Threshold BBS+ From Pseudorandom Correlations

Sebastian Faust[1], Carmit Hazay[2], David Kretzler[1], and Benjamin Schlosser[1]

[1] Technical University of Darmstadt, Germany
{first.last}@tu-darmstadt.de
[2] Bar-Ilan University, Israel
carmit.hazay@biu.ac.il

Abstract. The BBS+ signature scheme is one of the most prominent solutions for realizing anonymous credentials. In particular, due to properties like selective disclosure and efficient protocols for creating and showing possession of credentials. In recent years, research in cryptography has increasingly focused on the distribution of cryptographic tasks to mitigate attack surfaces and remove single points of failure.

In this work, we present a threshold BBS+ protocol in the preprocessing model. Our protocol supports an arbitrary t-out-of-n threshold and achieves non-interactive signing in the online phase. It relies on a new pseudorandom correlation-based offline protocol producing preprocessing material with sublinear communication complexity in the number of signatures. Both our offline and online protocols are actively secure under the Universal Composability framework. Finally, we estimate the concrete efficiency of our protocol, including an implementation of the online phase. The online protocol without network latency takes less than $15ms$ for $t \leq 30$ and credentials sizes up to 10. Further, our results indicate that the influence of t on the online signing is insignificant, $< 6\%$ for $t \leq 30$, and the overhead of the thresholdization occurs almost exclusively in the offline phase.

Keywords: Threshold Signature · BBS+ · Pseudorandom Correlation Functions · Pseudorandom Correlation Generators

1 Introduction

Anonymous credential schemes are becoming increasingly important in today's digital world, where privacy and security are significant concerns. They were introduced by Chaum in 1985 [Cha85] and refined by a line of work [Che95, LRSW99, CL01, CL04, Cam06, CDHK15, CKL+15, BBDE19, YAY19]. Such schemes allow an issuing party to create credentials for users, which then can prove specific attributes about themselves without revealing their identities. This selective disclosure is particularly beneficial when individuals want to keep their personal information private but still need to prove that they are authorized to access specific resources or services. The essential properties satisfied by these schemes are *unlinkability* and *unforgeability.* While unlinkability ensures that verifiers cannot link two disclosures of credentials to the same identity, unforgeability guarantees that only the issuer can generate credentials.

The BBS+ signature scheme [ASM06, CDL16] named after the group signature scheme of Boneh, Boyen, and Shacham [BBS04] is one of the most prominent solutions for realizing anonymous credential schemes. BBS+ signatures are specifically suited for anonymous credentials because of their appealing features, including the ability to sign an array of messages while keeping the signature size constant, efficient protocols for blind signing and proving possession of a signature in zero knowledge, and selective disclosure. The scheme is already being implemented by companies, such as Trinsic [Tri23], MATTR [MAT23] and Microsoft [Mic23], and to build applications like direct anonymous attestation (DAA) [Che09, BL10, CDL16], k-times anonymous authentication [ASM06] and Intel SGX's EPID [BL11]. Recently, the scheme also attracted renewed attention in the research community [TZ23, DKL$^+$23]. Moreover, due to the prominence and real-life usage of BBS+, several organizations are actively working on a standardized specification of it, including the W3C Verifiable Credentials Group [LS23] and the IETF [LKWL23].

In recent years, research in cryptography has increasingly focused on the distribution (also called thresholdization) of cryptographic tasks. One example of this paradigm shift is the huge line of work on threshold ECDSA, including [Lin17, GG18, LN18, DKLS19, SA19, CCL$^+$20, CGG$^+$20, KMOS21, ANO$^+$22] and more. Instead of running the task on a single machine, a set of servers runs a potentially interactive protocol producing the desired output without leaking any additional information. A major reason for this development is the increased use of digital payment methods, which, in addition to apparent advantages, also bring additional security risks. In a traditional, non-distributed system, an attacker who breaks into that system gains full access to the system's capabilities, e.g., can authorize arbitrary payments. This is referred to as a single point of failure. In a distributed system, single intrusions can be tolerated; the attacker must break into various devices to access the system's capabilities. Also, in the context of credentials, it is highly desirable to avoid a single point of failure as an attacker who learns the single key can create arbitrary credentials. Due to anonymity, credentials created by the attacker cannot be detected, and due to its use in authorizing access to potential sensible data and services, the consequences might be severe. A thresholdization of the signing algorithm of the BBS+ scheme allows the replacement of the single certificate issuer by a committee of issuing servers and hence reduces the risk of an attacker getting access to sufficient secret key material to forge credentials. Additionally, threshold BBS+ enables more use cases, such as two-factor authentication, and supports enforcing company policies, such as the four-eye principle.

The thresholdization of cryptographic tasks often comes with significant overhead in computation, communication, and round complexity. This is also the case with BBS+. The BBS+ signature algorithm requires the exponentiation of the inverse of the secret key added to some random nonce. This operation is highly non-linear and hence hard to distribute. A popular approach in secure distributed computation to cope with the high complexities of protocols is to split the computation into an input-independent offline and input-dependent online phase [DPSZ12, NNOB12, WRK17a, WRK17b]. The offline phase provides precomputation material used to accelerate the online phase, which computes the desired outcome efficiently. In the setting of signature schemes, we call this precomputation material presignatures [EGM96]. In recent years, Boyle et al. [BCGI18, BCG$^+$19b, BCG$^+$20a] introduced the concept of pseudorandom correlation-based precomputation (PCP). This concept allows the generation of precomputation material with sublinear communication complexity in the amount of generated precomputation material. Recently, this technique also attracted interest in the realm of threshold signature protocols [ANO$^+$22, KOR23]. In PCP, precomputed values are generated by a pseudorandom correlation generator (PCG) or a pseudorandom correlation function (PCF). These primitives include a potentially interactive setup phase where short keys are generated and distributed. Then, in the evaluation phase, every party locally evaluates on its key and a common input. The resulting outputs look pseudorandom but still satisfy some correlation, such as oblivious linear evaluation (OLE), oblivious transfer (OT), or multiplication triples (MT). The type of correlation determines the efficiency of the instantiations.

In this work, we explore the opportunities of designing an efficient threshold BBS+ scheme in the PCP setting. We aim for a non-interactive online phase as we assess it unlikely for the servers to be closely located. The major reasons for thresholdizing the credential issuance are to avoid a single point of failure and provide robustness, which means that the system should continue functioning even if some servers are unavailable. When avoiding a single point of failure, we often want to distribute the trust among several machines and involve several legal entities, e.g., several companies, most likely not within the same local network. For robustness, it makes sense to distribute the servers over a larger area to prevent environmental influences, such as network outages, from affecting too many servers at once.

1.1 Contribution

We propose a novel t-out-of-n threshold BBS+ signature scheme secure against active corruption and arbitrary security threshold $t \leq n$. Our threshold BBS+ scheme is the first in the offline-online model and has been developed concurrently and independently to the first overall threshold BBS+ signature scheme [DKL$^+$23]. We deliberately design preprocessing in the offline phase to achieve a non-interactive online signing process. Upon receiving a signing request, servers reply with a single message without additional cross-server communication.

The communication of the offline phase is restricted to the execution of a seed generation protocol. Using techniques known as *silent preprocessing*, i.e., PCGs or PCFs, we realize the seed generation protocol with sublinear complexity in the number of signatures. Our scheme has no additional per-signature interaction during the offline phase. In particular, parties expand their seeds to valid presignatures without any communication.

While our precomputation can be instantiated with both PCGs and PCFs, we focus on PCFs for the protocol specification. Conceptually, PCFs are better suited for preprocessing signatures as PCFs allow servers to compute one presignature after another. At the same time, PCGs require the generation of a large batch of presignatures at once that need to be stored. Unlike prior work using a silent preprocessing in the context of threshold signatures [ANO$^+$22], we use the PCF primitive in a black-box way allowing for more modularity. In this process, we identify several issues in using the PCF primitive in a black-box way, extend the definitional framework of PCFs accordingly, and prove the security of existing constructions under the adapted properties.

Finally, we provide an extensive evaluation of our protocol, including an implementation and experimental evaluation of the online phase. The total runtime of the online signing protocol, is below 13.595 ms plus one round trip time of the slowest client-server connection for $t \leq 30$ signers and message arrays of size $k \leq 10$. Our benchmarks show that the influence of the number of signers on the runtime of the online protocol is minimal; increasing the number of signers from 2 to 30 increases the runtime by just $1.14\% - 5.52\%$ (for message array sizes between 2 and 50). Further, our results show that the cost of thresholdization occurs almost exclusively in the offline phase; a threshold signature on a single message array takes 7.536 ms in our protocol, while a non-threshold signature, including verification of the received signature, takes 7.248 ms; ignoring network delays which are the same in both settings.

1.2 Technical Overview

BBS+ Signatures. Let $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ be a bilinear pairing of order p with generators $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$. A BBS+ signature on a message array $\{m_\ell\}_{\ell \in [k]}$ is a tuple (A, e, s) with $A = (g_1 \cdot h_0^s \cdot \prod_{\ell \in [k]} h_\ell^{m_\ell})^{\frac{1}{x+e}}$ for random nonces $e, s \in_R \mathbb{Z}_p$, secret key $x \in \mathbb{Z}_p$ and a set of random $\mathbb{G}_1$ elements $\{h_\ell\}_{\ell \in [0..k]}$.

Distributed inverse calculation. The main difficulty in thresholdizing the BBS+ signature algorithm comes from the signing operation requiring the computation of the inverse of $x + s$ without leaking x. This highly non-linear operation is elaborate to be computed in a distributed way. Similar challenges are known from other signature schemes relying on exponentiation (or a scalar multiplication in additive notion) of the inverse of secret values, e.g., ECDSA [AHS20, CGG$^+$20, ANO$^+$22, WMYC23, BS23]. The typical approach to compute $M^{\frac{1}{y}}$ for a group element M and a secret y is to separately open $B = M^a$ and $\delta = a \cdot y$ for a secret shared random a based on the classical inversion protocol by Bar-Ilan and Beaver [BB89]. The desired result can be reconstructed by computing $M^{\frac{1}{y}} = B^{\frac{1}{\delta}}$.

Computing δ as the product of two secret shared values is still a non-linear operation requiring interaction between the parties. Nevertheless, as δ is independent of the actual message, several such values can be precomputed in an *offline* phase. As explained next, a similar, yet more involved, approach can be applied to the BBS+ protocol allowing an efficient online signing based on correlated precomputation material.

The threshold BBS+ protocol. We describe a simplified, n-out-of-n version of our threshold BBS+ protocol. Assume a BBS+ secret key x, $\mathbb{G}_1$ elements $\{h_\ell\}_{\ell\in[0..k]}$ and n servers, each having access to a preprocessed tuple $(a_i, e_i, s_i, \delta_i, \alpha_i) \in \mathbb{Z}_p^5$, in the following called presignature, such that

$$\begin{aligned} \sum_{i\in[n]} \delta_i = a(x+e), \qquad \sum_{i\in[n]} \alpha_i = as \\ \text{for } a = \sum_{i\in[n]} a_i, \qquad e = \sum_{i\in[n]} e_i, \qquad s = \sum_{i\in[n]} s_i. \end{aligned} \tag{1}$$

To sign a message array $\{m_\ell\}_{\ell\in[k]}$, each server computes $A_i := (g_1 \cdot \prod_{\ell\in[k]} g_\ell^{m_\ell})^{a_i} \cdot h_0^{\alpha_i}$ and outputs a partial signature $\sigma_i := (e_i, s_i, \delta_i, A_i)$. This allows the receiver of the partial signatures to reconstruct δ, e and s and compute

$$A = (\prod_{i\in[n]} A_i)^{\frac{1}{\delta}} = ((g_1 \cdot \prod_{\ell\in[k]} h_\ell^{m_\ell})^a \cdot h_0^{as})^{\frac{1}{a(x+e)}}$$

such that the tuple (A, e, s) constitutes a valid BBS+ signature. Each signature requires a new preprocessed tuple to prevent straightforward forgeries.

The specialized layout of our presignatures allows us to realize a non-interactive signing procedure. Using plain multiplication triples, as often done in multi-party computation protocols [Bea91, DPSZ12], would require an additional round of communication, which is highly undesirable. Our non-interactive signing procedure enables a highly efficient online signature creation and provides active security at a low cost. Given that the presignatures are created securely, it is sufficient for the signature receiver to validate the received signature to achieve active security.

The preprocessing protocol. An appealing choice for instantiating the preprocessing protocol is the promising technique of pseudorandom correlation generators (PCG) or functions (PCF), as they enable the efficient generation of correlated random tuples. More precisely, PCGs and PCFs allow two parties to expand short seeds to fresh correlated random tuples locally. While the distributed generation of the seeds requires more involved protocols and general-purpose multi-party computation, the seed size and the communication complexity of the generating protocols are sublinear in the size of the expanded correlated tuples [BCGI18, BCG$^+$19b].

When using PCGs, the parties must expand all the PCG's outputs at once in a single batch, which has to be kept in storage. These batches need to be rather large to amortize the cost of the expensive setup procedure; prior work using similar correlations reports 2^{16} - 2^{25}[ANO$^+$22, BCG$^+$20b] tuples to be reasonable. In contrast, PCFs allow for generating individual tuples ad-hoc, removing the necessity of storing a large amount of preprocessing material. We do not expect thousands of signatures to be issued in short intervals, so we assess PCFs as better suited for preprocessing threshold signatures. While state-of-the-art solutions of PCFs incur high storage costs for PCF keys, we deem them conceptually better suited. Moreover, the research on PCF is a young field. Thus we expect further progress in this direction.

The correlated pseudorandom presignatures required by our online signing procedure are specifically tailored to the BBS+ setting (cf. (1)). For these specific presignatures, there exist no tailored PCG or PCF constructions. Instead, we show how to obtain these presignatures from simple correlations. Specifically, we leverage *oblivious linear evaluation* (OLE) and *vector oblivious linear evaluation* (VOLE) correlations. For both of these correlations, there exist PCG and PCF constructions [BCGI18, BCG$^+$19b, BCG$^+$20a, BCG$^+$20b, CRR21, OSY21, BCG$^+$22]. An OLE tuple is a 2-party correlation, in which party P_1 gets random values (a, u) and party P_2 gets random values (s, v) such that $a \cdot s = u + v$. A VOLE tuple provides the same correlation but fixes b over all tuples computed by the particular PCG or PCF instance. In these tuples, we call a and s the *input* value of party P_1 and P_2. Further, the PCGs/PCFs used by our protocol provide a so-called *reusability* feature, allowing parties to fix the *input* values over several PCG/PCF instances.

If parties want to compute shares of $\alpha = \sum_{i\in[n]} a_i \cdot \sum_{i\in[n]} s_i$, each party P_i computes $a_i s_i$ locally and uses an OLE correlation to get an additive share of the cross terms $a_i s_j$ and $a_j s_i$ for $j \neq i$. As a- and s-values are input values, they can be fixed over several PCG/PCF instances. By summing up $a_i s_i$ with all the additive shares of the OLE relations, P_i gets an additive share of α. Shares of $a \cdot e$ and $a \cdot x$ are computed accordingly and combined to shares of $a \cdot (e + x)$.

Using PCFs in a black-box way. Boyle et al. [BCG$^+$20a] define pseudorandom correlation functions (PCFs) and provide constructions for different correlations, such as VOLE, based on function secret sharing of a family of weak pseudorandom functions (PRFs). They differentiate between weak and strong PCFs. Similar to weak and strong PRFs, the definition of weak PCF considers random inputs, while a strong PCF allows the adversary to query PCF outputs on arbitrary inputs. [BCG$^+$20a] also shows a generic transformation from weak to strong PCF in the programmable random oracle model.

In our work, we aim to deal with PCFs in a black-box way such that we can instantiate our protocols with arbitrary PCFs fulfilling our requirements. These requirements include the active security setting and the opportunity to reuse inputs, as emphasized above. We rely on strong PCFs to cover active security and allow the adversary to choose arbitrary input. While Boyle et al. [BCG$^+$20a] lay out the foundations for the reusability property, which they call programmability, they define the property only in the passive security setting. We make the following changes to cover active security.

First, we allow the adversary to specify its input in generating PCF keys. To capture this behavior, we introduce a new *key indistinguishability* property, informally stating that the adversary cannot learn any information about the other party's input from its key. Second, the basic correctness and security properties must also hold if the adversary specifies its input to the key generation. Therefore, we extend the existing definitions of these properties by the additional power of the adversary. Note that this extension is only required for PCFs with reusable inputs, as the adversary cannot provide input to the key generation otherwise. As the definitions of strong PCFs and reusable PCFs are linked together, we merge them. Finally, our resulting definition of *strong reusable PCFs* captures the reusability feature in the active security setting. We prove that the VOLE PCF construction by Boyle et al. [BCG$^+$20a] fulfills our new definition. Additionally, we present an extension of this construction for OLE correlations and again show its security.

The t-out-of-n setting. So far, we discussed a setting where n-out-of-n servers must contribute to the signature creation. However, it is highly desirable for many use cases to support the more flexible t-out-of-n setting with $t \leq n$. In this setting, the secret key material is distributed to n servers, but only t must contribute to the signing protocol. A threshold $t \leq n$ improves the flexibility and robustness of the signing process, as not all servers must be online.

The typical approach in the t-out-of-n setting is to share the secret key material using Shamir's secret sharing [Sha79] instead of an additive sharing as done above. While additive shares are reconstructed by summation, Shamir-style shares must be aggregated using Lagrange interpolation, either on the client- or server-side. While reconstruction on the client side is favorable as it increases flexibility and reduces coordination, our PCF-based precomputation poses challenges that urge us to reconstruct on the server side. While this design choice is not optimal, prior threshold signature schemes leveraging PCF/PCGs (e.g., [ANO$^+$22, KOR23]) achieves only n-out-of-n, in contrast to a flexible t-out-of-n setting.

On a technical level, the challenge for client-side reconstruction is due to (V)OLE correlations providing us with 2-party additive sharing of multiplications, e.g., $u_{i,j} + v_{i,j} = a_i s_j$. For a product of two additively shared values $a \cdot s$, we can rewrite the product as $\sum_{i\in[n]} a_i \cdot \sum_{i\in[n]} s_i = \sum_{i\in[n]} \sum_{j\in[n]} a_i s_j = \sum_{i\in[n]} \sum_{j\in[n]} u_{i,j} + v_{i,j}$. Here, $u_{i,j}$ and $v_{i,j}$ can be interpreted as additive shares of the product. These additive shares are sufficient for the n-out-of-n setting. However, it is unclear how (V)OLE outputs can be transformed to Shamir-style sharing of $a \cdot s$ required for t-out-of-n.

We therefore incorporate a share conversion mechanism from Shamir-style shared key material into additively shared presignatures on the server side. Our mechanism consists of the servers applying the corresponding Lagrange interpolation directly to the outputs of the VOLE correlation. More precisely, as described above, each party P_i gets additive shares of the cross terms $a_i x_j$ and $a_j x_i$ for every other party P_j. Let $c_{i,j}$ be the additive share of $a_i x_j$, then party P_i multiplies the Lagrange coefficient $L_{j,\mathcal{T}}$ to this share and $L_{i,\mathcal{T}}$ to $c_{j,i}$, where $\mathcal{T}$ is the set of t signers. To enable the servers to compute the interpolation, the client provides the set of servers as part of the signing request. Note that the signer set can be obtained by hashing the message to reduce bandwidth complexity.

1.3 Related Work

Most related to our work is the work by Gennaro et al. [GGI19] and Doerner et al. [DKL$^+$23], proposing threshold protocols for the BBS+ signing algorithm. While [GGI19] focuses on a group signature scheme with threshold issuance based on the BBS signatures, their techniques can be directly applied to BBS+. [DKL$^+$23] presents a threshold anonymous credential scheme based on BBS+. Both schemes compute the inverse using classical techniques of Bar-Ilan and Beaver [BB89]. Moreover, they realize the multiplication of two secret shared values by multiplying each pair of shares. While [GGI19] uses a three-round multiplication protocol based on an additively homomorphic encryption scheme, [DKL$^+$23] integrates a two-round OT-based multiplier. Although the OT-based multiplier requires a one-time setup, both schemes do not use precomputed values per signing request. This is in contrast to our scheme, but at the cost of requiring several rounds of communication during signing. In addition, they show security in a model tailored to the BBS+ signature scheme, while we consider a more generic threshold signature ideal functionality.

In the non-threshold setting, Tessaro and Zhu [TZ23] show that short BBS+ signatures, where the signature consists only of A and e, are also secure under the q-SDH assumption. Their results suggest removing s to reduce the signature size to one group element and a scalar. Like prior proofs of BBS+, their security proof in the standard model incurs a multiplicative loss. However, they present a tight proof in the Algebraic Group Model [FKL18]. We elaborate on the influence of their work on our evaluation in Appendix J.

A different signature scheme for anonymous credentials is proposed by Caminsch and Lysyanskaya [CL04]. However, these CL signatures are based on RSA. Therefore, they have large keys, credentials, and files for revocation. Furthermore, the generation of signing keys takes a long time (10-20 seconds) due to finding a large set of random prime numbers.

Another anonymous credential scheme with threshold issuance, called Coconut, is proposed by Sonnino et al. [SAB$^+$19] and the follow-up work by Rial and Piotrowska [RP22]. Their scheme is based on the Pointcheval-Sanders (PS) signature scheme, which is less popular than BBS+ and is secure under an interactive assumption similar but different to the LRSW assumption. However, the structure of PS signatures enables a non-interactive issuance phase without coordination or precomputation. While their scheme also supports multi-attribute credentials and selective disclosure, in the case of multi-attribute credentials, the secret and public key size increases linearly in the number of attributes. In BBS+, the key size is constant. The security of Coconut was not shown under concurrent composition while our scheme is analyzed in the Universal Composability framework.

Like our work, [ANO$^+$22] and [KOR23] leverage pseudorandom correlation for threshold signatures. [ANO$^+$22] presents a ECDSA scheme, while [KOR23] focuses on Schnorr signatures. In contrast to our scheme, [ANO$^+$22] constructs a tailored PCG, presents an n-out-of-n protocol without a flexible threshold, and requires interaction in the presigning phase. [KOR23] introduces the new notion of a discrete log PCF, and construct a 2-party protocol based on this primitive. In contrast to our work, [KOR23] captures only the 2-out-of-2 setting and takes two rounds for signing.

2 Preliminaries

Throughout this work, we denote the security parameter by $\lambda \in \mathbb{N}$, the set $\{1, \ldots, k\}$ as $[k]$, the set $\{0, 1, \ldots, k\}$ as $[0..k]$, the number of parties by n and a specific party by P_i. The set of indices of corrupted parties is denoted by $\mathcal{C} \subsetneq [n]$ and honest parties are denoted by $\mathcal{H} = [n] \setminus \mathcal{C}$.

We model our protocol in the Universal Composability (UC) framework by Canetti [Can01]. We refer to Appendix A for a brief introduction to UC. We model a malicious adversary corrupting up to $t \leq n$ parties. We consider static corruption and a rushing adversary. Moreover, our protocols are in the synchronous communication model.

2.1 Bilinear Maps

We briefly recall the basics of bilinear maps following [BF01, BBS04]. Let BGen be a randomized algorithm that on input a security parameter λ outputs a prime p, such that $\log_2(p) = O(\kappa)$, three cyclic groups $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ of prime order p, generators g_1 of $\mathbb{G}_1$ and g_2 of $\mathbb{G}_2$, and a pairing $\mathsf{e} : \mathbb{G}_1 \times \mathbb{G} \rightarrow \mathbb{G}_T$.

We call e a bilinear map if the following properties hold:

- Bilinearity: For all $u \in \mathbb{G}_1$, $v \in \mathbb{G}_2$ and $a, b \in \mathbb{Z}_p$, we have $\mathsf{e}(u^a, v^b) = \mathsf{e}(u, v)^{ab}$.
- Non-degeneracy: For generators g_1 of $\mathbb{G}_1$ and g_2 of $\mathbb{G}_2$ it holds that $\mathsf{e}(g_1, g_2) \neq 1$. Since $\mathbb{G}_T$ is of prime order p, this implies that $\mathsf{e}(g_1, g_2)$ is a generator of $\mathbb{G}_T$.
- Efficiency: e can be computed efficiently in polynomial time in λ.

The literature differentiates between three types of pairings [GPS06]: Type-1 with $\mathbb{G}_1 = \mathbb{G}_2$, Type-2 with $\mathbb{G}_1 \neq \mathbb{G}_2$ and existence of an efficiently computable isomorphism $\phi : \mathbb{G}_2 \rightarrow \mathbb{G}_1$, and Type-3 with $\mathbb{G}_1 \neq \mathbb{G}_2$ and no such isomoprhism ϕ.

2.2 The BBS+ Signature Scheme

Let k be the size of the message arrays, $\mathcal{G} = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, g_1, g_2, \mathsf{e})$ be a bilinear mapping tuple and $\{h_\ell\}_{\ell \in [0..k]}$ be random elements of $\mathbb{G}_1$. The BBS+ signature scheme is defined as follows:

- $\mathsf{KeyGen}(\lambda)$: Sample $x \overset{\$}{\leftarrow} \mathbb{Z}_p^*$, compute $y = g_2^x$, and output $(\mathsf{pk}, \mathsf{sk}) = (y, x)$.
- $\mathsf{Sign}_{\mathsf{sk}}(\{m_\ell\}_{\ell \in [k]} \in \mathbb{Z}_p^k)$: Sample $e, s \overset{\$}{\leftarrow} \mathbb{Z}_p$, compute $A := (g_1 \cdot h_0^s \cdot \prod_{\ell \in [k]} h_\ell^{m_\ell})^{\frac{1}{x+e}}$ and output $\sigma = (A, e, s)$.
- $\mathsf{Verify}_{\mathsf{pk}}(\{m_\ell\}_{\ell \in [k]} \in \mathbb{Z}_p^k, \sigma)$: Output 1 iff $\mathsf{e}(A, y \cdot g_2^e) = \mathsf{e}(g_1 \cdot h_0^s \cdot \prod_{\ell \in [k]} h_\ell^{m_\ell}, g_2)$

The BBS+ signature scheme is proven strong unforgeable under the q-strong Diffie Hellman (SDH) assumption for pairings of type 1, 2, and 3 [ASM06, CDL16, TZ23]. Intuitively, strong unforgeability means that the attacker is not possible to come up with a forgery even for messages that have been signed before. We refer to [TZ23] for further details.

Optimized scheme of Tessaro and Zhu [TZ23] Concurrently to our work, Tessaro and Zhu showed an optimized version of the BBS+ signatures, reducing the signature size. In their scheme, the signer samples only one random value, $e \overset{\$}{\leftarrow} \mathbb{Z}_p$, computes $A := (g_1 \cdot \prod_{\ell \in [k]} h_\ell^{m_\ell})^{\frac{1}{x+e}}$, and outputs $\sigma = (A, e)$. The verification works as before, with the only difference that the term h_0^s is removed. Note that if the first message m_1 is sampled randomly, then the short version is equal to the original version. While we describe our protocol in the original BBS+ scheme by Au et al. [ASM06], we elaborate on the influence of [TZ23] on our evaluation in Appendix J.

3 Reusable Pseudorandom Correlation Function

On a high level, a pseudorandom correlation function (PCF) allows two parties to generate a large amount of correlated randomness from short seeds. PCF extends the notion of a pseudorandom correlation generator (PCG) in a similar way as a pseudorandom function extends a pseudorandom generator. While a PCG generates a large batch of correlated randomness during one-time expansion, a PCF allows the creation of correlation samples on the fly.

A PCF consists of two algorithms, Gen and Eval. The Gen algorithm computes a pair of short keys distributed to two parties. Then, each party can locally evaluate the Eval algorithm using its key and public input to generate an output of the target correlation. One example of such a correlation is the oblivious linear evaluation (OLE) correlation, defined by a pair of random values (y_0, y_1) where $y_0 = (a, u)$ and $y_1 = (b, v)$ such that $v = ab + u$. Other meaningful correlations are oblivious transfer (OT) and multiplication triples.

PCFs are helpful in two- and multi-party protocols, where parties first set up correlated randomness and then use this data to speed up the computation [DILO22, ANO$^+$22, KOR23].

This section presents our definition of reusable PCFs, extending the definition of a programmable PCF from [BCG$^+$20a], which is stated in Appendix B for completeness. Furthermore, we state constructions of reusable PCFs and argue why they satisfy our new definition in Appendix C.

Our modifications and extensions of the definition [BCG$^+$20a] reflect the challenges we faced when using PCFs as black-box primitives in our threshold BBS+ protocol. We present our definition and highlight these challenges and changes in the following.

3.1 Definition

As mentioned above, a pseudorandom correlation function (PCF) realizes a target correlation $\mathcal{Y}$. For some correlations, like VOLE, parts of the correlation outputs are fixed over all outputs. In the example of VOLE, where the correlation is $v = ab + u$ over some ring R, the b value is fixed for all correlation tuples.

Additionally, in a multi-party setting, we like PCF constructions that allow parties to obtain the same values for parts of the correlation output in multiple PCF instances. Concretely, assume party P_i evaluates one VOLE PCF instance with party P_j and one with party P_k. P_i evaluates the PCF to $(a_{i,j}, u_{i,j})$ for the first instance and $(a_{i,k}, u_{i,k})$ for the second instance. Here, we want to give party P_i the opportunity to get $a_{i,j} = a_{i,k}$ when applied on the same input. This property is necessary to construct multi-party correlations from two-party PCF instances.

To formally model the abovementioned properties, we define a *target correlation* as a tuple of probabilistic algorithms $(\mathsf{Setup}, \mathcal{Y})$, where Setup takes two inputs and creates a master key mk. These inputs enable fixing parts of the correlation, e.g., the fixed value b. Algorithm $\mathcal{Y}$ uses the master key and an index i to sample correlation outputs. The index i helps to sample the same value if one of the Setup inputs is identical for multiple invocations.

Finally, we follow [BCG$^+$20a] and require a target correlation to be reverse-sampleable to facilitate a suitable definition of PCFs. In contrast to [BCG$^+$20a], our definition of a target correlation explicitly considers the reusability of values over multiple invocations.

In the following, we formally define a *reverse-sampleable and indexable correlation with setup.*

Definition 1 (Reverse-sampleable and indexable correlation with setup). *Let $\ell_0(\lambda), \ell_1(\lambda) \leq \mathsf{poly}(\lambda)$ be output length functions. Let $(\mathsf{Setup}, \mathcal{Y})$ be a tuple of probabilistic algorithms, such that Setup on input 1^λ and two parameters ρ_0, ρ_1 returns a master key mk and $\mathcal{Y}$ on input 1^λ, mk, and index i returns a pair of outputs $(y_0^{(i)}, y_1^{(i)}) \in \{0,1\}^{\ell_0(\lambda)} \times \{0,1\}^{\ell_1(\lambda)}$.*

We say that the tuple $(\mathsf{Setup}, \mathcal{Y})$ defines a reverse-sampleable and indexable correlation with setup *if there exists a probabilistic polynomial time algorithm $\mathsf{RSample}$ that takes as input $1^\lambda, \mathsf{mk}, \sigma \in \{0,1\}, y_\sigma^{(i)} \in \{0,1\}^{\ell_\sigma(\lambda)}$ and i, and outputs $y_{1-\sigma}^{(i)} \in \{0,1\}^{\ell_{1-\sigma}(\lambda)}$, such that for all $\mathsf{mk}, \mathsf{mk}'$ in the range of Setup, all $\sigma \in \{0,1\}$ and all $i \in \{0,1\}^*$ the following distributions are statistically close:*

$$\{(y_0^{(i)}, y_1^{(i)}) | (y_0^{(i)}, y_1^{(i)}) \xleftarrow{\$} \mathcal{Y}(1^\lambda, \mathsf{mk}, i)\}$$

$$\{(y_0^{(i)}, y_1^{(i)}) | (y_0'^{(i)}, y_1'^{(i)}) \xleftarrow{\$} \mathcal{Y}(1^\lambda, \mathsf{mk}', i),$$
$$y_\sigma^{(i)} \leftarrow y_\sigma'^{(i)}, y_{1-\sigma}^{(i)} \leftarrow \mathsf{RSample}(1^\lambda, \mathsf{mk}, \sigma, y_\sigma, i)\}$$

Given the definition of a reverse-sampleable and indexable correlation with setup, we define our primitive called *strong reusable PCF* (srPCF). Our definition builds on the definition of a strong PCF of Boyle et al. [BCG$^+$20a] and extends it by a reusability feature. Note that [BCG$^+$20a] presents a separate definition of this reusability feature for PCFs, but this property also affects the other properties of a PCF. Therefore, we merge these definitions. Additionally, the reusability definition of Boyle et al. works only for the semi-honest setting, while our definition covers malicious adversaries.

A PCF must fulfill two properties. First, the pseudorandomness property intuitively states that the joint outputs of the Eval algorithm are computationally indistinguishable from outputs of the correlation $\mathcal{Y}$. Second, the security property intuitively guarantees the output being pseudorandomly even when knowing one key.

Similarly to the notions of weak and strong PRFs, there exist the notions of *weak* and *strong* PCFs. For a weak PCF, we consider the Eval algorithm to be executed on randomly chosen inputs, while for a strong PCF, we consider arbitrarily chosen inputs. Boyle et al. [BCG$^+$20a] showed a generic transformation from a weak to a strong PCF using a hash function modeled as a programmable random oracle. We use this transformation later in constructing srPCFs.

A PCF needs to meet two additional requirements to satisfy the reusability features. First, an adversary cannot learn any information about the other party's input used for the key generation from its own key. This is modeled by the key indistinguishability property and the corresponding game in Figure 3. On a high level, the game captures the fact that the adversary cannot tell what value was used by the honest party for the PCF key generation, given the key of the corrupted party. To model this, the challenger samples two random values and uses one for the key generation. Then, given the corrupted party's key and random values, the adversary has to identify which value was used. Second, two efficiently computable functions must exist to compute the reusable parts of the correlation from the setup input and the public evaluation input. Formally, we state the definition of a strong reusable PCF next.

Definition 2 (Strong reusable pseudorandom correlation function (srPCF)). *Let* $(\mathsf{Setup}, \mathcal{Y})$ *be a reverse-sampleable and indexable correlation with setup which has output length functions* $\ell_0(\lambda), \ell_1(\lambda)$*, and let* $\lambda \leq n(\lambda) \leq \mathsf{poly}(\lambda)$ *be an input length function. Let* $(\mathsf{PCF.Gen}, \mathsf{PCF.Eval})$ *be a pair of algorithms with the following syntax:*

- $\mathsf{PCF.Gen}(1^\lambda, \rho_0, \rho_1)$ *is a probabilistic polynomial-time algorithm that on input the security parameter* 1^λ *and reusable inputs* ρ_0, ρ_1 *outputs a pair of keys* $(\mathsf{k}_0, \mathsf{k}_1)$*.*
- $\mathsf{PCF.Eval}(\sigma, \mathsf{k}_\sigma, x)$ *is a deterministic polynomial-time algorithm that on input* $\sigma \in \{0,1\}$*, key* k_σ *and input value* $x \in \{0,1\}^{n(\lambda)}$ *outputs a value* $y_\sigma \in \{0,1\}^{\ell_\sigma(\lambda)}$*.*

We say $(\mathsf{PCF.Gen}, \mathsf{PCF.Eval})$ *is a strong reusable* (N, B, ϵ)*-secure pseudorandom correlation function (srPCF) for* $(\mathsf{Setup}, \mathcal{Y})$*, if the following conditions hold:*

- ***Strong pseudorandom*** $\mathcal{Y}$***-correlated outputs.*** *For every non-uniform adversary* $\mathcal{A}$ *of size* $B(\lambda)$ *asking at most* $N(\lambda)$ *queries to the oracle* $\mathcal{O}_b(\cdot)$*,* *it holds*

$$\left|\Pr[\mathsf{Exp}^{\mathsf{s\text{-}pr}}_{\mathcal{A}}(\lambda)=1]-\frac{1}{2}\right|\leq\epsilon(\lambda)$$

for all sufficiently large λ, where $\mathsf{Exp}^{\mathsf{s\text{-}pr}}_{\mathcal{A}}(\lambda)$ is as defined in Figure 1.

- ***Strong security.*** *For each $\sigma\in\{0,1\}$ and non-uniform adversary $\mathcal{A}$ of size $B(\lambda)$ asking at most $N(\lambda)$ queries to oracle $\mathcal{O}_b(\cdot)$, it holds*

$$\left|\Pr[\mathsf{Exp}^{\mathsf{s\text{-}sec}}_{\mathcal{A},\sigma}(\lambda)=1]-\frac{1}{2}\right|\leq\epsilon(\lambda)$$

for all sufficiently large λ, where $\mathsf{Exp}^{\mathsf{s\text{-}sec}}_{\mathcal{A},\sigma}(\lambda)$ is as defined in Figure 2.

- ***Programmability.*** *There exist public efficiently computable functions f_0, f_1 for which*

$$\Pr\left[\begin{array}{l}\rho_0,\rho_1\overset{\$}{\leftarrow}\{0,1\}^*,\\ (\mathsf{k}_0,\mathsf{k}_1)\leftarrow\mathsf{PCF.Gen}(1^\lambda,\rho_0,\rho_1)\\ (a,c)\leftarrow\mathsf{PCF.Eval}(0,\mathsf{k}_0,x),\\ (b,d)\leftarrow\mathsf{PCF.Eval}(1,\mathsf{k}_1,x)\end{array}:\begin{array}{c}a=f_0(\rho_0,x)\\ b=f_1(\rho_1,x)\end{array}\right]\geq 1-\mathsf{negl}(\lambda).$$

- ***Key indistinguishability.*** *For any $\sigma\in\{0,1\}$ and non-uniform adversary $\mathcal{A}=(\mathcal{A}_0,\mathcal{A}_1)$, it holds*

$$\Pr[\mathsf{Exp}^{\mathsf{key\text{-}ind}}_{\mathsf{PCF},\mathcal{A},\sigma}(\lambda)=1]\leq\frac{1}{2}+\mathsf{negl}(\lambda)$$

for all sufficiently large λ, where $\mathsf{Exp}^{\mathsf{key\text{-}ind}}_{\mathsf{PCF},\mathcal{A},\sigma}$ is as defined in Figure 3.

We say that $(\mathsf{PCF.Gen},\mathsf{PCF.Eval})$ *is a srPCF for $\mathcal{Y}$ if it is a $(p,1/p,p)$-secure sPCF for $\mathcal{Y}$ for every polynomial p. If $B=N$, we write (B,ϵ)-secure sPCF for short.*

3.2 Correlations

Next, we state the correlations that are used in our preprocessing protocol. These are the oblivious linear evaluation (OLE) and vector OLE (VOLE) correlations. We present PCF constructions realizing these correlations in Appendix C.

Our OLE correlation over ring R is given by $c_1=ab+c_0$, where $a,b,c_0,c_1\in R$. Moreover, we require a and b being computed by a weak psuedorandom function (PRF). Formally, we define the reverse-sampleable and indexable target correlation with setup $(\mathsf{Setup}_{\mathsf{OLE}},\mathcal{Y}_{\mathsf{OLE}})$ over ring R as

$$\begin{aligned}(k,k')&\leftarrow\mathsf{Setup}_{\mathsf{OLE}}(1^\lambda,k,k')\,,\\ ((F_k(i),u),(F_{k'}(i),v))&\leftarrow\mathcal{Y}_{\mathsf{OLE}}(1^\lambda,(k,k'),i)\quad\text{such that}\\ v&=F_k(i)\cdot F_{k'}(i)+u\,,\end{aligned}\tag{2}$$

```
$\underline{\mathsf{Exp}^{\text{s-pr}}_{\mathcal{A}}(\lambda):}$
$(\rho_0, \rho_1) \leftarrow \mathcal{A}_0(1^\lambda)$
$\mathsf{mk} \leftarrow \mathsf{Setup}(1^\lambda, \rho_0, \rho_1)$
$(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{PCF.Gen}(1^\lambda, \rho_0, \rho_1)$
$\mathcal{Q} = \emptyset$
$b \xleftarrow{\$} \{0,1\}$
$b' \leftarrow \mathcal{A}_1^{\mathcal{O}_b(\cdot)}(1^\lambda)$
if $b = b'$return $1$
else return $0$

$\underline{\mathcal{O}_0(x):}$
if $(x, y_0, y_1) \in \mathcal{Q}$ :
    return $(y_0, y_1)$
else :
    $(y_0, y_1) \leftarrow \mathcal{Y}(1^\lambda, \mathsf{mk}, x)$
    $\mathcal{Q} = \mathcal{Q} \cup \{(x, y_0, y_1)\}$
    return $(y_0, y_1)$

$\underline{\mathcal{O}_1(x):}$
for $\sigma \in \{0,1\}$ :
    $y_\sigma \leftarrow \mathsf{PCF.Eval}(\sigma, \mathsf{k}_\sigma, x)$
return $(y_0, y_1)$
```

Fig. 1: Strong pseudorandom $\mathcal{Y}$-correlated outputs of a PCF.

```
$\underline{\mathsf{Exp}^{\text{s-sec}}_{\mathcal{A},\sigma}(\lambda):}$
$(\rho_0, \rho_1) \leftarrow \mathcal{A}_0(1^\lambda)$
$\mathsf{mk} \leftarrow \mathsf{Setup}(1^\lambda, \rho_0, \rho_1)$
$(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{PCF.Gen}(1^\lambda, \rho_0, \rho_1)$
$b \xleftarrow{\$} \{0,1\}$
$b' \leftarrow \mathcal{A}_1^{\mathcal{O}_b(\cdot)}(1^\lambda, \sigma, \mathsf{k}_\sigma)$
if $b = b'$return $1$
else return $0$

$\underline{\mathcal{O}_0(x):}$
$y_{1-\sigma} \leftarrow \mathsf{PCF.Eval}(1-\sigma, \mathsf{k}_{1-\sigma}, x)$
return $y_{1-\sigma}$

$\underline{\mathcal{O}_1(x):}$
$y_\sigma \leftarrow \mathsf{PCF.Eval}(\sigma, \mathsf{k}_\sigma, x)$
$y_{1-\sigma} \leftarrow \mathsf{RSample}(1^\lambda, \mathsf{mk}, \sigma, y_\sigma, x)$
return $y_{1-\sigma}$
```

Fig. 2: Strong security of a PCF.

where $u, v \in R$ and F being a (PRF) with key k, k'. Note that while the Setup algorithm for our OLE and also VOLE is essentially the identity function, the algorithm might be more complex for other correlations. The reverse-sampling algorithm is defined such that $(F_{k'}(x), F_k(i) \cdot F_{k'}(x) + u) \leftarrow \mathsf{RSample}_{\mathsf{OLE}}(1^\lambda, (k, k'), 0, (F_k(i), u), i)$ and
$(F_k(i), u) \leftarrow \mathsf{RSample}_{\mathsf{OLE}}(1^\lambda, (k, k'), 1, (F_{k'}(x), v), i)$.

Next, we state the VOLE correlation. In contrast to OLE, the value b is fixed over multiple correlation samples, i.e., $\vec{c_1} = \vec{a}b + \vec{c_0}$, where each correlation sample contains one component of the vectors. We formally define the reverse-sampleable and indexable target correlation with setup $(\mathsf{Setup}_{\mathsf{VOLE}}, \mathcal{Y}_{\mathsf{VOLE}})$ over

```
$\mathsf{Exp}^{\mathsf{key\text{-}ind}}_{\mathcal{A},\sigma}(\lambda)$ :
$b \xleftarrow{\$} \{0,1\}$
$\rho^{(0)}_{1-\sigma}, \rho^{(1)}_{1-\sigma} \xleftarrow{\$} \{0,1\}^*$
$\rho_{1-\sigma} \leftarrow \rho^{(b)}_{1-\sigma}$
$\rho_\sigma \leftarrow \mathcal{A}_0(1^\lambda)$
$(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{PCF.Gen}_\mathsf{p}(1^\lambda, \rho_0, \rho_1)$
$b' \leftarrow \mathcal{A}_1(1^\lambda, \mathsf{k}_\sigma, \rho^{(0)}_{1-\sigma}, \rho^{(1)}_{1-\sigma})$
if $b' = b$ return 1
else return 0
```

Fig. 3: Key Indistinguishability of a reusable PCF.

ring R as

$$\begin{aligned}(k,b) &\leftarrow \mathsf{Setup}_{\mathsf{VOLE}}(1^\lambda, k, b)\,,\\ ((F_k(i),u),(b,v)) &\leftarrow \mathcal{Y}_{\mathsf{VOLE}}(1^\lambda, (k,b), i) \quad \text{such that}\\ v &= F_k(i)\cdot b + u\,,\end{aligned} \tag{3}$$

where $b, u, v \in R$ and F being a weak pseudorandom function (PRF) with key k. The reverse-sampling algorithm is defined such that $(b, F_k(i)\cdot b + u) \leftarrow \mathsf{RSample}_{\mathsf{VOLE}}(1^\lambda, (k,b), 0, (F_k(i), u), i)$ and $(F_k(i), u) \leftarrow \mathsf{RSample}_{\mathsf{VOLE}}(1^\lambda, (k,b), 1, (b,v), i)$.

We state PCF constructions realizing these definitions of OLE and VOLE correlations in Appendix C.

4 Threshold Online Protocol

In this section, we present our threshold BBS+ protocol. This protocol yields a signing phase without interaction between the signers and a flexible threshold parameter t. Moreover, we show the security of our protocol against a malicious adversary statically corrupting up to $t-1$ parties in the UC framework.

Section 4.1 states our modifications to the ideal functionality for threshold signature schemes introduced by Canetti et al. [CGG+20]. The full functionality is given in Appendix D. We use this functionality to prove UC security of our scheme. To be more generic, we deliberately chose the generic threshold signature functionality by Canetti et al. [CGG+20] over a specific BBS+ functionality such as the one used in [DKL+23]. Proving security under a generic threshold functionality enables our threshold BBS+ protocol to be used whenever a threshold signature scheme is required and not only when a BBS+ scheme is required.

Our protocol uses precomputation to accelerate online signing. An intuitive description of the used precomputation is given in Section 1.2. We formally model the precomputation by describing our protocol in a hybrid model where parties can access a hybrid preprocessing functionality $\mathcal{F}_{\mathsf{Prep}}$. Section 4.2 states the hybrid functionality $\mathcal{F}_{\mathsf{Prep}}$. Using a hybrid model allows us to abstract from the concrete instantiation of the preprocessing functionality. We present a concrete instantiation of $\mathcal{F}_{\mathsf{Prep}}$ in Section 5.

Finally, Section 4.3 formally states our threshold BBS+ protocol and provides proof in the UC framework. We refer the reader to the technical overview in Section 1.2 for an intuitive description of our protocol.

4.1 Ideal Threshold Signature Functionality

We base our security analysis on the ideal threshold signature functionality $\mathcal{F}_{\mathsf{tsig}}$ of Canetti et al. [CGG$^+$20]. We slightly modify the functionality in the following aspects. First, we allow the parties to specify a set of signers $\mathcal{T}$ during the signing request. Specifying $\mathcal{T}$ helps us to account for a flexible threshold of signers instead of requiring all n parties to sign. Second, we model the signed message as an array of messages. This change accounts for signature schemes allowing signing k messages simultaneously, such as BBS+. Third, we remove the identifiability property, the key-refresh, and the corruption/decorruption interface. The key-refresh and the corruption/decorruption interface are not required in our scenario as we consider a static adversary in contrast to the mobile adversary in [CGG$^+$20]. Should we add the following: Fourth, we allow only one signature per ssid to prevent attacks due to same randomness used in multiple signatures. Fifth, at the end of the signing phase, honest parties might output $\mathtt{abort}$ instead of a valid signature. This modification is due to our protocol not providing robustness or identifiable abort. The later is achieved by the protocol of [CGG$^+$20].

The full formal description is presented in Appendix D.

4.2 Ideal Preprocessing Functionality

The preprocessing functionality consists of two phases. First, the *Initialization* phase samples a private/public key pair. Second, the *Tuple* phase provides correlated tuples upon request. In this phase, the output values of the honest parties are reverse sampled, given the corrupted parties' outputs. To explicitly model the Tuple phase as non-interactive, we require the simulator to specify a function Tuple during the Initialization. This function defines the corrupted parties' output values in the Tuple phase and is computed first to reverse sample the honest parties' outputs.

Functionality $\mathcal{F}_{\mathsf{Prep}}$

The functionality $\mathcal{F}_{\mathsf{Prep}}$ interacts with parties $P_1, \ldots, P_n$ and ideal-world adversary $\mathcal{S}$. The functionality is parameterized by a threshold parameter t. During the initialization, $\mathcal{S}$ provides a tuple function $\mathsf{Tuple}(\cdot,\cdot,\cdot) \to \mathbb{Z}_p^5$.
Initialization. Upon receiving $(\mathtt{init}, \mathsf{sid})$ from all parties,

- sample the secret key $\mathsf{sk} \overset{\$}{\leftarrow} \mathbb{Z}_p$
- send $\mathsf{pk} = (g_2^{\mathsf{sk}})$ to $\mathcal{S}$. Upon receiving $(\mathtt{ok}, \mathsf{Tuple}(\cdot,\cdot,\cdot))$ from $\mathcal{S}$, send pk to every honest party.

Tuple. On input $(\mathtt{tuple}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T})$ from party P_l where $l \in \mathcal{T}$, $\mathcal{T} \subseteq [n]$ of size t do:

- If $(\mathsf{ssid}, \mathcal{T}, \{(a_i, e_i, s_i, \delta_i, \alpha_i)\}_{i\in\mathcal{T}})$ is stored, send $(a_l, e_l, s_l, \delta_l, \alpha_l)$ to P_l.
- Else, compute $(a_j, e_j, s_j, \delta_j, \alpha_j) \leftarrow \mathsf{Tuple}(\mathsf{ssid}, \mathcal{T}, j)$ for every corrupted party P_j where $j \in \mathcal{C} \cap \mathcal{T}$. Next, sample $a, e, s \overset{\$}{\leftarrow} \mathbb{Z}_p$ and tuples $(a_j, e_j, s_j, \delta_j, \alpha_j)$ over $\mathbb{Z}_p$ for $j \in \mathcal{H} \cap \mathcal{T}$ such that

$$\begin{gathered}\sum_{i\in\mathcal{T}} a_i = a \qquad \sum_{i\in\mathcal{T}} e_i = e \qquad \sum_{i\in\mathcal{T}} s_i = s \\ \sum_{\ell\in\mathcal{T}} \delta_i = a(\mathsf{sk} + e) \qquad \sum_{i\in\mathcal{T}} \alpha_i = as\end{gathered} \tag{4}$$

  Store $(\mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \{(a_i, e_i, s_i, \delta_i, \alpha_i)\}_{i\in\mathcal{T}})$ and send $(\mathsf{sid}, \mathsf{ssid}, a_l, e_l, s_l, \delta_l, \alpha_l)$ to honest party P_l.

Abort. On input $(\mathtt{abort}, \mathsf{sid})$ from $\mathcal{S}$, send $\mathtt{abort}$ to all honest parties and halt.

4.3 Online Signing Protocol

We formally state our threshold BBS+ protocol next and show its security afterward.

Construction 1: $\pi_{\mathsf{TBBS+}}$

We describe the protocol from the perspective of an honest party P_i.
Public Parameters. Number of parties n, size of message arrays k, security threshold t, a bilinear mapping tuple $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, g_1, g_2, \mathsf{e})$ and randomly sampled $\mathbb{G}_1$ elements $\{h_\ell\}_{\ell\in[0..k]}$. Let $\mathsf{Verify}_{\mathsf{pk}}(\cdot,\cdot)$ be the BBS+ verification algorithm as defined above.
KeyGen.

- Upon receiving $(\mathtt{keygen}, \mathsf{sid})$ from $\mathcal{Z}$, send $(\mathtt{init}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{Prep}}$ and receive pk in return.
- Upon receiving $(\mathtt{pubkey}, \mathsf{sid})$ from $\mathcal{Z}$ output $(\mathtt{pubkey}, \mathsf{sid}, \mathsf{Verify}_{\mathsf{pk}}(\cdot,\cdot))$.

Sign. Upon receiving $(\mathtt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m} = \{m_\ell\}_{\ell\in[k]})$ from $\mathcal{Z}$ with $P_i \in \mathcal{T}$ and no tuple $(\mathsf{sid}, \mathsf{ssid})$ is stored, perform the following steps:

1. Send $(\mathtt{tuple}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T})$ to $\mathcal{F}_{\mathsf{Prep}}$ and receive tuple $(a_i, e_i, s_i, \delta_i, \alpha_i)$.
2. Store $(\mathsf{sid}, \mathsf{ssid})$ and send $(\mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, A_i := (g_1 \cdot \prod_{\ell\in[k]} h_\ell^{m_\ell})^{a_i} \cdot h_0^{\alpha_i}, \delta_i, e_i, s_i)$ to each party $P_j \in \mathcal{T}$.
3. Once $(\mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, A_j, \delta_j, e_j, s_j)$ is received from every party $P_j \in \mathcal{T} \setminus \{P_i\}$,

(a) compute $e = \sum_{\ell \in \mathcal{T}} e_\ell$, $s = \sum_{\ell \in \mathcal{T}} s_\ell$, $\epsilon = \left(\sum_{\ell \in \mathcal{T}} \delta_\ell\right)^{-1}$, and $A = (\Pi_{\ell \in \mathcal{T}} A_\ell)^\epsilon$.

(b) If $\mathsf{Verify}_{\mathsf{pk}}(m, (A, e, s)) = 1$, set $\mathsf{out} = \sigma = (A, e, s)$. Otherwise, set $\mathsf{out} = \mathtt{abort}$. Then, output $(\mathtt{sig}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, \mathsf{out})$.

Verify. Upon receiving $(\mathtt{verify}, \mathsf{sid}, \mathbf{m} = \{m_\ell\}_{\ell \in [k]}, \sigma, \mathsf{Verify}_{\mathsf{pk}'}(\cdot, \cdot))$ from $\mathcal{Z}$ output $(\mathtt{verified}, \mathsf{sid}, \mathbf{m}, \sigma, \mathsf{Verify}_{\mathsf{pk}'}(\mathbf{m}, \sigma))$.

Theorem 1. *Assuming the strong unforgeability of BBS+, it holds that protocol $\pi_{\mathsf{TBBS+}}$ UC-realizes $\mathcal{F}_{\mathsf{tsig}}$ in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model in the presence of malicious adversaries controlling up to $t - 1$ parties.*

The proof is given in Appendix E.

4.4 Anonymous Credentials and Blind Signing

BBS+ signatures can be used to design anonymous credential schemes as follows. To receive a credential, a client sends a signing request to the servers containing its public and private credential information as a message array. Public parts of the credentials are sent in clear, while private information is blinded. The client can add zero-knowledge proofs that blinded messages satisfy some predicate. These proofs enable the issuing servers to enforce a signing policy even though they blindly sign parts of the messages. Once holding credentials, clients can prove in zero knowledge that their credential fulfills certain predicates without leaking their signature.

Our scheme must be extended by a blind-signing property to realize the described blueprint. Precisely, we require a property called *partially blind* signatures [AO00]. This property prevents the issuer from learning more details about the message to be signed besides some explicitly declared public information.

To transform our scheme into a partially blind one, we follow the approach of [ASM06]. Let $\{m_\ell\}_{\ell \in [k]}$ be the set of messages representing the client's credential information. Without loss of generality, we assume that m_k is the public part. In order to blind its messages, the client computes a Pedersen Commitment [Ped91] on the private messages: $C = g_1^{s'} \cdot \prod_{\ell \in [k-1]} h_\ell^{m_\ell}$ for a random s' and a zero-knowledge proof π that C is well-formed, i.e., that the client knows $(s', \{m_\ell\}_{\ell \in [k-1]})$. The client sends $(\mathcal{T}, C, \pi, m_k)$ and potential zero-knowledge proofs for signing policy enforcement to the servers. Each server P_i for $i \in \mathcal{T}$ replies with $(A_i = (g_1 \cdot C \cdot h_k^{m_k})^{a_i} \cdot h_0^{\alpha_i}, \delta_i, e_i, s_i)$. The client computes e, s, and A as before but outputs signature $(A, e, s^* = s' + s)$ which constitutes a valid signature.

As the blinding mechanism and the resulting signatures are equivalent in the non-threshold BBS+ setting, we can use existing zero-knowledge proofs for policy enforcement and credential usage from the non-threshold setting.

5 Threshold Preprocessing Protocol

We state our threshold BBS+ signing protocol in Section 4 in a $\mathcal{F}_{\mathsf{Prep}}$-hybrid model. Now, we present an instantiation of the $\mathcal{F}_{\mathsf{Prep}}$ functionality using pseudorandom correlation functions (PCFs). In particular, our π_{Prep} protocol builds on PCFs for VOLE and OLE correlations. The resulting protocol consists of an interactive *Initialization* and a non-interactive *Tuple* phase, consisting only of the local PCF evaluations and additional local computation. We now give an intuition of our preprocessing protocol and present formal definitions in Section 5.1-5.3. In Section 5.4 we briefly give an intuition about instantiating our precomputation pseudorandom correlation generators (PCGs) instead of PCFs.

Our preprocessing protocol consists of three steps: the first two are part of the Initialization phase, and the third one builds the Tuple phase. First, the parties set up a secret and corresponding public key. For the BBS+ signature scheme, the public key is $\mathsf{pk} = h_0^x$, while the secret key is $\mathsf{sk} = x$, which is secret-shared using Shamir's secret sharing, i.e., party P_i knows $\mathsf{sk}_i = F(i)$ for a random polynomial P with $P(0) = \mathsf{sk}$. This procedure constitutes a standard distributed key generation protocol for a DLOG-based cryptosystem. Therefore, we abstract from the concrete instantiation of this protocol and model the key generation as a hybrid functionality $\mathcal{F}_{\mathsf{KG}}$.

Second, the parties set up the keys for the PCF instances. The protocol uses two-party PCFs, meaning each pair of parties sets up required instances. At the time of writing, no PCF construction with a tailored MPC protocol for setting up the keys exists. Therefore, we model the PCF key generation as a hybrid functionality $\mathcal{F}_{\mathsf{Setup}}$.

Third, every party in the signer set of a signing request executes the Tuple phase. In this phase party P_i generates $(a_i, e_i, s_i, \delta_i, \alpha_i)$, where the values fulfill correlation (4). To this end, each party samples a_i, e_i, s_i such that the a_i values constitute an additive secret sharing of a. The same holds for e and s. Then, $\sum_{\ell \in \mathcal{T}} \alpha_\ell = as$ can be rewritten as $as = \sum_{\ell \in \mathcal{T}} a_\ell \sum_{\ell \in \mathcal{T}} s_\ell = \sum_{\ell \in \mathcal{T}} \sum_{k \in \mathcal{T}} a_\ell s_k$. Each multiplication $a_\ell s_k$ is turned into additive shares using an OLE correlation, i.e., $c_1 - c_0 = as$. The parties use PCF instances to compute this OLE correlation. Finally, party P_i locally adds $a_i s_i$ and the outputs of its PCF evaluations to get an additive sharing of as. The same idea works for computing δ_i such that $\sum_{\ell \in \mathcal{T}} \delta_\ell = a(\mathsf{sk} + e) = a\mathsf{sk} + ae$. Note that while the values a, e, s are fresh random values for each signing request, sk is fixed. Therefore, the parties use VOLE correlations to compute $a\mathsf{sk}$ instead of OLE correlations.

Note that party P_i uses PCF instances for computing additive shares of $a_i s_j$ and $a_i s_k$ for two different parties P_j and P_k. Since a_i must be the same for both products, we use reusable PCFs so parties can fix a_i over multiple PCF instances. In addition, parties evaluate the PCFs on ssid as input. As ssid is provided by the environment, we require strong PCFs. Based on these two requirements, our protocol relies on strong reusable PCFs defined in Section 3.

Next, we present the hybrid key generation functionality in Section 5.1 and the hybrid setup functionality in Section 5.2. Then, we formally state and prove our PCF-based preprocessing protocol in the $(\mathcal{F}_{\mathsf{KG}}, \mathcal{F}_{\mathsf{Setup}})$-hybrid model in Section 5.3.

5.1 Key Generation Functionality

We abstract from the concrete instantiation of the key generation. Therefore, we state a very simple key generation functionality for discrete logarithm-based cryptosystems similar to the functionality of [Wik04]. The functionality describes a standard distributed key generation for discrete logarithm-based cryptosystems and can be realized by [GJKR99, Wik04] or the key generation phase of [CGG^{+}20].

Functionality $\mathcal{F}_{\mathsf{KG}}$

The functionality is parameterized by the order of the group from which the secret key is sampled p, a generator for the group of the public key h, and a threshold parameter t. The key generation functionality interacts with parties $P_1, \ldots, P_n$ and ideal-world adversary $\mathcal{S}$.

Key Generation:

Upon receiving $(\texttt{keygen}, \mathsf{sid})$ from every party P_i and $(\texttt{corruptedShares}, \mathsf{sid}, \{\mathsf{sk}_j\}_{j \in \mathcal{C}})$ from $\mathcal{S}$:

– Sample random polynomial $F \in \mathbb{Z}_p[X]$ of degree $t-1$ such that $F(j) = \mathsf{sk}_j$ for every $j \in \mathcal{C}$.

– Set $\mathsf{sk} = F(0)$, $\mathsf{pk} = h^{\mathsf{sk}}$, $\mathsf{sk}_\ell = F(\ell)$ and $\mathsf{pk}_\ell = h^{\mathsf{sk}_\ell}$ for $\ell \in [n]$.

– Send $(\mathsf{sid}, \mathsf{sk}_\ell, \mathsf{pk}, \{\mathsf{pk}_k\}_{k \in [n]})$ to every party P_ℓ.

5.2 Setup Functionality

The setup functionality gets random values, secret key shares, and partial public keys as input from every party. Then, it first checks if the secret key shares and the partial public key match and next generates the PCF keys using the random values. Finally, it returns the generated PCF keys to the parties.

At the time of writing, no PCF construction with a tailored key generation protocol exists. Therefore, we abstract from a concrete instantiation by specifying this functionality. Nevertheless, $\mathcal{F}_{\mathsf{Setup}}$ can be instantiated using general-purpose MPC.

Functionality $\mathcal{F}_{\mathsf{Setup}}$

Let $(\mathsf{PCF}_{\mathsf{VOLE}}.\mathsf{Gen}, \mathsf{PCF}_{\mathsf{VOLE}}.\mathsf{Eval})$ be a srPCF for VOLE correlations and let $(\mathsf{PCF}_{\mathsf{OLE}}.\mathsf{Gen}, \mathsf{PCF}_{\mathsf{OLE}}.\mathsf{Eval})$ be a srPCF for OLE correlations. The setup functionality interacts with parties $P_1, \ldots, P_n$.

Setup:

Upon receiving $(\texttt{setup}, \mathsf{sid}, \rho_a^{(i)}, \rho_s^{(i)}, \rho_e^{(i)}, \mathsf{sk}_i, \{\mathsf{pk}_k^{(i)}\}_{k \in [n]})$ from every party P_i:

– Check if $h^{\mathsf{sk}_\ell} = \mathsf{pk}_\ell^{(k)}$ for every $\ell, k \in [n]$. If the check fails, send $\texttt{abort}$ to all parties.

– Else, compute for every pair of parties (P_i, P_j):

  • $(\mathsf{k}_{i,j,0}^{\mathsf{VOLE}}, \mathsf{k}_{i,j,1}^{\mathsf{VOLE}}) \leftarrow \mathsf{PCF}_{\mathsf{VOLE}}.\mathsf{Gen}(1^\lambda, \rho_a^{(i)}, \mathsf{sk}_j)$,

- $(\mathsf{k}_{i,j,0}^{(\mathsf{OLE},1)}, \mathsf{k}_{i,j,1}^{(\mathsf{OLE},1)}) \leftarrow \mathsf{PCF}_{\mathsf{OLE}}.\mathsf{Gen}(1^\lambda, \rho_a^{(i)}, \rho_s^{(j)})$, and
- $(\mathsf{k}_{i,j,0}^{(\mathsf{OLE},2)}, \mathsf{k}_{i,j,1}^{(\mathsf{OLE},2)}) \leftarrow \mathsf{PCF}_{\mathsf{OLE}}.\mathsf{Gen}(1^\lambda, \rho_a^{(i)}, \rho_e^{(j)})$.

– Send keys $(\mathsf{sid}, \mathsf{k}_{i,j,0}^{\mathsf{VOLE}}, \mathsf{k}_{j,i,1}^{\mathsf{VOLE}}, \mathsf{k}_{i,j,0}^{(\mathsf{OLE},1)}, \mathsf{k}_{j,i,1}^{(\mathsf{OLE},1)}, \mathsf{k}_{i,j,0}^{(\mathsf{OLE},2)}, \mathsf{k}_{j,i,1}^{(\mathsf{OLE},2)})_{j \neq i}$ to every party P_i.

5.3 PCF-based Preprocessing Protocol

In this section, we formally present our PCF-based preprocessing protocol in the $(\mathcal{F}_{\mathsf{KG}}, \mathcal{F}_{\mathsf{Setup}})$-hybrid model.

Construction 2: π_{Prep}

Let $(\mathsf{PCF}_{\mathsf{VOLE}}.\mathsf{Gen}, \mathsf{PCF}_{\mathsf{VOLE}}.\mathsf{Eval})$ be a srPCF for VOLE correlations and let $(\mathsf{PCF}_{\mathsf{OLE}}.\mathsf{Gen}, \mathsf{PCF}_{\mathsf{OLE}}.\mathsf{Eval})$ be a srPCF for OLE correlations.
We describe the protocol from the perspective of P_i.
Initialization. Upon receiving input $(\texttt{init}, \mathsf{sid})$, do:

1. Send $(\texttt{keygen}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{KG}}$.
2. Upon receiving $(\mathsf{sid}, \mathsf{sk}_i, \mathsf{pk}, \{\mathsf{pk}_k^{(i)}\}_{k\in[n]})$ from $\mathcal{F}_{\mathsf{KG}}$, sample $\rho_a^{(i)}, \rho_s^{(i)}, \rho_e^{(i)} \in \{0,1\}^\lambda$ and send $(\texttt{setup}, \mathsf{sid}, \rho_a^{(i)}, \rho_s^{(i)}, \rho_e^{(i)}, \mathsf{sk}_i, \{\mathsf{pk}_k^{(i)}\}_{k\in[n]})$ to $\mathcal{F}_{\mathsf{Setup}}$.
3. Upon receiving $(\mathsf{sid}, \mathsf{k}_{i,j,0}^{\mathsf{VOLE}}, \mathsf{k}_{j,i,1}^{\mathsf{VOLE}}, \mathsf{k}_{i,j,0}^{(\mathsf{OLE},1)}, \mathsf{k}_{j,i,1}^{(\mathsf{OLE},1)}, \mathsf{k}_{i,j,0}^{(\mathsf{OLE},2)}, \mathsf{k}_{j,i,1}^{(\mathsf{OLE},2)})_{j \neq i}$ from $\mathcal{F}_{\mathsf{Setup}}$, output pk.

Tuple. Upon receiving input $(\texttt{tuple}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T})$, compute:

4. for $j \in \mathcal{T} \setminus \{i\}$:
   - $(a_i, c_{i,j,0}^{\mathsf{VOLE}}) = \mathsf{PCF}_{\mathsf{VOLE}}.\mathsf{Eval}(0, \mathsf{k}_{i,j,0}^{\mathsf{VOLE}}, \mathsf{ssid})$,
   - $(\mathsf{sk}_i, c_{j,i,1}^{\mathsf{VOLE}}) = \mathsf{PCF}_{\mathsf{VOLE}}.\mathsf{Eval}(1, \mathsf{k}_{j,i,0}^{\mathsf{VOLE}}, \mathsf{ssid})$,
   - $(a_i, c_{i,j,0}^{(\mathsf{OLE},1)}) = \mathsf{PCF}_{\mathsf{OLE}}.\mathsf{Eval}(0, \mathsf{k}_{i,j,0}^{(\mathsf{OLE},1)}, \mathsf{ssid})$,
   - $(s_i, c_{j,i,1}^{(\mathsf{OLE},1)}) = \mathsf{PCF}_{\mathsf{OLE}}.\mathsf{Eval}(1, \mathsf{k}_{j,i,1}^{(\mathsf{OLE},1)}, \mathsf{ssid})$,
   - $(a_i, c_{i,j,0}^{(\mathsf{OLE},2)}) = \mathsf{PCF}_{\mathsf{OLE}}.\mathsf{Eval}(0, \mathsf{k}_{i,j,0}^{(\mathsf{OLE},2)}, \mathsf{ssid})$, and
   - $(e_i, c_{j,i,1}^{(\mathsf{OLE},2)}) = \mathsf{PCF}_{\mathsf{OLE}}.\mathsf{Eval}(1, \mathsf{k}_{j,i,1}^{(\mathsf{OLE},2)}, \mathsf{ssid})$.
5. $\delta_i = a_i(e_i + L_{i,\mathcal{T}}\mathsf{sk}_i) + \sum_{j\in\mathcal{T}\setminus\{i\}} \left(L_{i,\mathcal{T}} c_{j,i,1}^{\mathsf{VOLE}} - L_{j,\mathcal{T}} c_{i,j,0}^{\mathsf{VOLE}} + c_{j,i,1}^{(\mathsf{OLE},2)} - c_{i,j,0}^{(\mathsf{OLE},2)} \right)$
6. $\alpha_i = a_i s_i + \sum_{j\in\mathcal{T}\setminus\{i\}} \left(c_{j,i,1}^{(\mathsf{OLE},1)} - c_{i,j,0}^{(\mathsf{OLE},1)} \right)$

Finally, output $(\mathsf{sid}, \mathsf{ssid}, a_i, e_i, s_i, \delta_i, \alpha_i)$.

Theorem 2. *Let $\mathsf{PCF}_{\mathsf{VOLE}}$ be a srPCF for VOLE correlations and let $\mathsf{PCF}_{\mathsf{OLE}}$ be a srPCF for OLE correlations. Then, protocol π_{Prep} UC-realizes $\mathcal{F}_{\mathsf{Prep}}$ in the $(\mathcal{F}_{\mathsf{KG}}, \mathcal{F}_{\mathsf{Setup}})$-hybrid model in the presence of malicious adversaries controlling up to $t-1$ parties.*

We state our simulator in Appendix F, provide a sketch in Appendix G, and the full indistinguishability proof in Appendix H.

5.4 PCG-based Preprocessing

Instead of using PCFs, we can also use PCGs to instantiate our preprocessing phase. On a high level, our protocol presented in Section 5.3 uses VOLE and OLE PCFs. For VOLE and OLE correlations, PCG constructions were proposed in [BCGI18, BCG$^+$19b, BCG$^+$19a, SGRR19, BCG$^+$20b, YWL$^+$20, CRR21]. It remains to show that these constructions fulfill a notion similar to strong reusability defined in Section 3.

In a practical setting, a PCG-based precomputation requires the parties to perform the PCG expansion directly after the seed generation. Then, the parties store the expanded correlation outputs and use one for each signing request.

6 Evaluation

For the evaluation, we split our protocol into two phases: online and offline. Given a signing request determined by the message and the signer set, the online phase captures the signing request-dependent parts. In contrast, the offline phase covers the signing request-independent preprocessing. This separation does not fully reflect the protocol specification's separation in in Key Generation and Signing. In the protocol specificaiton, servers evaluate the PCFs on the fly as part of the Signing protocol. However, the PCF evaluation is signing request-independent, and hence, can be precomputed in the offline phase. Servers precompute the PCF for upcoming requests and store the results to respond to signing requests even faster. In the case of PCG-based preprocessing, servers evaluate the PCG directly after seed generation. Upon receiving a signing request, the servers aggregate the preprocessed PCF/PCG outputs to a valid signer set-dependent presignature and use this presignature to perform the actual signing. More precisely, servers compute Step 4 of protocol π_{Prep} as part of the offline phase[3], and start the online phase with Step 5 and Step 6. Steps 5 and 6 cannot be executed earlier as they depend on the signer set.

For the online, signing request-dependent phase, we implement the protocol and run benchmarks to test its practicality, reporting both the runtime and the communication complexity. For comparison, we also implement and benchmark the non-threshold BBS+ signing algorithm. We open-source our prototype implementation to foster future research in this area[4].

For the offline, signing request-independent phase, we compute the communication, storage, and computation complexity. Due to the lack of efficient instantiations of PCFs for the required correlations, we focus the evaluation on a PCG-based precomputation.

In the following, we denote the security parameter by λ, the number of servers by n, the security threshold by t, the size of the signed message arrays by k, the number of generated precomputation tuples by N, the order of the elliptic curve's groups $\mathbb{G}_1$ and $\mathbb{G}_2$ by p and assume PCGs based on the Ring LPN problem with

[3] Now, they evaluate this step for all $j \in [n] \setminus \{i\}$ instead of all $j \in \mathcal{T} \setminus \{j\}$.

[4] `https://github.com/AppliedCryptoGroup/NI-Threshold-BBS-Plus-Code`

static leakage and security parameters c and τ, i.e., the $R^c - LPN_{p,\tau}$ assumption[5]. This assumption is common to state-of-the-art PCG instantiations for OLE correlations [BCG+20b].

As [TZ23] published an optimization of the BBS+ signature scheme concurrently to our work, we repeat our evaluation, including implementation and benchmarks, for an optimized version of our protocol and present the results in Appendix J

6.1 Online, Signing Request-Dependent Phase

Our implementation and benchmarks of the online phase are written in Rust and based on the BLS12_381 curve[6]. Our code, including the benchmarks and rudimentary tests, comprises 1.400 lines. We compiled our code using rustc 1.68.2 (9eb3afe9e).

Setup. For our benchmarks, we split the protocol in four phases: *Adapt* (Steps 5 and 6 of protocol π_{Prep}), *Sign* (Step 2 of $\pi_{\mathsf{TBBS+}}$), *Reconstruct* (Step 3a of $\pi_{\mathsf{TBBS+}}$) and *Verify* (Step 3b of $\pi_{\mathsf{TBBS+}}$). Adapt and Sign are executed by the servers. Reconstruct and Verify are executed by the client. Together, these phases cover the whole online signing protocol. The runtime of our protocol is influenced by the security threshold t and the message array size k. We perform benchmarks for $2 \leq t \leq 30$ and $1 \leq k \leq 50$. The influence of the total number of servers n is insignificant to non-existent. Our benchmarks do not account for network latency which heavily depends on the location of clients and servers. Network latency, in our protocol, incurs the same overhead as in the non-threshold setting. It can be incorporated by adding the Round-Trip-Time of messages up to 2kB over the client's (slowest) server connection to the total runtime. As the online phase of our protocol is non-interactive, we benchmark servers and clients individually. More precisely, we execute all benchmarks on a single machine which has a 14-core Intel Xeon Gold 5120 CPU @ 2.20GHz processor and 64GB of RAM. To account for statistical deviations, we repeat each benchmark 100 times and report the average. For comparability, we report the runtime of basic arithmetic operations on our machine in Table 1 in Appendix I.

Results. We report the results of our benchmarks in Figure 4. These results reflect our expectations as outlined in the following. The *Adapt* phase transforming PCF/PCG outputs to signing request-dependent presignatures involves only field operations and is much faster than the other phases for small t. The runtime increase for larger t stems from the number of field operations scaling quadratically with the number of signers. Signers have to compute a LaGrange coefficient for each other signer. The computation of the LaGrange coefficient scales with t as well. The *Sign* phase requires the servers to compute $k+2$ scalar multiplications in $\mathbb{G}_1$, each taking 100 times more time than the slowest field

[5] For 128-bit security, [BCG+20b] reports $(c,t) = (2,76)$, $(c,t) = (4,16)$, and $(c,t) = (8,5)$ for $N = 2^{20}$.

[6] We have used Algorand's pairing-plus library [Alg23] for all curve operations.

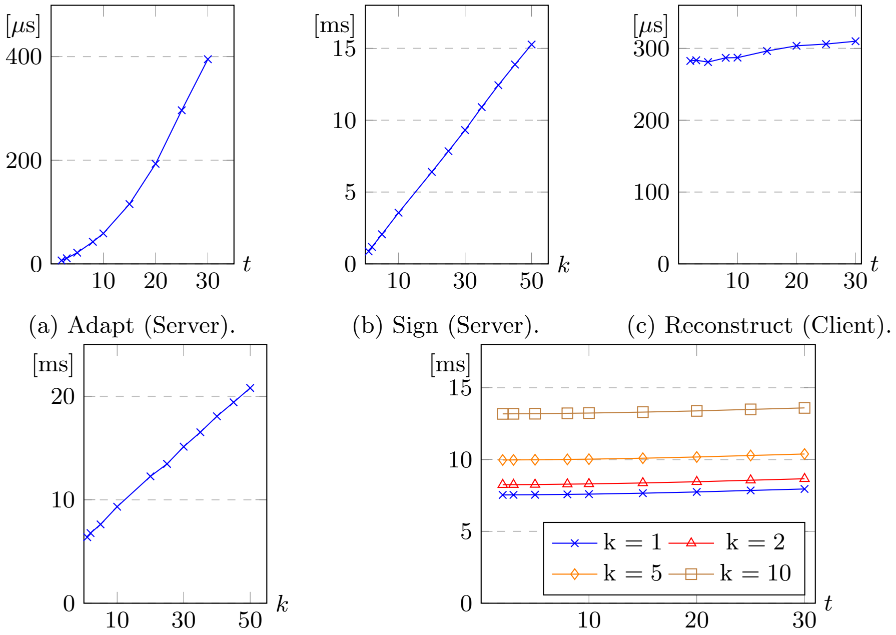
operation (cf. Appendix I). The *Reconstruct* phase involves a single $\mathbb{G}_1$ scalar multiplication, field operations, and $\mathbb{G}_1$ additions, depending on the threshold t. The scalar multiplication, being responsible for more than 90% of the phase's runtime for $t \leq 30$, dominates the cost of this phase. The *Verify* phase requires the client to compute the pairing operation, a single scalar multiplication in $\mathbb{G}_2$, $k+1$ scalar multiplications $\mathbb{G}_1$, and multiple additions in $\mathbb{G}_1$ and $\mathbb{G}_2$. The pairing operation and the scalar multiplication in $\mathbb{G}_2$ are responsible for the constant costs visible in the graph. The scalar multiplications in $\mathbb{G}_1$ cause the linear increase. The influence of $\mathbb{G}_1$ and $\mathbb{G}_2$ additions is insignificant because they take at most 1.4% of scalar multiplication in $\mathbb{G}_1$. The *Total* runtime mainly depends on the size of the signed message array due to the scalar multiplications in the signing and verification step. The number of signers, t, has only a minor influence on the online runtime; increasing the number of signers from 2 to 30 increases the runtime by $1.14\% - 5.52\%$. Following, the online protocol can essentially tolerate any amount of servers as long as the preprocessing, which is expected to scale worse, can be instantiated efficiently for the number of servers and the storage complexity of the generated preprocessing material does not exceed the servers' capacities (cf. Section 6.2).

To measure the *overhead of thresholdization*, we compare the runtime of our online protocol to the runtime of signature creation (and verification) in the non-threshold setting in Figure 5. When considering clients that verify signatures, even if created by a single server, the overhead of our protocol consists only of a single scalar multiplication in $\mathbb{G}_1$ and is essentially free. This observation reflects our protocol pushing all the overhead of the thresholdization, except one signature verification, to the offline phase.

Communication-wise, the client has to send one signing request of size $(k \cdot \lceil \log p \rceil) + (t \cdot \lceil \log n \rceil)$ bits to each of the t selected servers. By deriving the signer set via a random oracle, we can reduce the size of the request to $(k \cdot \lceil \log p \rceil)$. Each of the selected servers has to send a partial signature of size $(3\lceil \log p \rceil + |\mathbb{G}_1|)$. While our protocol requires the partial signatures to be send to all other servers in the signer set due to some subtle details in the UC modeling, we expect it to be sufficient for real-world applications to send the partial signature to the client only. We emphasize that this request-response behavior is the minimum interaction for MPC protocols. As there is no interaction between the servers, this setting is referred to as non-interactive in the literature [CGG$^+$20, ANO$^+$22]. In case of the BLS12_381 curve, $\lceil \log p \rceil$ equals 255 bits whereas $|\mathbb{G}_1|$ equals 768 bits. Parties can also encode $\mathbb{G}_1$ elements with 384 bits by only sending the x-coordinate of the curve point. In this case, it is necessary that the client computes the y-coordinate of the received $\mathbb{G}_1$ elements itself.

6.2 Offline, Signing Request-Independent Phase

For the evaluation of the offline, signing request-independent phase, we focus on a PCG-based preprocessing, analyzing the communication complexity of the distributed PCG seed generation, the storage complexity of the PCG seeds and the generated tuples, and the computation complexity of the seed expansions.



(a) Adapt (Server).

(b) Sign (Server).

(c) Reconstruct (Client).

(d) Verify (Client).

(e) Total.

Fig. 4: The runtime of individual protocol phases (a)-(d) and the total online protocol (e). The *Adapt* phase, describing Steps 5 and 6 of protocol π_{Prep}, and the *Reconstruct* phase, describing Step 3a of $\pi_{\mathsf{TBBS+}}$, depend on security threshold t. The *Sign* phase, describing Step 2 of $\pi_{\mathsf{TBBS+}}$, and the signature verification, describing Step 3b of $\pi_{\mathsf{TBBS+}}$, depend on the message array size k.

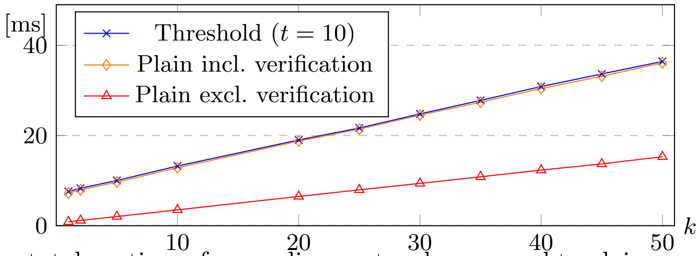


Fig. 5: The total runtime of our online protocol compared to plain, non-threshold signing with and without signature verification in dependence of k. The number of signers t is insignificant (cf. Figure 4e).

Existing fully distributed PCG constructions for OLE-correlations [BCG+20b, ANO+22] do not separate between the PCG seed generation and the PCG evaluation phase. Instead, they merge both phases into one distributed protocol. These distributed protocols make use of secret sharing-based general-purpose MPC protocols optimized for different kinds of operations (binary [NNOB12], field [DPSZ12, DKL+13], or elliptic curve [DKO+20]) as well as a special-purpose protocol for the computation of a two-party distributed point function (DPF) presented in [BCG+20b]. As the PCG-generated preprocessing material utilized in [ANO+22] shows similarities to the material required by our online signing protocol, we derive a distributed PCG protocol for our setting from theirs and analyze the communication complexity accordingly. This yields that the communication complexity of a PCG-based preprocessing instantiating our offline, signing request-independent protocol, is dominated by

$$26(nc\tau)^2 \cdot (\log N + \log p) + 8n(c\tau)^2 \cdot \lambda \cdot \log N.$$

bits of communication per party.

Instead of merging the PCG setup with the PCG evaluation in one setup protocol, it is also possible generate the PCG seeds first, either via a trusted party or another dedicated protocol, and execute the expansion at a later point in time, e.g., when the next batch of presignatures is required. In this scenario, each server stores seeds with a size of at most

$$\begin{aligned}&\log p + 3c\tau \cdot (\lceil \log p \rceil + \lceil \log N \rceil)\\ &+2 \cdot (n-1) \cdot c\tau \cdot (\lceil \log N \rceil \cdot (\lambda + 2) + \lambda + \lceil \log p \rceil)\\ &+4(n-1) \cdot (c\tau)^2 \cdot (\lceil \log 2N \rceil \cdot (\lambda + 2) + \lambda + \lceil \log p \rceil)\end{aligned}$$

bits if the PCGs are instantiated with the constructions of [BCG+20b].

When instantiating the precomputation with PCGs, servers have to evaluate all of the PCGs' outputs at once and keep the resulting precomputation material in storage which occupies

$$\log p \cdot (1 + N \cdot (3 + 6 \cdot (n-1)))$$

bits of storage. In [ANO+22], the authors report $N = 94\,019$ as a reasonable parameter for a PCG-based setup protocol. In [BCG+20b], the authors base their analysis on $N = 2^{20} = 1\,048\,576$. To efficiently apply Fast Fourier Transformation algorithms during the seed expansion, it is necessary to choose N such that it divides $p - 1$. Figure 6 reports the storage complexity depending on the number of servers n for different N that are close or equal to the ones used by prior work and which divide the group order of the BLS12_381 curve used by our implementation. Note that the dependency on the number of servers n stems from the fact that we support any threshold $t \leq n$. In a n-out-of-n settings, servers can execute Steps 5 and 6 of protocol π_{Prep} during the preprocessing, and hence, only store $\log p \cdot (1 + 5N)$ bits of preprocessing material.

The computation cost of the seed expansion is dominated by the ones of the PCGs for OLE correlations. In [BCG+20b], the authors report the computation complexity of expanding a seed of an OLE PCG to involve at most

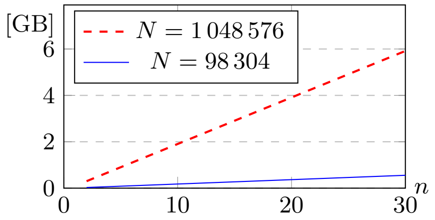


Fig. 6: Storage complexity of the precomputation material required for $N \in \{98\,304, 1\,048\,576\}$ signatures depending on the number of servers n.

$N(ct)^2(4+2\lfloor\log(p/\lambda)\rfloor)$ PRG operations and $O(c^2 N \log N)$ operations in $\mathbb{Z}_p$. In our protocol, each server P_i has to evaluate 4 OLE-generating PCGs for each other server P_j; one for each cross term $(a_i \cdot e_j)$, $(a_j \cdot e_i)$, $(a_i \cdot s_j)$, and $(a_j \cdot s_i)$. It follows that the seed expansion in our protocol is dominated by

$$4 \cdot (n-1) \cdot (4+2\lfloor\log(p/\lambda)\rfloor) \cdot N \cdot (c\tau)^2$$

PRG evaluations and $O(nc^2 N \log N)$ operations in $\mathbb{Z}_p$.

6.3 Comparison to [DKL+23]

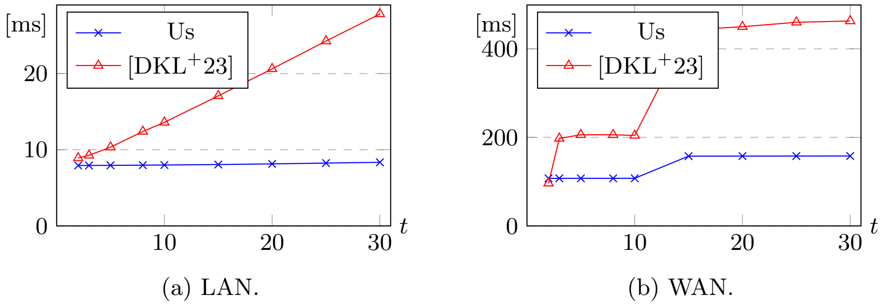


Fig. 7: Runtime of the signing protocol of [DKL+23] compared to the network adjusted runtime of our signing protocol in the LAN and WAN setting.

Concurrently to our work, [DKL+23] presented the first t-out-of-n threshold BBS+ protocol. While we achieve a non-interactive online signing phase at the cost of a computationally intensive offline phase, their protocol incorporates a lightweight setup independent from the number of generated signatures but requires an interactive signing protocol. [DKL+23] provide an experimental evaluation of the interactive signing protocol, which we will compare to our online signing in the following. We thank the authors of [DKL+23] for sharing concrete numbers of their evaluation.

Equivalent to our implementation, their implementation is in Rust and based on the BLS12_381 curve. When comparing the benchmarking machines, $\mathbb{G}_1$ and $\mathbb{G}_2$ scalar multiplications are $20 - 30\%$ faster on our machine, while signature verifications are 20% faster on their machine. Although not explicitly stated, the numbers strongly indicate the choice $k = 1$ in [DKL+23]; the reported runtime of non-threshold BBS+ signing is slightly larger than three $\mathbb{G}_1$ scalar multiplications. Due to the interactivity of their protocol, their benchmarks incorporate network delays for different settings (LAN, WAN). We add network delays to our results to compare our benchmarks to theirs. All machines used in their evaluation are *Google Cloud c2d-standard-4* instances. In the LAN setting, all instances are located at the us-east1-c zone. [DP20] reports a LAN latency of 0.146 ms for this zone. We add a delay of 0.3 ms to our results. In the WAN setting, the first 12 instances in their benchmarks are located in the US, while other machines are in Europe or the US. According to [Kum22], we add 100 ms to our results for $t < 13$ and 150 ms for $t \leq 13$.

In Figure 7, we compare the runtime, including latency, of our online signing protocol to the runtimes reported in [DKL+23] for the LAN and the WAN setting. The graphs show that our protocol outperforms the one of [DKL+23] in both settings for every number of servers. The only exception is the runtime for $t = 2$ in the WAN setting. This exception seems caused by an unusually low connection latency between the first two servers and the client in [DKL+23]. The overhead of [DKL+23] is mainly caused by the two additional rounds of cross-server interaction. This overhead rises with the number of servers as each server has to communicate with each other servers and is especially severe in the WAN setting.

We conclude that, due to the high efficiency and non-interactivity of our online phase, our protocol is more suited for settings where servers have a sufficiently long setup interval and storage capacities to deal with the complexity of the preprocessing phase. On the other hand, the protocol of [DKL+23] is more suited for use cases with more lightweight servers, especially in a LAN environment where the network delay of the additional communication is less significant.

References

AHS20. Jean-Philippe Aumasson, Adrian Hamelink, and Omer Shlomovits. A survey of ECDSA threshold signing. *IACR Cryptol. ePrint Arch.*, 2020.

Alg23. Algorand. BLS12-381 Rust crate. `https://github.com/algorand/pairing-plus`, 04 2023. (Accessed on 04/18/2023).

ANO+22. Damiano Abram, Ariel Nof, Claudio Orlandi, Peter Scholl, and Omer Shlomovits. Low-bandwidth threshold ECDSA via pseudorandom correlation generators. In *IEEE SP*, 2022.

AO00. Masayuki Abe and Tatsuaki Okamoto. Provably secure partially blind signatures. In *CRYPTO*, 2000.

ASM06. Man Ho Au, Willy Susilo, and Yi Mu. Constant-size dynamic k-TAA. In *SCN*, 2006.

BB89. Judit Bar-Ilan and Donald Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In *PODC*, 1989.

BBDE19. Johannes Blömer, Jan Bobolz, Denis Diemert, and Fabian Eidens. Updatable anonymous credentials and applications to incentive systems. In *CCS*, 2019.

BBS04. Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In *CRYPTO*, 2004.

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

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

BCG$^+$20a. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Correlated pseudorandom functions from variable-density LPN. In *FOCS*, 2020.

BCG$^+$20b. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators from ring-lpn. In *CRYPTO*, 2020.

BCG$^+$22. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Nicolas Resch, and Peter Scholl. Correlated pseudorandomness from expand-accumulate codes. In *CRYPTO*, 2022.

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

Bea91. Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, 1991.

BF01. Dan Boneh and Matthew K. Franklin. Identity-based encryption from the weil pairing. In *CRYPTO*, 2001.

BL10. Ernie Brickell and Jiangtao Li. A pairing-based DAA scheme further reducing TPM resources. In *TRUST*, 2010.

BL11. Ernie Brickell and Jiangtao Li. Enhanced privacy ID from bilinear pairing for hardware authentication and attestation. *Int. J. Inf. Priv. Secur. Integr.*, 2011.

BS23. Alexandre Bouez and Kalpana Singh. One round threshold ECDSA without roll call. In *CT-RSA*, 2023.

Cam06. Jan Camenisch. Anonymous credentials: Opportunities and challenges. In *SEC*, 2006.

Can01. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.

CCL$^+$20. Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. Bandwidth-efficient threshold EC-DSA. In *PKC*, 2020.

CDHK15. Jan Camenisch, Maria Dubovitskaya, Kristiyan Haralambiev, and Markulf Kohlweiss. Composable and modular anonymous credentials: Definitions and practical constructions. In *ASIACRYPT*, 2015.

CDL16. Jan Camenisch, Manu Drijvers, and Anja Lehmann. Anonymous attestation using the strong diffie hellman assumption revisited. In *TRUST*, 2016.

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

Cha85. David Chaum. Security without identification: Transaction systems to make big brother obsolete. *Commun. ACM*, 1985.

Che95. Lidong Chen. Access with pseudonyms. In *Cryptography: Policy and Algorithms*, 1995.

Che09. Liqun Chen. A DAA scheme requiring less TPM resources. In *Information Security and Cryptology*, 2009.

CKL^{+}15. Jan Camenisch, Stephan Krenn, Anja Lehmann, Gert Læssøe Mikkelsen, Gregory Neven, and Michael Østergaard Pedersen. Formal treatment of privacy-enhancing credential systems. In *SAC*, 2015.

CL01. Jan Camenisch and Anna Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *EUROCRYPT*, 2001.

CL04. Jan Camenisch and Anna Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In *CRYPTO*, 2004.

CRR21. Geoffroy Couteau, Peter Rindal, and Srinivasan Raghuraman. Silver: Silent VOLE and oblivious transfer from hardness of decoding structured LDPC codes. In *CRYPTO*, 2021.

DILO22. Samuel Dittmer, Yuval Ishai, Steve Lu, and Rafail Ostrovsky. Authenticated garbling from simple correlations. In *CRYPTO*, 2022.

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

DKL^{+}23. Jack Doerner, Yash Kondi, Eysa Lee, abhi shelat, and LakYah Tyner. Threshold bbs+ signatures for distributed anonymous credential issuance. In *IEEE SP*, 2023.

DKLS19. Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. Threshold ECDSA from ECDSA assumptions: The multiparty case. In *SP*, 2019.

DKO^{+}20. Anders Dalskov, Marcel Keller, Claudio Orlandi, Kris Shrishak, and Haya Shulman. Securing dnssec keys via threshold ecdsa from generic mpc, 2020.

DP20. Rick Jones Derek Phanekham. How much is google cloud latency (gcp) between regions? `https://cloud.google.com/blog/products/networking/using-netperf-and-ping-to-measure-network-latency`, June 2020. (Accessed on 05/04/2023).

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

EGM96. Shimon Even, Oded Goldreich, and Silvio Micali. On-line/off-line digital signatures. *J. Cryptol.*, 1996.

FKL18. Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In *CRYPTO*, 2018.

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

GGI19. Rosario Gennaro, Steven Goldfeder, and Bertrand Ithurburn. Fully distributed group signatures, 2019.

GJKR99. Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *EUROCRYPT*, 1999.

GPS06. S. D. Galbraith, K. G. Paterson, and N. P. Smart. Pairings for cryptographers, 2006.

KMOS21. Yashvanth Kondi, Bernardo Magri, Claudio Orlandi, and Omer Shlomovits. Refresh when you wake up: Proactive threshold wallets with offline devices. In *SP*, 2021.

KOR23. Yashvanth Kondi, Claudio Orlandi, and Lawrence Roy. Two-round stateless deterministic two-party schnorr signatures from pseudorandom correlation functions. *IACR Cryptol. ePrint Arch.*, 2023.

Kum22. Chandan Kumar. How much is google cloud latency (gcp) between regions? `https://geekflare.com/google-cloud-latency/`, March 2022. (Accessed on 05/04/2023).

Lin17. Yehuda Lindell. Fast secure two-party ECDSA signing. In *CRYPTO*, 2017.

LKWL23. Tobias Looker, Vasilis Kalos, Andrew Whitehead, and Mike Lodder. The BBS Signature Scheme. Internet-Draft draft-irtf-cfrg-bbs-signatures-02, Internet Engineering Task Force, March 2023. (Work in Progress).

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

LRSW99. Anna Lysyanskaya, Ronald L. Rivest, Amit Sahai, and Stefan Wolf. Pseudonym systems. In *SAC*, 1999.

LS23. Tobias Looker and Orie Steele. Bbs cryptosuite v2023. `https://w3c.github.io/vc-di-bbs/`, May 2023. (Accessed on 05/04/2023).

MAT23. MATTR. mattrglobal/bbs-signatures: An implementation of bbs+ signatures for node and browser environments. `https://github.com/mattrglobal/bbs-signatures`, 04 2023. (Accessed on 04/18/2023).

Mic23. Microsoft. microsoft/bbs-node-reference: Typescript/node reference implementation of bbs signature. `https://github.com/microsoft/bbs-node-reference`, 04 2023. (Accessed on 04/18/2023).

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

OSY21. Claudio Orlandi, Peter Scholl, and Sophia Yakoubov. The rise of paillier: Homomorphic secret sharing and public-key silent OT. In *EUROCRYPT*, 2021.

Ped91. Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*, 1991.

RP22. Alfredo Rial and Ania M. Piotrowska. Security analysis of coconut, an attribute-based credential scheme with threshold issuance. *IACR Cryptol. ePrint Arch.*, 2022.

SA19. Nigel P. Smart and Younes Talibi Alaoui. Distributing any elliptic curve based protocol. In *IMA*, 2019.

SAB$^+$19. Alberto Sonnino, Mustafa Al-Bassam, Shehar Bano, Sarah Meiklejohn, and George Danezis. Coconut: Threshold issuance selective disclosure credentials with applications to distributed ledgers. In *NDSS*, 2019.

SGRR19. Phillipp Schoppmann, Adrià Gascón, Leonie Reichert, and Mariana Raykova. Distributed vector-ole: Improved constructions and implementation. In *CCS*, 2019.

Sha79. Adi Shamir. How to share a secret. *Commun. ACM*, 1979.

Tri23. Trinsic. Credential api - documentation. `https://docs.trinsic.id/reference/services/credential-service/`, 04 2023. (Accessed on 04/18/2023).

TZ23. Stefano Tessaro and Chenzhi Zhu. Revisiting bbs signatures. In *IEEE SP*, 2023.

Wik04. Douglas Wikström. Universally composable DKG with linear number of exponentiations. In *SCN*, 2004.

WMYC23. Harry W. H. Wong, Jack P. K. Ma, Hoover H. F. Yin, and Sherman S. M. Chow. Real threshold ECDSA. In *NDSS*, 2023.

WRK17a. Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *CCS*, 2017.

WRK17b. Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *CCS*, 2017.

YAY19. Zuoxia Yu, Man Ho Au, and Rupeng Yang. Accountable anonymous credentials. In *Advances in Cyber Security: Principles, Techniques, and Applications*. 2019.

YWL$^+$20. Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In *CCS*, 2020.

A Universal Composability Framework ([Can01])

We formally model and prove the security of our protocols in the Universal Composability framework (UC). The framework was introduced by Canetti in 2001 [Can01] to analyze the security of protocols formally. The universal composability property guarantees the security of a protocol holds even under sequential and parallel composition. We give a brief intuition and defer the reader to [Can01] for all details.

Like simulation-based proofs, the framework differentiates between real-world and ideal-world execution. The real-world execution consists of n parties $P_1, \ldots, P_n$ executing protocol π, an adversary $\mathcal{A}$, and an environment $\mathcal{Z}$. All parties are initialized with security parameter λ and a random tape, and $\mathcal{Z}$ runs on some advice string z. In this work, we consider only static corruption, where the adversary corrupts parties at the onset of the execution. After corruption, the adversary may instruct the corrupted parties to deviate arbitrarily from the protocol specification. The environment provides inputs to the parties, instructs them to continue the execution of π, and receives outputs from the parties. Additionally, $\mathcal{Z}$ can interact with the adversary.

The real-world execution finishes when $\mathcal{Z}$ stops activating parties and outputs a decision bit. We denote the output of the real-world execution by $\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(\lambda, z)$.

The ideal-world execution consists of n dummy parties, an ideal functionality $\mathcal{F}$, an ideal adversary $\mathcal{S}$, and an environment $\mathcal{Z}$. The dummy parties forward messages between $\mathcal{Z}$ and $\mathcal{F}$, and $\mathcal{S}$ may corrupt dummy parties and act on their behalf in the following execution. $\mathcal{S}$ can also interact with $\mathcal{F}$ directly according to the specification of $\mathcal{F}$. Additionally, $\mathcal{Z}$ and $\mathcal{S}$ may interact. The goal of $\mathcal{S}$ is to simulate a real-world execution such that the environment cannot tell apart if it is running in the real or ideal world. Therefore, $\mathcal{S}$ is also called the simulator.

Again, the ideal-world execution ends when $\mathcal{Z}$ outputs a decision bit. We denote the output of the ideal-world execution by $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\lambda, z)$.

Intuitively, a protocol is secure in the UC framework if the environment cannot distinguish between real-world and ideal-world execution. Formally, protocol

π UC-realizes $\mathcal{F}$ if for every probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ there exists a PPT simulator $\mathcal{S}$ such that for every PPT environment $\mathcal{Z}$

$$\{\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(1^\lambda, z)\}_{\lambda\in\mathbb{N},z\in\{0,1\}^*} = \{\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(1^\lambda, z)\}_{\lambda\in\mathbb{N},z\in\{0,1\}^*}\,.$$

B PCF Definition of [BCG+20a]

Definition 3 (Pseudorandom correlation function (PCF)). *Let* $(\mathsf{Setup}, \mathcal{Y})$ *be a reverse-sampleable correlation with setup which has output length functions* $\ell_0(\lambda), \ell_1(\lambda)$*, and let* $\lambda \leq n(\lambda) \leq \mathsf{poly}(\lambda)$ *be an input length function. Let* $(\mathsf{PCF.Gen}, \mathsf{PCF.Eval})$ *be a pair of algorithms with the following syntax:*

- $\mathsf{PCF.Gen}(1^\lambda)$ *is a probabilistic polynomial-time algorithm that on input* 1^λ *outputs a pair of keys* (k_0, k_1)*.*
- $\mathsf{PCF.Eval}(\sigma, k_\sigma, x)$ *is a deterministic polynomial-time algorithm that on input* $\sigma \in \{0,1\}$*, key* k_σ *and input value* $x \in \{0,1\}^{n(\lambda)}$ *outputs a value* $c_\sigma \in \{0,1\}^{\ell_\sigma(\lambda)}$*.*

We say $(\mathsf{PCF.Gen}, \mathsf{PCF.Eval})$ *is a (weak)* (N, B, ϵ)*-secure pseudorandom correlation function (PCF) for* $\mathcal{Y}$*, if the following conditions hold:*

- ***Pseudorandom*** $\mathcal{Y}$***-correlated outputs.*** *For every non-uniform adversary* $\mathcal{A}$ *of size* $B(\lambda)$*, it holds*

$$\left|\Pr[\mathsf{Exp}^{\mathsf{pr}}_{\mathcal{A},N,0}(\lambda) = 1] - \Pr[\mathsf{Exp}^{\mathsf{pr}}_{\mathcal{A},N,1}(\lambda) = 1]\right| \leq \epsilon(\lambda)$$

  for all sufficiently large λ*, where* $\mathsf{Exp}^{\mathsf{pr}}_{\mathcal{A},N,b}(\lambda)$ *for* $b \in \{0,1\}$ *is as defined in Figure 8. In particular, the adversary is given access to* $N(\lambda)$ *samples.*
- ***Security.*** *For each* $\sigma \in \{0,1\}$ *and non-uniform adversary* $\mathcal{A}$ *of size* $B(\lambda)$*, it holds*

$$\left|\Pr[\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},N,\sigma,0}(\lambda) = 1] - \Pr[\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},N,\sigma,1}(\lambda) = 1]\right| \leq \epsilon(\lambda)$$

  for all sufficiently large λ*, where* $\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},N,\sigma,b}(\lambda)$ *for* $b \in \{0,1\}$ *is as defined in Figure 9 (again, with* $N(\lambda)$ *samples).*

We say that $(\mathsf{PCF.Gen}, \mathsf{PCF.Eval})$ *is a PCF for* $\mathcal{Y}$ *if it is a* $(p, 1/p, p)$*-secure PCF for* $\mathcal{Y}$ *for every polynomial* p*. If* $B = N$*, we write* (B, ϵ)*-secure PCF for short.*

C Reusable PCF Constructions

This sections presents construction of reusable PCFs for VOLE and OLE correlations as defined in Section 3.2. We first present the reusable PCF for VOLE and then for OLE.

```
$\underline{\mathsf{Exp}^{\mathsf{pr}}_{\mathcal{A},N,0}(\lambda):}$
$\mathsf{mk} \leftarrow \mathsf{Setup}(1^\lambda)$
$\textbf{for } i = 1 \textbf{ to } N(\lambda)$
    $x^{(i)} \xleftarrow{\$} \{0,1\}^{n(\lambda)}$
    $(y_0^{(i)}, y_1^{(i)}) \leftarrow \mathcal{Y}(1^\lambda, \mathsf{mk})$
$b \leftarrow \mathcal{A}(1^\lambda, (x^{(i)}, y_0^{(i)}, y_1^{(i)})_{i \in [N(\lambda)]})$
$\textbf{return } b$
```

```
$\underline{\mathsf{Exp}^{\mathsf{pr}}_{\mathcal{A},N,1}(\lambda):}$
$(k_0, k_1) \leftarrow \mathsf{PCF.Gen}(1^\lambda)$
$\textbf{for } i = 1 \textbf{ to } N(\lambda)$
    $x^{(i)} \xleftarrow{\$} \{0,1\}^{n(\lambda)}$
    $\textbf{for } \sigma \in \{0,1\} : y_\sigma^{(i)} \leftarrow \mathsf{PCF.Eval}(\sigma, k_\sigma, x^{(i)})$
$b \leftarrow \mathcal{A}(1^\lambda, (x^{(i)}, y_0^{(i)}, y_1^{(i)})_{i \in [N(\lambda)]})$
$\textbf{return } b$
```

Fig. 8: Pseudorandom $\mathcal{Y}$-correlated outputs of a PCF.

```
$\underline{\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},N,\sigma,0}(\lambda):}$
$(k_0, k_1) \leftarrow \mathsf{PCF.Gen}(1^\lambda)$
$\textbf{for } i = 1 \textbf{ to } N(\lambda)$
    $x^{(i)} \xleftarrow{\$} \{0,1\}^{n(\lambda)}$
    $y_{1-\sigma}^{(i)} \leftarrow \mathsf{PCF.Eval}(1-\sigma, k_{1-\sigma}, x^{(i)})$
$b \leftarrow \mathcal{A}(1^\lambda, \sigma, k_\sigma, (x^{(i)}, y_{1-\sigma}^{(i)})_{i \in [N(\lambda)]})$
$\textbf{return } b$
```

```
$\underline{\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},N,\sigma,1}(\lambda):}$
$(k_0, k_1) \leftarrow \mathsf{PCF.Gen}(1^\lambda)$
$\mathsf{mk} \xleftarrow{\$} \mathsf{Setup}(1^\lambda)$
$\textbf{for } i = 1 \textbf{ to } N(\lambda)$
    $x^{(i)} \xleftarrow{\$} \{0,1\}^{n(\lambda)}$
    $y_\sigma^{(i)} \leftarrow \mathsf{PCF.Eval}(\sigma, k_\sigma, x^{(i)})$
    $y_{1-\sigma}^{(i)} \leftarrow \mathsf{RSample}(1^\lambda, \mathsf{mk}, \sigma, y_\sigma^{(i)})$
$b \leftarrow \mathcal{A}(1^\lambda, \sigma, k_\sigma, (x^{(i)}, y_{1-\sigma}^{(i)})_{i \in [N(\lambda)]})$
$\textbf{return } b$
```

Fig. 9: Security of a PCF.

The VOLE construction heavily builds on the constructions of [BCG$^+$20a], which provides only weak PCF. However, Boyle et al. presented a generic transformation from weak to strong PCF using a programmable random oracle. This transformation is also straightforwardly applicable to reusable PCFs. Therefore, we state a weak reusable PCF in the following and emphasize that this construction can be extended to a strong reusable PCF in the programmable random oracle model.

The following construction is taken from [BCG$^+$20a, Fig. 22]. It builds on a weak PRF F and a function secret sharing for the multiplication of F with a scalar.

Construction 3: Reusable PCF for $\mathcal{Y}_{\mathsf{VOLE}}$

```
Let $\mathcal{F} = \{F_k : \{0,1\}^n \to R\}_{k \in \{0,1\}^\lambda}$ be a weak PRF and $\mathsf{FFS} = (\mathsf{FFS.Gen}, \mathsf{FFS.Eval})$
an FSS scheme for $\{c \cdot F_k\}_{c \in R, k \in \{0,1\}^\lambda}$ with weak pseudorandom outputs. Let further
$\rho_0 \in \{0,1\}^\lambda, \rho_1 \in R$.
$\mathsf{PCF.Gen_p}(1^\lambda, \rho_0, \rho_1)$:

  1. Set the weak PRF key $k \leftarrow \rho_0$ and $b \leftarrow \rho_1$.
  2. Sample a pair of FSS keys $(K_0^{\mathsf{FFS}}, K_1^{\mathsf{FFS}}) \leftarrow \mathsf{FFS.Gen}(1^\lambda, b \cdot F_k)$.
  3. Output the keys $\mathsf{k}_0 = (K_0^{\mathsf{FFS}}, k)$ and $\mathsf{k}_1 = (K_1^{\mathsf{FFS}}, b)$.

$\mathsf{PCF.Eval}(\sigma, \mathsf{k}_\sigma, x)$: On input a random $x$:

– If $\sigma = 0$:
    1. Let $c_0 = -\mathsf{FFS.Eval}(0, K_0^{\mathsf{FFS}}, x)$.
    2. Let $a = F_k(x)$.
    3. Output $(a, c_0)$.
– If $\sigma = 1$:
    1. Let $c_1 = \mathsf{FFS.Eval}(1, K_1^{\mathsf{FFS}}, x)$.
    2. Output $(b, c_1)$.
```

Theorem 3. *Let $R = R(\lambda)$ be a finite commutative ring. Suppose there exists an FSS scheme for scalar multiples of a family of weak pseudorandom functions $\mathcal{F} = \{F_k : \{0,1\}^n \to R\}_{k \in \{0,1\}^\lambda}$. Then, there is a reusable PCF for the VOLE correlation over R, given by Construction 3.*

Proof. Boyle et al. showed in their proof of [BCG$^+$20a, Theorem 5.3] that Construction 3 satisfies pseudorandom $\mathcal{Y}_{\mathsf{VOLE}}$-correlated outputs and security. Although we slightly adapted our definition to consider reusable inputs, their argument still holds. Further, it is easy to see that programmability holds for functions $f_0(\rho_0, x) = F_{\rho_0}(x)$ and $f_1(\rho_1, x) = \rho_1$. Finally, key indistinguishability follows from the secrecy property of the FSS scheme. The secrecy property states that for every function f of the function family, there exists a simulator $\mathcal{S}(1^\lambda)$ such that the output of $\mathcal{S}$ is indistinguishable from the FSS keys generated correctly using the $\mathsf{FFS.Gen}$-algorithm.

To briefly sketch the proof of key indistinguishability, we define a hybrid experiment, where inside the PCF key generation, we use $\mathcal{S}$ to simulate FSS keys. These simulated FSS keys are used inside the PCF key, which is given to $\mathcal{A}_1$.

We can show via a reduction to the FSS secrecy that the original $\mathsf{Exp}^{\mathsf{key\text{-}ind}}$ game is indistinguishable from the hybrid experiment. For the hybrid experiment, it is easy to see that the adversary can only guess bit b' since the simulated PCF key is independent of $\rho^{(0)}_{1-\sigma}, \rho^{(1)}_{1-\sigma}$ and hence also independent of b. It follows that $\Pr[\mathsf{Exp}^{\mathsf{key\text{-}ind}}_{\mathcal{A},\sigma}(\lambda) = 1] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$.

The construction of the reusable PCF for OLE correlations follows the same blueprint as our PCF construction for VOLE.

The following construction is generically based on a weak PRF and function secret sharing (FSS) for products of two weak PRFs.

Construction 4: Reusable PCF for $\mathcal{Y}_{\mathsf{OLE}}$

Let $\mathcal{F} = \{F_k : \{0,1\}^n \to R\}_{k \in \{0,1\}^\lambda}$ be a weak PRF and $\mathsf{FFS} = (\mathsf{FFS.Gen}, \mathsf{FFS.Eval})$ an FSS scheme for $\{F_{k_0} \cdot F_{k_1}\}_{k_0,k_1 \in \{0,1\}^\lambda}$ with weak pseudorandom outputs. Let further $\rho_0, \rho_1 \in \{0,1\}^\lambda$.
$\mathsf{PCF.Gen_p}(1^\lambda, \rho_0, \rho_1)$:

1. Set the weak PRF keys $k \leftarrow \rho_0$ and $k' \leftarrow \rho_1$.
2. Sample a pair of FSS keys $(K_0^{\mathsf{FFS}}, K_1^{\mathsf{FFS}}) \leftarrow \mathsf{FFS.Gen}(1^\lambda, F_k F_{k'})$.
3. Output the keys $\mathsf{k}_0 = (K_0^{\mathsf{FFS}}, k)$ and $\mathsf{k}_1 = (K_1^{\mathsf{FFS}}, k')$.

$\mathsf{PCF.Eval}(\sigma, \mathsf{k}_\sigma, x)$: On input a random x:

- If $\sigma = 0$:
  1. Let $c_0 = -\mathsf{FFS.Eval}(0, K_0^{\mathsf{FFS}}, x)$.
  2. Let $a = F_k(x)$.
  3. Output (a, c_0).
- If $\sigma = 1$:
  1. Let $c_1 = \mathsf{FFS.Eval}(1, K_1^{\mathsf{FFS}}, x)$.
  2. Let $b = F_{k'}(x)$.
  3. Output (b, c_1).

Theorem 4. *Let $R = R(\lambda)$ be a finite commutative ring. Suppose there exists an FSS scheme for multiplications of two elements of a family of weak pseudorandom functions $\mathcal{F} = \{F_k : \{0,1\}^n \to R\}_{k \in \{0,1\}^\lambda}$. Then, there is a reusable PCF for the OLE correlation over R, given by Construction 4.*

We omit the proof as it follows the same arguments as the proof of Theorem 3.

D Ideal Threshold Signature Functionality

Next, we state our ideal threshold functionality $\mathcal{F}_{\mathsf{tsig}}$, which is a modification of the functionality proposed by Canetti et al. [CGG+20]. We explain our modifications in Section 4.1.

Functionality $\mathcal{F}_{\mathsf{tsig}}$

The functionality is parameterized by a threshold parameter t. We denote a set of t parties by $\mathcal{T}$. For a specific session id sid, the sub-procedures *Signing* and *Verification* can only be executed once a tuple $(\mathsf{sid}, \mathcal{V})$ is recorded.

Key-generation:

1. Upon receiving $(\mathtt{keygen}, \mathsf{sid})$ from some party P_i, interpret $\mathsf{sid} = (\dots, \mathbf{P})$, where $\mathbf{P} = (P_1, \dots, P_n)$.
   - If $P_i \in \mathbf{P}$, send to $\mathcal{S}$ and record $(\mathtt{keygen}, \mathsf{sid}, i)$.
   - Otherwise ignore the message.
2. Once $(\mathtt{keygen}, \mathsf{sid}, i)$ is recorded for all $P_i \in \mathbf{P}$, send $(\mathtt{pubkey}, \mathsf{sid})$ to the adversary $\mathcal{S}$ and do:
   (a) Upon receiving $(\mathtt{pubkey}, \mathsf{sid}, \mathcal{V})$ from $\mathcal{S}$, record $(\mathsf{sid}, \mathcal{V})$.
   (b) Upon receiving $(\mathtt{pubkey}, \mathsf{sid})$ from $P_i \in \mathbf{P}$, output $(\mathtt{pubkey}, \mathsf{sid}, \mathcal{V})$ if it is recorded. Else ignore the message.

Signing:

1. Upon receiving $(\mathtt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m} = (m_1, \dots, m_k))$ with $\mathcal{T} \subseteq \mathbf{P}$, from $P_i \in \mathcal{T}$ and no tuple $(\mathtt{sign}, \mathsf{sid}, \mathsf{ssid}, \cdot, \cdot, i)$ is stored, send to $\mathcal{S}$ and record $(\mathtt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, i)$.
2. Upon receiving $(\mathtt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m} = (m_1, \dots, m_k), i)$ from $\mathcal{S}$, record $(\mathtt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, i)$ if $P_i \in \mathcal{C}$. Else ignore the message.
3. Once $(\mathtt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, i)$ is recorded for all $P_i \in \mathcal{T}$, send $(\mathtt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m})$ to the adversary $\mathcal{S}$.
4. Upon receiving $(\mathtt{sig}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, \sigma, \mathcal{I})$ from $\mathcal{S}$, where $\mathcal{I} \subseteq \mathcal{T} \setminus \mathcal{C}$, do:
   - If there exists a record $(\mathsf{sid}, \mathbf{m}, \sigma, 0)$, output an error.
   - Else, record $(\mathsf{sid}, \mathbf{m}, \sigma, \mathcal{V}(\mathbf{m}, \sigma))$, send $(\mathtt{sig}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, \sigma)$ to all $P_i \in \mathcal{T} \setminus (\mathcal{C} \cup \mathcal{I})$ and send $(\mathtt{sig}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, \mathtt{abort})$ to all $P_i \in \mathcal{T} \cap \mathcal{I}$.

Verification:

Upon receiving $(\mathtt{verify}, \mathsf{sid}, \mathbf{m} = (m_1, \dots, m_k), \sigma, \mathcal{V}')$ from a party Q, send the tuple $(\mathtt{verify}, \mathsf{sid}, \mathbf{m}, \sigma, \mathcal{V}')$ to $\mathcal{S}$ and do:
- If $\mathcal{V}' = \mathcal{V}$ and a tuple $(\mathsf{sid}, \mathbf{m}, \sigma, \beta')$ is recorded, then set $\beta = \beta'$.
- Else, if $\mathcal{V}' = \mathcal{V}$ and less than t parties in $\mathbf{P}$ are corrupted, set $\beta = 0$ and record $(\mathsf{sid}, \mathbf{m}, \sigma, 0)$.
- Else, set $\beta = \mathcal{V}'(\mathbf{m}, \sigma)$.

Output $(\mathtt{verified}, \mathsf{sid}, \mathbf{m}, \sigma, \beta)$ to Q.

E Proof of Theoreom 1

This section presents the proof of our online protocol, i.e., Theorem 1.

Proof. We construct a simulator $\mathcal{S}$ that interacts with the environment and the ideal functionality $\mathcal{F}_{\mathsf{tsig}}$. Since the security statement for UC requires that for every real-world adversary $\mathcal{A}$, there is a simulator $\mathcal{S}$, we allow $\mathcal{S}$ to execute $\mathcal{A}$ internally. In the internal execution of $\mathcal{A}$, $\mathcal{S}$ acts as the environment and the honest parties. In particular, $\mathcal{S}$ forwards all messages between its environment and $\mathcal{A}$. The adversary $\mathcal{A}$ creates messages for the corrupted parties. These messages are sent to $\mathcal{S}$ in the internal execution. Note that this scenario also covers dummy adversaries, which just forward messages received from the environment. An output of $\mathcal{S}$ indistinguishable from the output of $\mathcal{A}$ in the real-world execution is created by simulating a protocol transcript towards $\mathcal{A}$ that is indistinguishable from the real-world execution and outputting whatever $\mathcal{A}$ outputs in the simulated execution. Since the protocol $\pi_{\mathsf{TBBS+}}$ is executed in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model, $\mathcal{S}$ impersonates the hybrid functionality $\mathcal{F}_{\mathsf{Prep}}$ in the internal execution.

We start with presenting our simulator $\mathcal{S}$.

Simulator $\mathcal{S}$

KeyGen.

1. Upon receiving $(\mathtt{init}, \mathsf{sid})$ from corrupted party P_j, send $(\mathtt{keygen}, \mathsf{sid})$ on behalf of P_j to $\mathcal{F}_{\mathsf{tsig}}$.
2. Upon receiving $(\mathtt{pubkey}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{tsig}}$ simulate the initialization phase of $\mathcal{F}_{\mathsf{Prep}}$ to get pk. In particular, sample $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_p$ and send $\mathsf{pk} = g_2^{\mathsf{sk}}$ to $\mathcal{A}$.
3. Upon receiving $(\mathtt{ok}, \mathsf{Tuple}(\cdot,\cdot,\cdot))$ from $\mathcal{A}$, send $(\mathtt{pubkey}, \mathsf{sid}, \mathsf{Verify}_{\mathsf{pk}}(\cdot,\cdot))$ to $\mathcal{F}_{\mathsf{tsig}}$.

Sign.

1. Upon receiving $(\mathtt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m} = \{m_\ell\}_{\ell\in[k]}, i)$ from $\mathcal{F}_{\mathsf{tsig}}$ for honest party P_i, simulate the tuple phase of $\mathcal{F}_{\mathsf{Prep}}$ to get $(a_i, e_i, s_i, \delta_i, \alpha_i)$ for P_i. Then, compute $(A_i := (g_1 \cdot \prod_{\ell\in[k]} h_\ell^{m_\ell})^{a_i} \cdot h_0^{\alpha_i}, \delta_i, e_i, s_i)$ and send it to the corrupted parties in $\mathcal{T}$ in the internal execution.
2. Upon receiving $(\mathtt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m})$ from $\mathcal{Z}$ to corrupted party P_j, send message to P_j in the internal execution an do:
   (a) Upon receiving $(\mathtt{tuple}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T})$ on behalf of $\mathcal{F}_{\mathsf{Prep}}$ from corrupted party P_j with $j \in \mathcal{T}$ return $(a_j, e_j, s_j, \delta_j, \alpha_j) \leftarrow \mathsf{Tuple}(\mathsf{ssid}, \mathcal{T}, j)$ to P_j.
   (b) Forward $(\mathtt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, j)$ to $\mathcal{F}_{\mathsf{tsig}}$ and define an empty set $\widehat{\mathcal{I}}_j = \emptyset$ of honest parties that received signature shares from corrupted party P_j.
   (c) Upon receiving $(\mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, A'_{j,i}, \delta'_{j,i}, e'_{j,i}, s'_{j,i})$ from P_j to honest party P_i in the internal execution, add P_i to $\widehat{\mathcal{I}}_j$.
3. Upon receiving $(\mathtt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m})$ from $\mathcal{F}_{\mathsf{tsig}}$, do:
   - Use tuple $(a_j, e_j, s_j, \delta_j, \alpha_j)$ to compute honestly generated $(A_j, \delta_j, e_j, s_j)$ for $P_j \in \mathcal{T} \cap \mathcal{C}$. Compute honestly generated signature $\sigma = (A, e, s)$ as honest parties do using $(A_\ell, \delta_\ell, e_\ell, s_\ell)$ for $P_\ell \in \mathcal{T}$.

- For each honest party P_i recompute signature σ_i obtained by P_i as honest parties do by using $A'_{j,i}, \delta'_{j,i}, e'_{j,i}, s'_{j,i}$ for $P_j \in \mathcal{T} \cap \mathcal{C}$.
- We define set $\mathcal{I}$ of honest parties that obtained no or an invalid signature. First set, $\mathcal{I} = (\mathcal{T} \setminus \mathcal{C}) \setminus (\bigcap_{j \in \mathcal{T} \cap \mathcal{C}} \widehat{\mathcal{I}}_j)$, i.e., add all honest parties to $\mathcal{I}$ that did not receive signature shares from all corrupted parties in $\mathcal{T}$. Next, compute $\mathcal{I} = \mathcal{I} \cup \{i : \sigma_i \neq \sigma\}$, i.e., add all honest parties that obtained a signature different to the honestly generated signature. If there exists $\sigma_i \neq \sigma$ such that $\mathsf{Verify}_{\mathsf{pk}}(\mathbf{m}, \sigma_i) = 1$ and $(\texttt{sig}, \mathsf{sid}, \mathsf{ssid}, \cdot, \mathbf{m}, \sigma_i, \cdot)$ was not sent to $\mathcal{F}_{\mathsf{tsig}}$ before, output `fail` and stop the execution.
- Finally, send $(\texttt{sig}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, \sigma, \mathcal{I})$ to $\mathcal{F}_{\mathsf{tsig}}$.

Verify. Upon receiving $(\texttt{verify}, \mathsf{sid}, \mathbf{m}, \sigma, \mathsf{Verify}_{\mathsf{pk}'}(\cdot,\cdot))$ from $\mathcal{F}_{\mathsf{tsig}}$ check if

- $\mathsf{Verify}_{\mathsf{pk}'}(\cdot,\cdot) = \mathsf{Verify}_{\mathsf{pk}}(\cdot,\cdot)$,
- $(\texttt{sig}, \mathsf{sid}, \mathsf{ssid}, \cdot, \mathbf{m}, \sigma, \cdot)$ was not sent to $\mathcal{F}_{\mathsf{tsig}}$ before
- $\mathsf{Verify}_{\mathsf{pk}}(\mathbf{m}, \sigma) = 1$.

If the checks hold, output `fail` and stop the execution.

Lemma 1. *If simulator $\mathcal{S}$ does not outputs* `fail`*, protocol $\pi_{\mathsf{TBBS+}}$ UC-realizes $\mathcal{F}_{\mathsf{tsig}}$ in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model in the presence of malicious adversaries controlling up to $t-1$ parties.*

Proof. If the simulator $\mathcal{S}$ does not outputs `fail`, it behaves precisely as the honest parties in real-world execution. Therefore, the simulation is perfect, and no environment can distinguish between the real and ideal worlds.

Lemma 2. *Assuming the strong unforgeability of BBS+, the probability that $\mathcal{S}$ outputs* `fail` *is negligible.*

Proof. We show Lemma 2 via contradiction. Given a real-world adversary $\mathcal{A}$ such that simulator $\mathcal{S}$ outputs `fail` with non-negligible probability, we construct an attacker $\mathcal{B}$ against the strong unforgeability (SUF) of BBS+ with non-negligible success probability. $\mathcal{B}$ simulates the protocol execution towards $\mathcal{A}$ like $\mathcal{S}$ except the following aspects:

1. During the simulation of the initialization phase of $\mathcal{F}_{\mathsf{Prep}}$, instead of sampling $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_p$ and computing $\mathsf{pk} = g_2^{\mathsf{sk}}$, $\mathcal{B}$ returns pk^* obtained from the SUF-challenger. Since the SUF-challenger samples the key exactly as the simulator $\mathcal{S}$, this step of the simulations is indistinguishable towards $\mathcal{A}$.
2. During the *Sign* phase, upon receiving $(\texttt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, i)$ from $\mathcal{F}_{\mathsf{tsig}}$ for honest party P_i, the computation of signature shares of the honest parties is modified as follows:
   - Request the signing oracle of the SUF-game on message $\mathbf{m}$ to obtain signature $\sigma = (A, e, s)$. This signature is forwarded to $\mathcal{F}_{\mathsf{tsig}}$ on receiving $(\texttt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m})$ from $\mathcal{F}_{\mathsf{tsig}}$.

- Compute $(a_j, e_j, s_j, \delta_j, \alpha_j) \leftarrow \mathsf{Tuple}(\mathsf{ssid}, \mathcal{T}, j)$ and (A_j, e_j, s_j) according to the protocol specification for every corrupted party $P_j \in \mathcal{T} \cap \mathcal{C}$.
- Sample random index $k \overset{\$}{\leftarrow} \mathcal{T} \setminus \mathcal{C}$.
- For all honest parties except P_k sample random signature share, i.e., $\forall P_i \in (\mathcal{T} \setminus \mathcal{C}) \setminus \{P_k\} : (A_i, \delta_i, e_i, s_i) \overset{\$}{\leftarrow} (\mathbb{G}_1, \mathbb{Z}_p, \mathbb{Z}_p, \mathbb{Z}_p)$.
- For P_k sample random $\delta_k \overset{\$}{\leftarrow} \mathbb{Z}_p$ and compute $e_k = e - \sum_{\ell \in \mathcal{T} \setminus \{k\}} e_\ell$, $s_k = s - \sum_{\ell \in \mathcal{T} \setminus \{k\}} s_\ell$, and

$$A_k = \frac{A^{\sum_{\ell \in \mathcal{T}} \delta_\ell}}{\prod_{\ell \in \mathcal{T} \setminus \{k\}} A_\ell}.$$

  It is easy to see that e_i and s_i are sampled at random by both, $\mathcal{S}$ and $\mathcal{B}$. Moreover, δ_i is a share of $a(\mathsf{sk} + e)$ in the simulation by $\mathcal{S}$ and since the random value a works as a random mask, it has the same distribution as in the simulation by $\mathcal{B}$. Finally, the A_i values yield a valid signature in $\mathcal{B}$. Therefore, the simulation of the *Sign* phase of $\mathcal{B}$ and $\mathcal{S}$ are indistinguishable to $\mathcal{A}$.

Finally, $\mathcal{B}$ needs to provide a strong forgery to the SUF-challenger. Here, we use the fact that $\mathcal{S}$ outputs `fail` with non-negligible probability either in the *Sign* or the *Verify* phase. As the interaction of $\mathcal{B}$ with $\mathcal{A}$ is indistinguishable, $\mathcal{B}$ outputs `fail` with non-negligible probability as well. Whenever $\mathcal{B}$ outputs `fail`, it forwards the pair $(\mathbf{m}^*, \sigma^*)$ obtained in the *Sign* or *Verify* phase to the SUF-challenger.

It remains to show that $\mathcal{B}$ successfully wins the SUF-game. In order to be a valid forgery, it must hold that (1) $\mathsf{Verify}_{\mathsf{pk}^*}(\mathbf{m}^*, \sigma^*) = 1$ and (2) $(\mathbf{m}^*, \sigma^*)$ was not returned by the signing oracle before. (1) is trivially true, since $\mathcal{B}$ only outputs `fail` if this condition holds. For (2), we note that $\mathcal{A}$ has never seen σ^* as output from $\mathcal{F}_{\mathsf{tsig}}$, since $\mathcal{B}$ checks that $(\mathtt{sig}, \mathsf{sid}, \mathsf{ssid}, \cdot, \mathbf{m}^*, \sigma^*, \cdot)$ was not sent to $\mathcal{F}_{\mathsf{tsig}}$ before. However, it might happen that $\mathcal{B}$ obtained σ^* as response to a signing request for message $\mathbf{m}^*$ without forwarding it the to $\mathcal{F}_{\mathsf{tsig}}$ (this happens if the environment does not instruct all parties in $\mathcal{T}$ to sign). Since the signing oracle samples e and s at random from $\mathbb{Z}_p$, the probability that σ^* was returned by the signing oracle is $\leq \frac{q}{p}$, where q is the number of oracle requests and p is the size of the field. While q is a polynomial, p is exponential in the security parameter. Thus, the probability that σ^* hits an unseen response from the signing oracle is negligible in the security parameter. It follows that $(\mathbf{m}^*, \sigma^*)$ is a valid forgery and $\mathcal{B}$ wins the SUF-game.

Since this contradicts the strong unforgeability of BBS+, it follows that the probability that $\mathcal{S}$ outputs `fail` is negligible.

F Simulator for PCF-based Precprocessing

Here, we state our simulator for proving security of our PCF-based preprocessing. Formally, the security is stated in Theorem 2. We provide a proof sketch of our indistinguishability argument in Appendix G and state the full proof in Appendix H

Simulator for Preprocessing $\mathcal{S}$

Without loss of generality, we assume the adversary corrupts parties $P_1, \dots, P_{t-1}$ and parties $P_t, \dots, P_n$ are honest. $\mathcal{S}$ internally uses adversary $\mathcal{A}$.

Initialization:

1: - Upon receiving $(\texttt{keygen}, \mathsf{sid})$ on behalf of $\mathcal{F}_{\mathsf{KG}}$ from corrupted party P_j, send $(\texttt{init}, \mathsf{sid})$ on behalf of corrupted P_j to $\mathcal{F}_{\mathsf{Prep}}$. Then, wait to receive $(\texttt{corruptedShares}, \mathsf{sid}, \{\mathsf{sk}_j\}_{j \in \mathcal{C}})$ from $\mathcal{A}$.

2: - Upon receiving pk from $\mathcal{F}$, set $\mathsf{pk}_j = h_0^{\mathsf{sk}_j}$ for $j \in \mathcal{C}$ and compute $\mathsf{pk}_i = \left(\mathsf{pk}/(\mathsf{pk}_1^{L_{1,\mathcal{T}}} \cdot \ldots \cdot \mathsf{pk}_{t-1}^{L_{1,\mathcal{T}}})\right)^{1/L_{i,\mathcal{T}}}$, where $\mathcal{T} := \mathcal{C} \cup \{i\}$, for every honest party P_i. Then, send $(\mathsf{sid}, \mathsf{sk}_j, \mathsf{pk}, \{\mathsf{pk}_k\}_{k \in [n]})$ to every corrupted party P_j.
   - Upon receiving $(\texttt{setup}, \mathsf{sid}, \rho_a^{(j)}, \rho_s^{(j)}, \rho_e^{(j)}, \mathsf{sk}'_j, \{\mathsf{pk}_k^{(j)}\}_{k \in [n]})$ on behalf of $\mathcal{F}_{\mathsf{Setup}}$ from every corrupted party P_j, check that $\mathsf{pk}_k^{(j)} = \mathsf{pk}_k$ and $h^{\mathsf{sk}'_j} = \mathsf{pk}_j$ for $j \in \mathcal{C}$ and $k \in [n]$. If any check fails, send $(\texttt{abort}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{Prep}}$.

     Otherwise sample $\rho_a^{(i)}, \rho_s^{(i)}, \rho_e^{(i)}$ and a dummy secret key share $\widehat{\mathsf{sk}}_i$ for every honest party P_i and simulate the computation of $\mathcal{F}_{\mathsf{Setup}}$ (i.e., compute all the PCF keys using the values received from the corrupted parties and the values sampled for the honest parties).

3: - Send keys $(\mathsf{sid}, \mathsf{k}_{j,\ell,0}^{\mathsf{VOLE}}, \mathsf{k}_{\ell,j,1}^{\mathsf{VOLE}}, \mathsf{k}_{j,\ell,0}^{(\mathsf{OLE},1)}, \mathsf{k}_{\ell,j,1}^{(\mathsf{OLE},1)}, \mathsf{k}_{j,\ell,0}^{(\mathsf{OLE},2)}, \mathsf{k}_{\ell,j,1}^{(\mathsf{OLE},2)})_{\ell \neq j}$ to every corrupted party P_j.
   - Send $(\texttt{ok}, \mathsf{Tuple}(\cdot,\cdot,\cdot))$ to $\mathcal{F}$, where $\mathsf{Tuple}(\mathsf{ssid}, \mathcal{T}, j)$ computes $(a_j, e_j, s_j, \delta_j, \alpha_j)$ as follows:

     First sample for every $\ell \in \mathcal{T} \setminus \{j\}$

$$((a_j, c_{j,\ell,0}^{\mathsf{VOLE}}), \cdot) \xleftarrow{\$} \mathcal{Y}_{\mathsf{VOLE}}(1^\lambda, (\rho_a^{(j)}, \mathsf{sk}_\ell), \mathsf{ssid}),$$
$$(\cdot, (\mathsf{sk}_j, c_{\ell,j,1}^{\mathsf{VOLE}})) \xleftarrow{\$} \mathcal{Y}_{\mathsf{VOLE}}(1^\lambda, (\rho_a^{(\ell)}, \mathsf{sk}_j), \mathsf{ssid}),$$
$$((a_j, c_{j,\ell,0}^{(\mathsf{OLE},1)}), \cdot) \xleftarrow{\$} \mathcal{Y}_{\mathsf{OLE}}(1^\lambda, (\rho_a^{(j)}, \rho_s^{(\ell)}), \mathsf{ssid}),$$
$$(\cdot, (s_j, c_{\ell,j,1}^{(\mathsf{OLE},1)})) \xleftarrow{\$} \mathcal{Y}_{\mathsf{OLE}}(1^\lambda, (\rho_a^{(\ell)}, \rho_s^{(j)}), \mathsf{ssid}),$$
$$((a_j, c_{j,\ell,0}^{(\mathsf{OLE},2)}), \cdot) \xleftarrow{\$} \mathcal{Y}_{\mathsf{OLE}}(1^\lambda, (\rho_a^{(j)}, \rho_e^{(\ell)}), \mathsf{ssid}),$$
$$(\cdot, (e_j, c_{\ell,j,1}^{(\mathsf{OLE},2)})) \xleftarrow{\$} \mathcal{Y}_{\mathsf{OLE}}(1^\lambda, (\rho_a^{(\ell)}, \rho_e^{(j)}), \mathsf{ssid}).$$

Take a_j, e_j, s_j from the samples and compute

$$\alpha_j = a_j s_j + \sum_{\ell \in \mathcal{T} \setminus \{j\}} c_{\ell,j,1}^{(\mathsf{OLE},1)} - c_{j,\ell,0}^{(\mathsf{OLE},1)} ,$$

$$\delta_j = a_j(L_{j,\mathcal{T}} \mathsf{sk}_j + e_j) + \sum_{\ell \in \mathcal{T} \setminus \{j\}} \left(L_{j,\mathcal{T}} c_{\ell,j,1}^{\mathsf{VOLE}} - L_{\ell,\mathcal{T}} c_{j,\ell,0}^{\mathsf{VOLE}} + c_{\ell,j,1}^{(\mathsf{OLE},2)} - c_{j,\ell,0}^{(\mathsf{OLE},2)} \right) .$$

Tuple:
Upon receiving $(\mathtt{tuple}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T})$ from $\mathcal{Z}$ on behalf of corrupted party P_j, forward message $(\mathtt{tuple}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T})$ to $\mathcal{A}$ and output whatever $\mathcal{A}$ outputs.

G Indistinguishability Proof Sketch of Theorem 2

We prove indistinguishability between the ideal-world execution and the real-world execution via a sequence of hybrid experiments. We start with Hybrid_0 which is the ideal-world execution and end up in Hybrid_7 being identical to the real-world execution. By showing indistinguishability between each subsequent pair of hybrids, it follows that the ideal and real-world execution are indistinguishable. In particular, we show indistinguishability between the joint distribution of the adversary's view and the outputs of the honest parties in Hybrid_i and Hybrid_{i+1} for $i = 0, \ldots, 6$. In the following we sketch the proof outline and defer the full proof to Appendix H.

$\underline{\mathsf{Hybrid}_1}$: In this hybrid experiment, we inline the description of the simulator $\mathcal{S}$, the ideal functionality $\mathcal{F}_{\mathsf{Prep}}$ and the outputs of the honest parties. Since this is only a syntactical change, the distribution is identical to the one of Hybrid_0.

$\underline{\mathsf{Hybrid}_2}$: In the second experiment, we modify the computation inside the tuple function Tuple. Instead of using outputs of the $\mathcal{Y}\mathsf{VOLE}$ and $\mathcal{Y}_{\mathsf{OLE}}$ correlations, we run the $\mathsf{PCF}_{\mathsf{VOLE}}$ and $\mathsf{PCF}_{\mathsf{OLE}}$ evaluations. For running the PCF evaluations, we use the keys sent to the corrupted parties in step 3.

This change aligns the output of the Tuple function with the tuple values of corrupted parties if they follow the protocol specification. Note that although the PCF keys are generated using dummy key shares for the honest parties, the final tuple values of honest parties are reverse sampled to match the tuple correlation using the correct secret key.

Indistinguishability between Hybrid_1 and Hybrid_2 can be shown via reductions to the strong pseudorandom $\mathcal{Y}_{\mathsf{VOLE}}$-correlated output property of the $\mathsf{PCF}_{\mathsf{VOLE}}$ primitive and to the strong pseudorandom $\mathcal{Y}_{\mathsf{OLE}}$-correlated output property of the $\mathsf{PCF}_{\mathsf{OLE}}$ primitive, respectively. More precisely, a series of intermediate hybrids can be introduce, where in each hop only a single correlation output is replaced by the output of PCF evaluations.

Hybrid_3: Instead of sampling the secret key sk at random from $\mathbb{Z}_p$, we sample a random polynomial $F(x) \in \mathbb{Z}_p[X]$ of degree $t-1$ such that $F(j) = \mathsf{sk}_j$ for every $j \in \mathcal{C}$. The secret key is then defined as $\mathsf{sk} = F(0)$.

Note that the adversary knows only $t-1$ shares of the polynomial which give no information about sk. This is due to the information-theoretically secrecy of Shamir's secret sharing. It follows that Hybrid_2 and Hybrid_3 are indistinguishable.

Hybrid_4: In this hybrid, we change the way honest parties' secret key shares are defined. Instead of sampling random dummy key shares, we derive the key shares from the polynomial introduced in the last hybrid. In more detail, the key share of honest party P_i is computed as $\mathsf{sk}_i = F(i)$. This change effects the PCF key generation as the dummy key share is replaced by a sk_i.

To show indistinguishability between Hybrid_3 and Hybrid_4, we reduce to the key indistinguishability property of the $\mathsf{PCF}_{\mathsf{VOLE}}$ primitive. More specifically, we again introduce a sequence of intermediate hybrids where we only change the secret key of a single honest party.

Hybrid_5: In this hybrid, we change the computation of the honest party P_i's public key share pk_i. Instead of interpolating pk_i it is defined as $\mathsf{pk}_i = h_0^{\mathsf{sk}_i}$. As both ways are equivalent, Hybrid_5 is indistinguishable from Hybrid_4.

Hybrid_6: Next, we get rid of the reverse-sampling of the honest parties tuple values. Instead, we compute these values using outputs of the $\mathcal{Y}_{\mathsf{VOLE}}$ and $\mathcal{Y}_{\mathsf{OLE}}$ correlations. For instance, for computing α_i for an honest party P_i, we sample

$$((a_i, c_{i,\ell,0}^{(\mathsf{OLE},1)}), \cdot) \in \mathcal{Y}_{\mathsf{OLE}}(1^\lambda, (\rho_a^{(i)}, \rho_s^{(\ell)}), x)\,, \tag{5}$$

$$(\cdot, (s_i, c_{\ell,i,1}^{(\mathsf{OLE},1)})) \in \mathcal{Y}_{\mathsf{OLE}}(1^\lambda, (\rho_a^{(\ell)}, \rho_s^{(i)}), x)\,, \tag{6}$$

for every $\ell \in \mathcal{T}$ and compute

$$\alpha_i = a_i s_i + \sum_{\ell \in \mathcal{T} \setminus \{i\}} c_{\ell,i,1}^{(\mathsf{OLE},1)} - c_{i,\ell,0}^{(\mathsf{OLE},1)}\,. \tag{7}$$

Similar process is done for the computation of δ_i and e_i. A straightforward calculation shows that resulting tuple values satisfy correlation (4). Thus, the view of the environment is indistinguishable in Hybrid_5 and Hybrid_6.

Hybrid_7: Now, we replace the sampling of correlation outputs for calculating honest parties' tuples with the evaluations of PCFs. This change is the same as applied in Hybrid_2 but now for the calculation of the honest parties' tuples.

Indistinguishability follows the same argument as sketched in Hybrid_2.

Hybrid_7 is the real-world execution, which concludes the proof.

H Full Indistinguishability Proof of Theorem 2

In this section, we provide the full indistinguishability proof of Theorem 2. The simulator is given in Appendix F.

Hybrid_0: The initial experiment Hybrid_0 denotes the ideal-world execution where simulator $\mathcal{S}$ is interacting with the corrupted parties, ideal functionality $\mathcal{F}_{\mathsf{Prep}}$ and internally runs real-world adversary $\mathcal{A}$.

$\underline{\mathsf{Hybrid}_1}$: In this hybrid, we inline the description of the simulator $\mathcal{S}$, the ideal functionality $\mathcal{F}_{\mathsf{Prep}}$ and the outputs of the honest parties. Since this is only a syntactical change, the joint distribution of the adversary's view and the output of the honest parties is identical to the one of Hybrid_0. We state Hybrid_1 as the starting point, and emphasize only on the changes in the following hybrids.

$\underline{\mathsf{Hybrid}_1}$

Without loss of generality, we assume the adversary corrupts parties $P_1, \ldots, P_{t-1}$ and parties $P_t, \ldots, P_n$ are honest. $\mathcal{S}$ internally uses adversary $\mathcal{A}$.

Initialization:

1:
- Upon receiving $(\texttt{keygen}, \mathsf{sid})$ on behalf of $\mathcal{F}_{\mathsf{KG}}$ from corrupted party P_j, store $(\texttt{init}, \mathsf{sid}, P_j)$. Then, wait to receive $(\texttt{corruptedShares}, \mathsf{sid}, \{\mathsf{sk}_j\}_{j \in \mathcal{C}})$ from $\mathcal{A}$.
- Upon receiving $(\texttt{init}, \mathsf{sid})$ from every honest party, sample the secret key $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_p$ and set $\mathsf{pk} = h_0^{\mathsf{sk}}$. Further, set $\mathsf{pk}_j = h_0^{\mathsf{sk}_j}$ for $j \in \mathcal{C}$ and compute $\mathsf{pk}_i = \left(\mathsf{pk}/(\mathsf{pk}_1^{L_{1,\mathcal{T}}} \cdot \ldots \cdot \mathsf{pk}_{t-1}^{L_{1,\mathcal{T}}})\right)^{1/L_{i,\mathcal{T}}}$, where $\mathcal{T} := \mathcal{C} \cup \{i\}$, for every honest party P_i.

2:
- Send $(\mathsf{sid}, \mathsf{sk}_j, \mathsf{pk}, \{\mathsf{pk}_k\}_{k \in [n]})$ to every corrupted party P_j.
- Upon receiving $(\texttt{setup}, \mathsf{sid}, \rho_a^{(j)}, \rho_s^{(j)}, \rho_e^{(j)}, \mathsf{sk}'_j, \{\mathsf{pk}_k^{(j)}\}_{k \in [n]})$ on behalf of $\mathcal{F}_{\mathsf{Setup}}$ from every corrupted party P_j, check that $\mathsf{pk}_k^{(j)} = \mathsf{pk}_k$ and $h^{\mathsf{sk}'_j} = \mathsf{pk}_j$ for $j \in \mathcal{C}$ and $k \in [n]$. If any check fails, honest parties output $\texttt{abort}$.
  Otherwise sample $\rho_a^{(i)}, \rho_s^{(i)}, \rho_e^{(i)}$ and a dummy secret key share $\widehat{\mathsf{sk}}_i$ for every honest party P_i and simulate the computation of $\mathcal{F}_{\mathsf{Setup}}$ (i.e., compute all the PCF keys using the values received from the corrupted parties and the values sampled for the honest parties).

3:
- Send keys $(\mathsf{sid}, \mathsf{k}_{j,\ell,0}^{\mathsf{VOLE}}, \mathsf{k}_{\ell,j,1}^{\mathsf{VOLE}}, \mathsf{k}_{j,\ell,0}^{(\mathsf{OLE},1)}, \mathsf{k}_{\ell,j,1}^{(\mathsf{OLE},1)}, \mathsf{k}_{j,\ell,0}^{(\mathsf{OLE},2)}, \mathsf{k}_{\ell,j,1}^{(\mathsf{OLE},2)})_{\ell \neq j}$ to every corrupted party P_j.
- Store $(\texttt{ok}, \mathsf{Tuple}(\cdot,\cdot,\cdot))$, where $\mathsf{Tuple}(\mathsf{ssid}, \mathcal{T}, j)$ computes $(a_j, e_j, s_j, \delta_j, \alpha_j)$ as follows:
  First sample for every $\ell \in \mathcal{T} \setminus \{j\}$

$$((a_j, c_{j,\ell,0}^{\mathsf{VOLE}}), \cdot) \xleftarrow{\$} \mathcal{Y}_{\mathsf{VOLE}}(1^\lambda, (\rho_a^{(j)}, \mathsf{sk}_\ell), \mathsf{ssid})\,, \tag{8}$$

$$(\cdot, (\mathsf{sk}_j, c_{\ell,j,1}^{\mathsf{VOLE}})) \xleftarrow{\$} \mathcal{Y}_{\mathsf{VOLE}}(1^\lambda, (\rho_a^{(\ell)}, \mathsf{sk}_j), \mathsf{ssid})\,, \tag{9}$$

$$((a_j, c_{j,\ell,0}^{(\mathsf{OLE},1)}), \cdot) \xleftarrow{\$} \mathcal{Y}_{\mathsf{OLE}}(1^\lambda, (\rho_a^{(j)}, \rho_s^{(\ell)}), \mathsf{ssid})\,, \tag{10}$$

$$(\cdot, (s_j, c_{\ell,j,1}^{(\mathsf{OLE},1)})) \xleftarrow{\$} \mathcal{Y}_{\mathsf{OLE}}(1^\lambda, (\rho_a^{(\ell)}, \rho_s^{(j)}), \mathsf{ssid})\,, \tag{11}$$

$$((a_j, c_{j,\ell,0}^{(\mathsf{OLE},2)}), \cdot) \xleftarrow{\$} \mathcal{Y}_{\mathsf{OLE}}(1^\lambda, (\rho_a^{(j)}, \rho_e^{(\ell)}), \mathsf{ssid})\,, \tag{12}$$

$$(\cdot, (e_j, c_{\ell,j,1}^{(\mathsf{OLE},2)})) \xleftarrow{\$} \mathcal{Y}_{\mathsf{OLE}}(1^\lambda, (\rho_a^{(\ell)}, \rho_e^{(j)}), \mathsf{ssid})\,. \tag{13}$$

Then, take a_j, e_j, s_j from the samples and compute

$$\alpha_j = a_j s_j + \sum_{\ell \in \mathcal{T}\setminus\{j\}} c^{(\mathsf{OLE},1)}_{\ell,j,1} - c^{(\mathsf{OLE},1)}_{j,\ell,0}, \tag{14}$$

$$\begin{aligned}\delta_j &= a_j(L_{j,\mathcal{T}}\mathsf{sk}_j + e_j) \\ &\quad + \sum_{\ell \in \mathcal{T}\setminus\{j\}} \left(L_{j,\mathcal{T}} c^{\mathsf{VOLE}}_{\ell,j,1} - L_{\ell,\mathcal{T}} c^{\mathsf{VOLE}}_{j,\ell,0} + c^{(\mathsf{OLE},2)}_{\ell,j,1} - c^{(\mathsf{OLE},2)}_{j,\ell,0}\right).\end{aligned} \tag{15}$$

- The honest parties $P_t, \ldots, P_n$ output pk.

Tuple:

- Upon receiving $(\texttt{tuple}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T})$ from $\mathcal{Z}$ on behalf of corrupted party P_j, forward message $(\texttt{tuple}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T})$ to $\mathcal{A}$ and output whatever $\mathcal{A}$ outputs.
- Upon receiving $(\texttt{tuple}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T})$ from $\mathcal{Z}$ on behalf of honest party P_i, if $(\mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \{(a_\ell, e_\ell, s_\ell, \delta_\ell, \alpha_\ell)\}_{\ell \in \mathcal{T}})$ is stored, output $(\mathsf{sid}, \mathsf{ssid}, a_i, e_i, s_i, \delta_i, \alpha_i)$. Otherwise, compute $(a_j, e_j, s_j, \delta_j, \alpha_j) \leftarrow \mathsf{Tuple}(\mathsf{ssid}, \mathcal{T}, j)$ for every corrupted party P_j where $j \in \mathcal{C} \cap \mathcal{T}$ and sample $a, e, s \overset{\$}{\leftarrow} \mathbb{Z}_p$ and tuples $(a_i, e_i, s_i, \delta_i, \alpha_i)$ over $\mathbb{Z}_p$ for $i \in \mathcal{H} \cap \mathcal{T}$ such that

$$\sum_{\ell \in \mathcal{T}} a_\ell = a \qquad \sum_{\ell \in \mathcal{T}} e_\ell = e \qquad \sum_{\ell \in \mathcal{T}} s_\ell = s$$
$$\sum_{\ell \in \mathcal{T}} \delta_\ell = a(\mathsf{sk} + e) \qquad \sum_{\ell \in \mathcal{T}} \alpha_\ell = as$$

Store $(\mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \{(a_\ell, e_\ell, s_\ell, \delta_\ell, \alpha_\ell)\}_{\ell \in \mathcal{T}})$ and honest party P_i outputs $(\mathsf{sid}, \mathsf{ssid}, a_i, e_i, s_i, \delta_i, \alpha_i)$.

$\underline{\mathsf{Hybrid}_2}$: As a next step, we align the computation of Tuple to the behavior of corrupted parties that behave honestly in the real-world execution. More precisely, we replace the sampling of correlation tuples from $\mathcal{Y}_{\mathsf{VOLE}}$ and $\mathcal{Y}_{\mathsf{OLE}}$ with the evaluation of the $\mathsf{PCF}_{\mathsf{VOLE}}$ and $\mathsf{PCF}_{\mathsf{OLE}}$ primitives. The strong pseudorandom $\mathcal{Y}_{\mathsf{VOLE}}$-correlated respectively $\mathcal{Y}_{\mathsf{OLE}}$-correlated outputs property of the srPCF primitives yield indistinguishability between Hybrid_1 and Hybrid_2.

To formally show this, we introduce a sequence of intermediate hybrids $\mathsf{Hybrid}_{1,a}$ to $\mathsf{Hybrid}_{1,f}$. In $\mathsf{Hybrid}_{1,a}$, we change Hybrid_1 only in the generation of the first VOLE correlation outputs, i.e., the tuple $(a_j, c^{\mathsf{VOLE}}_{j,\ell,0})$ in Equation (8) is computed as $\mathsf{PCF}_{\mathsf{VOLE}}.\mathsf{Eval}(0, \mathsf{k}^{\mathsf{VOLE}}_{j,\ell,0}, \mathsf{ssid})$. Next, in $\mathsf{Hybrid}_{1,b}$ we build on $\mathsf{Hybrid}_{1,a}$ and replace the computation of the second VOLE correlation output, i.e., Equation (9). We continue this procedure until all outputs of Equations (8) - (13) are computed using PCF evaluations in $\mathsf{Hybrid}_{1,f}$ which is equal to Hybrid_2.

Since every equation from (8)-(13) is computed for ever $\ell \in \mathcal{T} \setminus \{j\}$, we introduce additional intermediate hybrids denoted by additional subscript $k \in \{0, \ldots, t-1\}$. $\mathsf{Hybrid}_{1,a,k}$ means that correlation sampling is replaced by PCF evaluations for the first k parties in $\mathcal{T} \setminus \{j\}$. Note that $\mathsf{Hybrid}_{1,a,0} = \mathsf{Hybrid}_1$ and $\mathsf{Hybrid}_{1,f,t-1} = \mathsf{Hybrid}_2$.

For the same of presentation, we show that for every $k \in \{0, \ldots, t-2\}$, indistinguishability between $\mathsf{Hybrid}_{1,a,k}$ and $\mathsf{Hybrid}_{1,a,k+1}$ can be derived from strong pseudorandom $\mathcal{Y}_{\mathsf{VOLE}}$-correlated outputs property of $\mathsf{PCF}_{\mathsf{VOLE}}$. The argumentation for $\mathsf{Hybrid}_{1,b}$ to $\mathsf{Hybrid}_{1,f}$ is analogously.

We construct an adversary $\mathcal{A}^{\mathsf{s-pr}}$ against the strong pseudorandom $\mathcal{Y}_{\mathsf{VOLE}}$-correlated outputs property from a distinguisher $\mathcal{D}$ between $\mathsf{Hybrid}_{1,a,k}$ and $\mathsf{Hybrid}_{1,a,k+1}$. First, note that the only difference between these hybrids is the computation of $(a_j, c^{\mathsf{VOLE}}_{j,\ell,0})$. While the tuple is sampled from $\mathcal{Y}_{\mathsf{VOLE}}(1^\lambda, (\rho_a^{(j)}, \mathsf{sk}_{k+1}), \mathsf{ssid})$ in $\mathsf{Hybrid}_{1,a,k}$, it is computed from $\mathsf{PCF}_{\mathsf{VOLE}}.\mathsf{Eval}(0, \mathsf{k}^{\mathsf{VOLE}}_{j,k+1,0}, \mathsf{ssid})$ in $\mathsf{Hybrid}_{1,a,k+1}$.

$\mathcal{A}^{\mathsf{s-pr}}$ simulates the hybrid experiment and sends $(\rho_a^{(j)}, \mathsf{sk}_{k+1})$ to the security game. Then, whenever a tuple $(a_j, c^{\mathsf{VOLE}}_{j,\ell,0})$ is required, $\mathcal{A}^{\mathsf{s-pr}}$ asks its oracle $\mathcal{O}_b(\mathsf{ssid})$. Note that if $b = 0$, then the oracle samples the tuple from the correlation and if $b = 1$, then the PCF is evaluation. Thus, if $b = 0$, the simulated hybrid is identical to $\mathsf{Hybrid}_{1,a,k}$ and otherwise it is $\mathsf{Hybrid}_{1,a,k+1}$. It is easy to see that $\mathcal{A}^{\mathsf{s-pr}}$ has the same advantage in winning the security game as $\mathcal{D}$ in distingushing between $\mathsf{Hybrid}_{1,a,k}$ and $\mathsf{Hybrid}_{1,a,k+1}$. Given that $\mathsf{PCF}_{\mathsf{VOLE}}$ is a srPCF, the two hybrids are indistinguishable.

$\underline{\mathsf{Hybrid}_3}$: In this hybrid, we change the sampling of the secret key sk. Instead of sampling sk in step 1 from $\mathbb{Z}_p$, we sample a random polynomial $F \in \mathbb{Z}_p[X]$ of degree $t-1$ such that $F(j) = \mathsf{sk}_j$ for every $j \in \mathcal{C}$. Further, we define $\mathsf{sk} = F(0)$. Since the polynomial is of degree $t-1$, t evaluation points are required to fully determine $F(x)$. As the adversary knows only $t-1$ shares, it cannot learn anything about sk. In detail, for every $\mathsf{sk}' \in \mathbb{Z}_p$ there exists a t-th share that defined the polynomial $F(x)$ such that $F(x) = \mathsf{sk}'$. It follows that the views of the adversary are distributed identically and hence Hybrid_2 and Hybrid_3 are indistinguishable.

$\underline{\mathsf{Hybrid}_4}$: Next, we use the polynomial $F(x)$ sampled in step 1 to determine the honest parties' secret key shares. In particular, for every honest party P_i the experiment samples $\mathsf{sk}_i = F(i)$. The secret key shares $\{\mathsf{sk}_i\}_{i \in \mathcal{H}}$ are then used for the simulation of $\mathcal{F}_{\mathsf{Setup}}$ instead of the dummy key shares. In particular, the correctly sampled key shares of the honest parties are used as input to $\mathsf{PCF}_{\mathsf{VOLE}}.\mathsf{Gen}$ whenever a secret key share of the honest party is used. Since the experiment does not use the dummy key shares at all after these changes, we remove them completely. Note that the sampling of the honest parties' key shares and the generation of the PCF keys are exactly as in the real-world execution.

Indistinguishability between Hybrid_3 and Hybrid_4 can be shown via a series of reductions to the key indistinguishability of the reusability property of the VOLE PCF. We briefly sketch the proof outline in the following. We define intermediate hybrids $\mathsf{Hybrid}_{3,\ell,k}$ for $\ell \in \{0, \ldots, n-(t-1)\}$ and $k \in [n]$, which only differ in the honest parties' key shares that are used in the generation of the VOLE PCF keys. Recall that for every party P_ℓ we generate a VOLE PCF for every other party P_k, where P_ℓ uses its secret key shares as input. We define $\mathsf{Hybrid}_{3,\ell,k}$ such that the key shares derived from polynomial $F(x)$ are used for the first ℓ honest parties in all VOLE PCF instances and for the $(\ell+1)$-th honest party in the VOLE PCF instances with the first k other parties. For all other VOLE PCF instances, the dummy key shares are used for the honest parties' key shares.

Note that $\mathsf{Hybrid}_{3,0,0} = \mathsf{Hybrid}_3$ and $\mathsf{Hybrid}_{3,n-(t-1),n} = \mathsf{Hybrid}_4$. To show indistinguishability between $\mathsf{Hybrid}_{3,\ell,k}$ and $\mathsf{Hybrid}_{3,\ell,k+1}$ for every $\ell \in \{0, \ldots, n-(t-1)\}$, we make a reduction to the key indistinguishability of the reusability property of the VOLE PCF. In particular, we construct an adversary $\mathcal{A}^{\mathsf{key-ind}}$ from a distinguisher $\mathcal{D}_\ell$ which distinguishes between $\mathsf{Hybrid}_{2,\ell,k}$ and $\mathsf{Hybrid}_{2,\ell,k+1}$. Upon receiving the shares of the corrupted parties in the hybrid execution, $\mathcal{A}^{\mathsf{key-ind}}$ forwards the key share of the $k+1$-th corrupted party to the security game. Then, the security game samples two possible key shares for the ℓ-th honest party $\rho_1^{(0)}, \rho_1^{(1)}$, uses one of them in the VOLE PCF key generation and sends the key k_1 for the corrupted party and the two possible key shares back to $\mathcal{A}^{\mathsf{key-ind}}$. Next, $\mathcal{A}^{\mathsf{key-ind}}$ continues the simulation of hybrid $\mathsf{Hybrid}_{3,\ell,k}$ or $\mathsf{Hybrid}_{3,\ell,k+1}$ by sampling the polynomial $F(x)$ using the corrupted key shares and $\rho_1^{(0)}$. Since $\rho_1^{(0)}$ is a random value in $\mathbb{Z}_p$, $F(x)$ is also a random polynomial. Finally, $\mathcal{A}^{\mathsf{key-ind}}$ uses k_1 as the output of the simulation of $\mathcal{F}_{\mathsf{Setup}}$.

If k_1 was sampled using $\rho_1^{(0)}$, then the simulated experiment is identical to $\mathsf{Hybrid}_{3,\ell,k+1}$ and otherwise it is identical to $\mathsf{Hybrid}_{3,\ell,k}$. It is easy to see that a successful distinguisher between these two hybrids allows to easily win the key indistinguishability game. Since we assume the VOLE PCF to support reusability, this leads to a contradiction. Thus, the two hybrids are indistinguishable.

$\underline{\mathsf{Hybrid}_5}$: In this hybrid, we derive the honest parties public key shares pk_i from the secret key shares sk_i instead of interpolating them from pk and the corrupted shares. More precisely, in Hybrid_4 the public key share of honest party P_i was computed as

$$\mathsf{pk}_i = \Big(\mathsf{pk}/(\mathsf{pk}_1^{L_{1,\mathcal{T}}} \cdot \ldots \cdot \mathsf{pk}_{t-1}^{L_{1,\mathcal{T}}})\Big)^{1/L_{i,\mathcal{T}}} ,$$

where $\mathcal{T} := \mathcal{C} \cup \{i\}$. In Hybrid_5 the public key share is instead computed as $\mathsf{pk}_i = h_0^{\mathsf{sk}_i}$. We show that both definitions are equivalent.

To this end, note that $\mathsf{sk} = \sum_{\ell \in \mathcal{T}} L_{\ell,\mathcal{T}} \mathsf{sk}_\ell$ for every set $\mathcal{T}$ of size t, $\mathsf{pk} = h_0^{\mathsf{sk}}$ and $\mathsf{pk}_j = h_0^{\mathsf{sk}_j}$ for $j \in \mathcal{C}$. Using this equation we get for $\mathcal{T} = \mathcal{C} \cup \{i\}$

$$
\begin{aligned}
\mathsf{pk}_i &= \left(\frac{\mathsf{pk}}{\mathsf{pk}_1^{L_{1,\mathcal{T}}} \cdot \ldots \cdot \mathsf{pk}_{t-1}^{L_{1,\mathcal{T}}}} \right)^{1/L_{i,\mathcal{T}}} \\
\Leftrightarrow \mathsf{pk}_i &= \left(\frac{h_0^{\mathsf{sk}}}{h_0^{L_{1,\mathcal{T}} \mathsf{sk}_1} \cdot \ldots \cdot h_0^{L_{1,\mathcal{T}} \mathsf{sk}_{t-1}}} \right)^{1/L_{i,\mathcal{T}}} \\
\Leftrightarrow \mathsf{pk}_i &= \left(\frac{h_0^{\sum_{\ell \in \mathcal{T}} L_{\ell,\mathcal{T}} \mathsf{sk}_\ell}}{h_0^{L_{1,\mathcal{T}} \mathsf{sk}_1} \cdot \ldots \cdot h_0^{L_{1,\mathcal{T}} \mathsf{sk}_{t-1}}} \right)^{1/L_{i,\mathcal{T}}} \\
\Leftrightarrow \mathsf{pk}_i &= \left(h_0^{L_{i,\mathcal{T}} \mathsf{sk}_i} \right)^{1/L_{i,\mathcal{T}}} \\
\Leftrightarrow \mathsf{pk}_i &= h_0^{\mathsf{sk}_i}
\end{aligned}
$$

As public key shares are equivalent in both hybrids, the view of the adversary is identical distributed. Hence, Hybrid_4 and Hybrid_5 are indistinguishable.

$\underline{\mathsf{Hybrid}_6}$: In this hybrid, instead of reverse-sampling the tuple values of the honest parties, we compute them in the same way using Equations (8)-(15).

We show that the resulting tuple outputs satisfy the same correlation as before. In particular, we show $\sum_{\ell \in \mathcal{T}} \alpha_\ell = as$ and $\sum_{\ell \in \mathcal{T}} \delta_\ell = a(\mathsf{sk}+e)$, where $a = \sum_{\ell \in \mathcal{T}} a_\ell = \sum_{\ell \in \mathcal{T}} F_{\rho_a^{(\ell)}}(x)$, $e = \sum_{\ell \in \mathcal{T}} e_\ell = \sum_{\ell \in \mathcal{T}} F_{\rho_e^{(\ell)}}(x)$ and $s = \sum_{\ell \in \mathcal{T}} s_\ell = \sum_{\ell \in \mathcal{T}} F_{\rho_s^{(\ell)}}(x)$. First, we show $\sum_{\ell \in \mathcal{T}} \alpha_\ell = as$:

$$
\begin{aligned}
\sum_{\ell \in \mathcal{T}} \alpha_\ell &= \sum_{\ell \in \mathcal{T}} \left(a_\ell s_\ell + \sum_{k \in \mathcal{T} \setminus \{\ell\}} (c_{k,\ell,1}^{(\mathsf{OLE},1)} - c_{\ell,k,0}^{(\mathsf{OLE},1)}) \right) \\
&= \sum_{\ell \in \mathcal{T}} a_\ell s_\ell + \sum_{\ell \in \mathcal{T}} \sum_{k \in \mathcal{T} \setminus \{\ell\}} \left(c_{k,\ell,1}^{(\mathsf{OLE},1)} - c_{k,\ell,0}^{(\mathsf{OLE},1)} \right) \\
&= \sum_{\ell \in \mathcal{T}} a_\ell s_\ell + \sum_{\ell \in \mathcal{T}} \sum_{k \in \mathcal{T} \setminus \{\ell\}} \left(F_{\rho_a^{(k)}}(x) \cdot F_{\rho_s^{(\ell)}}(x) \right) \\
&= \sum_{\ell \in \mathcal{T}} a_\ell s_\ell + \sum_{\ell \in \mathcal{T}} \sum_{k \in \mathcal{T} \setminus \{\ell\}} a_k s_\ell \\
&= \sum_{\ell \in \mathcal{T}} \sum_{k \in \mathcal{T}} a_k s_\ell \\
&= \sum_{\ell \in \mathcal{T}} a_k \sum_{k \in \mathcal{T}} s_\ell \\
&= as
\end{aligned}
$$

Next, we show $\sum_{\ell\in\mathcal{T}} \delta_\ell = a(\mathsf{sk}+e)$:

$$\begin{aligned}
\sum_{\ell\in\mathcal{T}} \delta_\ell &= \sum_{\ell\in\mathcal{T}} \Bigg(a_\ell(L_{\ell,\mathcal{T}}\mathsf{sk}_\ell + e_\ell) + \sum_{k\in\mathcal{T}\setminus\{\ell\}} L_{\ell,\mathcal{T}} c^{\mathsf{VOLE}}_{k,\ell,1} - L_{k,\mathcal{T}} c^{\mathsf{VOLE}}_{\ell,k,0} \\
&\qquad\qquad\qquad\qquad + c^{(\mathsf{OLE},2)}_{k,\ell,1} - c^{(\mathsf{OLE},2)}_{\ell,k,0} \Bigg) \\
&= \sum_{\ell\in\mathcal{T}} a_\ell(L_{\ell,\mathcal{T}}\mathsf{sk}_\ell + e_\ell) + \sum_{\ell\in\mathcal{T}} \sum_{k\in\mathcal{T}\setminus\{\ell\}} L_{\ell,\mathcal{T}} c^{\mathsf{VOLE}}_{k,\ell,1} - L_{\ell,\mathcal{T}} c^{\mathsf{VOLE}}_{k,\ell,0} \\
&\qquad\qquad\qquad\qquad + c^{(\mathsf{OLE},2)}_{k,\ell,1} - c^{(\mathsf{OLE},2)}_{k,\ell,0} \\
&= \sum_{\ell\in\mathcal{T}} a_\ell(L_{\ell,\mathcal{T}}\mathsf{sk}_\ell + e_\ell) + \sum_{\ell\in\mathcal{T}} \sum_{k\in\mathcal{T}\setminus\{\ell\}} L_{\ell,\mathcal{T}} a_k \mathsf{sk}_\ell + a_k e_\ell \\
&= \sum_{\ell\in\mathcal{T}} a_\ell(L_{\ell,\mathcal{T}}\mathsf{sk}_\ell + e_\ell) + \sum_{\ell\in\mathcal{T}} \sum_{k\in\mathcal{T}\setminus\{\ell\}} a_k(L_{\ell,\mathcal{T}}\mathsf{sk}_\ell + e_\ell) \\
&= \sum_{\ell\in\mathcal{T}} \sum_{k\in\mathcal{T}} a_k(L_{\ell,\mathcal{T}}\mathsf{sk}_\ell + e_\ell) \\
&= \sum_{k\in\mathcal{T}} \sum_{\ell\in\mathcal{T}} a_k(L_{\ell,\mathcal{T}}\mathsf{sk}_\ell + e_\ell) \\
&= \sum_{k\in\mathcal{T}} a_k \sum_{\ell\in\mathcal{T}} (L_{\ell,\mathcal{T}}\mathsf{sk}_\ell + e_\ell) \\
&= \sum_{k\in\mathcal{T}} a_k \left(\sum_{\ell\in\mathcal{T}} L_{\ell,\mathcal{T}}\mathsf{sk}_\ell + \sum_{\ell\in\mathcal{T}} e_\ell \right) \\
&= a(\mathsf{sk}+e)
\end{aligned}$$

As the tuple values of the honest parties still satisfy the same correlation as in Hybrid_5, Hybrid_5 and Hybrid_6 are indistinguishable.

$\underline{\mathsf{Hybrid}_7}$: In this hybrid, instead of sampling values from the VOLE and OLE correlations for computing the parties' tuple values we compute them using the PCF instances. For instance, instead of sampling $((a_j, c^{\mathsf{VOLE}}_{j,\ell,0}), (\mathsf{sk}_\ell, c^{\mathsf{VOLE}}_{j,\ell,1})) \in \mathcal{Y}_{\mathsf{VOLE}}(1^\lambda, (\rho_a^{(j)}, \mathsf{sk}_\ell), x)$, we compute $(a_j, c^{\mathsf{VOLE}}_{j,\ell,0}) \leftarrow \mathsf{PCF}_{\mathsf{VOLE}}.\mathsf{Eval}(0, \mathsf{k}^{\mathsf{VOLE}}_{j,\ell,0}, x)$ and $(\mathsf{sk}_\ell, c^{\mathsf{VOLE}}_{j,\ell,1}) \leftarrow \mathsf{PCF}_{\mathsf{VOLE}}.\mathsf{Eval}(1, \mathsf{k}^{\mathsf{VOLE}}_{j,\ell,1}, x)$. The same modification is applied for both OLE correlations.

Indistinguishability between Hybrid_6 and Hybrid_7 can be shown via a series of reductions to the strong pseudorandom $\mathcal{Y}$-correlated outputs property of the VOLE and OLE PCF instances. The proof is analogous to the indistinguishability proof between Hybrid_1 and Hybrid_2. Therefore, we omit the details here.

We end up in Hybrid_7 where all correlation outputs are replaced by PCF evaluations. This holds for the calculation of honest parties outputs as well as for the computation inside Tuple. As this hybrid does not use any reverse-sampling anymore, we get rid of the tuple function Tuple.

Hybrid_7 is identical to the real-world execution which concludes the proof.

I Benchmarks of Basic Arithmetic Performance

We report the runtime of basic arithmetic operations in Table 1. The presented numbers might help the reader to assess the performance of system used for benchmarking and provides details for comparisons.

Table 1: Runtime of basic arithmetic operations in the BLS12_381 curve on our evaluation machine. The bit-size of the curve's group order p is 255. The error terms report standard deviation.

Operation	Time
$\mathbb{Z}_p$ addition	5.092 ns ±1.049 ns
$\mathbb{Z}_p$ multiplication	32.045 ns ±1.556 ns
$\mathbb{Z}_p$ inverse	2.713 μs ±101.973 ns
$\mathbb{G}_1$ addition	1.102 μs ±48.571 ns
$\mathbb{G}_2$ addition	3.668 μs ±96.867 ns
$\mathbb{G}_1$ scalar multiplication	279.146 μs ±14.763 μs
$\mathbb{G}_2$ scalar multiplication	0.952 ms ±0.04 μs
Pairing	2.403 ms ±56.976 μs

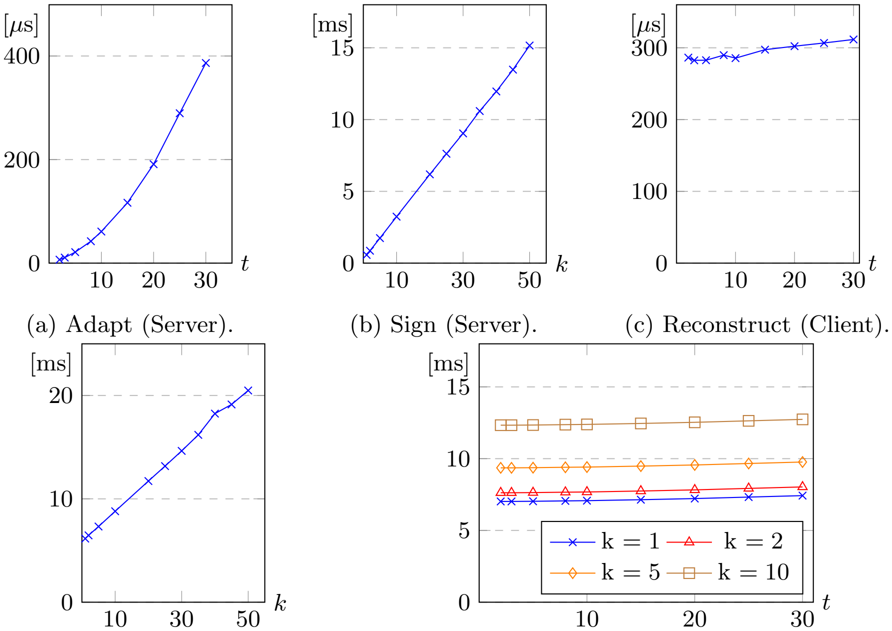
J Evaluation Considering [TZ23]

Concurrently to our work, Tessaro and Zhu [TZ23] proposed and proved security of a more compact BBS+ signature scheme removing the nonce s, and hence, reducing the signature size by one element in $\mathbb{Z}_p$. The proposed extension translates to our protocol in a straight-forward way as follows. We do no longer need public parameter h_0. The preprocessing protocol does not generate the shares s_i or α_i. When answering a signing request, the servers compute A_i differently, i.e., $A_i := (g_1 \cdot \prod_{\ell \in [k]} h_\ell^{m_\ell})^{a_i}$, and do not send s_i. The reconstruction of a signature ignores s and outputs the tuple (A, e). When verifying a signature, parties now check if $\mathsf{e}(A, y \cdot g_2^e) = \mathsf{e}(g_1 \cdot \prod_{\ell \in [k]} h_\ell^{m_\ell}, g_2)$. In the following we call the described protocol as the *lean version of our protocol.*

For us, their optimization has the advantage of removing the necessity of the α values computed during the preprocessing and the computation of the g^{s_i} and g^s term in the signing and verification process. In order to quantify the benefits of this optimization, we have repeated the evaluation presented in Section 6, including implementation and benchmarks, for the lean version of our protocol and report the changes here. The scope of the implementation and the setup of our benchmarks remains unchanged.

Online, Signing Request-Dependent Phase. The results of our benchmarks of the lean version of our protocol are reported in Figure 10. The comparison to the non-threshold protocol, also optimized according to [TZ23] is displayed in Figure 11.

The size of signing requests does not change in the lean version of our protocol. The size of partial signatures sent by the servers reduces to $(2\lceil \log p \rceil + |\mathbb{G}_1|)$.



(a) Adapt (Server).

(b) Sign (Server).

(c) Reconstruct (Client).

(d) Verify (Client).

(e) Total.

Fig. 10: The runtime of individual phases (a)-(d) and the total online protocol (e) in the protocol version optimized according to [TZ23]. The *Adapt* phase, describing Steps 5 and 6 of protocol π_{Prep}, and the *Reconstruct* phase, describing Step 3a of $\pi_{\mathsf{TBBS+}}$, depend on security threshold t. The *Sign* phase, describing Step 2 of $\pi_{\mathsf{TBBS+}}$, and the signature verification, describing Step 3b depend on the message array size k.

Offline, Signing Request-Independent Phase. The communication complexity of a distributed PCG-based preprocessing protocol instantiating the offline, signing request-independent phase of the lean version of our protocol is dominated by a factor of

$$13(nc\tau)^2 \cdot (\log N + \log p) + 4n(c\tau)^2 \cdot \lambda \cdot \log N.$$

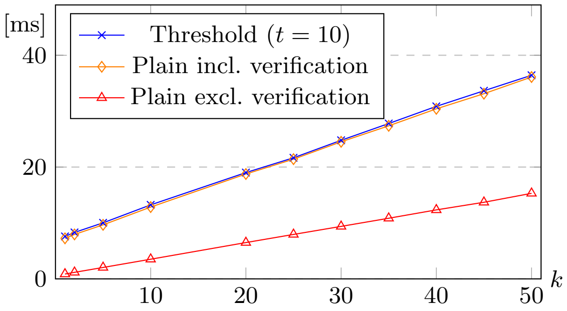


Fig. 11: The total runtime of the lean version of our online protocol in comparison to plain, non-threshold signing (also optimied according to [TZ23]) with and without signature verification in dependence of the size of the message array k. As depicted in Figure 10e, the influence of the number of signers t is insignificant. We choose $t = 10$.

In case, the preprocessing decouples seed generation from seed evaluation, servers have to store seeds with a size of at most

$$\begin{aligned} &\log p + 2c\tau \cdot (\lceil \log p \rceil + \lceil \log N \rceil) \\ &+2 \cdot (n-1) \cdot c\tau \cdot (\lceil \log N \rceil \cdot (\lambda + 2) + \lambda + \lceil \log p \rceil) \\ &+2(n-1) \cdot (c\tau)^2 \cdot (\lceil \log 2N \rceil \cdot (\lambda + 2) + \lambda + \lceil \log p \rceil) \end{aligned}$$

bits. The expanded precomputation material occupies

$$\log p \cdot (1 + N \cdot (2 + 4 \cdot (n-1)))$$

bits of storage. In Figure 12, we report the concrete storage complexity of the preprocessing material of the lean version of our protocol when instantiating the with $N \in \{98\,304, 1\,048\,576\}$ and $p = 255$ according to the BLS12_381 curve used by our implementation.

The computation cost of the seed expansion is still dominated by the ones of the PCGs for OLE correlations. However, we do no longer need the OLE-generating PCGs for the cross terms $a_i \cdot s_j$, and $a_j \cdot s_i$. It follows that the computation complexity of the seed expansion in the lean version of our protocol is dominated by

$$2 \cdot (n-1) \cdot (4 + 2\lfloor \log(p/\lambda) \rfloor) \cdot N \cdot (ct)^2$$

PRG evaluations and $O(nc^2 N \log N)$ operations in $\mathbb{Z}_p$.

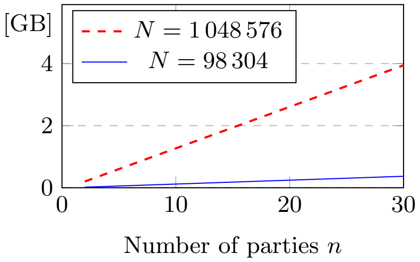


Fig. 12: Storage complexity of the preprocessing material in the lean version of our protocol required for $N \in \{98\,304, 1\,048\,576\}$ signatures depending on the number of servers n.