r_i\mathtt{G} + z_i\mathtt{H}\} \wedge \\ \forall i \in \{1, \ldots, \psi\} : \{\mathtt{Q}_i = f_i\mathtt{H} + \sum_{j=1}^{\mu} u_{\mu(i-1)+j}\mathtt{G}_j\} \wedge \\ \mathtt{P} = \gamma\mathtt{H} + \beta\mathtt{G}_{6t+n+\psi+\mu+1} + \sum_{i=1}^{n} u_i\mathtt{G}_{6t+i} + \\ \sum_{i=1}^{\psi} f_i\mathtt{G}_{6t+n+i} + \sum_{i=1}^{\mu} \alpha_i\mathtt{G}_{6t+n+\psi+i} + \\ \sum_{i=1}^{t} (a_i\mathtt{G}_i + b_i\mathtt{G}_{t+i} + r_i\mathtt{G}_{2t+i} + x_i\mathtt{G}_{3t+i} + y_i\mathtt{G}_{4t+i} + z_i\mathtt{G}_{5t+i}) \end{array} \right\}$$

We again use $\mathcal{R}_{P\text{-}\mathtt{Lin}}$ for the consistency check, and the security argument from above directly carries over. We will require $\mu + 1$ calls to $\mathcal{R}_{P\text{-}\mathtt{Lin}}$, instead of just 2.

1. The verifier knows $\mathtt{A}_i$, $\mathtt{B}_i$ and $\mathtt{R}_i$ such that $\mathtt{A}_i = a_i\mathtt{G} + x_i\mathtt{H}$, $\mathtt{B}_i = b_i\mathtt{G} + y_i\mathtt{H}$ and $\mathtt{R}_i = r_i\mathtt{G} + z_i\mathtt{H}$. The prover knows a_i, b_i, r_i, x_i, y_i and z_i for $i = 1, \ldots, t$. It also knows $u_1, \ldots, u_n$.

2. The prover samples $f_1, \ldots, f_\psi$ uniformly at random and for $i = 1, \ldots, \psi$ sets

$$\mathtt{Q}_i := u_{\mu(i-1)+1}\mathtt{G}_1 + \cdots + u_{\mu(i-1)+\mu}\mathtt{G}_\mu + f_i\mathtt{H}.$$

3. Let $N = 6t + n + \psi + \mu + 1$. The prover samples $\alpha_1, \ldots, \alpha_\mu, \beta, \gamma \xleftarrow{\$} \mathbb{Z}_q^{\mu+2}$ and defines the vector

$$\vec{x} = \begin{pmatrix} a_1, \ldots, a_t, b_1, \ldots, b_t, r_1, \ldots, r_t, x_1, \ldots, x_t, y_1, \ldots, y_t, z_1, \ldots, z_t, \\ u_1, \ldots, u_n, f_1, \ldots, f_\psi, \alpha_1, \ldots, \alpha_\mu, \beta \end{pmatrix}.$$

   The prover computes $\mathtt{P} = \vec{x} \star \vec{\mathtt{G}} + \gamma\mathtt{H}$ and sends $\mathtt{P}$ as well as $\mathtt{R} = \alpha_1\mathtt{G}_1 + \cdots + \alpha_\mu\mathtt{G}_\mu + \beta\mathtt{H}$ and $\mathtt{Q}_1, \ldots, \mathtt{Q}_\psi$ to the verifier.

4. The verifier returns a challenge $\mathsf{ch}_\mathrm{T} \in \mathbb{Z}_q$ to the prover.

5. Recall that $\mathsf{un}(i, n)$ outputs a standard unit vector of size n. We will use this function below to insert 0's around a particular index. Define

$$\vec{\mathsf{ch}}_{\mathrm{T},1} = (\mathsf{ch}_\mathrm{T}, \ldots, \mathsf{ch}_\mathrm{T}^{3t}, \vec{0}^{3t}, \mathsf{ch}_\mathrm{T}^{3t+1} \cdot \mathsf{un}(1, \mu), \ldots, \mathsf{ch}_\mathrm{T}^{3t+\psi} \cdot \mathsf{un}(1, \mu), \vec{0}^{\psi}, 1, \vec{0}^{\mu})$$

   and for $i = 2, \ldots, \mu$ define

$$\vec{\mathsf{ch}}_{\mathrm{T},i} = (\vec{0}^{6t}, \mathsf{ch}_\mathrm{T}^{3t+1} \cdot \mathsf{un}(i, \mu), \ldots, \mathsf{ch}_\mathrm{T}^{3t+\psi} \cdot \mathsf{un}(i, \mu), \vec{0}^{\psi+i-1}, 1, \vec{0}^{\mu-i+1}).$$

   Additionally define

$$\vec{\mathsf{ch}}_{\mathrm{T},o} = (\vec{0}^{3t}, \mathsf{ch}_\mathrm{T}, \ldots, \mathsf{ch}_\mathrm{T}^{3t}, \vec{0}^{n}, \mathsf{ch}_\mathrm{T}^{3t+1}, \ldots, \mathsf{ch}_\mathrm{T}^{3t+\psi}, \vec{0}^{\mu}, 1).$$

   Then let L_i be $L_i(\vec{x}) \mapsto \vec{x} \star \vec{\mathsf{ch}}_{\mathrm{T},i}$ as well as $\tilde{L}(\vec{x}) \mapsto \vec{x} \star \vec{\mathsf{ch}}_{\mathrm{T},o}$.

6. The prover computes $\tilde{y} = \tilde{L}(\vec{x})$ as well as $y_i = L_i(\vec{x})$ for $i = 1, \ldots, \mu$. It then sends $y_1, \ldots, y_\mu, \tilde{y}$ to the verifier.

7. Prover and verifier then run $\Pi^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ on $\mathtt{P}$ to show that the linear forms $L_1, \ldots, L_\mu, \tilde{L}$ hold with respect to the committed secret $\vec{x}$. (Here we abuse notation a bit and consider a slightly altered protocol $\Pi^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ which works on multiple linear forms simultaneously, instead of just one.)

8. Finally, the verifier accepts if the proof from $\Pi^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ accepts, and if

$$\mathtt{R} + \left(\sum_{i=1}^{t} \mathsf{ch}_{\mathrm{T}}^{i}\mathtt{A}_i + \mathsf{ch}_{\mathrm{T}}^{i+t}\mathtt{B}_i + \mathsf{ch}_{\mathrm{T}}^{i+2t}\mathtt{R}_i\right) + \sum_{i=1}^{\psi} \mathsf{ch}_{\mathrm{T}}^{3t+i}\mathtt{Q}_i = \tilde{y}\mathtt{H} + \sum_{i=1}^{\mu} y_i\mathtt{G}_i$$

holds.

Lemma 3. *Assume that $\Pi^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ is a Zero-Knowledge Proof of Knowledge. Then the aforementioned protocol runs in 3 rounds and outputs $\mathtt{Q}_1, \ldots, \mathtt{Q}_\psi$ that satisfy $\mathcal{R}_{\mathtt{auxcom}}$ except with probability $(\mu+1)(3t+\psi)/q$.*

Proof. Correctness of the protocol can be seen by evaluating the linear forms $L_i, \tilde{L}$ on the vector $\vec{x}$. For L_1, we perform the same linear combination as in the first protocol from Section 5.1, but additionally consider all u_i that are committed with respect to the generator $\mathtt{G}_1$ in $\mathtt{Q}_1, \ldots, \mathtt{Q}_\psi$. For $L_2, \ldots, L_\mu$ we then consider the values committed with respect to $\mathtt{G}_2, \ldots, \mathtt{G}_\mu$, respectively. Here we only have to include values u_i that are in the commitments $\mathtt{Q}_1, \ldots, \mathtt{Q}_\psi$ but nothing else except for a blinding factor provided by $\mathtt{R}$, as $\mathtt{A}_i, \mathtt{B}_i, \mathtt{R}_i$ only contain secrets committed to in $\mathtt{G}_1$. From completeness of $\Pi^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ the completeness then follows.

The argument why soundness of the proven linear relations translates into the desired security against a dishonest prover among the commitments follows identically as in the proof of Lemma 2, except there are now $\mu+1$ polynomials of degree $3t+\psi$ leading to a soundness error $(\mu+1)(3t+\psi)/q$.

Towards simulatability against a dishonest verifier, observe that each L_i contains an independent uniformly random value α_i that is added only to y_i, while $\tilde{y}$ contains the additional uniformly random value β. Hence these values can trivially be simulated, and based on the Zero-Knowledge property of $\Pi^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ the statement follows. □

5.3 Proving multiplications

Next, we will consider how to equip a commitment $\mathtt{P}$ such that it can be used to prove correctness of some multiplicative relation of committed values. This will require Lagrange interpolation, for which we will introduce a bit of notation. Consider a polynomial f of degree d, which is uniquely defined by the evaluations $(0, y_0), \ldots, (d, y_d)$. To evaluate f in a random point $\mathsf{ch}_{\mathrm{M}} \in \mathbb{Z}_q \setminus \{1, \ldots, d\}$ define the Lagrange coefficients

$$\lambda_i^{(d)}(\mathsf{ch}_{\mathrm{M}}) = \prod_{m \in \{0,\ldots,d\}\setminus\{i\}} \frac{\mathsf{ch}_{\mathrm{M}} - m}{i - m}$$

for $i \in \{0, \ldots, d\}$. Then $f(\mathsf{ch}_{\mathrm{M}}) = \sum_{i=0}^{d} y_i \lambda_i^{(d)}(\mathsf{ch}_{\mathrm{M}})$. Note that we require $\mathsf{ch}_{\mathrm{M}} \notin \{1, \ldots, d\}$ as this event would later otherwise reveal information about the secret polynomial f.

Let $\vec{a}, \vec{b}, \vec{c} \in \mathbb{Z}_q^t$ and assume the prover wishes to show that $c_i = a_i \cdot b_i$ for $i = 1, \ldots, t$. It can achieve this by interpreting $\vec{a}, \vec{b}$ and $\vec{c}$ as evaluations of polynomials in points $1, \ldots, t$ and showing the following relation on $\mathtt{P}$:

$$\mathcal{R}_{\mathtt{amult}} = \left\{ \left((\mathtt{P}), \begin{pmatrix} \gamma, A(X), \\ B(X), C(X) \end{pmatrix} \right) \;\middle|\; \begin{array}{l} \deg(A(X)) \leq t \wedge \deg(B(X)) \leq t \wedge \deg(C(X)) \leq 2t \wedge \\ \forall i \in \{0, \ldots, t\} : \{a_i = A(i) \wedge b_i = B(i)\} \wedge \\ \forall i \in \{0, \ldots, 2t\} : \{c_i = C(i)\} \wedge \\ C(X) = A(X) \cdot B(X) \wedge \\ \mathtt{P} = \gamma\mathtt{H} + \sum_{i=0}^{t}(a_i\mathtt{G}_{i+1} + b_i\mathtt{G}_{t+2+i}) + \sum_{i=0}^{2t} c_i\mathtt{G}_{2t+3+i} \end{array} \right\}$$

The following protocol can be used to generate $\mathtt{P}$ and prove the relation:

1. Let $N = 4t + 3$. The prover has inputs a_i, b_i for $i \in \{1, \ldots, t\}$.

2. The prover picks $\gamma, a_0, b_0 \xleftarrow{\$} \mathbb{Z}_q^3$ and sets $c_0 \leftarrow a_0 b_0$.

3. The prover locally computes degree t polynomials $A(X)$, $B(X)$ such that $a_i = A(i)$ and $b_i = B(i)$ for $i = 0, \ldots, t$. Define the degree $2t$ polynomial C as $C(X) = A(X)B(X)$. For $i = t+1, \ldots, 2t$ let $c_i = C(i)$.

4. The prover defines $\vec{x} = (a_0, \ldots, a_t, b_0, \ldots, b_t, c_0, \ldots, c_{2t})$, computes $\mathtt{P} = \vec{x} \star \vec{\mathtt{G}} + \gamma \mathtt{H}$ and sends $\mathtt{P}$ to the verifier.

5. The verifier picks $\mathsf{ch}_\mathrm{M} \xleftarrow{\$} \mathbb{Z}_q \setminus \{1, \ldots, t\}$ and sends ch_M to the prover.

6. Define the vectors

$$\begin{aligned}
\vec{\mathsf{ch}}_{\mathrm{M},a} &= (\lambda_0^{(t)}(\mathsf{ch}_\mathrm{M}), \ldots, \lambda_t^{(t)}(\mathsf{ch}_\mathrm{M}), \vec{0}^{3t+2}), \\
\vec{\mathsf{ch}}_{\mathrm{M},b} &= (\vec{0}^{t+1}, \lambda_0^{(t)}(\mathsf{ch}_\mathrm{M}), \ldots, \lambda_t^{(t)}(\mathsf{ch}_\mathrm{M}), \vec{0}^{2t+1}), \\
\vec{\mathsf{ch}}_{\mathrm{M},c} &= (\vec{0}^{2t+2}, \lambda_0^{(2t)}(\mathsf{ch}_\mathrm{M}), \ldots, \lambda_{2t}^{(2t)}(\mathsf{ch}_\mathrm{M})).
\end{aligned}$$

7. Let $L_a(\vec{x}) = \vec{x} \star \vec{\mathsf{ch}}_{\mathrm{M},a}$, and similar for L_b and L_c.

8. The prover computes $y_a \leftarrow A(\mathsf{ch}_\mathrm{M})$, $y_b \leftarrow B(\mathsf{ch}_\mathrm{M})$ and $y_c \leftarrow C(\mathsf{ch}_\mathrm{M})$ and sends (y_a, y_b, y_c) to the verifier.

9. The prover and verifier runs an instance of $\Pi^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ on $\mathtt{P}$ showing that $L_a(\vec{x}) = y_a$, $L_b(\vec{x}) = y_b$ and $L_c(\vec{x}) = y_c$.

10. Finally, the verifier accepts if the proof from $\Pi^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ accepts, and if $y_c = y_a y_b$.

Lemma 4. *Assuming that $\Pi^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ is a Zero-Knowledge Proof of Knowledge for $\mathcal{R}^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ (this relation is identical to $\mathcal{R}_{P\text{-}\mathtt{Lin}}$ except that it concerns multiple linear forms), then the aforementioned protocol runs in 3 rounds and creates a commitment $\mathtt{P}$ fulfilling relation $\mathcal{R}_{\mathtt{amult}}$ except with probability $2t/(q-t)$.*

Proof. Correctness follows trivially based on Lagrange interpolation. Security against a dishonest prover is similar to Lemma 2: Consider the first $t+1$ committed indices in the vector commitment $\mathtt{P}$ and denote them as evaluation points of the degree-t polynomial $A(X)$ identical to the one defined in the protocol. Consider the next $t+1$ indices of the vector commitment $\mathtt{P}$ as the evaluation points of the degree-t polynomial $B(X)$ and the remaining ones as the evaluation points of the degree-$2t$ polynomial $C(X)$. Now assume that for some $i \in \{0, \ldots, t\}$ that $a_i \cdot b_i \neq c_i$, so it must be that $A(X) \cdot B(X) \neq C(X)$. The prover reveals y_a, y_b, y_c which are evaluations of the aforementioned polynomials A, B, C in ch_M, and by proving the relation $\mathcal{R}^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ using $\Pi^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ for the 3 linear forms L_a, L_b, L_c, the soundness of $\Pi^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ and the correctness of Lagrange interpolation, the prover shows that the evaluations are consistent with the polynomials. But then, $A(X) \cdot B(X)$ can only agree with $C(X)$ in at most $2t$ points without the polynomials being identical, leading to the stated soundness error.

The protocol is secure against a dishonest verifier due to the additional choice of a_0, b_0 which makes the values y_a, y_b, y_c perfectly simulatable, and because $\Pi^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ is Zero-Knowledge. □

5.4 $\Pi_{\mathtt{APOCM}}$ for a single secret u

We now modify the multiplication check from Section 5.3 to something more akin to the relation $\mathcal{R}_{\mathtt{APOCM}}$. Namely, assume that the verifier knows the value θ and we want to prove

that $\mathtt{P}$ contains $u = \sum_{i=1}^{t} \theta^i(a_i b_i - r_i)$. More concretely, the relation that we want to show is

$$\mathcal{R}_{\mathtt{umult}} = \left\{ \left((\mathtt{P}, \theta), \begin{pmatrix} \gamma, u, A(X), \\ B(X), C(X), \\ r_1, \ldots, r_t \end{pmatrix} \right) \;\middle|\; \begin{array}{l} \deg(A(X)) \leq t \wedge \deg(B(X)) \leq t \wedge \\ \deg(C(X)) \leq 2t \wedge C(X) = A(X) \cdot B(X) \wedge \\ \forall i \in \{0, \ldots, t\} : \{a_i = A(i) \wedge b_i = B(i)\} \wedge \\ \forall i \in \{0, \ldots, 2t\} : \{c_i = C(i)\} \wedge \\ u = \sum_{i=1}^{t} \theta^i (c_i - r_i) \wedge \\ \mathtt{P} = \gamma \mathtt{H} + u\mathtt{G}_{5t+4} + \sum_{i=0}^{t} (a_i \mathtt{G}_{i+1} + b_i \mathtt{G}_{t+2+i}) + \\ \sum_{i=0}^{2t} c_i \mathtt{G}_{2t+3+i} + \sum_{i=1}^{t} r_i \mathtt{G}_{4t+3+i} \end{array} \right\}$$

The protocol is as follows:

1. Let $N = 5t + 4$. The prover has inputs a_i, b_i, r_i for $i \in \{1, \ldots, t\}$, while the verifier knows θ.
2. The prover picks $\gamma, a_0, b_0 \xleftarrow{\$} \mathbb{Z}_q$ and sets $c_0 \leftarrow a_0 \cdot b_0$.
3. The prover locally computes degree t polynomials $A(X)$, $B(X)$ such that $a_i = A(i)$ and $b_i = B(i)$ for $i = 0, \ldots, t$. As above, set $C(X) = A(X) \cdot B(X)$, and for $i = t+1, \ldots, 2t$ let $c_i = C(i)$.
4. The prover defines
$$\vec{x} = (a_0, \ldots, a_t, b_0, \ldots, b_t, c_0, \ldots, c_{2t}, r_1, \ldots, r_t, u),$$
computes $\mathtt{P} = \vec{x} \star \vec{\mathtt{G}} + \gamma \mathtt{H}$ and sends $\mathtt{P}$ to the verifier.
5. The verifier picks $\mathsf{ch}_{\mathrm{M}} \in \mathbb{Z}_q \setminus \{1, \ldots, t\}$ and sends ch_{M} to the prover.
6. Define the vectors
$$\begin{aligned} \vec{\mathsf{ch}}_{\mathrm{M},a} &= (\lambda_0^{(t)}(\mathsf{ch}_{\mathrm{M}}), \ldots, \lambda_t^{(t)}(\mathsf{ch}_{\mathrm{M}}), \vec{0}^{4t+3}), \\ \vec{\mathsf{ch}}_{\mathrm{M},b} &= (\vec{0}^{t+1}, \lambda_0^{(t)}(\mathsf{ch}_{\mathrm{M}}), \ldots, \lambda_t^{(t)}(\mathsf{ch}_{\mathrm{M}}), \vec{0}^{3t+2}), \\ \vec{\mathsf{ch}}_{\mathrm{M},c} &= (\vec{0}^{2t+2}, \lambda_0^{(2t)}(\mathsf{ch}_{\mathrm{M}}), \ldots, \lambda_{2t}^{(2t)}(\mathsf{ch}_{\mathrm{M}}), \vec{0}^{t+1}), \\ \overrightarrow{\mathsf{proj}} &= (\vec{0}^{2t+3}, \theta^1, \ldots, \theta^t, \vec{0}^t, -\theta^1, \ldots, -\theta^t, -1). \end{aligned}$$
7. Let $L_a(\vec{x}) = \vec{x} \star \vec{\mathsf{ch}}_{\mathrm{M},a}$, and similar for L_b, L_c and L_u.
8. The prover computes $y_a \leftarrow L_a(\vec{x})$, $y_b \leftarrow L_b(\vec{x})$, and $y_c \leftarrow L_c(\vec{x})$, and sends (y_a, y_b, y_c) to the verifier.
9. The prover and verifier run $\Pi^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ on $\mathtt{P}$ showing that $L_a(\vec{x}) = y_a$, $L_b(\vec{x}) = y_b$, $L_c(\vec{x}) = y_c$ and $L_u(\vec{x}) = 0$.
10. The verifier accepts if $\Pi^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ accepts and if $y_c = y_a \cdot y_b$.

Lemma 5. *Assume that $\Pi^{\mathtt{Am}}_{P\text{-}\mathtt{Lin}}$ is a Zero-Knowledge Proof of Knowledge. Then the aforementioned protocol runs in 3 rounds and outputs $\mathtt{P}$ that satisfies $\mathcal{R}_{\mathtt{umult}}$ except with probability $2t/(q-t)$.*

Proof. Correctness and security against the dishonest prover follows identically as Lemma 4, and the construction of $\overrightarrow{\mathsf{proj}}$. In it, we just check that this linear relation involving the θ^i holds on the respective values in $\vec{x}$ that is committed to in $\mathtt{P}$.

Security against a dishonest verifier likewise follows as before. □

The full proof for $\Pi_{\texttt{APOCM}}$. We can now combine all the techniques from Sections 5.1,5.2,5.3 and 5.4 into a full proof protocol for the $\mathcal{R}_{\texttt{APOCM}}$ relation. This protocol will rely on a protocol $\Pi_{P\text{-}\texttt{Lin}}$ for the $\mathcal{R}_{P\text{-}\texttt{Lin}}$ relation, meaning that we additionally take care of combining multiple forms that are tested simultaneously (which we denoted $\Pi^{\texttt{Am}}_{P\text{-}\texttt{Lin}}$ above) into a single form $L(\cdot)$ using a standard random linear combination technique, whenever possible. That is, after $\texttt{P}$ is generated and all linear forms which we want to check are fixed, the verifier samples an additional challenge ch_{L} which allows to combine all forms into one using an additional random linear combination of all forms with coefficients $1, \mathsf{ch}_{\mathrm{L}}, \mathsf{ch}^2_{\mathrm{L}}$ etc.

As mentioned earlier, the overall proof for $\mathcal{R}_{\texttt{APOCM}}$ consists of two separate ZK proofs: one showing the correctness of all the relations between the commitments $\texttt{A}_i, \texttt{B}_i, \texttt{R}_i, \texttt{P}, \texttt{Q}_j$, showing that the committed u_i are well-formed. In addition, a second proof opens the $\texttt{Q}_j$ at the position which contains the right u_ι value that the receiving party P_ι should obtain, thus showing that the share is correct. The first proof is independent of the index of the receiver, and can therefore be sent to all parties.

Protocol 6: $\Pi_{\texttt{APOCM}}$

The verifier knows commitments $\texttt{A}_i$, $\texttt{B}_i$, $\texttt{R}_i$ for $i = 1, \dots, t$ and θ_j, for $j = 1, \dots, n$, as well as the value u_ι and index $\iota \in [n]$. Note that we can write $\iota = h \cdot \mu + k + 1$ where $h \in \{0, \dots, \psi - 1\}$ and $k \in \{0, \dots, \mu - 1\}$.

The prover knows the openings (a_i, x_i), (b_i, y_i) and (r_i, z_i), for $i = 1, \dots, t$, as well as the values $u_1, \dots, u_n$. Let $\Pi_{P\text{-}\texttt{Lin}}$ be a zero-knowledge proof of knowledge for $\mathcal{R}_{P\text{-}\texttt{Lin}}$.

1. Let $N = 8t + n + \psi + \mu + 4$
2. The prover samples $\alpha_1, \dots, \alpha_\mu, \beta, \gamma, a_0, b_0, f_1, \dots, f_\psi \in \mathbb{Z}_q^{\psi+\mu+4}$ and sets $c_0 \leftarrow a_0 b_0$.
3. The prover computes degree t polynomials $A(X)$, $B(X)$ satisfying $A(i) = a_i$, $B(i) = b_i$ for $i \in \{0, \dots, t\}$. It sets $C(X) = A(X)B(X)$ and lets $c_i = C(i)$ for $i \in \{t+1, \dots, 2t\}$.
4. The prover defines
$$\vec{x} = \begin{pmatrix} a_0, \dots, a_t, b_0, \dots, b_t, c_0, \dots, c_{2t}, r_1, \dots, r_t, x_1, \dots, x_t, \\ y_1, \dots, y_t, z_1, \dots, z_t, u_1, \dots, u_n, f_1, \dots, f_\psi, \alpha_1, \dots, \alpha_\mu, \beta \end{pmatrix},$$
computes $\texttt{P} = \vec{x} \star \vec{\texttt{G}} + \gamma \texttt{H}$, $\texttt{R} = \alpha_1 \texttt{G}_1 + \dots + \alpha_\mu \texttt{G}_\mu + \beta \texttt{H}$ as well as $\texttt{Q}_i := u_{\mu(i-1)+1}\texttt{G}_1 + \dots + u_{\mu(i-1)+\mu}\texttt{G}_\mu + f_i \texttt{H}$ for $i = 1, \dots, \psi$ and sends these to the verifier.
5. The verifier samples $\mathsf{ch}_{\mathrm{M}} \xleftarrow{\$} \mathbb{Z}_q \setminus \{1, \dots, t\}$ and sends it to the prover.
6. The prover computes $y_a \leftarrow A(\mathsf{ch}_{\mathrm{M}})$, $y_b \leftarrow B(\mathsf{ch}_{\mathrm{M}})$, $y_c \leftarrow C(\mathsf{ch}_{\mathrm{M}})$ and sends them to the verifier.
7. The verifier samples $\mathsf{ch}_{\mathrm{T}} \xleftarrow{\$} \mathbb{Z}_q$ and sends it to the prover.
8. The prover computes
$$y_{\alpha,1} \leftarrow \alpha_1 + \sum_{i=1}^{t} \left(\mathsf{ch}_{\mathrm{T}}^i a_i + \mathsf{ch}_{\mathrm{T}}^{t+i} b_i + \mathsf{ch}_{\mathrm{T}}^{2t+i} r_i \right) + \sum_{j=1}^{\psi} \mathsf{ch}_{\mathrm{T}}^{3t+j} u_{(j-1)\mu+1}$$
as well as $y_{\alpha,i} \leftarrow \alpha_i + \sum_{j=1}^{\psi} \mathsf{ch}_{\mathrm{T}}^{3t+j} u_{(j-1)\mu+i}$ for $i = 2, \dots, \mu$. It additionally computes
$$y_\beta \leftarrow \beta + \sum_{i=1}^{t} \mathsf{ch}_{\mathrm{T}}^i x_i + \mathsf{ch}_{\mathrm{T}}^{t+i} y_i + \mathsf{ch}_{\mathrm{T}}^{2t+i} z_i + \sum_{i=1}^{\psi} \mathsf{ch}_{\mathrm{T}}^{3t+i} f_i$$
and sends $y_{\alpha,1}, \dots, y_{\alpha,\mu}, y_\beta$ to the verifier

9. The verifier samples $\mathsf{ch}_{\mathrm{L}} \xleftarrow{\$} \mathbb{Z}_q$ and sends it to the prover.
10. Prover and verifier define the vectors

$$\begin{aligned}
\vec{\mathsf{ch}}_{\mathrm{M},a} &= (\lambda_0^{(t)}(\mathsf{ch}_{\mathrm{M}}), \ldots, \lambda_t^{(t)}(\mathsf{ch}_{\mathrm{M}}), \vec{0}^{N-t-1}),\\
\vec{\mathsf{ch}}_{\mathrm{M},b} &= (\vec{0}^{t+1}, \lambda_0^{(t)}(\mathsf{ch}_{\mathrm{M}}), \ldots, \lambda_t^{(t)}(\mathsf{ch}_{\mathrm{M}}), \vec{0}^{N-2t-2}),\\
\vec{\mathsf{ch}}_{\mathrm{M},c} &= (\vec{0}^{2t+2}, \lambda_0^{(2t)}(\mathsf{ch}_{\mathrm{M}}), \ldots, \lambda_{2t}^{(2t)}(\mathsf{ch}_{\mathrm{M}}), \vec{0}^{N-4t-3}),\\
\vec{\mathsf{proj}}_1 &= (\vec{0}^{2t+3}, \theta_1^1, \ldots, \theta_1^t, \vec{0}^t, -\theta_1^1, \ldots, -\theta_1^t, \vec{0}^{3t}, -1, \vec{0}^{n+\psi+\mu}),\\
&\vdots\\
\vec{\mathsf{proj}}_n &= (\vec{0}^{2t+3}, \theta_n^1, \ldots, \theta_n^t, \vec{0}^t, -\theta_n^1, \ldots, -\theta_n^t, \vec{0}^{3t}, \vec{0}^{n-1}, -1, \vec{0}^{\psi+\mu+1}),\\
\vec{\mathsf{ch}}_{\mathrm{T},1} &= \begin{pmatrix} 0, \mathsf{ch}_{\mathrm{T}}, \ldots, \mathsf{ch}_{\mathrm{T}}^t, 0, \mathsf{ch}_{\mathrm{T}}^{t+1}, \ldots, \mathsf{ch}_{\mathrm{T}}^{2t}, \vec{0}^{2t+1}, \mathsf{ch}_{\mathrm{T}}^{2t+1}, \ldots, \mathsf{ch}_{\mathrm{T}}^{3t},\\ \vec{0}^{3t}, \mathsf{ch}_{\mathrm{T}}^{3t+1} \cdot \mathsf{un}(1,\mu), \ldots, \mathsf{ch}_{\mathrm{T}}^{3t+\psi}\mathsf{un}(1,\mu), \vec{0}^{\psi}, 1, \vec{0}^{\mu} \end{pmatrix},\\
\vec{\mathsf{ch}}_{\mathrm{T},i>1} &= (\vec{0}^{8t+3}, \mathsf{ch}_{\mathrm{T}}^{3t+1} \cdot \mathsf{un}(i,\mu), \ldots, \mathsf{ch}_{\mathrm{T}}^{3t+\psi} \cdot \mathsf{un}(i,\mu), \vec{0}^{\psi+i-1}, 1, \vec{0}^{\mu-i+1}),\\
\vec{\mathsf{ch}}_{\mathrm{T},o} &= (\vec{0}^{5t+3}, \mathsf{ch}_{\mathrm{T}}, \ldots, \mathsf{ch}_{\mathrm{T}}^{3t}, \vec{0}^n, \mathsf{ch}_{\mathrm{T}}^{3t+1}, \ldots, \mathsf{ch}_{\mathrm{T}}^{3t+\psi}, \vec{0}^{\mu}, 1)
\end{aligned}$$

11. Prover and verifier set

$$\vec{\mathsf{query}} \leftarrow \vec{\mathsf{ch}}_{\mathrm{M},a} + \mathsf{ch}_{\mathrm{L}}\vec{\mathsf{ch}}_{\mathrm{M},b} + \mathsf{ch}_{\mathrm{L}}^2\vec{\mathsf{ch}}_{\mathrm{M},c} + \sum_{i=1}^{n} \mathsf{ch}_{\mathrm{L}}^{2+i}\vec{\mathsf{proj}}_i + \sum_{i=1}^{\mu} \mathsf{ch}_{\mathrm{L}}^{n+2+i}\vec{\mathsf{ch}}_{\mathrm{T},i} + \mathsf{ch}_{\mathrm{L}}^{n+\mu+3}\vec{\mathsf{ch}}_{\mathrm{T},o}$$

and define $L(\vec{x}) = \vec{x} \star \vec{\mathsf{query}}$. Additionally, they set

$$y \leftarrow y_a + \mathsf{ch}_{\mathrm{L}} y_b + \mathsf{ch}_{\mathrm{L}}^2 y_c + \sum_{i=1}^{\mu} \mathsf{ch}_{\mathrm{L}}^{n+2+i} y_{\alpha,i} + \mathsf{ch}_{\mathrm{L}}^{n+\mu+3} y_{\beta}.$$

12. The prover runs $\Pi_{P\text{-}\mathtt{Lin}}$ on $\mathtt{P}$ with the verifier to prove that $y = L(\vec{x})$
13. Set $\vec{\mathsf{sel}} = (\vec{0}^k, 1, \vec{0}^{\mu-k-1})$ as well as $\vec{u}_P = (u_{h\mu+1}, \ldots, u_{h\mu+\mu})$. The prover runs $\Pi_{P\text{-}\mathtt{Lin}}$ on $\mathtt{Q}_{h+1}$ with the verifier to prove that $u_\iota = L_u(\vec{u}_P)$ for the linear form $L_u(\vec{u}_P) = \vec{u}_P \star \vec{\mathsf{sel}}$.
14. The verifier accepts if
    (a) The two $\Pi_{P\text{-}\mathtt{Lin}}$ instances were accepting;
    (b) $y_c = y_a y_b$;
    (c) It holds that

$$\begin{aligned}
&\mathtt{R} + \sum_{i=1}^{t} \left(\mathsf{ch}_{\mathrm{T}}^{i}\mathtt{A}_i + \mathsf{ch}_{\mathrm{T}}^{i+t}\mathtt{B}_i + \mathsf{ch}_{\mathrm{T}}^{i+2t}\mathtt{R}_i \right) + \sum_{i=1}^{\psi} \mathsf{ch}_{\mathrm{T}}^{3t+i}\mathtt{Q}_i\\
&= y_{\alpha,1}\mathtt{G}_1 + \cdots + y_{\alpha,\mu}\mathtt{G}_\mu + y_\beta \mathtt{H}
\end{aligned}$$

From Lemmas 2-5 we then obtain the following:

Theorem 5. *Assume that the Discrete Logarithm problem is hard in $\mathbb{G}$, that $\Pi_{P\text{-}\mathtt{Lin}}$ is a Public-Coin Honest-Verifier Zero-Knowledge proof of Knowledge for the relation $\mathcal{R}_{P\text{-}\mathtt{Lin}}$, then the protocol $\Pi_{\mathtt{APOCM}}$ is a Public-Coin Honest-Verifier Zero-Knowledge proof of knowledge of the relation $\mathcal{R}_{\mathtt{APOCM}}$ except with probability*

$$\epsilon = \frac{2t}{q-t} + \frac{(\mu+1)*(3t+\psi)+n+\mu+4}{q}.$$

Note that the theorem statement, as well as the proof, does not take any soundness error

for $\Pi_{P\text{-}\mathtt{Lin}}$ into consideration. It instead assumes a *hybrid* model with perfect knowledge extraction from $\Pi_{\mathtt{APOCM}}$ proof instances. This is just for the sake of simplicity, so that the analysis of the theorem can focus on the knowledge error incurred through the protocol $\Pi_{\mathtt{APOCM}}$ itself. After instantiating $\Pi_{P\text{-}\mathtt{Lin}}$ below we will include the knowledge error in the summarizing Corollary 1.

Proof. The construction runs the protocols from Lemma 3 and Lemma 5 sequentially, and then opens u_i from the respective auxiliary commitment $\mathtt{Q}_j$. The opening of u_i is just a linear relation and does not have an additional soundness error, while proving correctness of all u_i is sound except with a probability derived from the union bound on both checks. However, the base protocol only supports one linear query. In step 9 of the protocol we generate an additional challenge ch_{L} that is only chosen once all linear relations and their outputs are fixed. Let $\overrightarrow{\mathsf{query}}_1, \ldots, \overrightarrow{\mathsf{query}}_{n+\mu+4}$ be the queries and $y_1, \ldots, y_{n+\mu+4}$ be the claimed outputs. Define $y_i = \vec{x} \star \overrightarrow{\mathsf{query}}_i + \delta_i$ where $\delta_i \neq 0$ for at least one $i \in [n+\mu+4]$. Then it must hold that $0 = \sum_{i=1}^{n+\mu+4} \mathsf{ch}_{\mathrm{L}}^{i-1}\delta_i$ for the prover to accept a false witness $\vec{x}$. However, by the Schwartz-Zippel Lemma, this event only happens with probability $(n+\mu+4)/q$. Note that for the attacker to win with a false witness, at least one of the 3 checks must fail, the probability of which can be obtained from a union bound which yields the claimed probability. □

In the protocol, the prover communicates $\psi + 2$ commitments as well as $\mu + 4$ $\mathbb{Z}_q$-elements. All that remains is to provide an efficient construction for $\Pi_{P\text{-}\mathtt{Lin}}$. We do so in the following subsection.

5.5 Proving Linear forms $\mathcal{R}_{P\text{-}\mathtt{Lin}}$

To prove statements for the relation $\mathcal{R}_{P\text{-}\mathtt{Lin}}$, we will use the "compressed Σ-protocol theory" from [AC20]. The resulting proof is a multi-round Sigma protocol which has sublinear communication and a small verification overhead. We include the protocol here for completeness, as we implemented them in our construction, and describe the underlying idea on a high level. We however refer to [AC20] for the security analysis.

A trivial proof for $\mathcal{R}_{P\text{-}\mathtt{Lin}}$. We start with the standard Σ protocol $\Pi_{\mathtt{S}}$ for $\mathcal{R}_{P\text{-}\mathtt{Lin}}$, i.e. a ZKPoK:

1. The prover samples $\vec{r} \in \mathbb{Z}_q^N$ as well as $\rho \in \mathbb{Z}_q$ uniformly at random.
2. The prover computes $u = L(\vec{r})$ and $\mathtt{A} = \vec{r} \star \vec{\mathtt{G}} + \rho\mathtt{H}$ and sends $u, \mathtt{A}$ to the verifier.
3. The verifier chooses a challenge $\mathsf{ch} \in \mathbb{Z}_q$ uniformly at random and sends it to the prover.
4. The prover computes $\vec{z} = \vec{x}\mathsf{ch} + \vec{r}$, $\phi = \gamma\mathsf{ch} + \rho$ and sends $\vec{z}, \phi$ to the verifier.
5. The verifier checks that $\vec{z} \star \vec{\mathtt{G}} + \phi\mathtt{H} = \mathsf{ch}\mathtt{P} + \mathtt{A}$ and $L(\vec{z}) = y\mathsf{ch} + u$.

The protocol is perfectly complete, honest-verifier ZK and has soundness error $1/q$.

The computational bottleneck is for both prover and verifier to compute a statement of the form $\vec{x} \star \vec{\mathtt{G}}$ from a vector of group elements $\vec{\mathtt{G}}$ and a vector of scalars $\vec{x}$. This can be sped up using Pippengers algorithm.[9]

The communication bottleneck is Step 4 of $\Pi_{\mathtt{S}}$ where the prover has to send a message of length $O(N)$. The observation of [AC20] is that Steps 4 and 5 are actually a proof of knowledge for a modified relation that does not have to be Zero-Knowledge. Concretely, write $\vec{\mathtt{G}}^{(E)} := (\vec{\mathtt{G}}, \mathtt{H})$, $\vec{x}^{(E)} := (\vec{z}, \phi)$, $\overrightarrow{\mathsf{query}}^{(E)} := (\overrightarrow{\mathsf{query}}, 0)$ and let $L^{(E)}(\vec{x}^{(E)}) : \mathbb{Z}_q^{N+1} \to$

[9] See e.g. https://jbootle.github.io/Misc/pippenger.pdf for an introduction.

$\mathbb{Z}_q$ compute $\vec{x}^{(E)} \star \overrightarrow{\mathsf{query}}^{(E)}$ mod q. Additionally denote $\mathtt{P}^{(E)} := \mathsf{ch}\mathtt{P} + \mathtt{A}$ and $y^{(E)} := \mathsf{ch} \cdot y + u$. Then one can replace Steps 4 and 5 with a PoK of the relation $\mathcal{R}_{P\text{-}\mathtt{Lin}}$.

Modifying the trivial proof. Let $\mathtt{K}$ be an additional generator from $\mathbb{G}$ such that the discrete logarithm between $\mathtt{K}$ and any element of $\vec{\mathtt{G}}^{(E)}$ is not known. We rewrite the aforementioned relation such that it is easier to prove. Towards this, let

$$\mathcal{R}_{E\text{-}\mathtt{Lin}} = \left\{ ((\mathtt{P}^{(E)}, L^{(E)}), \vec{x}^{(E)}) \;\middle|\; \vec{x}^{(E)} \in \mathbb{Z}_q^{N+1} : \mathtt{P}^{(E)} = \vec{x}^{(E)} \star \vec{\mathtt{G}}^{(E)} + L^{(E)}(\vec{x}^{(E)})\mathtt{K} \right\}.$$

Clearly, any instance from $\mathcal{R}_{P\text{-}\mathtt{Lin}}$ can easily be turned into a (randomized) instance of $\mathcal{R}_{E\text{-}\mathtt{Lin}}$. We now present a protocol $\Pi_{P\text{-}\mathtt{Lin}}$ which is a version of $\Pi_{\mathtt{S}}$, but that uses a proof for $\mathcal{R}_{E\text{-}\mathtt{Lin}}$ as a subprotocol. This sub-proof is denoted as $\Pi_{E\text{-}\mathtt{Lin}}$. We make the protocol non-interactive for simplicity.

Protocol 7: $\Pi_{P\text{-}\mathtt{Lin}}$

Both prover and verifier know $\vec{\mathtt{G}} = (\mathtt{G}_1, \ldots, \mathtt{G}_N)$, $\mathtt{H}$, $\mathtt{K}$, commitment $\mathtt{P} = \vec{x} \star \vec{\mathtt{G}} + \gamma\mathtt{H}$ and target $y = L(\vec{x})$. The prover knows $\vec{x}$ and γ. Let $\Pi_{E\text{-}\mathtt{Lin}}$ be a Zero-Knowledge proof of Knowledge for $\mathcal{R}_{E\text{-}\mathtt{Lin}}$.

1. The prover picks $\vec{r} \xleftarrow{\$} \mathbb{Z}_q^N$, $\rho \xleftarrow{\$} \mathbb{Z}_q$, computes $t = L(\vec{r})$ and $\mathtt{F} = \vec{r} \star \vec{\mathtt{G}} + \rho\mathtt{H}$ and sends $t, \mathtt{F}$ to the verifier.
2. The verifier chooses $\mathsf{ch}_0, \mathsf{ch}_1 \xleftarrow{\$} \mathbb{Z}_q$ and sends them to the prover.
3. The prover computes $\vec{z} = \vec{x}\mathsf{ch}_0 + \vec{r}$, $\phi = \gamma\mathsf{ch}_0 + \rho$ and defines $\vec{x}^{(E)} = (\vec{z}, \phi)$.
4. The prover and verifier define $\vec{\mathtt{G}}^{(E)} = (\vec{\mathtt{G}}, \mathtt{H})$, $\mathtt{P}^{(E)} = \mathtt{F} + \mathsf{ch}_0\mathtt{P} + (\mathsf{ch}_1(\mathsf{ch}_0 y + t))\mathtt{K}$ and $L^{(E)}(\vec{x}^{(E)}) = L^{(E)}(\vec{z}, \phi) = \mathsf{ch}_1 L(\vec{z})$.
5. Both parties run $\Pi_{E\text{-}\mathtt{Lin}}$ on $\vec{\mathtt{G}}^{(E)}$, $\mathtt{K}$, $\mathtt{P}^{(E)}$ and linear form $L^{(E)}$ using private input $\vec{x}^{(E)}$.
6. The verifier accepts if $\Pi_{E\text{-}\mathtt{Lin}}$ accepts.

The protocol $\Pi_{P\text{-}\mathtt{Lin}}$ communicates one $\mathbb{G}$-element and one $\mathbb{Z}_q$-element, in addition to the communication of the proof $\Pi_{E\text{-}\mathtt{Lin}}$. All that remains now is to present a construction for $\Pi_{E\text{-}\mathtt{Lin}}$, which we do below.

An efficient protocol for $\mathcal{R}_{E\text{-}\mathtt{Lin}}$. We now use a special-sound proof of knowledge from "compressed Σ-protocol theory" to prove $\mathcal{R}_{E\text{-}\mathtt{Lin}}$ efficiently. The proof is a multi-round Sigma protocol-based proof which has sublinear communication and small verification overhead. It first appeared in [AC20], and we refer to the reference for the security analysis.

To define the protocol $\Pi_{E\text{-}\mathtt{Lin}}$ for $\mathcal{R}_{E\text{-}\mathtt{Lin}}$, we consider the same generators $\mathtt{G}_1, \ldots, \mathtt{G}_N, \mathtt{H}$, $\mathtt{K} \in \mathbb{G}$ as above, and define $\vec{\mathtt{G}}^{(E)} = (\mathtt{G}_1, \ldots, \mathtt{G}_N, \mathtt{H})$ as before.

Let the vector $\vec{\mathtt{G}}^{(E)}$ be of length M that is a multiple[10] of 2. We define $\vec{\mathtt{G}}_L^{(E)}$ to be the first $M/2$ elements of $\vec{\mathtt{G}}^{(E)}$ and $\vec{\mathtt{G}}_R^{(E)}$ to be the remaining $M/2$ (pad $\vec{\mathtt{G}}_R^{(E)}$ with 1 identity element if $\vec{\mathtt{G}}^{(E)}$ isn't a multiple of 2). For a vector $\vec{x}^{(E)}$ of scalars of length M, we equivalently define $\vec{x}_L^{(E)}, \vec{x}_R^{(E)}$ but pad $\vec{x}_R^{(E)}$ with 0s instead. Similarly, for $L^{(E)} : \mathbb{Z}_q^M \to \mathbb{Z}_q$ we define $L_L^{(E)}, L_R^{(E)} : \mathbb{Z}_q^{M/2} \to \mathbb{Z}_q$ computing $L_L^{(E)}(\vec{x}) = L^{(E)}(\vec{x}, \vec{0})$ and $L_R^{(E)}(\vec{x}) = L^{(E)}(\vec{0}, \vec{x})$ respectively, where $\vec{0}$ is of length $M/2$. In each iteration of the protocol, an instance for a commitment $\mathtt{P}^{(E)}$ is randomly folded into an instance for a commitment $\mathtt{P}^{(\mathsf{c})}$ where the committed vector has half the length.

[10] Otherwise, pad with identity elements.

Protocol 8: $\Pi_{E\text{-}\mathtt{Lin}}$

Let $\vec{\mathtt{G}}^{(E)} \in \mathbb{G}^M$ for $M \mod 2 = 0$. Both prover and verifier know $\vec{\mathtt{G}}^{(E)}$, $\mathtt{K}$ and the commitment $\mathtt{P}^{(E)} = \vec{x}^{(E)} \star \vec{\mathtt{G}}^{(E)} + L^{(E)}(\vec{x}^{(E)}) \cdot \mathtt{K}$. The prover knows $\vec{x}^{(E)}$.

Prover

1. The prover computes $\mathtt{A} = \vec{x}_L^{(E)} \star \vec{\mathtt{G}}_R^{(E)} + L_R^{(E)}(\vec{x}_L^{(E)})\mathtt{K}$ and $\mathtt{B} = \vec{x}_R^{(E)} \star \vec{\mathtt{G}}_L^{(E)} + L_L^{(E)}(\vec{x}_R^{(E)})\mathtt{K}$ and sends them to the verifier.
2. The verifier chooses $\mathsf{ch} \xleftarrow{\$} \mathbb{Z}_q$ and sends it to the prover.
3. Prover and verifier set $\vec{\mathtt{G}}^{(\mathsf{c})} = \mathsf{ch}\vec{\mathtt{G}}_L^{(E)} + \vec{\mathtt{G}}_R^{(E)}$, $\mathtt{P}^{(\mathsf{c})} = \mathtt{A} + \mathsf{ch}\mathtt{P}^{(E)} + \mathsf{ch}^2\mathtt{B}$, $L^{(\mathsf{c})} = \mathsf{ch}L_L^{(E)} + L_R^{(E)}$.
4. The prover computes $\vec{x}^{(\mathsf{c})} = \vec{x}_L^{(E)} + \mathsf{ch}\vec{x}_R^{(E)}$. If $\vec{x}^{(\mathsf{c})} \in \mathbb{Z}_q^2$ then it sends $\vec{x}^{(\mathsf{c})}$ to the verifier, otherwise both parties run $\Pi_{E\text{-}\mathtt{Lin}}(\mathtt{P}^{(\mathsf{c})}, L^{(\mathsf{c})}, \vec{x}^{(\mathsf{c})})$ with public parameters $\vec{\mathtt{G}}^{(\mathsf{c})}, \mathtt{K}$.
5. If the prover sent $\vec{x}^{(\mathsf{c})} \in \mathbb{Z}_q^2$, then the verifier accepts if $\vec{x}^{(\mathsf{c})} \star \vec{\mathtt{G}}^{(\mathsf{c})} + L^{(\mathsf{c})}(\vec{x}^{(\mathsf{c})})\mathtt{K} = \mathtt{P}^{(\mathsf{c})}$. If $\vec{x}^{(\mathsf{c})}$ was not sent then the verifier accepts if the call to $\Pi_{E\text{-}\mathtt{Lin}}$ accepts.

In each iteration of $\Pi_{E\text{-}\mathtt{Lin}}$, the prover sends either 2 $\mathbb{Z}_q$-elements and 2 $\mathbb{G}$-elements or two $\mathbb{G}$-elements. Hence, the proof runs in at most $\log_2(M) - 1$ rounds and communicates $2\log_2(M) - 2$ $\mathbb{G}$-elements and 2 $\mathbb{Z}_q$-elements.

5.6 Making the protocols non-interactive

Theorem 5 only applies to an interactive protocol. In our implementation, we instead use a non-interactive version by applying the Fiat-Shamir transform (hashing the full statement to ensure adaptive soundness), which we also apply to $\Pi_{P\text{-}\mathtt{Lin}}$. For this transformation, one can show the following:

Corollary 1. *Let H_{FS} be a cryptographic hash function modeled as a Random Oracle. Let $\Pi^{FS}_{\mathtt{APOCM}}$ be a version of $\Pi_{\mathtt{APOCM}}$ after applying the Fiat-Shamir transform. Then for any PPT prover making at most Q queries to H_{FS} and assuming hardness of the Discrete Logarithm problem in $\mathbb{G}$, $\Pi^{FS}_{\mathtt{APOCM}}$ is a NIZK proof of knowledge in the Random Oracle model except with probability*

$$(Q+1) \cdot \left(\frac{2t}{q-t} + \frac{(\mu+1)(3t+\psi) + n + \mu + 7 + 2\lceil \log_2(N) \rceil + 2\lceil \log_2(\mu+1) \rceil}{q} \right).$$

Proof. The statement can be obtained by applying [AFK22]. From the proof of Theorem 5, it follows that obtaining valid responses for $n + \mu + 5$ choices of ch_{L} implies correctness of all the individual linear relations. If the linear relations are correct, then only $2t$ choices of ch_{M} yield transcripts that are accepting for false committed values. Thus by obtaining transcripts for $2t + 1$ choices of ch_{M}, the extracted solution must always have a correct multiplicative relation (following Lemma 5). Finally, towards consistency of $\mathtt{P}$ with all the other commitments, we know that there can be at most $(\mu+1)(3t+\psi)$ choices of ch_{T} such that $\mathtt{P}$ might be inconsistent by Lemma 3. Thus, if one was able to extract strictly more accepting transcripts with different choices of ch_{T} then the witness is guaranteed to be valid. This coincides with the notion of $(k_1, \ldots, k_\ell)$-out-of-$(N_1, \ldots, N_\ell)$ special soundness as defined in [AFK22]. By observing that [AC20] follows the same notion where $\Pi_{P\text{-}\mathtt{Lin}}$ is $(2,2)$-out-of-(q,q) special sound and $\Pi_{E\text{-}\mathtt{Lin}}$ is ($\underbrace{3, \ldots, 3}_{\lceil \log_2(M) \rceil \text{ times}}$)-out-of-($\underbrace{q, \ldots, q}_{\lceil \log_2(M) \rceil \text{ times}}$) special

sound, we can combine them into the bound mentioned above. □

6 Experimental Evaluation

We implemented the $\Pi_{\texttt{sign}}$, $\Pi_{\texttt{mul}}$ and $\Pi_{\texttt{rand}}$ protocols, which allows us to present benchmarks for the most complex parts of this work. The implementation was done in C++ using the SCL framework [Dal24], and all experiments were performed using the simulation functionality of SCL. In a nutshell, the simulator in SCL runs each party of a protocol sequentially such that no parties occupy the CPU at the same time. The execution time of each party is timed separately and adjusted whenever data is sent or received on a simulated network. The adjustments made depend on a set of configurable network parameters, and we use a 1MBit bandwidth, 100ms latency network for all experiments thus obtaining results which correspond to running our protocol in WAN-like network.[11] All experiments were run on a laptop with an AMD Ryzen 5 PRO 5650U processor and 45 GB of memory. No multithreading was used. All code and data is available on github.[12]

Modelling a broadcast channel. Our protocols rely on a broadcast channel, but we opted to use an idealized version as opposed to implementing a real broadcast protocol. Since the SCL simulation framework runs all parties locally it is possible to implement a broadcast channel as a buffer that is shared between all parties. Messages can then be programmatically hidden based on a configurable delay (the time it takes for a message to be broadcast). At a high level, this works in the following way: Suppose P_i places message m in the buffer at local time Δ_i. If the broadcast delay is t, then any other party P_j would be able to read m from the buffer when its local time satisfies $\Delta_j \geq \Delta_i + t$. (At a more low level, this shared buffer is implemented as an additional protocol party that is co-located with all other parties.) This functionality is used to implement a notion of a "broadcast round": If P_i is the first to send a k'th message at time Δ_i, then all other k'th messages sent within $\leq \Delta_i + t_r$ belong to the same round. Here, the parameter t_r is the "round delay". In our experiments, we use $t_r = 5$s. All our protocols are coded such that parties wait for a round to finish before any action is taken.

6.1 Experiments

We perform five experiments in total, aimed at understanding the performance characteristics of $\Pi_{\texttt{sign}}$, $\Pi_{\texttt{mul}}$ and $\Pi_{\texttt{rand}}$ and their parts. In addition, we perform a comparison to the threshold ECDSA protocol currently used in production by Partisia Blockchain. A brief description of each experiment follows here. Our experiments focus on the "happy path" of the protocols. That is, corrupt party behavior (e.g., crashes, malformed messages, selective responses, and so on) is not considered as a parameter in our experiments.

Experiment 1: $\Pi_{\texttt{sign}}$. The first experiment examines the execution time and communication complexity of $\Pi_{\texttt{sign}}$ as the number of parties grows. We vary the number of parties from $n = 4$ to $n = 100$ in increments of 3. For each n, we run $\Pi_{\texttt{sign}}$ five times and take the median execution time of a party that reconstructs the final signature. Regarding communication we report both the amount of data sent by this party, and the amount of data received. We note that the amount of data received is only the *effective* amount—i.e., the data used by the party to recover the signature.

[11] See: https://github.com/anderspkd/secure-computation-library/blob/master/include/scl/simulation/config.h.

[12] https://github.com/anderspkd-partisia/large-oracle-cic-code/tree/main

Experiment 2: $\Pi_{\texttt{mul}}$. The second experiment examines the execution time and communication complexity of $\Pi_{\texttt{mul}}$, in a similar fashion as experiment 1. That is, we measure a single execution of $\Pi_{\texttt{mul}}$ while varying the number of parties. We use values of $n = 10$ to $n = 100$ in increments such that n is even (this is due to how $\Pi_{\texttt{APOCM}}$ was implemented). For each n, we repeat the experiment three times. For each repetition, we compute the average execution time of all parties, and the final values we report are the median of these averages. Because $\Pi_{\texttt{mul}}$ works in batches, we report both total execution times, respectively communication complexity, as well as amortized execution time / communication complexity.

Experiment 3: $\Pi_{\texttt{APOCM}}$. As a complement to the second experiment, we also perform benchmarks of $\Pi_{\texttt{APOCM}}$ in order to better understand how the choice of ψ affects its performance.

Experiment 4: $\Pi_{\texttt{rand}}$. The fourth experiment examines the execution time and communication complexity of $\Pi_{\texttt{rand}}$. This too is done in a similar fashion as experiments 1 and 2. We run $\Pi_{\texttt{rand}}$ once while varying the number of parties. We run the protocol for $n = 4$ to $n = 88$ in increments of 3, the somewhat lower limit being a result of memory limitations on the benchmark platform.[13] We repeat each experiment two times and report the median of the average as before. In addition to measuring execution time and communication complexity, we also examine the **Complain** and **Fill** step, in order to better understand particularities that arise due to the use of verifiable secret-sharing. We report both total execution time, respectively communication complexity, as well as amortized execution time, respectively communication complexity.

Experiment 5: Comparison to Partisia Blockchain. Finally, we perform a comparison against the threshold signature protocol deployed on Partisia Blockchain, using their implementation located at [Blo24]. Their implementation does not implement communication as part of the protocol, and so it was fairly straightforward for us to implement an SCL compatible wrapper around `pbcts.so` in C++, using the functions exposed in `pbcts.h`. We refer to this protocol as PBCTS. The PBCTS protocol is, as mentioned back in Section 1.2, the robust $t \leq (n-1)/3$ protocol described in [GJKR96, GJKR01], with minor modifications to accommodate pre-signature creation. We benchmark the execution time and communication complexity of generating 10 pre-signatures for $n = 4$ to $n = 32$. Each experiment is repeated three times.

6.2 Results

Results of experiment 1. The results from experiment 1 can be seen in Figure 1, where the figure on the left shows communication complexity, while the two figures on the right show the execution time. The results are not surprising. With larger n, more data needs to be sent by each party. Similarly, a larger n implies that each party needs more shares to reconstruct the signature. Note that the gap between the "sent" and "received" values in Figure 1 is roughly 2/3. I.e., each party sends their share to $n-1$ other parties, but only needs to receive $t+2$ shares to reconstruct the signature. Regarding execution time, we report both total running time (figure on the right bottom), as well as time it takes to receive shares, respectively reconstruct the signature. Receiving data can be done in parallel, hence the flat line. On the other hand, reconstruction time increases with n. This is not surprising, given that the running time of Lagrange interpolation depends on

[13] During execution, SCL needs to keep the state of all n parties in memory. For large n, and for protocols which keep a large amount of state per party, this can result in quite high memory usage.

the degree of the polynomial being interpolated. In any event, the time it takes to sign a message is roughly the network latency, which is to be expected.

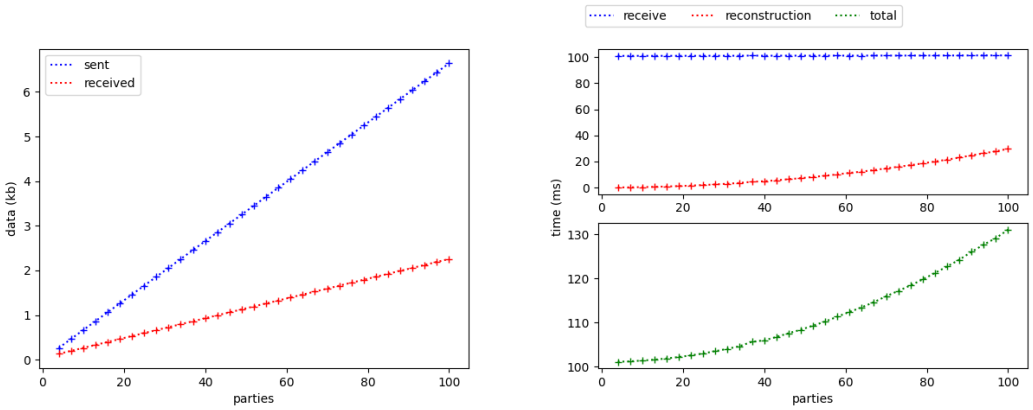

Figure 1: Execution time (right) and communication (left) of $\Pi_{\texttt{sign}}$.

Results of experiment 2. The results from experiment 2 can be seen in Figure 2 (execution time of $\Pi_{\texttt{mul}}$) and Figure 3 (communication complexity of $\Pi_{\texttt{mul}}$). Looking at Figure 2, we see that the execution time of $\Pi_{\texttt{mul}}$ is dominated by the "recv & verify" step, which corresponds to step 5 in $\Pi_{\texttt{mul}}$ (Protocol 3). This shows that verifying the Zero-Knowledge proof is the most expensive step—by a large margin—in our multiplication protocol. On the other hand, proof creation is essentially negligible due to how we split proof creation into a receiver-specific and receiver-agnostic part. We will return to this in experiment 3. Despite the increase in *total* execution time, we see that the *per-multiplication* execution time is more flat. This is to be expected as $\Pi_{\texttt{mul}}$ performs t multiplications per invocation, and thus more multiplications are performed as n increases.

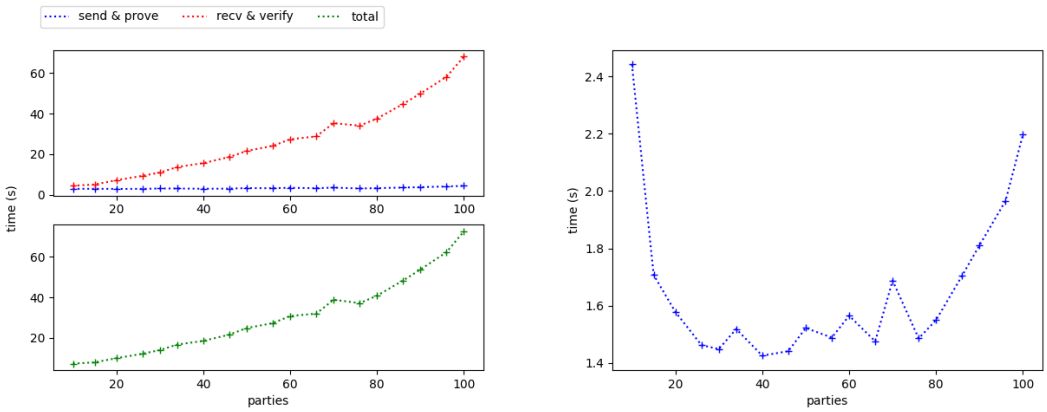

Figure 2: $\Pi_{\texttt{mul}}$ execution time. Left is total time, while right is the time per-multiplication.

Results of experiment 3. Experiment 3 complements experiment 2, by supporting the observation that proof verification is the bottleneck. See Table 1 for a breakdown of how different choices of ψ affect the prover and verifier time, as well as the size of the proofs.

Not surprisingly, we see that the verification time is not affected much by the fact that the proof has been split into two parts. This also supports the observation made

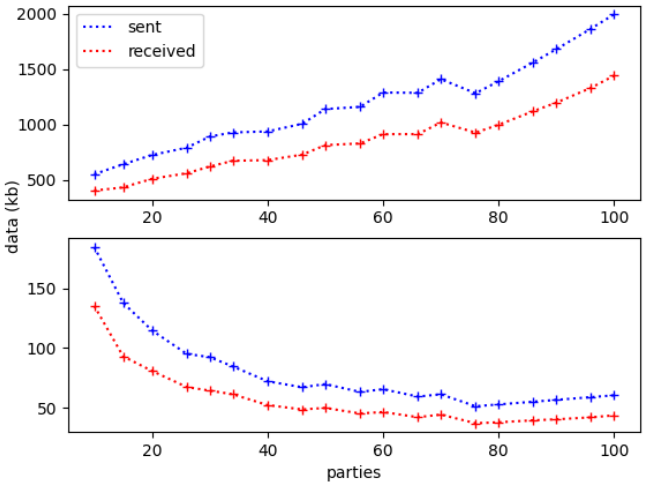

Figure 3: $\Pi_{\texttt{mul}}$: Top is total sent/received. Bottom is sent/received per multiplication.

Table 1: Timings and proof sizes for $\Pi_{\texttt{APOCM}}$ for $n = 64$. "Prover ind" denotes the party-agnostic part of the proof, while "Prover dep" is party dependent proof. For the verifier row under proof size, the size given is simply the combined size of the two proofs.

	ψ	2	4	8	16	32
time (ms)	Prover ind.	492.7	485.4	475.6	482.7	504.5
	Prover dep.	28.0	15.4	8.5	4.8	2.5
	Verifier	137.4	131.1	126.8	128.0	138.1
size (bytes)	Prover ind.	2633	2251	2125	2647	3623
	Prover dep.	823	693	563	433	303
	Verifier	3456	2944	2688	3080	3926

regarding $\Pi_{\texttt{mul}}$ earlier. Table 1 also shows the smallest party dependent proof is obtained for ψ around $n/2$, whereas the smallest proof comes from ψ close to $\sqrt{n}$. However, given that verification is a bottleneck, ψ close to $\sqrt{n}$ seems like the optimal choice, since that gives the smallest verification time.

Results from experiment 4. The results from experiment 4 can be seen in Figure 4 (execution time) and Figure 5 (communication complexity). We see that, by far, the most time is spent in $\Pi_{\texttt{agree}}$. This is not surprising, given that $\Pi_{\texttt{agree}}$ is the most expensive part of $\Pi_{\texttt{rand}}$. This holds true for both the communication complexity and execution time of $\Pi_{\texttt{rand}}$.

We further unpack the main bottleneck of $\Pi_{\texttt{rand}}$ in Figure 6. This figure shows the running time for the **Complain** and **Fill** step in $\Pi_{\texttt{agree}}$, for party P_1 and P_n, respectively. Figure 6 presents an interesting observation: Parties with small indices spend very little time in **Complain**, but a long time in **Fill**; for parties with a large index, the situation is reversed. This is a consequence of how shares are verified in the verifiable secret-sharing scheme. In particular, parties P_i with $1 \leq i \leq t$ can verify their share very efficiently—only a single exponentiation is required since their share is explicitly committed to in $\vec{\mathtt{C}}$. On the other hand, P_i with $i > t$ need to perform an interpolation in the exponent for every share they receive. However, not much is gained for parties with fast verification—they still need to wait for the slower parties in the next step.

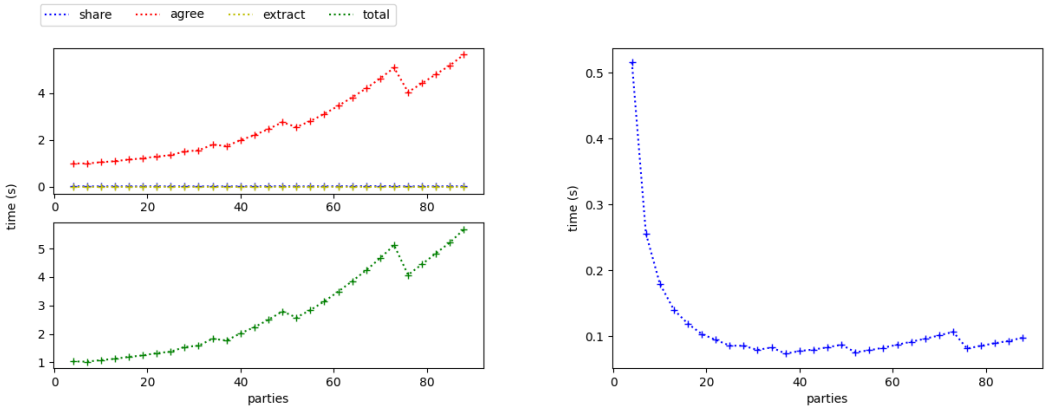

Figure 4: Left is total running time of $\Pi_{\texttt{rand}}$; right is time per random share.

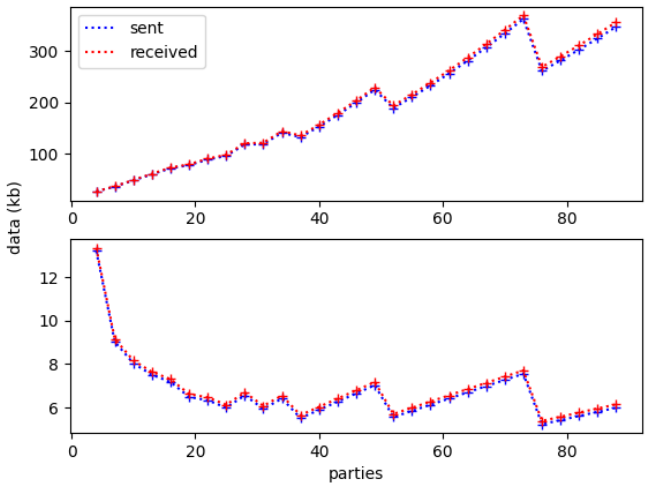

Figure 5: Top is total communication of $\Pi_{\texttt{rand}}$; bottom is communication per random share.

Results from experiment 5. The result of the final experiment is presented in Figure 7. The main observation we can note here is the positive effect that batching has on the complexity. This holds true for both the communication complexity, as well as execution time of our protocol. The lack of batching in PBCTS can clearly be seen by the more or less linear increase in both data required, as well as time needed, to produce pre-signatures. On the other hand, our protocol maintains a more flat profile, which was also visible in our previous experiments (specifically the results from experiments 2 and 4).

6.3 Discussion

Scalability. Our protocol appears to scale quite well, and if we take the comparison to a real threshold ECDSA protocol (Experiment 5) as an indication, then our protocol would be performant in a real system as well. However, we take this experiment as merely indicative. The PBCTS implementation is written in Go, and as such includes a not insignificant overhead. This overhead remains when the Go code is compiled to a shared

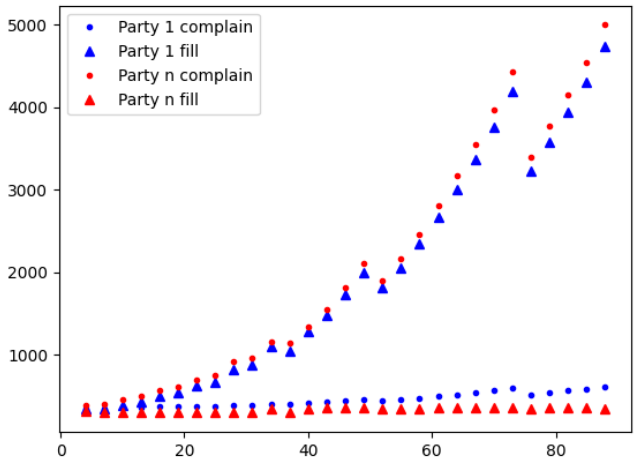

Figure 6: Execution time of P_1 vs P_n in the **Complain** and **Fill** steps of $\Pi_{\texttt{rand}}$.

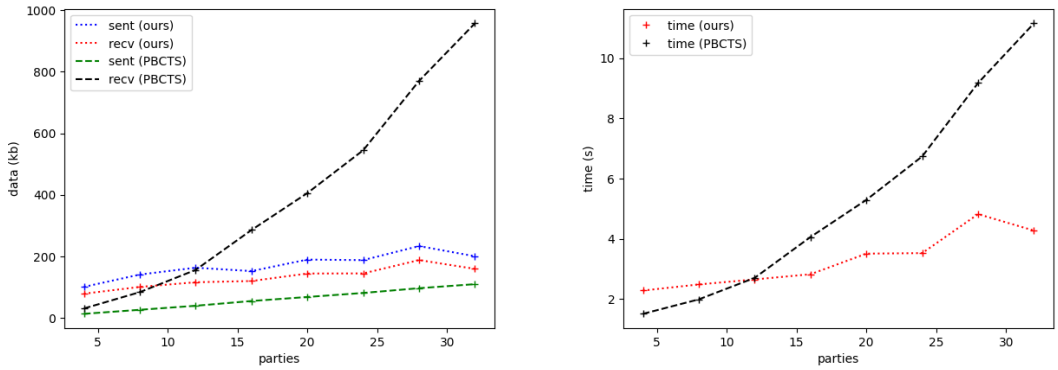

Figure 7: Experiment 5.

library (the `pbcts.so` file) since `cgo` includes the Go runtime as part of the library. That said, we believe the trend illustrated in Figure 7 would remain even if the protocol of Partisia Blockchain were re-implemented directly in C++.

Local computation appears to be a bottleneck. For both $\Pi_{\texttt{rand}}$ and $\Pi_{\texttt{mul}}$, local computation appears to be the bottleneck. For $\Pi_{\texttt{mul}}$ there is evidence for this given the expensiveness of proof verification, as shown in Experiment 3. In addition, it is not difficult to see that receiving the around 1500 kB required when running $\Pi_{\texttt{mul}}$ with $n = 100$ cannot fully explain an execution time of around 70s. For $\Pi_{\texttt{rand}}$, local computation can clearly be seen to be the bottleneck in Figure 6. The fact that local computation appears to be the bottleneck is a welcome discovery, as it is a problem which can be solved by increasing the computational power of the parties. We may also expect our protocol can be sped up relatively easily by simply using more threads.

Use of Zero-Knowledge proofs. The results in Experiments 2 and 3 show that we managed to essentially eliminate the cost of *creating* the Zero-Knowledge proofs and

thereby obtain a very efficient prover. However, we did not manage to do the same for the verifier, which is the main bottleneck. Whether or not it is possible to optimize the verification to a similar degree is an open problem.

Use of Verifiable Secret-Sharing. The results in Figure 6 showed that verification of a received share can be quite expensive, if the share is not explicitly encoded in the vector of commitments. This implies a trade-off, where verification time of parties can be reduced by instead storing more commitments. I.e., instead of storing just $t+1$ commitments (thus giving t parties fast verification), we could instead store all $n+1$ commitments, thereby giving *all* parties fast verification. The downside is of course that more data has to be sent around.

Optimizing communication patterns. The asymmetry in e.g., Figure 3 shows that our implementation sends more data than is needed. This is the reason for the gap between the "sent" and the "received" lines. This gap corresponds directly to the amount of data that a party is sent, but that it does not use. An open question is whether this can be exploited somehow. I.e., is it possible to devise a more sophisticated communication pattern such that our protocols communicate no more than is absolutely necessary. We believe this to be an interesting and non-trivial engineering task.

7 Conclusion

This work presented a robust threshold ECDSA protocol. Although our protocol utilizes techniques that date back to [FH06] and [BTH08], combining them in a way which results in a demonstrably efficient protocol required some non-trivial engineering. The protocols we presented in this work were designed not just from a theoretical perspective, but with a real world deployment in mind.

We benchmarked our protocol in a number of ways in order to support the claim that our protocol is scalable to large number of parties. Although other threshold ECDSA protocols have shown similar scalability, e.g., [DKLs19, FMM+27, DGNP25], to the best of our knowledge, we are the first to experimentally support this claim for a *robust* threshold ECDSA protocol for the setting of $t \leq (n-1)/3$ and large n. This conclusion is moreover supported by a concrete comparison against a deployed threshold ECDSA protocol.

References

[AC20] Thomas Attema and Ronald Cramer. Compressed Σ-protocol theory and practical application to plug & play secure algorithmics. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 513–543. Springer, Cham, August 2020. doi:10.1007/978-3-030-56877-1_18.

[ADEO21] Diego F. Aranha, Anders P. K. Dalskov, Daniel Escudero, and Claudio Orlandi. Improved threshold signatures, proactive secret sharing, and input certification from LSS isomorphisms. In Patrick Longa and Carla Ràfols, editors, *LATINCRYPT 2021*, volume 12912 of *LNCS*, pages 382–404. Springer, Cham, October 2021. doi:10.1007/978-3-030-88238-9_19.

[AFK22] Thomas Attema, Serge Fehr, and Michael Klooß. Fiat-Shamir transformation of multi-round interactive proofs. In Eike Kiltz and Vinod Vaikuntanathan, editors, *TCC 2022, Part I*, volume 13747 of *LNCS*, pages 113–142. Springer, Cham, November 2022. doi:10.1007/978-3-031-22318-1_5.

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

[ANO+22] Damiano Abram, Ariel Nof, Claudio Orlandi, Peter Scholl, and Omer Shlomovits. Low-bandwidth threshold ECDSA via pseudorandom correlation generators. In *2022 IEEE Symposium on Security and Privacy*, pages 2554–2572. IEEE Computer Society Press, May 2022. doi:10.1109/SP46214.2022.9833559.

[BCH+20] Christian Badertscher, Ran Canetti, Julia Hesse, Björn Tackmann, and Vassilis Zikas. Universal composition with global subroutines: Capturing global setup within plain UC. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part III*, volume 12552 of *LNCS*, pages 1–30. Springer, Cham, November 2020. doi:10.1007/978-3-030-64381-2_1.

[BIB89] Judit Bar-Ilan and Donald Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In Piotr Rudnicki, editor, *8th ACM PODC*, pages 201–209. ACM, August 1989. doi:10.1145/72981.72995.

[Blo24] Partisia Blockchain. threshold-signature-oracle. https://gitlab.com/partisiablockchain/governance/threshold-signature-oracle/-/tree/main/go, 2024.

[BMP22] Constantin Blokh, Nikolaos Makriyannis, and Udi Peled. Efficient asymmetric threshold ECDSA for MPC-based cold storage. Cryptology ePrint Archive, Report 2022/1296, 2022. URL: https://eprint.iacr.org/2022/1296.

[BTH08] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure MPC with linear communication complexity. In Ran Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 213–230. Springer, Berlin, Heidelberg, March 2008. doi:10.1007/978-3-540-78524-8_13.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001. doi:10.1109/SFCS.2001.959888.

[CC25] Ran Canetti and Megan Chen. Universally composable succinct vector commitments and applications. In Benny Applebaum and Huijia (Rachel) Lin, editors, *TCC 2025, Part IV*, volume 16271 of *LNCS*, pages 642–674. Springer, Cham, December 2025. doi:10.1007/978-3-032-12290-2_21.

[CGG+20] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1769–1787. ACM Press, November 2020. doi:10.1145/3372297.3423367.

[CL15] Guilhem Castagnos and Fabien Laguillaumie. Linearly homomorphic encryption from DDH. In Kaisa Nyberg, editor, *CT-RSA 2015*, volume 9048 of *LNCS*, pages 487–505. Springer, Cham, April 2015. doi:10.1007/978-3-319-16715-2_26.

[CL24] Yi-Hsiu Chen and Yehuda Lindell. Feldman's verifiable secret sharing for a dishonest majority. *CiC*, 1(1):16, 2024. doi:10.62056/ak2isgvtw.

[CSW22] Ran Canetti, Pratik Sarkar, and Xiao Wang. Triply adaptive UC NIZK. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part II*, volume 13792 of *LNCS*, pages 466–495. Springer, Cham, December 2022. doi:10.1007/978-3-031-22966-4_16.

[Dal24] Anders Dalskov. SCL (Secure Computation Library). utility library for prototyping MPC applications. https://github.com/anderspkd/secure-computation-library, 2024.

[Des88] Yvo Desmedt. Society and group oriented cryptography: A new concept. In Carl Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, pages 120–127. Springer, Berlin, Heidelberg, August 1988. doi:10.1007/3-540-48184-2_8.

[DGNP25] Hila Dahari-Garbian, Ariel Nof, and Luke Parker. Trout: Two-round threshold ECDSA from class groups. In Chun-Ying Huang, Jyh-Cheng Chen, Shiuh-Pyng Shieh, David Lie, and Véronique Cortier, editors, *ACM CCS 2025*, pages 380–393. ACM Press, October 2025. doi:10.1145/3719027.3765192.

[DJN+20] Ivan Damgård, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Jakob Illeborg Pagter, and Michael Bæksvang Østergaard. Fast threshold ECDSA with honest majority. In Clemente Galdi and Vladimir Kolesnikov, editors, *SCN 20*, volume 12238 of *LNCS*, pages 382–400. Springer, Cham, September 2020. doi:10.1007/978-3-030-57990-6_19.

[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*, pages 980–997. IEEE Computer Society Press, May 2018. doi:10.1109/SP.2018.00036.

[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*, pages 1051–1066. IEEE Computer Society Press, May 2019. doi:10.1109/SP.2019.00024.

[DKLs24] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. Threshold ECDSA in three rounds. In *2024 IEEE Symposium on Security and Privacy*, pages 3053–3071. IEEE Computer Society Press, May 2024. doi:10.1109/SP54263.2024.00178.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 572–590. Springer, Berlin, Heidelberg, August 2007. doi:10.1007/978-3-540-74143-5_32.

[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 Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider, editors, *ESORICS 2020, Part II*, volume 12309 of *LNCS*, pages 654–673. Springer, Cham, September 2020. doi:10.1007/978-3-030-59013-0_32.

[eic25] eichhorl. Chain-key signing performance improvements. https://forum.dfinity.org/t/chain-key-signing-performance-improvements/64672, 2025.

[Fel87] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th FOCS*, pages 427–437. IEEE Computer Society Press, October 1987. doi:10.1109/SFCS.1987.4.

[FH06] Matthias Fitzi and Martin Hirt. Optimally efficient multi-valued Byzantine agreement. In Eric Ruppert and Dahlia Malkhi, editors, *25th ACM PODC*, pages 163–168. ACM, July 2006. doi:10.1145/1146381.1146407.

[Fis05] Marc Fischlin. Communication-efficient non-interactive proofs of knowledge with online extractors. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 152–168. Springer, Berlin, Heidelberg, August 2005. doi:10.1007/11535218_10.

[FKP16] Manuel Fersch, Eike Kiltz, and Bertram Poettering. On the provable security of (EC)DSA signatures. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1651–1662. ACM Press, October 2016. doi:10.1145/2976749.2978413.

[FKP17] Manuel Fersch, Eike Kiltz, and Bertram Poettering. On the one-per-message unforgeability of (EC)DSA and its variants. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part II*, volume 10678 of *LNCS*, pages 519–534. Springer, Cham, November 2017. doi:10.1007/978-3-319-70503-3_17.

[FMM+24] Offir Friedman, Avichai Marmor, Dolev Mutzari, Omer Sadika, Yehonatan C. Scaly, Yuval Spiizer, and Avishay Yanai. 2PC-MPC: Emulating two party ECDSA in large-scale MPC. Cryptology ePrint Archive, Report 2024/253, 2024. URL: https://eprint.iacr.org/2024/253.

[FMM+27] Offir Friedman, Avichai Marmor, Dolev Mutzari, Yehonatan C. Scaly, and Yuval Spiizer. Practical zero-trust threshold signatures in large-scale asynchronous networks. In Kangkook Jee and Aniket Kate, editors, *Applied Cryptography and Network Security*, pages 473–500, Cham, 2027. Springer Nature Switzerland. doi:10.1007/978-3-032-32560-0_17.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Berlin, Heidelberg, August 1987. doi:10.1007/3-540-47721-7_12.

[GG18] Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1179–1194. ACM Press, October 2018. doi:10.1145/3243734.3243859.

[GJKR96] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Robust threshold DSS signatures. In Ueli M. Maurer, editor, *EUROCRYPT'96*, volume 1070 of *LNCS*, pages 354–371. Springer, Berlin, Heidelberg, May 1996. doi:10.1007/3-540-68339-9_31.

[GJKR01] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Robust threshold dss signatures. *Information and Computation*, 164(1):54–84, 2001. URL: https://www.sciencedirect.com/science/article/pii/S0890540100928815, doi:10.1006/inco.2000.2881.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *17th ACM STOC*, pages 291–304. ACM Press, May 1985. doi:10.1145/22145.22178.

[GS22a] Jens Groth and Victor Shoup. Design and analysis of a distributed ECDSA signing service. Cryptology ePrint Archive, Report 2022/506, 2022. URL: https://eprint.iacr.org/2022/506.

[GS22b] Jens Groth and Victor Shoup. On the security of ECDSA with additive key derivation and presignatures. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part I*, volume 13275 of *LNCS*, pages 365–396. Springer, Cham, May / June 2022. doi:10.1007/978-3-031-06944-4_13.

[GS24] Jens Groth and Victor Shoup. Fast batched asynchronous distributed key generation. In Marc Joye and Gregor Leander, editors, *EUROCRYPT 2024, Part V*, volume 14655 of *LNCS*, pages 370–400. Springer, Cham, May 2024. doi:10.1007/978-3-031-58740-5_13.

[HK23] Dominik Hartmann and Eike Kiltz. Limits in the provable security of ECDSA signatures. In Guy N. Rothblum and Hoeteck Wee, editors, *TCC 2023, Part IV*, volume 14372 of *LNCS*, pages 279–309. Springer, Cham, November / December 2023. doi:10.1007/978-3-031-48624-1_11.

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1:36–63, 2001. doi:10.1007/s102070100002.

[KMOS21] Yashvanth Kondi, Bernardo Magri, Claudio Orlandi, and Omer Shlomovits. Refresh when you wake up: Proactive threshold wallets with offline devices. In *2021 IEEE Symposium on Security and Privacy*, pages 608–625. IEEE Computer Society Press, May 2021. doi:10.1109/SP40001.2021.00067.

[KU25] Jonathan Katz and Antoine Urban. Honest-majority threshold ECDSA with batch generation of key-independent presignatures. *CiC*, 2(1):8, 2025. doi:10.62056/a0l5wol7.

[LN18] Yehuda Lindell and Ariel Nof. Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1837–1854. ACM Press, October 2018. doi:10.1145/3243734.3243788.

[Mer88] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, pages 369–378. Springer, Berlin, Heidelberg, August 1988. doi:10.1007/3-540-48184-2_32.

[Ped92] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 129–140. Springer, Berlin, Heidelberg, August 1992. doi:10.1007/3-540-46766-1_9.

[Pet21] Michaella Pettit. Efficient threshold-optimal ECDSA. In Mauro Conti, Marc Stevens, and Stephan Krenn, editors, *CANS 21*, volume 13099 of *LNCS*, pages 116–135. Springer, Cham, December 2021. doi:10.1007/978-3-030-92548-2_7.

[Sho25] Victor Shoup. Threshold ecdsa: The key ingredient behind the internet computer's bitcoin and ethereum integrations. https://medium.com/dfinity/threshold-ecdsa-the-key-ingredient-behind-the-internet-computers-bitcoin-and-ethereum-cf22649b98a1, 2025.

[THO20a] THORChain. Thorchain publish tss whitepapers. https://medium.com/thorchain/thorchain-publish-tss-whitepapers-d3ea66913721, 2020.

[THO20b] THORChain. Thorchain tss benchmark report. https://github.com/thorchain/Resources/blob/master/Whitepapers/THORChain-TSS-Benchmark-July2020.pdf, 2020.

[TX25] Guofeng Tang and Haiyang Xue. Robust threshold ECDSA with online-friendly design in three rounds. In Marina Blanton, William Enck, and Cristina Nita-Rotaru, editors, *2025 IEEE Symposium on Security and Privacy*, pages 203–221. IEEE Computer Society Press, May 2025. doi:10.1109/SP61157.2025.00115.

[WMC24] Harry W. H. Wong, Jack P. K. Ma, and Sherman S. M. Chow. Secure multiparty computation of threshold signatures made more efficient. In *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society, 2024. URL: https://www.ndss-symposium.org/ndss-paper/secure-multiparty-computation-of-threshold-signatures-made-more-efficient/, doi:https://dx.doi.org/10.14722/ndss.2024.24601.

[WMYC23] Harry W. H. Wong, Jack P. K. Ma, Hoover H. F. Yin, and Sherman S. M. Chow. Real threshold ECDSA. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023. URL: https://www.ndss-symposium.org/ndss-paper/real-threshold-ecdsa/, doi:https://dx.doi.org/10.14722/ndss.2023.24817.

[XAX+21] Haiyang Xue, Man Ho Au, Xiang Xie, Tsz Hon Yuen, and Handong Cui. Efficient online-friendly two-party ECDSA signature. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 558–573. ACM Press, November 2021. doi:10.1145/3460120.3484803.