TECHNISCHE UNIVERSITÄT DARMSTADT

Technische Universität Darmstadt
Department of Computer Science
Chair of Applied Cryptography

Master Thesis

Implementation and Evaluation of Pseudorandom Correlation Generators in the Context of Threshold BBS+

Leandro Rometsch

April 08, 2024

Supervisors: Prof. Sebastian Faust, Ph.D.
Benjamin Schlosser

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit erkläre ich, Leandro Rometsch, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB TU Darmstadt selbstständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

English translation for information purposes only:

Thesis Statement pursuant to § 22 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Leandro Rometsch, have written the submitted thesis independently pursuant to § 22 paragraph 7 of APB TU Darmstadt without any outside support and using only the quoted literature and other sources. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I have clearly marked and separately listed in the text the literature used literally or in terms of content and all other sources I used for the preparation of this academic work. This also applies to sources or aids from the Internet.

This thesis has not been handed in or published before in the same or similar form.

I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once.

For a thesis of the Department of Architecture, the submitted electronic version corresponds to the presented model and the submitted architectural plans.

Datum / Date: 08.04.2024 Unterschrift/Signature:

Abstract

Secure multi-party computation (MPC) protocols often use correlated randomness to improve efficiency, allowing the scheme to be split into a computationally intensive offline preprocessing phase followed by a highly efficient input-dependent online phase. This approach is appealing because parties can now use previously unused idle time for preprocessing to speed up the time-critical part of their computation. Some schemes go further by realizing a non-interactive online phase, making them very attractive for high-latency environments, such as when participants are geographically dispersed. *Pseudorandom correlation generators* (PCGs), as initiated by Boyle et al. (Crypto 2019, Crypto 2020), facilitate the preprocessing approach by allowing the generation of short correlated seeds that expand into long instances of a target correlation. PCGs offer sublinear communication complexity for seed generation and local expansion, reducing the communication required in the preprocessing phase. Despite the theoretical value of the primitive and its use in various MPC protocols in recent years, implementations of PCGs have been lacking.

This work addresses this gap by recalling the PCG construction proposed by Boyle et al., analyzing its building blocks, deriving practical considerations, and finally presenting an efficient implementation. Existing MPC protocols that use the PCG primitive for OLE and Vector-OLE correlations can use our implementation as a blueprint to efficiently implement their offline phase. We demonstrate the value of our practical considerations by implementing a PCG that realizes the offline phase of a non-interactive threshold BBS+ signature scheme by Faust et al. The PCG is the first for a threshold signature scheme to support the t-out-of-n setting. It provides a quasilinear runtime for the number of presignatures generated, with a linear increase in runtime as participants are added. We provide benchmarks for up to 10 parties and find that in the 2-out-of-2 setting, our PCG generates 2^{17} presignatures in 100ms per presignature while adding about 80ms per additional party. This significantly outperforms (6x to 10x) the only other available PCG implementation for threshold ECDSA by Abram et al. (SP 2022) while incorporating an additional correlation.

Contents

1 Introduction **1**
- 1.1 Our Contribution . . . 2
- 1.2 Related Work . . . 2

2 Preliminaries **4**
- 2.1 LPN-based Cryptography . . . 4
  - 2.1.1 Primal- and Dual-LPN . . . 4
  - 2.1.2 Ring-LPN . . . 5
- 2.2 Threshold Signatures . . . 5
- 2.3 Pseudorandom Correlation Generators . . . 6
- 2.4 Building Blocks . . . 8
  - 2.4.1 Function Secret Sharing . . . 8
  - 2.4.2 OLE and Vector-OLE Correlations . . . 9
  - 2.4.3 Cyclotomic Polynomials . . . 9
  - 2.4.4 BBS+ Signature Scheme . . . 9

3 PCG for (Vector-)OLE **11**
- 3.1 PCG based on Dual-LPN . . . 11
- 3.2 PCG based on Ring-LPN . . . 12
  - 3.2.1 Compressing the Noise . . . 14
  - 3.2.2 Splitting the Ring through $F(X)$ . . . 14
  - 3.2.3 Complexity Improvements . . . 15
- 3.3 Construction . . . 15
  - 3.3.1 Adaption to VOLE . . . 18

4 Implementation **20**
- 4.1 Distributed Sum of Point Functions . . . 20
  - 4.1.1 Tree-Based Distributed Point Function . . . 20
  - 4.1.2 Constructing DSPFs . . . 21
- 4.2 Polynomial Operations . . . 22
  - 4.2.1 Multiplication via FFT . . . 22
  - 4.2.2 Evaluation via Horners Method . . . 23
  - 4.2.3 Exploiting Sparsity . . . 23
- 4.3 Considerations for F(X) in BLS12-381 . . . 24

5 Evaluation **25**
- 5.1 Building Blocks . . . 25
  - 5.1.1 DSPF Full Domain Evaluation . . . 25
  - 5.1.2 Operations on Polynomials of High Degree . . . 26
  - 5.1.3 Computing Roots of Unity . . . 28
- 5.2 PCG Construction . . . 28
  - 5.2.1 Generation . . . 29
  - 5.2.2 Expansion . . . 29

6 PCG for Threshold BBS+ **32**
- 6.1 Thresholdization . . . 32

6.2 Offline Preprocessing Phase . . . 33
6.2.1 PCG Construction for BBS+ . . . 34
6.2.2 Threshold Setting . . . 36
6.3 Evaluation . . . 36
6.3.1 n-out-of-n . . . 37
6.3.2 t-out-of-n . . . 37
6.3.3 Identifying Bottlenecks . . . 39
6.3.4 Implications for Non-Interactive Threshold BBS+ . . . 40

7 Conclusion **41**
7.1 Future Work . . . 41

1 Introduction

Correlated secret randomness is a valuable resource for enhancing the performance and security of secure multi-party computation (MPC) protocols [Ish+13]. The type of correlation determines the kind of MPC circuit supported; for example, oblivious transfer (OT) correlations support boolean circuits [GMW19], while oblivious linear-function evaluation (OLE) is used for arithmetic circuits [IPS09]. In order to generate correlated secret randomness, there is usually a distribution phase. In this phase, each party receives a sample from a joint random distribution. Although the samples are random, they adhere to the specified correlation. MPC protocols utilizing correlated secret randomness are therefore split up into an input-independent *offline* and input-dependent *online phase*. This is commonly referred to as *preprocessing* since the offline phase generates many instances of the required correlation, which are then consumed by the online phase. Preprocessing makes the online phase very efficient by shifting communication and computationally demanding tasks to the offline phase. In some instances, schemes manage to move all communication to the preprocessing phase [Abr+22; Fau+23], making the online phase non-interactive. Preprocessing is, therefore, very interesting for many real-world applications since it enables participants to utilize their idle times to speed up the critical (on-demand) part of their MPC protocol. Additionally, when the online phase is non-interactive, these schemes become especially appealing for use in high-latency environments. Further, the flexibility of a preprocessing phase provides significant cost savings when using elastic cloud computing environments [Cou+15], where providers offer substantial discounts to users willing to use spare computing capacity during off-peak periods[1].

Pseudorandom Correlation Generators. In recent years, Boyle et al. introduced the novel primitive of a *pseudorandom correlation generator* (PCG) [Boy+19; Boy+20], which provides a promising approach for the realization of a preprocessing phase. A PCG can be thought of as a form of (distributed) pseudorandom generator in which participants generate seeds and then locally expand them for pseudorandom bit streams. Crucially, the PCG expansion ensures that the generated bit streams are correlated across participants. Such a primitive provides several advantages to the preprocessing model, as it reduces offline communication through local seed expansion, while also reducing storage costs, as the correlations are compressed in the PCG seed and the parties can decide to expand them only when needed. Furthermore, only the seed generation must be protected against malicious parties since malicious seed expansion does not affect honest parties. The most practical PCG construction proposed by Boyle et al. [Boy+20] is based on the Learning Parity with Noise (LPN) assumption [Pie12]. The construction is of high practical relevance as it avoids the quadratic complexity of previous approaches. The PCG realizes OLE and Vector-OLE correlations, but the general approach can be modified to produce multiplication and authentication triples.

Application for Threshold Signature Schemes. Threshold signature schemes [Des87; Des92] distribute the ability to generate a digital signature among multiple participants (signers). In practice, any subset of t signers can collaborate to produce a signature, while signing is infeasible for any subset of less than t parties. This eliminates single points of failure and increases security. The rise of blockchain technology has led to increasing interest in threshold signature schemes for use cases such as securing digital wallets [GGN16]. In addition, threshold signature schemes are essential building blocks for distributed anonymous credentialing systems [GGM13]. However, many threshold signature schemes require interaction between signers during the signing process. This interaction results in high

[1] AWS Spot Instances: `https://aws.amazon.com/aws-cost-management/aws-cost-optimization/spot-instances/`
Azure Spot Instances: `https://azure.microsoft.com/en-us/products/virtual-machines/spot/`

latency, especially when the parties are geographically dispersed. For example, consider two parties located in Western Europe and the West Coast of the United States; Azure network latency alone can reach 147ms [2]. Geographic distance, therefore, creates a significant performance bottleneck for interactive threshold signature schemes, which worsens with each additional round of communication required. To mitigate this shortcoming through preprocessing, PCGs have recently received attention for use in threshold signature schemes [Abr+22; Fau+23]. These schemes use the PCG primitive to generate message-independent presignatures in the offline phase so that the actual signing can be facilitated in the online phase without interaction between the signing parties. The threshold BBS+ signature scheme proposed by Faust et al. [Fau+23] is particularly interesting because it supports non-interactive signing in the online phase while keeping the communication complexity in the offline phase sublinear. Other schemes, such as threshold ECDSA [Abr+22] or multi-party Schnorr signatures [KOR23], have not achieved both attributes simultaneously.

1.1 Our Contribution

This thesis focuses on the efficient implementation and evaluation of the LPN-based PCG proposed by Boyle et al. [Boy+20]. We present a theoretical and practical analysis of their PCG construction and evaluate its practicality by realizing the offline phase of Faust et al.'s threshold BBS+ scheme [Fau+23]. We summarize our contribution as follows:

- **PCG Derivation** In Chapter 3, we provide a step-by-step derivation of the PCG by Boyle et al. that leads to both OLE and Vector-OLE constructions. In this context, we present and prove a formula for efficiently generating so-called *roots of unity* by iteration for a specific group of cyclotomic polynomials.
- **Practical Considerations:** In Chapter 4, we investigate potential bottlenecks in the PCG's underlying building blocks and propose practical optimizations to improve their performance for the PCG use case.
- **Benchmarking and Evaluation:** In Chapter 5, we provide thorough benchmarks for the optimized building blocks and then proceed by evaluating the overall performance of our PCG implementation for both OLE and Vector-OLE correlations.
- **Implementing a PCG for BBS+:** In Chapter 6, we demonstrate the utility of PCGs under our practical considerations by proposing a construction that realizes the offline phase of Faust et al.'s threshold BBS+ signature scheme [Fau+23]. We implement this construction using the previously proposed optimizations and evaluate the performance of the schemes offline phase. Notably, our implementation is the first PCG for a threshold signature scheme that supports the t-out-of-n setting. Further, the runtime scales linear for additional parties, making our implementation the first to support higher participant counts without jeopardizing practicality.

1.2 Related Work

While Boyle et al. formally define PCGs in [Boy+19], earlier MPC literature has implicitly used PCG constructions. Examples include bilinear correlations derived from homomorphic secret sharing [Boy+17] and general correlations based on indistinguishability obfuscation [Hal+16]. However, these early PCGs are often inefficient, making them impractical. More useful constructions emerged for linear multi-party correlations [CDI05; GI99] and Vector-OLE correlations built upon the Learning Parity with Noise (LPN) assumption [Boy+18; Sch+19]. Based on this work, Boyle et al. develop an efficient PCG construction [Boy+19; Boy+20] that utilizes the `Ring-LPN` assumption (cf. Section 2.1.2) and advances in function secret sharing [BGI15; BGI16] (cf. Section 4.1.1). Subsequently, the

[2] `https://learn.microsoft.com/en-us/azure/networking/azure-network-latency` (accessed April 2, 2024)

PCG primitive is used prominently in recent MPC protocols that benefit from input-independent preprocessing [BC22; Abr+22; Wag22; Dit+22; Fau+23; KOR23; BC23]. The implementation provided in Abram et al.'s work on a non-interactive threshold ECDSA scheme [Abr+22] is particularly relevant to our work. However, their work focuses on the threshold signature scheme and less on implementation details. To our knowledge, there are no other publicly available implementations or comprehensive practical evaluations of PCGs. The only other threshold signature scheme that uses PCGs for preprocessing is the threshold BBS+ scheme of Faust et al. [Fau+23], for which this work provides the PCG construction and implementation.

2 Preliminaries

Notation. In this work, we denote the security parameter by λ and the set $[k]$ as $\{0, ..., k-1\}$. We denote vectors in bold lower-case $\mathbf{v}$, matrices in bold upper-case $\mathbf{M}$, the tensor product of two vectors as $\mathbf{v}_0 \otimes \mathbf{v}_1$, the outer sum as $\mathbf{v}_0 \boxplus \mathbf{v}_1$ and the inner product as $\langle \mathbf{v}_0, \mathbf{v}_1 \rangle$. We denote the i-th vector element by $\mathbf{v}[i]$. We consider a degree-n τ-sparse vector a vector of dimension n that contains τ non-zero elements. We consider a degree-n τ-sparse polynomial a polynomial of degree n containing τ non-zero coefficients. We refer to both as *dense* when they contain no non-zero elements/coefficients.

2.1 LPN-based Cryptography

The Learning Parity with Noise (LPN) problem has emerged as a popular hardness assumption in several disciplines, including cryptography, machine learning, and coding theory. The basic concept of LPN is the difficulty of solving linear equations when they are contaminated with noise. This noise significantly increases the complexity of deriving accurate solutions, embodying the hardness of the assumption, which can be related to the NP-complete problem of *decoding random linear codes* [YS16]. In the context of cryptography, the LPN assumption has found increasing application, particularly in the design of lightweight encryption and authentication schemes [Pie12]. Many of these applications are provably secure, meaning that any effective attack on such schemes would require overcoming the underlying LPN hardness assumption. In addition, LPN is gaining traction in post-quantum cryptography. Unlike other popular assumptions, such as prime factorization and the discrete logarithm problem, which are vulnerable to quantum attacks via Shor's algorithm [Sho99], the LPN problem has not yet been efficiently solved by quantum algorithms [ZYZ18]. In the following, we describe basic LPN and extend this description to LPN over rings with static leakage, which serves as the main security assumption in this work.

2.1.1 Primal- and Dual-LPN

Primal Learning Parity with Noise (`Primal-LPN`) represents the foundational version of the LPN problem, offering adversaries black-box access to two oracles. The first oracle, containing a secret vector $\mathbf{s} \in \mathbb{F}_2^n$, generates an error vector $\mathbf{e} \in \mathbb{F}_2^n$ with each element independently derived from a Bernoulli distribution of probability τ. It then outputs a pair: a uniformly random matrix $\mathbf{A} \in \mathbb{F}_2^{n \times n}$ and a vector $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e} \in \mathbb{F}_2^n$. The second oracle provides a uniformly random matrix $\mathbf{A}$ and vector $\mathbf{t}$, both in $\mathbb{F}_2^{n \times n}$ and $\mathbb{F}_2^n$ respectively, without the underlying $\mathbf{s}$ and $\mathbf{e}$ relationship. The adversary's task (in this decisional formulation of `Primal-LPN`) is to distinguish between pairs $(\mathbf{A}, \mathbf{t})$ that adhere to the linear equation $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e}$ and randomly generated pairs, which is computationally hard due to the obfuscating effect of the noise vector $\mathbf{e}$ [ZYZ18]. Notably, the problem description via Bernoulli implies that the problem remains hard for sparse noise vectors (i.e., containing a fixed number of zero elements), as proven by Liu et al. in [Liu+17].

Dual-LPN Assumption. The `Dual-LPN` assumption is equivalent to `Primal-LPN` [CRR21] and provides a simplified representation of the LPN challenge. In this formulation, the secret is embedded within the linear transformation itself. Given public LPN matrix $\mathbf{H} \in \mathbb{F}_2^{m \times n}$, the adversary now only needs to distinguish between the product $\mathbf{H} \cdot \mathbf{e}$ for a noise vector $\mathbf{e} \in \mathbb{F}_2^n$ and another matrix chosen uniformly at random. The complexity again arises from the noise vector $\mathbf{e}$ obfuscating the linear relationship.

2.1.2 Ring-LPN

Ring Learning Parity with Noise (`Ring-LPN`) is an extension of the LPN problem to rings and was first introduced by Heysen et al. [Hey+12] (although over $\mathbb{Z}_2$). Instead of dealing with vectors over a binary field, `Ring-LPN` operates over a polynomial ring, typically denoted $R = \mathbb{Z}_p[X]/F(X)$ for a prime p and degree-N monic polynomialc[1] $F(X) \in \mathbb{Z}_q[X]$. The error $e(X)$ is sampled from a uniform distribution $\mathcal{HW}_t^R$ as a t-sparse polynomial over R by sampling t noise positions $\mathbf{p} \leftarrow [N]^t$ and payloads $\mathbf{b} \leftarrow [\mathbb{Z}_p]^t$ uniformly for outputting ring element $e(X) = \sum_{j=0}^{t-1} \mathbf{b}[j] \cdot X^{\mathbf{p}[j]}$.

Generalizing Ring-LPN

As described by Boyle et al. in [Boy+20] `Ring-LPN` can be further generalized to R^c-LPN$_\tau$ (also called `Module-LPN`) by replacing $a^{(i)} \cdot e$ with the inner product $\langle \mathbf{a}^{(i)}, \mathbf{e} \rangle$ between length-$(c-1)$ vectors over R, for some constant $c \geq 2$. In this context, the adversary gains access to a leakage that allows him to guess any error coordinates as long as the guess is correct. Modeling this leakage is necessary because it allows for some flexibility in the PCG construction. For a more formal definition of R^c-LPN$_\tau$ *with static leakage*, we recall the security game of [Abr+22] in Figure 2.1. The R^c-LPN$_\tau$ problem *with static leakage* is hard if for every probabilistic polynomial-time adversary $\mathcal{A}$ it holds that

$$\left| \mathtt{Pr}\left(\mathcal{G}^{\text{Module-LPN}}_{R,\tau,c,\mathcal{A}}(\lambda) = 1 \right) - \frac{1}{2} \right| \leq \mathtt{negl}(\lambda).$$

In particular, [Boy+20] reports that the hardness of R^c-LPN$_\tau$ depends entirely on (c, τ) when a cyclotomic polynomial $F(X)$ is chosen for R due to a dimension reduction attack that renders N (from which the noise positions are sampled in $\mathcal{HW}_\tau^R$) irrelevant. The authors further report $(c, \tau) \in \{(4, 16), (8, 5), (2, 76)\}$ to achieve 128-bit security.

2.2 Threshold Signatures

Blakley and Shamir introduced the concept of threshold secret sharing [Bla79][Sha79], describing a system where a secret x is distributed into n parts. In such an t-out-of-n scheme, an efficient algorithm can reconstruct the secret x using any t of these parts, yet any collection of fewer than t parts reveals no information about x.

Threshold signature schemes. Bringing this concept over to signature schemes, Desmedt and Yair later proposed the first thresholdized signature scheme based on RSA [DF91]. Analogous to threshold secret sharing, in a t-out-of-n threshold signature scheme, a collective of any t participants can generate a valid signature on a message that is verifiable using a single, publicly known key. The distribution of secret shares can be facilitated either by a trusted dealer or by a dealer-less setup involving interactive protocols among the participants.

Definition 1 (**t-out-of-n Threshold Signature Scheme (TSS)**). *Let $P_0, ..., P_{n-1}$ denote the parties participating and let t be the threshold, such that $t \leq n-1$. We define a t-out-of-n TSS as a tuple of algorithms* (`ThreshKeyGen, ThreshSig, CombineSig, Verify`) *such that*

- `ThreshKeyGen`(1^λ) *is a probabilistic polynomial-time algorithm that takes a security parameter λ to output to each party $P_{i\in[n]}$ the public key pk as well as a private key share sk_i.*
- `ThreshSig`$_{sk_i}(m)$ *is a polynomial-time algorithm which takes a secret share sk_i and a message m to output a partial signature σ_i.*
- `CombineSig`$(\sigma_{i\in[n]}, ...)$ *is a polynomial-time algorithm that aggregates a minimum of t partial signatures $\sigma_{i\in[n]}$ to outputs a single signature σ.*

[1] A monic polynomial is one where the leading coefficient, the coefficient of the highest degree term, is 1.

Security Game: $\mathcal{G}^{\text{Module-LPN}}_{R,\tau,c,\mathcal{A}}(\lambda)$

Initialisation:

- The challenger initialises the adversary $\mathcal{A}$ with security parameter $\mathbb{1}^\lambda$
- The challenger samples a random bit b $\xleftarrow{\$} \{0,1\}$ and c ring elements $e_0, e_1, \ldots, e_{c-1} \xleftarrow{\$} \mathcal{HW}^R_\tau$, with the j-th noise position of e_i being denoted by $p_i[j]$.

Query Phase:

- The adversary $\mathcal{A}$ may adaptively issue queries (i, j, I), where $i \in [c]$, $j \in [\tau]$, and $I \subseteq [N]$.
- If $p_i[j] \in I$, the challenger responds with "Success" and waits for the next query; otherwise, it sends "Abort" and enters the next phase.

Challenge Phase:

- For each $i \in [c-1]$, the challenger samples a_i uniformly from R and fixes $a_{c-1} = 1$.
- It computes the challenge response (inner product) u_1 as follows:

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

- An independent sample u_0 is drawn uniformly from R.
- The challenger presents $\mathcal{A}$ with the tuple $(a_0, a_1, \ldots, a_{c-2}, u_b)$.

Response:

- The adversary $\mathcal{A}$ submits a guess b'.
- The game outputs 1 (indicating success) if $b = b'$, and 0 otherwise.

Figure 2.1: The R^c-LPN$_\tau$ Security Game

- $\texttt{Verify}_{pk}(\sigma, m)$ *is a polynomial-time algorithm that takes the public key pk, a signature* σ*, and a message* m*. It outputs 1 if and only if* σ *is a valid signature of* m *under the public key pk.*

The main security attribute of a TSS is *unforgeability*. We consider a TSS unforgeable under the condition that no adversary can, with non-negligible probability, forge a signature on any new, previously unsigned message m. This holds even if the adversary has compromised up to $t-1$ parties. The adversary's capabilities include access to the `ThreshKeyGen` and `ThreshSig` protocols for selecting input messages $m_1, \ldots, m_k$ adaptively and obtaining signatures on these messages which makes this notion analogous to the notion of *existential unforgeability* under chosen message attack for standard signature schemes as defined by Goldwasser et al. in [GMR88]. Informally, the security guarantee ensures that adversaries cannot derive secret key shares from observing legitimate signatures. A comprehensive formal definition of a secure TSS is detailed in [Gen96].

2.3 Pseudorandom Correlation Generators

A Pseudorandom Correlation Generator (PCG) is a distributed form of a pseudorandom generator that allows participants to generate seeds that can then be locally extended by each party to produce correlated pseudorandom bit streams. Realizing this functionality, we define a PCG as a pair of

algorithms (`PCG.Gen`, `PCG.Eval`) by using the notion of reverse-sampled correlation generators from [Boy+20] and following the general definition of [Abr+22].

Definition 2 (**Reverse-sampleable Correlation Generator**). *Let* `CGen` *be a PPT algorithm that implements an n-party correlation generator that takes the security parameter* 1^λ *as input and gives correlated outputs* $R_0, R_1, ..., R_{n-1}$ *for each party respectively. We consider* `CGen` *to be reverse samplabe if there exists a PPT algorithm* `RSample`*, s.t. for each set of corrupted parties* $\mathcal{C} \subseteq [n]$ *the distribution*

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

is computationally indistinguishable from $\mathcal{C}$.

Informally, the reverse-sampleable correlation generator allows to derive remaining correlations given a subset of the output of the correlation generator `CGen`. Since we are using correlations that are reverse-sampleable, this definition suits us to formalize the PCG.

Definition 3 (**Pseudorandom Correlation Generator (PCG)**). *Let* `CGen` *be a reverse-sampled correlation generator. A pseudorandom correlation generator for* `CGen` *is a pair of algorithms* (`PCG.Gen`, `PCG.Expand`) *such that*

- $\texttt{PCG.Gen}(1^\lambda)$ *is a probabilistic polynomial-time (PPT) algorithm that takes a security parameter* λ *to generate a pair of seeds* (k_0, k_1).
- $\texttt{PCG.Expand}(b, k_p)$ *is a deterministic polynomial-time algorithm that takes a seed* k_p *for party* p *and outputs a bit string* $R_p \in \{0,1\}^n$.

The scheme must satisfy the following properties:

- **Correctness:** *The respective outputs of* `PCG.Expand` *are correlated such that the following distribution is computationally indistinguishable from* $\texttt{CGen}(1^\lambda)$.

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

- **Security:** *For every subset of corrupted parties* $\mathcal{C} \subseteq [n]$*, the following distributions are computationally indistinguishable*

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

$$\left\{ \begin{array}{l} (\kappa_i)_{i\in\mathcal{C}} \\ \\ (R_i)_{i\in[n]} \end{array} \;\middle|\; \begin{array}{l} (\kappa_0, \kappa_1, \ldots, \kappa_{n-1}) \xleftarrow{\$} \texttt{CGen}\left(1^\lambda\right) \\ \forall i \in \mathcal{C}: \quad R_i \leftarrow \texttt{PCG.Expand}(\kappa_i, i) \\ (R_i)_{i\in[n]\setminus\mathcal{C}} \xleftarrow{\$} \texttt{RSample}\left(1^\lambda, \mathcal{C}, (R'_i)_{i\in\mathcal{C}}\right) \end{array} \right\}$$

The definition of correctness implies that the expansion yields the same correlations as generated by $\texttt{CGen}(1^\lambda)$. Security-wise, the information leaked by a subset of seeds about the rest of the outputs must be no more than what can be extracted from their expansion.

2.4 Building Blocks

2.4.1 Function Secret Sharing

Function Secret Sharing [BGI15] (FSS) schemes involve the random division of a secret function f into two or more component functions f_i, each represented by a unique key k_i, such that the sum of all component function outputs for any given input x, is equal to the output of the original function f, and any subset of the keys $\{k_i\}$ conceals the secret function f from any (partial) disclosure. We use 2-party FSS and recall the formalization of [Boy+20] in the following.

Definition 4 (**2-Party Function Secret Sharing**). *Let $C = \{f : I \to \mathbb{G}\}$ be a class of functions, where the description of each function f has the input domain I and the output domain $(\mathbb{G}, +)$. We define a 2-party FSS scheme for C as a pair of algorithms* `(FSS.Gen, FSS.Eval)` *such that*

- $\texttt{FSS.Gen}(1^\lambda, f)$ *is a probabilistic polynomial-time (PPT) algorithm that takes a security parameter λ and a function $f \in C$ to produce two keys (k_0, k_1) that imply both I and $\mathbb{G}$.*
- $\texttt{FSS.Eval}(b, k_b, x)$ *is a deterministic polynomial-time algorithm that takes a key k_b for party b and an input x to output a group element $y_b \in \mathbb{G}$.*

The scheme must satisfy the following properties:

- **Correctness:** *For any $f \in C$, $x \in I$ and generated keys $(k_0, k_1) \xleftarrow{\$} \texttt{FSS.Gen}(1^\lambda, f)$ it holds that $\sum_{b\in\{0,1\}} \texttt{FSS.Eval}(b, k_b, x) = f(x)$*
- **Security:** *For any $b \in \{0, 1\}$, the function $\texttt{Leak}(f)$ that reveals the input and output domains of f and any PPT distinguisher $\mathcal{D}$, the advantage of $\mathcal{D}$ in distinguishing between k_b generated from $\texttt{FSS.Gen}(1^\lambda, f)$ and a simulated key from $\texttt{Sim}(1^\lambda, \texttt{Leak}(f))$ is negligible in λ.*

Further, we define $\texttt{FSS.FullEval}(b, k_b)$ as the full-domain evaluation of the function f, which outputs a vector of $|I|$ elements in $\mathbb{G}$ representing running `FSS.Eval` on all $x \in I$. For the following instantiations of the FSS scheme for *point functions*, running `FSS.FullEval` is significantly faster than running individual instances of `FSS.Eval` for a full-domain evaluation.

Applying FSS to Point Functions

A *point function* is a function f that is characterized by a single special point and a corresponding non-zero value, such that all points that are not the special point result in the function evaluating to zero. We define the instantiation of FSS for a *point function* as follows.

Definition 5 (**Distributed Point Function (DPF)**). *For domain N, abelian group $\mathbb{G}$, special point $\alpha \in [N]$, and non-zero element $\beta \in \mathbb{G}$, the point function $f_{\alpha,\beta}$ is the function $f_{\alpha,\beta} : [N] \to \mathbb{G}$ such that $f_{\alpha,\beta}(x) = \beta$ if $x = \alpha$, and $f_{\alpha,\beta}(x) = 0$ otherwise. A distributed point function (DPF) is an FSS scheme for $f_{\alpha,\beta}$.*

We naturally extend this to a *sum of points function* so that the function contains t special points, each of which evaluates to a respective non-zero element.

Definition 6 (**Distributed Sum of Point Function (DSPF)**). *For $\boldsymbol{\alpha} = (\alpha_1, ..., \alpha_t) \in [N]^t$ and $\boldsymbol{\beta} = (\beta_1, ..., \beta_t) \in \mathbb{G}^t$ the sum of points function $f_{\boldsymbol{\alpha},\boldsymbol{\beta}}$ is defined by*

$$f_{\boldsymbol{\alpha},\boldsymbol{\beta}}(x) = \sum_{i=1}^{t} f_{\alpha_i,\beta_i}(x)$$

where each $f_{\alpha_i,\beta_i}(x) = \beta_i$ if $x = \alpha_i$ for some α_i in $\boldsymbol{\alpha}$, and $f_{\alpha_i,\beta_i}(x) = 0$ otherwise. A distributed sum of point function (DSPF) is an FSS scheme for $f_{\boldsymbol{\alpha},\boldsymbol{\beta}}$.

Note that our use of DSPFs for a `Ring-LPN` based PCG does not require the elements in $\boldsymbol{\alpha}$ to be unique [Boy+20]. In the following, we slightly adapt the FSS notation for readability and denote $\mathtt{DSPF}^t_N\mathtt{.Gen}(1^\lambda, \boldsymbol{\alpha}, \boldsymbol{\beta})$ for key generation and $\mathtt{DSPF}^t_N\mathtt{.Eval}(\sigma, k_\sigma, x)$ for evaluating a share of $f_{\boldsymbol{\alpha},\boldsymbol{\beta}}$ at a position x. Further, let $\mathtt{DSPF}^t_N\mathtt{.FullEval}(\sigma, k_\sigma)$ denote a *full domain evaluation* which means calling $\mathtt{DSPF}^t_N\mathtt{.Eval}(\sigma, k_\sigma, x)$ on all position $x \in [N]$. We reuse the same notation for plain DPFs.

2.4.2 OLE and Vector-OLE Correlations

Oblivious Linear Evaluation (OLE) is a two-party protocol wherein the receiver obtains a secret linear combination of two elements possessed by the sender. Specifically, the sender holds a linear function $f(x) = ax + b$, where a and b are secret coefficients. The receiver holds a value x and can compute $f(x)$ without learning anything about a and b, and simultaneously, the sender does not learn anything about x.

VOLE: Vector-Oblivious Linear Evaluation (VOLE) is a natural extension of OLE. In the VOLE setting, the sender holds a pair of secret vectors $(\mathbf{a}, \mathbf{b})$, which together define a sequence of linear functions. The functionality allows the receiver to learn $\mathbf{a}x + \mathbf{b}$ without sharing x with the sender or the sender revealing $\mathbf{a}$ and $\mathbf{b}$ beyond the computed result. This extension allows for the batch processing of linear functions, enhancing efficiency in scenarios that necessitate the simultaneous evaluation of multiple OLE over a constant x.

Correlations: Both constructions can be interpreted as generating correlated tuples. An OLE tuple thus represents a 2-party correlation in which party P_1 holds values (a, b) and party P_2 holds values (x, y) such that $a \cdot x = y + b$. For VOLE, x is a fixed constant and (a, b, y) become vectors such that $\mathbf{a} \cdot x = \mathbf{y} + \mathbf{b}$.

2.4.3 Cyclotomic Polynomials

In abstract algebra, cyclotomic polynomials are irreducible polynomials with integer coefficients that play a fundamental role in the study of field extensions and roots of unity. The N-th cyclotomic polynomial, denoted $\Phi_N(X)$ as defined in Equation 2.1, is the unique monic polynomial that divides $X^N - 1$, but does not divide $X^k - 1$ for any $k < N$. Its degree is equal to Euler's totient function $\varphi(N)$, which counts the number of integers less than N that are coprime to N.

$$\Phi_N(X) = \prod_{\substack{1 \le k \le N \\ \gcd(k,\eta)=1}} \left(X - e^{2i\pi\frac{k}{N}}\right). \tag{2.1}$$

Roots of Unity

A key characteristic of cyclotomic polynomials is the existence of special points known as *roots of unity*. These roots play a significant role in algebraic number theory, forming a finite cyclic group under multiplication. The cyclotomic polynomial serves as the minimal polynomial of these roots over the field of rational numbers. The number of roots of unity for a given cyclotomic polynomial is intrinsically tied to its degree. Since the degree is determined by Euler's totient function, $\varphi(N)$, we denote the roots of unity of the N-th cyclotomic polynomial as $\xi_j \in (\xi_0, \ldots, \xi_{\varphi(N)})$, where $\Phi_N(\xi_j) = 0$.

2.4.4 BBS+ Signature Scheme

Boneh, Boyen, and Shacham implicitly introduced the BBS signature scheme in the context of one of the first group signature schemes based on bilinear pairings [BBS04]. Later, Au et al. proposed BBS+ as a provably secure extension of this scheme [ASM06]. The algebraic structure of BBS+ makes it desirable for anonymous credentials since it supports efficient zero-knowledge proofs of knowledge

alongside selective disclosure while maintaining a small constant-size signature. We formalize the BBS+ signature scheme in the following way:

Construction 1 (**BBS+ Signature Scheme**). *Let $\mathcal{G} = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, g_1, g_2, e)$ be a bilinear group setting, where $\mathbb{G}_1$ and $\mathbb{G}_2$ are groups of prime order p with generators $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$. The map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a bilinear pairing. Let $h_{l\in[0..k]}$ be a set of random elements in $\mathbb{G}_1$. The BBS+ signature scheme is composed of the following polynomial-time algorithms:*

- $\texttt{KeyGen}(1^\lambda)$*: On input of a security parameter λ sample $x \xleftarrow{\$} \mathbb{Z}_p^*$ and compute $y = g_2^x$ to output the key pair $(pk, sk) = (y, x)$.*
- $\texttt{Sign}_{sk}(\{m_\ell\}_{\ell\in[k]})$*: Given secret key $sk = x$ and a message vector $\{m_\ell\}_{\ell\in[k]}$, sample $e, s \xleftarrow{\$} \mathbb{Z}_p$ and compute $A := \left(g_1 \cdot h_0^s \cdot \prod_{\ell\in[k]} h_\ell^{m_\ell}\right)^{\frac{1}{x+e}}$ to output signature $\sigma = (A, e, s)$.*
- $\texttt{Verify}_{pk}(\{m_\ell\}_{\ell\in[k]}, \sigma)$*: Parse the signature $\sigma = (A, e, s)$ and public key $pk = y$ to output 1 iff $e(A, y \cdot g_2^e) = e(g_1 \cdot h_0^s \cdot \prod_{\ell\in[k]} h_\ell^{m_\ell}, g_2)$.*

The correctness arises from the bilinearity property of the map e and the fact that A is computed as an exponentiation involving the inverse of $x + e$, effectively "canceling out" with $y \cdot g_2^e$ when paired in the bilinear map. Security-wise, the signature scheme is proven to be *strong unforgeable* under the q-strong Diffie Hellman (SDH) assumption [ASM06], which implies that an attacker cannot come up with signature forgeries, including forgeries for messages that have already been signed.

3 PCG for (Vector-)OLE

In this chapter, we recall a PCG variant based on the R^c-LPN$_\tau$ assumption (cf. Figure 2.1) first proposed by Boyle et al. [Boy+20], for which we derive practical considerations afterward. The PCG supports expansion to (Vector-)OLE correlations, giving it many applications in cryptographic constructions. We start with the first naive PCG construction proposed in [Boy+19], which is based on the `Dual-LPN` assumption. With this basic construction, although limited in practicality by its inefficiency, we derive the intuition for using LPN to realize a PCG. We then proceed to move this PCG construction to `Ring-LPN` and show how this step mitigates the bottlenecks from before. Since `Ring-LPN` itself does not provide enough randomness, we recall the secure PCG construction by Boyle et al. [Boy+20] that utilizes a generalization of `Ring-LPN` over multiple ring elements (so-called `Module-LPN` or R^c-LPN$_\tau$) for OLE and show an adaptation that realizes VOLE correlations.

3.1 PCG based on Dual-LPN

We first introduce a naive approach under a simplified setting. This PCG is based on LPN over a ring $\mathbb{Z}_p$ (notice, this is not `Ring-LPN`) and follows the `Dual-LPN` assumption. Under `Dual-LPN` over ring $\mathbb{Z}_p$, we assume that the distribution

$$\left\{\mathbf{H}, \mathbf{H}\cdot\mathbf{e} \;\middle|\; \mathbf{H} \xleftarrow{\$} \mathbb{Z}_p^{m\times n}, \mathbf{e} \xleftarrow{\$} \left\{\mathbf{v}\in\mathbb{Z}_p^n \;\middle|\; \|\mathbf{v}\|_0 = \tau\right\}\right\}$$

is computationally indistinguishable from uniform for $\mathbf{e}$ being a τ-sparse error vector. The intuition for realizing a PCG for an OLE correlation from this is as follows: both parties share the same `Dual-LPN` instance, meaning the LPN matrix $\mathbf{H}$ is public. However, each party $\sigma\in\{0,1\}$ holds a unique, secret error vector $\mathbf{e}_\sigma$. The parties additively secret share a product of their secret sparse error vectors, so they attain an OLE correlation (cf. Section 2.4.2) over them. The parties then use the LPN matrix to (locally) expand their OLE correlation over sparse vectors to OLE correlations over dense vectors. The parties, therefore, exploit the fact that the small true randomness in a sparse error vector $\mathbf{e}$ can be expanded to a dense pseudorandom vector by computing $\mathbf{H}\cdot\mathbf{e}$, although $\mathbf{H}$ being public. In the following, the approach is presented more formally.

Construction

We start with the generation of the PCG seeds. A τ-sparse error vector $\mathbf{e}_\sigma\in\mathbb{Z}_p^n$ is sampled for each party $\sigma\in\{0,1\}$. Then the tensor product $(\mathbf{e}_0\otimes\mathbf{e}_1)$ is computed, which results in a $(n\times n)$-matrix consisting of τ^2 non-zero elements. The matrix is transformed into vector notation $(e_0,\ldots e_{n^2-1})$, where each element represents a corresponding coordinate as depicted in Equation 3.1. By interpreting this vector as a multi-point function that maps the τ^2 vector indices (of the non-zero elements) onto their respective values, Function Secret Sharing (FSS) (cf. Definition 4) can be used for implicitly secret-sharing the result of $(\mathbf{e}_0\otimes\mathbf{e}_1)$ additively. Using FFS here is appealing since there exist efficient schemes for multi-point functions [BGI15; BGI16]. The keys generated by `FSS.Gen` and the error vectors $\mathbf{e}_\sigma$ serve as the PCG seed.

$$\mathbf{e}_0\otimes\mathbf{e}_1 = \begin{pmatrix} \tilde{z}_0 & \cdots & \tilde{z}_n \\ \vdots & \ddots & \vdots \\ \tilde{z}_{n^2-n} & \cdots & \tilde{z}_{n^2-1}\end{pmatrix} \xrightarrow{\texttt{FSS.Gen}} (k_0,k_1), \quad \begin{aligned}\tilde{\mathbf{z}}_0 &= \texttt{FSS.FullEval}(0,k_0) = (\tilde{z}_0^0,\ldots,\tilde{z}_{n^2-1}^0)\\ \tilde{\mathbf{z}}_1 &= \texttt{FSS.FullEval}(1,k_1) = (\tilde{z}_0^1,\ldots,\tilde{z}_{n^2-1}^1)\end{aligned} \tag{3.1}$$

By evaluating their respective FSS key k_σ, each party obtains additive shares $\tilde{z}_x^\sigma$ of matrix/vector element e_x, such that $\tilde{z}_x = \tilde{z}_x^0 + \tilde{z}_x^1$ for $x\in[n^2]$. Notice here, that $\tilde{z}_x$ itself is already multiplicative

shared through $\mathbf{e}_\sigma$ as it stems from $\mathbf{e}_0 \otimes \mathbf{e}_1$ (Equation 3.2). For a full reconstruction of $\mathbf{e}_0 \otimes \mathbf{e}_1$, each party must query `FSS.Eval` n^2 times, which we denote in simplified form by `FSS.FullEval`.

$$\begin{aligned}\mathbf{e}_0 \otimes \mathbf{e}_1 &= \tilde{\mathbf{z}}_0 + \tilde{\mathbf{z}}_1 \\ &= \texttt{FSS.FullEval}(0, k_0) + \texttt{FSS.FullEval}(1, k_1) \\ &= (\tilde{z}_0^0, \ldots, \tilde{z}_{n^2-1}^0) + (\tilde{z}_0^1, \ldots, \tilde{z}_{n^2-1}^1) \\ &= (\tilde{z}_0^0 + \tilde{z}_0^1, \ldots, \tilde{z}_{n^2-1}^0 + \tilde{z}_{n^2-1}^1) \\ &= (\tilde{z}_0, \ldots \tilde{z}_{n^2-1})\end{aligned} \tag{3.2}$$

Each party now holds an multiplicative share of $(\mathbf{e}_0 \otimes \mathbf{e}_1)$ through $\mathbf{e}_\sigma$ and an additive share of the same operation through $\tilde{\mathbf{z}}_\sigma$. From this, the parties utilize the relation to the LPN matrix $\mathbf{H}$ to (locally) expand a dense vector of pseudorandom OLE correlations. By expanding their values under LPN multiplicative share $\mathbf{x}_\sigma$ and additive share $\mathbf{z}_\sigma$ (Equation 3.3) are obtained, such that $\mathbf{x}_0 \otimes \mathbf{x}_1 = \mathbf{z}_0 + \mathbf{z}_1$ (Equation 3.4).

$$\begin{aligned}\mathbf{x_0} &= \mathbf{H} \cdot \mathbf{e}_0 \quad & \mathbf{z}_0 &= (\mathbf{H} \otimes \mathbf{H}) \cdot \tilde{\mathbf{z}}_0 \\ \mathbf{x_1} &= \mathbf{H} \cdot \mathbf{e}_1 \quad & \mathbf{z}_1 &= (\mathbf{H} \otimes \mathbf{H}) \cdot \tilde{\mathbf{z}}_1\end{aligned} \tag{3.3}$$

Notice that, at this point, the parties do not need to interact with each other. Additive share $\mathbf{z}_\sigma$ can be computed independently by using the previously computed additive shares of $(\mathbf{e}_0 \otimes \mathbf{e}_1)$ and exploiting the distributive property of matrix multiplication over vector addition as shown in Equation 3.4.

$$\begin{aligned}\mathbf{x_0} \otimes \mathbf{x_1} &= (\mathbf{H} \cdot \mathbf{e}_0) \otimes (\mathbf{H} \cdot \mathbf{e}_1) \\ &= (\mathbf{H} \otimes \mathbf{H}) \cdot (\mathbf{e}_0 \otimes \mathbf{e}_1) \\ &= (\mathbf{H} \otimes \mathbf{H}) \cdot (\tilde{\mathbf{z}}_0 + \tilde{\mathbf{z}}_1) \\ &= (\mathbf{H} \otimes \mathbf{H}) \cdot \tilde{\mathbf{z}}_0 + (\mathbf{H} \otimes \mathbf{H}) \cdot \tilde{\mathbf{z}}_1 \\ &= \mathbf{z}_0 + \mathbf{z}_1\end{aligned} \tag{3.4}$$

Note that both $\mathbf{x}_0$ and $\mathbf{x}_1$ are pseudorandom under the `Dual-LPN` assumption. The construction therefore extends the τ-sparse error vectors $\mathbf{e}_i$ to realize n^2 independent secret shared multiplications of pseudorandom elements as $\mathbf{x}_0 \otimes \mathbf{x}_1 = \mathbf{z}_0 + \mathbf{z}_1$, thus realizing n^2 OLE correlation such that each party σ holds the pair $(\mathbf{x}_\sigma, \mathbf{z}_\sigma)$. Although we achieve a PCG for OLE with this construction, the approach remains inefficient: The computation of (1) the multiplication $H \cdot \mathbf{e}_i$, (2) the tensor $(\mathbf{e}_0 \otimes \mathbf{e}_1)$, and (3) `FSS.FullEval` for evaluating each secret share is of quadratic complexity. Certain FSS schemes allow for more efficient full-domain evaluation (cf. Section 4.1.1). However, the construction is still computationally expensive when n is large, which is to be expected since the number of individual OLEs we extend must be as large as possible to compensate for the setup cost.

3.2 PCG based on Ring-LPN

Efficient polynomial algorithms can be employed to mitigate the computational limitations described above. Notice that any n dimensional vector has a polynomial representation of degree n, where each vector element serves as a coefficient for each monomial, with the element's index acting as the monomial's exponent. Applying this to the previous approach, we get the following: let the error vectors $(\mathbf{e}_0, \mathbf{e}_1)$ now be degree-n τ-sparse polynomials (e_0, e_1). The multiplicative shares (x_0, x_1) are derived independently from evaluating the respective error polynomial on position x. The additive shares (z_0, z_1) are computed as a secret-shared evaluation of the polynomial product $e_0 \cdot e_1$, which are then also evaluated at x.

Construction

Doing the above naively would break the security of the PCG since (x_0, x_1) must remain indistinguishable from random. To achieve this, Boyle et al. [Boy+20] proposed an approach that exploits

the `Ring-LPN` assumption by letting $R_p := \mathbb{Z}_p[X]/(F(X))$ be a ring for a degree-n polynomial $F(X)$. Thus, for a randomly sampled public polynomial $a \in R_p$ and sparse polynomials $e, f \in R_p$, we assume that $(a, a \cdot e + f \mod F(X))$ is indistinguishable from random under `Ring-LPN` (cf. Section 2.1.2). From this, we construct the PCG: Given degree-n τ-sparse polynomials (e_0, e_1) and (f_0, f_1) each party σ obtains (e_σ, f_σ) as their LPN secret. The parties additively secret share the element-wise (polynomial) multiplication of their secrets using FSS as depicted in Equation 3.5.

$$\begin{bmatrix} e_0 \\ f_0 \end{bmatrix} \otimes \begin{bmatrix} e_1 \\ f_1 \end{bmatrix} = \begin{bmatrix} e_0 \cdot e_1 \\ e_0 \cdot f_1 \\ f_0 \cdot e_1 \\ f_0 \cdot f_1 \end{bmatrix}, \quad \begin{aligned} (e_0 \cdot e_1) &\xrightarrow{\texttt{FSS.Gen}} (k_0^0, k_1^0) \\ (e_0 \cdot f_1) &\xrightarrow{\texttt{FSS.Gen}} (k_0^1, k_1^1) \\ (f_0 \cdot e_1) &\xrightarrow{\texttt{FSS.Gen}} (k_0^2, k_1^2) \\ (f_0 \cdot f_1) &\xrightarrow{\texttt{FSS.Gen}} (k_0^3, k_1^3) \end{aligned} \tag{3.5}$$

Each party σ receives their FFS keys $(k_\sigma^i)_{i\in[4]}$ and is then able to compute an additive share $(\tilde{z}_\sigma^i)_{i\in[4]}$ of each multiplication by performing a full domain evaluation $\tilde{z}_\sigma^i = \texttt{FSS.FullEval}(\sigma, k_\sigma^i)$. Note that the following, therefore, applies (Equation 3.6).

$$\begin{aligned} \tilde{z}_0^0 + \tilde{z}_1^0 &= \texttt{FSS.FullEval}(0, k_0^0) + \texttt{FSS.FullEval}(1, k_1^0) = (e_0 \cdot e_1) \\ \tilde{z}_0^1 + \tilde{z}_1^1 &= \texttt{FSS.FullEval}(0, k_0^1) + \texttt{FSS.FullEval}(1, k_1^1) = (e_0 \cdot f_1) \\ \tilde{z}_0^2 + \tilde{z}_1^2 &= \texttt{FSS.FullEval}(0, k_0^2) + \texttt{FSS.FullEval}(1, k_1^2) = (f_0 \cdot e_1) \\ \tilde{z}_0^3 + \tilde{z}_1^3 &= \texttt{FSS.FullEval}(0, k_0^3) + \texttt{FSS.FullEval}(1, k_1^3) = (f_0 \cdot f_1) \end{aligned} \tag{3.6}$$

Continuing from here, the parties store $(\tilde{z}_\sigma^i)_{i\in[4]}$ as a 4-dimensional polynomial vector $\tilde{\mathbf{z}}_\sigma$. The additive combination of $(\tilde{\mathbf{z}}_0, \tilde{\mathbf{z}}_1)$ recovers the element-wise product of the parties LPN secrets (Equation 3.7).

$$\tilde{\mathbf{z}}_0 + \tilde{\mathbf{z}}_1 = \begin{bmatrix} \tilde{z}_0^0 + \tilde{z}_1^0 \\ \tilde{z}_0^1 + \tilde{z}_1^1 \\ \tilde{z}_0^2 + \tilde{z}_1^2 \\ \tilde{z}_0^3 + \tilde{z}_1^3 \end{bmatrix} = \begin{bmatrix} e_0 \cdot e_1 \\ e_0 \cdot f_1 \\ f_0 \cdot e_1 \\ f_0 \cdot f_1 \end{bmatrix} = \begin{bmatrix} e_0 \\ f_0 \end{bmatrix} \otimes \begin{bmatrix} e_1 \\ f_1 \end{bmatrix} \tag{3.7}$$

Each party σ now holds a multiplicative share of the element-wise multiplication of their secrets through their LPN secret (e_σ, f_σ), as well as an additive share of the same operation through $\tilde{\mathbf{z}}_\sigma$. From this, the parties utilize the public LPN parameter a to generate their tuple $(x_\sigma, \mathbf{z}_\sigma)$ as shown in Equation 3.8.

$$x_\sigma = a \cdot e_\sigma + f_\sigma, \qquad \mathbf{z}_\sigma = \left(\begin{bmatrix} a \\ 1 \end{bmatrix} \otimes \begin{bmatrix} a \\ 1 \end{bmatrix} \right) \cdot \tilde{\mathbf{z}}_\sigma \tag{3.8}$$

This results in a single shared OLE correlation over R_p, as tuples $(x_\sigma, \mathbf{z}_\sigma)_{\sigma\in\{0,1\}}$ are correlated, such that $x_0 \otimes x_1 = \mathbf{z}_0 + \mathbf{z}_1$ as shown in Equation 3.9. Notice also that x_0 and x_1 are now pseudorandom under the `Ring-LPN` assumption.

$$\begin{aligned} x_0 \otimes x_1 &= (a \cdot e_0 + f_0) \otimes (a \cdot e_1 + f_1) \\ &= \begin{bmatrix} a^2 \cdot e_0 \cdot e_1 \\ a \cdot e_0 \cdot f_1 \\ a \cdot e_1 \cdot f_0 \\ f_0 \cdot f_1 \end{bmatrix} = \begin{bmatrix} a^2 \\ a \\ a \\ 1 \end{bmatrix} \cdot \begin{bmatrix} e_0 \cdot e_1 \\ e_0 \cdot f_1 \\ e_1 \cdot f_0 \\ f_0 \cdot f_1 \end{bmatrix} = \begin{bmatrix} a^2 \\ a \\ a \\ 1 \end{bmatrix} \cdot \begin{bmatrix} \tilde{z}_0^0 + \tilde{z}_1^0 \\ \tilde{z}_0^1 + \tilde{z}_1^1 \\ \tilde{z}_0^2 + \tilde{z}_1^2 \\ \tilde{z}_0^3 + \tilde{z}_1^3 \end{bmatrix} \\ &= \left(\begin{bmatrix} a \\ 1 \end{bmatrix} \otimes \begin{bmatrix} a \\ 1 \end{bmatrix} \right) \cdot (\tilde{\mathbf{z}}_0 + \tilde{\mathbf{z}}_1) = \left(\left(\begin{bmatrix} a \\ 1 \end{bmatrix} \otimes \begin{bmatrix} a \\ 1 \end{bmatrix} \right) \cdot \tilde{\mathbf{z}}_0 \right) + \left(\left(\begin{bmatrix} a \\ 1 \end{bmatrix} \otimes \begin{bmatrix} a \\ 1 \end{bmatrix} \right) \cdot \tilde{\mathbf{z}}_1 \right) \\ &= \mathbf{z}_0 + \mathbf{z}_1 \end{aligned} \tag{3.9}$$

3.2.1 Compressing the Noise

Note that Equation 3.8 includes '1' in the tensor product of the public LPN polynomial a. The reason for this is that in this notion, e_0 and e_1 technically serve as the LPN secret, with f_0 and f_1 being the noise. In practice, the LPN secret and the noise can be combined into a single polynomial, fixing a position of a to 1. In `Ring-LPN`, this allows $\mathbf{z}_\sigma$ to be expressed as a single polynomial z_σ when instead of computing the multiplication with vector $(a^2, a, a, 1)$, we compute the inner product. This idea becomes more clear when looking at the `Module-LPN` assumption and the R^c-LPN$_\tau$ security game (Figure 2.1), which represents a generalization of `Ring-LPN` (cf. Section 2.1.2). Given c randomly sampled ring elements $\tilde{e}_0, \tilde{e}_1, ...\tilde{e}_{c-1}$ and $\tilde{a}_0, \tilde{a}_1, ...\tilde{a}_{c-1}$ for $\tilde{a}_{c-1} = 1$ the challenger computes the inner product $\tilde{u}_1$ as

$$\tilde{u}_1 \leftarrow \sum_{i=0}^{c-1} \tilde{a}_i \cdot \tilde{e}_i = \sum_{i=0}^{c-2} \tilde{a}_i \cdot \tilde{e}_i + \tilde{e}_{c-1} \tag{3.10}$$

while $\tilde{u}_0$ is sampled uniformly at random. Any distinguisher, given access to $\tilde{a}_0, \tilde{a}_1, ..., \tilde{a}_{c-1}$, is considered to win if it can distinguish between $\tilde{u}_0$ and $\tilde{u}_1$ with a non-negligible probability. The challenger wins if the challenger is not able to do so. Through setting $\tilde{a}_{c-1}$ to 1 we implicitly utilize $\tilde{e}_{c-1}$ as the LPN noise.

3.2.2 Splitting the Ring through $F(X)$

To adapt this to a PCG for OLE over $\mathbb{Z}_p^n$, cipher-packing techniques from fully homomorphic encryption schemes can be used [SV14]. If the prime p and the polynomial $F(X)$ are carefully chosen, it is possible to decompose $F(X)$ into n linear factors modulo p to get an isomorphism between R_p and n elements in $\mathbb{Z}_p$. This implies that any OLE over R_p can be split up into n individual OLEs over $\mathbb{Z}_p$. Specifically, Boyle et al. [Boy+20] analyze different choices for $F(X)$ and find that cyclotomic polynomials (cf. Section 2.4.3) are the most appropriate since they allow for fast multiplication in R_p using Fast Fourier Transform (FFT). In order to compute the isomorphism, n unique roots of unity $(\xi_0, ..., \xi_{n-1})$ are needed, such that for each root ξ_l it holds that $F(\xi_l) = 0$. Then, each party σ can evaluate its pair of polynomials (x_σ, z_σ) on ξ_l to get its share of the l-th OLE correlation in $\mathbb{Z}_p$.

Choosing a Cyclotomic Polynomial

An interesting property of N-th cyclotomic polynomials (Equation 2.1) is that when $N = \eta$ for η being a power of two, they take on the following simplified form:

$$\Phi_\eta(X) = X^{\frac{\eta}{2}} + 1 \tag{3.11}$$

This subset of cyclotomic polynomials is particularly well-suited for use in the `Ring-LPN` based PCG approach, as it splits completely into $\eta/2$ linear factors modulo p and is easy to compute for any choice of η. For the construction described above, we require the ring R_p to split into n linear factors modulo p. Therefore, assuming n is a power of 2, $F(X)$ is chosen as $\Phi_\eta(X) = \Phi_{2n}(X) = X^n + 1$.

Computing the Roots of Unity

What remains is to calculate the roots of unity for $\Phi_{2n}(X)$ so that for any $\xi_j \in (\xi_0, \ldots, \xi_{n-1})$ it holds that $\Phi_{2n}(\xi_j) = 0$. Recall that in a finite field $\mathbb{Z}_p$ (where p is prime) all elements $\nu \in \mathbb{Z}_p$ satisfy Fermat's Little Theorem:

$$\nu^{p-1} \equiv 1 \pmod{p} \tag{3.12}$$

Further, let ω be the *primitive root of unity* for $\mathbb{Z}_p$. This means that the order of ω is exactly $(p-1)$ but simultaneously for any positive $k < (p-1)$ we get $\omega^k \neq 1 \pmod{p}$. From this property and the fact that $\omega^{p-1} = \omega^{\frac{p-1}{2}} \cdot \omega^{\frac{p-1}{2}} = 1$ we formulate Equation 3.13.

$$\omega^{(p-1)/2} \equiv -1 \pmod{p}. \tag{3.13}$$

Further, let s be the *smooth order* (the product of distinct prime factors) of $p-1$. We define the following generator g:

$$g \equiv \left(\omega^{\frac{p-1}{s}}\right)^{\frac{s}{2n}} \pmod{p} \tag{3.14}$$

Finally, we use generator g to derive n roots, $\xi_j \in \mathbb{Z}_p$ by iterating over $j \in [n]$. Notably, we require i to be odd such that $i = 2j+1$.

$$\xi_j = g^i = g^{2j+1} \tag{3.15}$$

Proof. We now prove that ξ_j (Equation 3.15) is indeed a root of unity for $\Phi_{2n}(X)$, meaning $\Phi_{2n}(\xi_j) = 0$. We denote the finite field $\mathbb{Z}_p$ for p being prime. Further, let ω be the primitive root of unity of $\mathbb{Z}_p$ and s be the smooth order of $p-1$. We begin by generalizing Fermat's Little Theorem for multiples of $(p-1)$:

$$\omega^{n\cdot(p-1)} = \left(\omega^{(p-1)}\right)^n = (1)^n \equiv 1 \pmod{p}. \tag{3.16}$$

Then, by integrating Equations 3.13, 3.15 and 3.16, we demonstrate that $\Phi_{2n}(\xi_j) = (\xi_j)^n + 1 = 0$:

$$\begin{aligned}
\Phi_{2n}(\xi_j) &= (\xi_j)^n + 1 \\
&= \left(g^i\right)^n + 1 \\
&= \omega^{\frac{p-1}{s}\cdot\frac{s}{2n}\cdot i\cdot n} + 1 \\
&= \omega^{\frac{p-1}{s}\cdot\frac{s}{2}\cdot i} + 1 \\
&= \omega^{\frac{p-1}{2}\cdot i} + 1 \\
&= \omega^{\frac{p-1}{2}\cdot(2j+1)} + 1 \\
&= \omega^{j\cdot(p-1)} \cdot \omega^{(p-1)/2} + 1 \\
&= 1 \cdot -1 + 1 = 0
\end{aligned} \tag{3.17}$$

This concludes our proof, showing that Equation 3.15 generates roots of unity for $\Phi_{2n}(X)$ under the finite field $\mathbb{Z}_p$. □

3.2.3 Complexity Improvements

The `Ring-LPN` based PCG improves upon the quadratic complexity of the naive `Primal-LPN` based approach presented Section 3.1. Specifically, the computation of `FSS.FullEval` is now in $O(n)$ since the domain size is limited to $[0, ..., 2n-1]$. Additionally, polynomial multiplication, which is equivalent to computing the tensor product of two vectors, allows for the utilization of fast multiplication algorithms such as FFT. Hence, when FFT is used, the quadratic cost of a tensor product is reduced to a polynomial multiplication of quasilinear complexity in $O(n \log n)$.

3.3 Construction

In the following, we present the PCG construction by Boyle et al. [Boy+20] that builds on the notion of the `Ring-LPN` based PCG discussed above. For this construction, the `Ring-LPN` assumption is generalized to `Module-LPN`, which we denote more specifically as R^c-LPN$_\tau$ as formalized in Figure 2.1. Analogous to how `Module-LPN` generalizes `Ring-LPN` for $c \geq 2$ ring elements, the PCG now uses c degree-N τ-sparse polynomials instead of a single sparse degree-N polynomials per party. Moving to `Module-LPN` makes this the first practical PCG construction since the security of this assumption

(and therefore the PCG) only depends on the choice of (c, τ) (cf. Section 2.1.2) and not on N. Therefore, the PCG expansion can generate arbitrarily (up to any choice of N) many correlations through splitting the resulting ring elements at N roots of unity (cf. Section 3.2.2).

The main building block of this construction is a Function Secret Sharing (FSS) scheme (cf. Definition 4). The FSS primitive must be able to realize secret sharing of the multiplication of degree-N τ-sparse polynomials. The result of such a multiplication is always a polynomial of degree $2N$ consisting of τ^2 non-zero coefficients. In vector notation, this results in a $2N$-vector with τ^2 non-zero elements. This vector can be interpreted as a multi-point function such that the positions of the non-zero elements in the vector are mapped to their respective value. Therefore, the function has a domain $[2N]$ mapping onto τ^2 non-zero elements. As its FSS scheme, this construction chooses a distributed sum of point functions (DSPFs) for secret sharing the described multi-point functions. Under the derived parameters, the DSPF is denoted as $\mathtt{DSPF}_{2N}^{\tau^2}$. A formal definition for DSPFs is provided in Definition 6. The concrete PCG construction by Boyle et al. [Boy+20] (although adapted to our notation) is presented in the following:

Construction 2: PCG for Oblivious Linear Evaluation (OLE)

Let λ be the security parameter, (c, τ) the parameters of the R^c-LPN$_\tau$ assumption, and p a prime. We denote N as the domain of the PCG. Let $R_p := \mathbb{Z}_p[X]/(F(X))$ be a ring for a degree-N polynomial $F(X) \in \mathbb{Z}_p[X]$ and ($\mathtt{DSPF}_{2N}^{\tau^2}$.Gen, $\mathtt{DSPF}_{2N}^{\tau^2}$.Eval) be a FSS scheme for multi-point functions that map a domain $[2N]$ onto τ^2 non-zero elements. Further, let $\boldsymbol{a} = (1, a_2, ..., a_c)$ for $a_2, ..., a_c \in R_p$ be a public input, chosen uniformly at random.

PCG.Gen$_{\mathtt{OLE}}(1^\lambda)$:

1: For every $r \in [c]$ sample $\boldsymbol{\alpha}_0^r, \boldsymbol{\alpha}_1^r \xleftarrow{\$} [N]^\tau$ and $\boldsymbol{\beta}_0^r, \boldsymbol{\beta}_1^r \xleftarrow{\$} [\mathbb{Z}_p]^\tau$ uniformly at random.
2: For $i, j \in [c]$, sample FSS keys:

$$\left(K_0^{(i,j)}, K_1^{(i,j)}\right) \xleftarrow{\$} \mathtt{DSPF}_{2N}^{\tau^2}.\mathtt{Gen}\left(\mathbb{1}^\lambda, \boldsymbol{\alpha}_0^i \boxplus \boldsymbol{\alpha}_1^j, \boldsymbol{\beta}_0^i \otimes \boldsymbol{\beta}_1^j\right).$$

3: For $\sigma \in \{0, 1\}$ set $k_\sigma = (\{\boldsymbol{\alpha}_\sigma^i, \boldsymbol{\beta}_\sigma^i\}_{i\in[c]}, \{K_\sigma^{i,j}\}_{i,j\in[c]})$.
4: Output (k_0, k_1).

PCG.Expand$_{\mathtt{OLE}}(\sigma, k_\sigma)$:

1: Parse k_σ as $(\{\boldsymbol{\alpha}_\sigma^i, \boldsymbol{\beta}_\sigma^i\}_{i\in[c]}, \{K_\sigma^{i,j}\}_{i,j\in[c]})$.
2: For $i \in [c]$, define the degree $< N$, τ-sparse polynomial:

$$e_\sigma^i(X) = \sum_{k\in[t]} \boldsymbol{\beta}_\sigma^i[k] \cdot X^{\boldsymbol{\alpha}_\sigma^i[k]}$$

and compose all e_σ^i to a single length-c vector $\boldsymbol{e}_\sigma$.
3: For $i, j \in [c]$ compute vector $\boldsymbol{u}_\sigma$ with each element being interpreted as a degree $< 2N$ polynomial:

$$\boldsymbol{u}_\sigma[i + c(j-1)] \leftarrow \mathtt{DSPF}_{2N}^{\tau^2}.\mathtt{FullEval}(\sigma, K_\sigma^{i,j}) \bmod F(X).$$

4: Compute polynomials $x_\sigma = \langle \boldsymbol{a}, \boldsymbol{e}_\sigma \rangle \bmod F(X)$ and $z_\sigma = \langle \boldsymbol{a} \otimes \boldsymbol{a}, \boldsymbol{u}_\sigma \rangle \bmod F(X)$.
5: Output (x_σ, z_σ).

Seed Generation. For each participant $\sigma \in \{0, 1\}$ the construction samples c τ-sparse vectors $\boldsymbol{\alpha}_\sigma$ as exponents and c τ-sparse vectors $\boldsymbol{\beta}_\sigma$ as coefficients representing the participant's respective LPN error vectors. Through sampling the coefficients and exponents separately the polynomial multipli-

cation can be directly secret shared as a point function such that the outer sum of the exponents $\boldsymbol{\alpha}_0^i \boxplus \boldsymbol{\alpha}_1^j$ represent the *special positions* and the tensor product of the coefficients $\boldsymbol{\beta}_0^i \otimes \boldsymbol{\beta}_1^j$ represent the respective *non-zero elements*. Ultimately $\text{PCG.Gen}_{\texttt{OLE}}$, outputs seeds including error polynomials $(\{\boldsymbol{\alpha}_\sigma^i, \boldsymbol{\beta}_\sigma^i)\}_{i\in[c]}$ and the secret sharing of their tensors in form of FSS keys $\{K_\sigma^{i,j}\}_{i,j\in[c]}$ for each party.

Seed Expansion. For expanding the PCG seeds in $\text{PCG.Expand}_{\texttt{OLE}}$ every party inputs its PCG seed and parses the components. The exponent and coefficient vectors are recombined to polynomial representation to compose then a length-c vector $\boldsymbol{e}_\sigma$ of those. Further, each party computes its shares of the tensor products by fully evaluating their part of the DSPF. Recall that the party does not learn anything about the other party's error vectors (and, therefore, the shared multiplication) through this evaluation. The result of each full-domain evaluation is a $2N$ degree polynomial, which is stored as an element of the c^2-length vector $\boldsymbol{u}_\sigma$. Analogue to Equation 3.8, each party computes their polynomial pair (x_σ, y_σ) by computing the inner products $x_\sigma = \langle \boldsymbol{a}, \boldsymbol{e}_\sigma \rangle$ and $z_\sigma = \langle \boldsymbol{a} \otimes \boldsymbol{a}, \boldsymbol{u}_\sigma \rangle$ which results in a single OLE over R_p. As described in Section 3.2 and shown in Equation 3.10, the error vectors implicitly include the LPN secret at the first position which is preserved during the calculation of the inner product by fixing an element of the public input $\boldsymbol{a}$ to 1.

In terms of security, the following theorem is presented, with a proof available in [Boy+20].

Theorem 1. *Suppose the R^c-LPN$_\tau$ assumption holds and `DSPF` is a secure FSS scheme as formalized in Definition 4. Then, Construction 1 is a secure PCG for OLE in R_p to Definition 3.*

Splitting the Ring

Holding the polynomial pairs (x_σ, y_σ), the parties can then exploit the isomorphism between R_p and $\mathbb{Z}_p$ to generate N individual OLE correlations in $\mathbb{Z}_p$ as described in Section 3.2.2. The parties evaluate their pair on the same roots of unity of $F(X)$, such that for any root of unity $\xi_i \in \{\xi_0, ..., \xi_{N-1}\}$

$$x_0(\xi_i) \cdot x_1(\xi_i) = z_0(\xi_i) + z_1(\xi_i) \tag{3.18}$$

is the i-th OLE correlation in $\mathbb{Z}_p$.

Computational Complexity

The computational complexity of this construction depends on the LPN security parameters (c, t) and the chosen domain N. In practice, the LPN parameters are rather small compared to N. For example, Boyle et al. [Boy+20] report $(c, \tau) \in \{(4, 16), (8, 5), (2, 76)\}$ to achieve 128-bit security under the R^c-LPN$_\tau$ assumption. N is expected to be much larger since it is desirable to get as many OLE correlations from a PCG instantiation as possible. Regarding a specific DSPF construction, we choose the tree-based DSPF construction of [BGI16] for which $\texttt{DSPF}_N^\tau\texttt{.Gen}$ is in $O(t)$ and $\texttt{DSPF}_N^\tau\texttt{.FullEval}$ is in $O(N)$. Multiplying degree-N polynomials via FFT is in $O(N \log N)$.

The computational complexity of $\text{PCG.Gen}_{\texttt{OLE}}$ is in $O(c^2\tau^2)$ as the costliest operation is the generation of the DSPF seeds. For the outer sum of $\boldsymbol{\alpha}$, we require τ^2 additions, and for the tensor product of $\boldsymbol{\beta}$, we require τ^2 multiplications. $\texttt{DSPF}_{2N}^{\tau^2}\texttt{.Gen}$ is in $O(\tau^2)$. As this is computed, c^2 times the overall complexity comes to $O(c^2\tau^2)$. Although we have quadratic complexity here, this is not a bottleneck in practice as (c, τ) are small.

The computational complexity of $\text{PCG.Eval}_{\texttt{OLE}}$ is in $O(N \log N \cdot c^2)$ for which the computation of z_σ represents the main computational burden. Mainly, the use of FFT allows the complexity to stay quasilinear for N. Using naive multiplication here would result in quadratic complexity for N, making the scheme undesirable for expanding large amounts of OLE correlations. The complexity can be derived from

- step 3, which is in $O(Nc^2)$, as we call $\texttt{DSPF}_{2N}^{\tau^2}\texttt{.FullEval}$ c^2 times.

- step 4, which is in $O(N \log N \cdot c^2)$, as we compute inner products for
  - x_σ which requires $c-1$ multiplications of degree-N polynomials.
  - z_σ which requires c^2-1 multiplications of dense degree-N polynomials via FFT.

Notice that using FFT to compute x_σ is not beneficial for performance. The reason is that $\boldsymbol{e}$ contains τ-sparse polynomials. Therefore, naive multiplication with degree-N polynomials in $\boldsymbol{a}$ requires only $\tau \cdot N$ operations, which is in $O(N)$ instead of $O(N \log N)$ when using FFT for this case. We analyze this effect more thoroughly in Section 5.1.2.

3.3.1 Adaption to VOLE

We present a PCG construction for Vector-OLE (VOLE) correlations in the following. This construction is derived from slightly adapting the PCG for OLE (Construction 2) described above. Recall that VOLE is an special case of OLE for N-dimensional vectors $(\mathbf{x}_0, \mathbf{z}_0, \mathbf{z}_1)$ and constant v such that $\mathbf{x}_0 \cdot v = \mathbf{z}_0 + \mathbf{z}_1$ holds $i \in [n]$ OLE tuples $\{(\mathbf{x}_0[i], \mathbf{z}_0[i]), (v, \mathbf{z}_1[i])\}$.

The intuition for this PCG is as follows: where we had to secret share the multiplication of two polynomials before, we now need to secret share the multiplication of a polynomial by a constant. As this does not change the degree of the polynomial, the result of the operation under a degree-N τ-sparse polynomial can be represented by a multi-point function of domain N mapping onto τ non-zero elements. Therefore, we can realize a secret sharing using $\mathtt{DSPF}^\tau_N$. In the following, we formalize the adapted construction.

Construction 3: PCG for Vector Oblivious Linear Evaluation (VOLE)

Let λ be the security parameter, (c, τ) the parameters of the R^c-LPN$_\tau$ assumption, and p a prime. We denote N as the domain of the PCG. Let $R_p := \mathbb{Z}_p[X]/(F(X))$ be a ring for a degree-N polynomial $F(X) \in \mathbb{Z}_p[X]$ and ($\mathtt{DSPF}^\tau_N$.Gen, $\mathtt{DSPF}^\tau_N$.Eval) be a FSS scheme for multi-point functions that map a domain $[N]$ onto τ non-zero elements. Further, let $\boldsymbol{a} = (1, a_2, ..., a_c)$ for $a_2, ..., a_c \in R_p$ be a public input, chosen uniformly at random.

PCG.Gen$_{\mathtt{VOLE}}(1^\lambda, v)$:

1: For every $r \in [c]$ sample $\boldsymbol{\alpha}^r \xleftarrow{\$} [N]^\tau$ and $\boldsymbol{\beta}^r \xleftarrow{\$} [\mathbb{Z}_p]^\tau$ uniformly at random.
2: For $r \in [c]$, sample FSS keys:

$$(K^r_0, K^r_1) \xleftarrow{\$} \mathtt{DSPF}^\tau_N.\mathtt{Gen}\left(\mathbb{1}^\lambda, \boldsymbol{\alpha}^r, v \cdot \boldsymbol{\beta}^r\right).$$

3: Set $k_0 = (v, \{K^r_0\}_{r\in[c]})$ and $k_1 = (\{\boldsymbol{\alpha}^r, \boldsymbol{\beta}^r\}_{r\in[c]}, \{K^r_1\}_{r\in[c]})$.
4: Output (k_0, k_1).

PCG.Expand$_{\mathtt{VOLE}}(\sigma, k_\sigma)$:

1: If $\sigma = 0$, parse k_0 as $(v, \{K^r_0\}_{r\in[c]})$
2: If $\sigma = 1$, parse k_1 as $(\{\boldsymbol{\alpha}^r, \boldsymbol{\beta}^r\}_{r\in[c]}, \{K^r_\sigma\}_{r\in[c]})$ and define for $r \in [c]$ the degree $< N$, τ-sparse polynomials

$$e^r(X) = \sum_{k\in[t]} \boldsymbol{\beta}^r[k] \cdot X^{\boldsymbol{\alpha}^r[k]}$$

to compose all e^r to a single length-c vector $\boldsymbol{e}$.
3: For each $i \in [c]$ compute vector $\boldsymbol{u}_\sigma$ with each element being a degree $< 2N$ polynomial:

$$\boldsymbol{u}_\sigma[i] \leftarrow \mathtt{DSPF}^\tau_N.\mathtt{FullEval}(\sigma, K^i_\sigma) \bmod F(X).$$

4: Compute $z_\sigma = \langle \boldsymbol{a}, \boldsymbol{u}_\sigma \rangle \bmod F(X)$.
5: If $\sigma = 0$ compute $x_0 = \langle \boldsymbol{a}, \boldsymbol{e} \rangle \bmod F(X)$ and output (x_0, z_0).
6: If $\sigma = 1$ output (v, z_1).

Seed Generation. For PCG.Gen$_{\texttt{VOLE}}$, we now only sample c τ-sparse LPN error polynomials for one party. The other party is assigned the constant term v. Instead of calculating the polynomial components' outer sum and tensor product, multiplying the coefficients (or non-zero elements) is sufficient to define the resulting point function. The seeds are generated accordingly, containing their respective shares of the point function $\{K_\sigma^r\}_{r\in[c]}$. For party $\sigma = 0$, the LPN error polynomial is included, while for party $\sigma = 1$, the constant is included, such that $k_0 = (v, \{K_0^r\}_{r\in[c]})$ and $k_1 = (\{\boldsymbol{\alpha}^r, \boldsymbol{\beta}^r\}_{r\in[c]}, \{K_1^r\}_{r\in[c]})$.

Seed Expansion. In PCG.Expand$_{\texttt{VOLE}}$, the parties evaluate their respective share $\boldsymbol{u}_\sigma$ of the multiplication by performing a full-domain evaluation of the multi-point function and computing the inner product of $\boldsymbol{u}_\sigma$ and the public input $\boldsymbol{a}$ results in z_σ. Party $\sigma = 0$ must do the same for its LPN error vectors to learn x_0. The functionality returns (x_0, z_0) to party $\sigma = 0$ and (v, z_0) to party $\sigma = 1$. Similar to the construction for PCG for OLE, this yields a single VOLE over R_p.

In terms of security, the following derives from Theorem 1:

Theorem 2. *Suppose the R^c-LPN$_\tau$ assumption holds and* `DSPF` *is a secure FSS scheme as formalized in Definition 4. Then, Construction 2 is a secure PCG for VOLE in R_p.*

Splitting the Ring

Given a single shared VOLE correlation over R_p through (x_0, z_0) and (v, z_0), the parties can again leverage the isomorphism between R_p and $\mathbb{Z}_p$ to split the ring into n individual VOLE correlations in $\mathbb{Z}_p$, similar to the approach outlined in Section 3.2.2. The ring is split by evaluating the tuples on the roots of unity of $F(X)$. For any root of unity $\xi_i \in \{\xi_0, ..., \xi_{N-1}\}$ of $F(X)$

$$x_0(\xi_i) \cdot v = z_0(\xi_i) + z_1(\xi_i) \tag{3.19}$$

is the i-th VOLE correlation in $\mathbb{Z}_p$.

Reduced Computational Complexity

Compared to the complexity of the PCG for OLE, the VOLE construction offers a significant reduction in complexity: PCG.Gen$_{\texttt{VOLE}}$ is now in $O(ct)$, which is a clear improvement over the previous quadratic complexity. Further, the expansion PCG.Expand$_{\texttt{VOLE}}$ is reduced to $O(N)$, enabled by the efficient use of naive multiplication (over FFT) for z_σ. Although counter-intuitive, this is possible because the dense, degree-N polynomials in $\boldsymbol{a}$ are multiplied by τ-sparse polynomials in $\boldsymbol{e}$, requiring only $\tau \cdot N$ integer multiplications. Notice that the DSPF now operates with only τ special positions and a smaller domain N. While this doesn't change computational complexity, it has a noticeable impact on the construction runtime. We explore this in more detail in Section 5.2.2.

4 Implementation

In this chapter, we describe the implementation of the PCG constructions introduced in Section 3.3. We outline our practical considerations regarding the necessary building blocks and show optimizations for implementing these primitives in the context of the PCG construction. We also analyze the complexity of each building block. The implementation of all the building blocks, as well as the PCG for OLE/VOLE correlations, is available on GitHub[1] and is provided in *golang* using multithreading via *go routines*. Besides the single PCG implementation, the repository also contains a PCG for BBS+, which we construct and evaluate in Chapter 6. Our code, including the benchmarks and tests, comprises 5467 lines. We compiled our code using *go 1.21.3*. To perform operations within the finite field $\mathbb{Z}_p$, we use a high-speed BLS12-381 library[2], which implies setting p as the group order of this curve. For benchmarks and a detailed evaluation of our implementation, we refer the reader to Chapter 5.

4.1 Distributed Sum of Point Functions

Let's revisit point functions from Section 2.4.1. A point function $f_{a,b}$ focuses on a single position a within its domain $[N]$. It assigns a non-zero value b to that special position a and 0 to all other positions in the domain. Distributed Point Functions (DPFs) enable additively secret sharing this type of function. Our PCG construction requires an extension of DPFs that support multiple special positions. We call this a Distributed Sum of Point Functions (DSPF – see Definition 6). A DSPF realizes a secret sharing of an injective function that maps τ special positions to τ non-zero values. In practice, we construct a DSPF by using τ individual DPFs. This abstraction makes implementation straightforward and allows for efficient multithreading.

4.1.1 Tree-Based Distributed Point Function

We adopt a tree-based approach for realizing DPFs, originally introduced by Boyle et al. in [BGI15] and subsequently refined in [BGI16]. At its core, the seed generation algorithm, $\texttt{DPF.Gen}(1^\lambda, a, b)$ outputs two seeds (s_0, s_1), one for each party. Each seed encodes a binary tree of depth n, encompassing a domain of size $N = 2^n$ for the point function. The binary representation of the input of the point function determines the path that is taken along the binary tree, resulting in N leaves, one for each possible input. The binary trees are identical except along the path determined by the binary representation of the special point a. Construction of these trees involves `DPF.Gen` selecting a random root value for each party. Nodes are computed deterministically using a pseudo-random generator (PRG) seeded with the value of the parent node. Along the path corresponding to a, correction words are strategically applied based on the i-th bit of a. These correction words guarantee that:

- Nodes outside the path designated by a remain identical in both trees.
- Nodes along the path diverge between the trees.
- The leaf node to which a points holds a value that when added to its counterpart in the other tree, yields the non-zero output b.

The correction terms form part of the seed, resulting in a seed of size $n(\lambda + 2)$ bits each. To evaluate the function, participants call $\texttt{DPF.Eval}(\sigma, s_\sigma, x) \rightarrow y$ to navigate their tree based on the binary

[1] `https://github.com/leandro-ro/Threshold-BBS-Plus-PCG`
[2] `https://github.com/kilic/bls12-381`

representation of an input x. Using the PRG and integrating correction terms where necessary, each participant calculates its share. When $x = a$, the participant acquires a leaf node containing an additive share of the non-zero value b. If $x \neq a$, the cumulative additive shares amount to zero, disclosing no information about the non-zero element other than the fact that x is not the special position of the underlying point function.

4.1.2 Constructing DSPFs

DSPFs can be constructed directly from tree-based DPFs. In the seed generation, $\texttt{DSPF.Gen}(1^\lambda, \boldsymbol{\alpha}, \boldsymbol{\beta})$ invokes the DPF seed generator $\texttt{DPF.Gen}(1^\lambda, a, b)$ for each position $a \in \boldsymbol{\alpha}$ and corresponding non-zero element $b \in \boldsymbol{\beta}$. The resulting DPF seeds form the DSPF seeds. For evaluation, $\texttt{DSPF.Eval}(\sigma, \boldsymbol{s}_\sigma, x)$ performs $\texttt{DPF.Eval}(\sigma, s_\sigma, x)$ on each seed $s_\sigma \in \boldsymbol{s}_\sigma$, yielding a result vector $\boldsymbol{r}_\sigma$ containing all DPF evaluations. Combining both parties' results gives $\boldsymbol{r} = \boldsymbol{r}_0 + \boldsymbol{r}_1$. If $x \in \boldsymbol{\alpha}$, the vector $\boldsymbol{r}$ holds one non-zero element b, otherwise $\boldsymbol{r}$ only contains zeros.

We can analogously construct $\texttt{DSPF.FullEval}(\sigma, \boldsymbol{s}_\sigma)$ by invoking $\texttt{DPF.FullEval}(\sigma, s_\sigma)$ on each DPF seed $s_\sigma \in \boldsymbol{s}_\sigma$. This yields a dense $(t \times N)$-matrix $\boldsymbol{R}_\sigma$, where each row represents the full domain evaluation of the corresponding underlying DPF. Combining results from both parties produces a sparse matrix $\boldsymbol{R}$ with τ non-zero elements (one per row). Multiplying $\boldsymbol{R}$ by an N-vector of ones gives us the full domain evaluation, i.e., all non-zero elements $\boldsymbol{\beta}$.

$$\boldsymbol{R} = \boldsymbol{R}_0 + \boldsymbol{R}_1 = \begin{bmatrix} r_{11} & \cdots & r_{1N} \\ \vdots & \ddots & \vdots \\ r_{t1} & \cdots & r_{tN} \end{bmatrix}, \quad \boldsymbol{R} \cdot \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} = \boldsymbol{\beta} \tag{4.1}$$

Optimizing Space-Complexity

Implementing the DSPF described above naively poses a space complexity bottleneck for large domains. Consider $N = 2^{20}$, $t = 16$, and 128-bit elements: each party's matrix $\boldsymbol{R}_\sigma$ requires 268MB of memory. This becomes impractical for PCG constructions with numerous DSPF instances. Temporary moving to storage is also infeasible due to latency. A more efficient approach compresses elements on creation, such that we omit storing the full matrix $\boldsymbol{R}_\sigma$. Instead, the elements within each row can be summed directly on generation (through $\texttt{DSPF.FullEval}$). Due to the distributivity of vector/matrix operations, summing these vectors still yields $\boldsymbol{\beta}$ while allowing the parties to store a much smaller $N * 128$ bit (for $N = 2^{20}$ now 16.8 MB) vector.

$$\boldsymbol{R} \cdot \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} = (\boldsymbol{R}_0 + \boldsymbol{R}_1) \cdot \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} = \boldsymbol{R}_0 \cdot \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} + \boldsymbol{R}_1 \cdot \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} = \boldsymbol{\beta} \tag{4.2}$$

Parallelization

The DSPF construction is suitable for parallelization by executing each underlying DPF in a separate thread. It's important to note that parallelizing $\texttt{DSPF.Gen}$ and $\texttt{DSPF.Eval}$ likely introduces overhead that outweighs any gains due to their relatively short execution times. However, the computational complexity of the full domain evaluation $\texttt{DSPF.FullEval}$ justifies its parallelization across separate threads. Our implementation employs a worker pool to achieve optimal performance regardless of the available threads. A worker pool is a software design pattern where a fixed number of threads ("workers") are created to handle tasks from a queue. This optimizes resource usage and avoids the overhead of constantly creating and destroying threads. If $\texttt{DSPF.FullEval}$ is called the underlying $\texttt{DPF.FullEval}$ tasks are placed in the queue and concurrently processed by the workers. We evaluate this approach in Section 5.1.1.

Efficient Full Domain Evaluation

In PCG constructions for OLE and VOLE, the `PCG.Expand` functionality (Construction 2 and 3) necessitates multiple full domain evaluations of DSPFs in step 3. Recall that a full domain evaluation in VOLE entails evaluating the DSPF for all positions $x \in [N]$ for its domain N (or $2N$ in OLE), which requires τ full domain evaluations of the underlying DPFs. Hence, optimizing this operation has a noticeable impact on the overall runtime.

A naive implementation of the tree-based DPF would involve $2^n \cdot n$ PRG evaluations for n being the depth of the binary tree and domain $N = 2^n$, leading to undesirable exponential complexity. However, the tree structure allows for an efficient optimization strategy by caching previously computed nodes. This approach reduces the complexity of a full domain evaluation to only $2^{n+1} - 1$ PRG invocations, achieving a favorable complexity of $O(N)$.

4.2 Polynomial Operations

Efficient polynomial manipulation is important for optimizing the performance of our constructions. Scaling to large domains (N) is particularly interesting, as it increases the number of OLE/VOLE correlations, which speeds up setup cost amortization. In practice, N can reach millions, requiring our implementation to handle polynomials of extremely high degrees. Two key operations must be optimized: *polynomial multiplication*, used extensively in steps 4 and 5 of PCG expansion, and *polynomial evaluation*, necessary for extracting OLE/VOLE correlations by splitting the resulting ring elements. We opted for a custom implementation for the following reasons:

- **High-Degree Polynomial Support**: Existing libraries often lack support for the extremely high degrees required by our use case.
- **Finite Field Exponents:** Our implementation necessitates exponents residing within a finite field of a large prime modulus, a feature not commonly available in standard libraries.
- **Data Structure Optimization:** A custom implementation allows us to tailor the underlying data structures for maximum efficiency in our specific application.

4.2.1 Multiplication via FFT

The Fast Fourier Transform (FFT), originally proposed by Cooley et al. in [CT65], is a divide-and-conquer algorithm that allows to efficiently perform multiplications of degree-n polynomials in the complex numbers given the n-th root of unity in that space. The algorithm achieves a computational complexity of $O(n \log n)$, given that n is a power of 2 or, more generally, an integer of smooth order (composed of small prime factors). Pollard extended the FFT's applicability to finite fields in [Pol71], which we adopt for our implementation.

We operate within the finite field defined by the group order of the BLS12-381 curve (a 381-bit prime). To enhance performance, we've precomputed matching roots of unity for various powers of 2, specifically $n = 2^i$ with $i \in \{8, 9, ..., 21\}$. This allows us to efficiently multiply polynomials with a maximum degree of $(n/2) - 1$ each. A crucial aspect of our implementation is strategically selecting the most suitable precomputed root of unity. Once a root of unity is chosen, the FFT's computational complexity of $O(n \log n)$ is fixed. This complexity remains independent of the actual polynomial degree, provided it does not exceed the limit determined by the selected root. This strategic selection optimizes computational efficiency within the defined constraints. We give benchmarks for our implementation in Section 5.1.2.

4.2.2 Evaluation via Horners Method

To extract all OLE/VOLE correlations from the polynomial pair generated by `PCG.Expand` each party must evaluate its pair on all N roots of unity. Imagine we have a dense degree-n polynomial $P(X)$ in standard form:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

To evaluate the polynomial at a specific value x, a naive approach would involve iterating through each term, computing its corresponding power of x, and multiplying it by the coefficient. Summing these products would provide the final result. The number of multiplications required scales quadratically with the number of terms. Utilizing this naive approach for our construction would make this a bottleneck as each party performs $2N$ evaluations of degree-N dense polynomials for extracting N OLE/VOLE correlations. Especially for large Ns, this would become infeasible.

Horner's Method

Horner's method offers a more efficient way to evaluate polynomials [Hor19]. Instead of directly computing each term's power of x, it rewrites the polynomial using a nested structure. This restructuring leverages the fact that many terms share a common factor of x. We start with the highest degree terms and factor out x from these.

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + ... = x(a_n x^{n-1} + a_{n-1} x^{n-2}) + ...$$

We repeat this for all other terms and nest the result to obtain the reformulated polynomial.

$$\begin{aligned} P(x) &= x(x(a_n x^{n-2} + a_{n-1} x^{n-3}) + a_{n-2} x^{n-3}) + ... \\ &= (...((a_n x + a_{n-1})x + a_{n-2})x + \ldots + a_1)x + a_0 \end{aligned}$$

This polynomial representation then allows us to avoid repeated calculations of powers of x. By factoring out x, our implementation re-uses computations rather than recalculating the same powers multiple times. This reduces the number of multiplications from $O(n^2)$ to $O(n)$, which significantly improves performance, especially for the high-degree polynomials for which we want to optimize our implementation. Further, we speed up polynomial evaluation by parallelizing Horner's Method. The polynomial is divided into chunks, each of which is evaluated concurrently by a separate thread. The threads share a precomputed set of powers of x. We present a detailed evaluation of this approach in Section 5.1.2.

4.2.3 Exploiting Sparsity

Notably, in many cases, the polynomials used in our construction are sparse polynomials of high degree. A sparse polynomial contains only a small number of monomials with non-zero coefficients (e.g., $x^n + 7x^4$ is a 2-sparse polynomial of degree n). We exploit this sparsity in multiple ways. First, we represent sparse polynomials using map-like data structures where keys represent exponents and values store the corresponding coefficients, improving space complexity by eliminating the need to allocate space for monomials with zero coefficients. Second, the map representation enables faster execution of several polynomial operations, especially multiplication. While the Fast Fourier Transform (FFT) offers theoretically optimal complexity for polynomial multiplication in $O(n \log n)$, this advantage depends on the density of the polynomials. Our implementation chooses between naive multiplication and FFT for sparse polynomials, dynamically applying naive multiplication over our map representation if $n \log n > t_0 \cdot t_1$ (where t_0 and t_1 are the number of non-zero terms in each polynomial) and using FFT otherwise. This tailored approach optimizes our implementation for the specific characteristics of the polynomials we encounter.

4.3 Considerations for F(X) in BLS12-381

As described in Section 3.2.2, the *roots of unity* of the cyclotomic polynomial $F(X)$ play a critical role within the PCG construction for splitting the single (V)OLE correlation in a ring to N separate (V)OLE correlations in a finite field. Our implementation utilizes the finite field $\mathbb{Z}_p$, defined by the group order of the BLS12-381 elliptic curve. This group order p is a 381-bit prime. For efficiently performing arithmetic operations within this field, we utilize a high-speed BLS12-381 library[3].

Smooth Order and Primitive Root of Unity

The *smooth order* of the BLS12-381 group order p is the product of distinct prime factors of $p-1$ under a certain threshold. While we could employ factorization algorithms, such as *Pollard's p-1 algorithm* [Pol74], to calculate the smooth order in code, we opt for hard coding the factors to improve performance within our BLS12-381-specific implementation. We report the factorization in Figure 4.1.

$$p - 1 = 2^{32} \cdot 3 \cdot 11 \cdot 19 \cdot 10177 \cdot 125527 \cdot 859267 \cdot 906349^2 \cdot 2508409 \cdot 2529403 \cdot 52437899 \cdot 254760293^2$$

Figure 4.1: Factorization of $p-1$ for BLS12-381 group order p

Furthermore, we find the *primitive root of unity* for BLS12-381 to be $\omega = 7$. This parameter fulfills Fermat's Little Theorem (Equation 3.12) for group order p.

Generating Roots Of Unity

We implement Equation 3.15 with the abovementioned parameters. Notice that when all parameters are (pre-)computed such that the multiplicative group generator g (Equation 3.14) is available, the generation of all N roots of unity is as simple as iterating with $j \in [2N]$ over g^i for $i = 2j + 1$. Although low in complexity, performing this iteration for large N might lead to high runtimes when computing all roots of unity at once. We can optimize this using Horner's idea for efficient polynomial evaluation (cf. Section 4.2.2), which caches the exponentiation of x in order to calculate the next higher exponentiation(s) efficiently. This idea can similarly be applied here, such that g^{i-2} is re-used to compute the next root $g^i = g^{i-2} \cdot g^2$ through only one additional multiplication. We evaluate this optimization for large N in Section 5.1.3.

Further Practical Considerations

Ultimately, we make the following practical observations: Once roots are calculated for a specific p and N, they can be stored and re-used. Therefore, it is sufficient to calculate the corresponding roots of unity once for multiple (V)OLE PCG instances with the same domain N that are employed in parallel. This also applies to the sequential case. Although our implementation does not cover this, parties could alternatively compute a specific i-th root on demand if storage complexity is a significant concern. To optimize this approach further, parties could store only the current computed root of unity (g^i) and generate the next root (g^{i+2}) on-demand by only performing one multiplication $g^i \cdot g^2$. This strategy limits storage complexity to one element in $\mathbb{Z}_p$ (381 bit in case of BLS12-381) while minimizing computational effort.

[3] `https://github.com/kilic/bls12-381`

5 Evaluation

This chapter assesses the performance of our implementation, with initial benchmarks for the core building blocks, followed by benchmarks for the full OLE and VOLE PCG construction. The optimum for each building block is to keep the runtime linear for the PCG domain N. Recall that N determines the number of individual (V)OLE correlations expandable from a single PCG instantiation. Linear scaling across building blocks is appealing to offset setup costs and reduce the amortized runtime per correlation as more (V)OLEs are generated. We demonstrate that most building blocks successfully achieve linearity through careful optimization. However, using FFT for high-degree polynomial multiplication introduces a quasilinear element within the overall PCG construction.

Setup. Our benchmark environment utilizes a Xeon Gold 5120 CPU @ 2.20GHz with 14 cores (multi-threading disabled) and 64GB of RAM. Parallelization is handled using *go routines* and a worker pool to optimize thread utilization. All benchmarks are repeated ten times to account for statistical irregularities. Benchmarking is facilitated by the *go* built-in testing package.

5.1 Building Blocks

We start by examining the individual building blocks and providing insight into the performance characteristics of each component, allowing us to identify potential bottlenecks and additional areas for optimization for the final PCG construction. Therefore, all test parameters are chosen based on the LPN security parameters $(c, \tau) = (4, 16)$ and the domain N used for the final construction. We also benchmark naive implementations for every component to quantify the practical considerations proposed in Chapter 4.

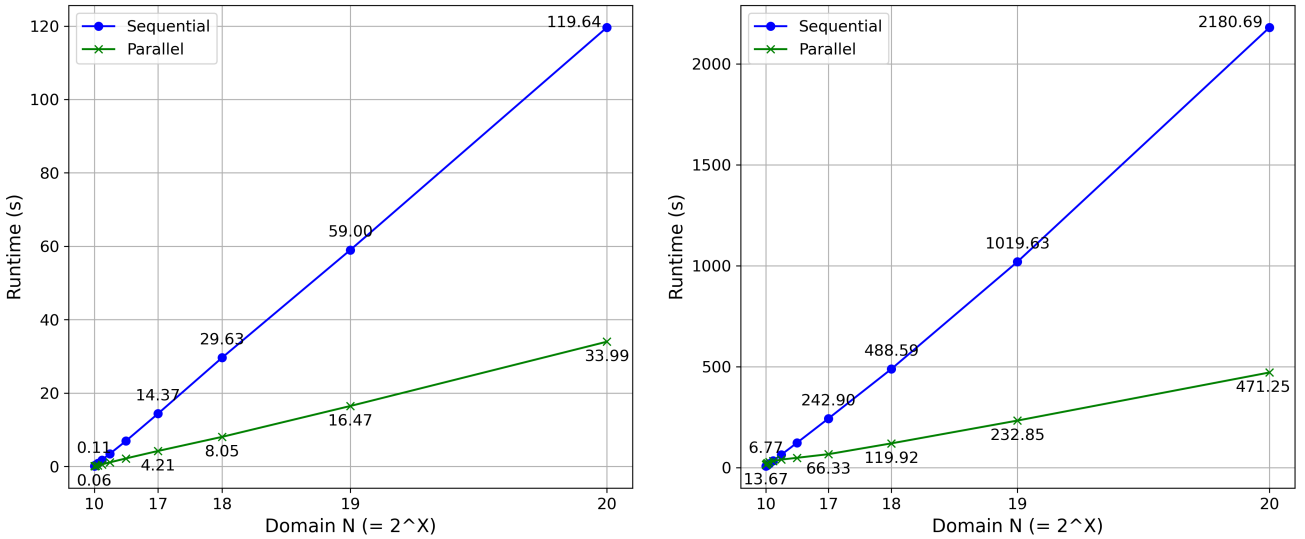
5.1.1 DSPF Full Domain Evaluation

We evaluate the performance of the DSPF full domain evaluation based on its differential usage in the OLE and VOLE construction. Hence, our benchmarks evaluate $\texttt{DSPF.FullEval}_N^l$ for $l \in \{16, 256\}$ and $N \in \{2^{10}, ..., 2^{20}\}$, reflecting the larger DSPF instanciation in the OLE case ($\texttt{DSPF}_{2N}^{\tau^2}\texttt{.FullEval}$) compared to the VOLE case ($\texttt{DSPF}_N^{\tau}\texttt{.FullEval}$) for $\tau = 16$. Results are presented in Figure 5.1.

Parameter l. The parameter l mainly influences the runtime of `DSPF.FullEval`. Choosing $l = \tau^2 = 256$ increases the runtime by around 16x compared to $l = \tau = 16$. This aligns with how DSPFs are constructed in our implementation, as each DSPF consists of l individual DPFs, each requiring its own full domain evaluation, resulting in a linear increase for larger l.

Parameter N. The domain of the DSPF N also has a significant effect on the runtime. Notice the logarithmic scaling of the x-axis here. The runtime for a full domain evaluation increases linearly for larger N, which is in line with the complexity we stated for the optimized approach for full domain evaluations of tree-based DPFs in Section 4.1.2. This property is favorable for the PCG construction as this building block does not influence the time per generated (V)OLE correlation; that is, it does not penalize choosing large N.

Potential Bottleneck. Notice that the runtime, especially for τ^2, can be rather high. For example, approximately 36 minutes for $N = 2^{20}$. For appropriate LPN parameters, the PCG construction for OLE requires several instances of $\texttt{DSPF}_{2N}^{\tau^2}\texttt{.FullEval}$. This poses the risk of making this primitive a



(a) $l = \tau = 16$

(b) $l = \tau^2 = 256$

Figure 5.1: Comparison of sequential and parallel processing of ($\texttt{DSPF}_N^l\texttt{.FullEval}$) for $\tau = 16$

major bottleneck for the final PCG construction, so optimizations here could significantly affect the final construction runtime.

Parallelization. We achieve substantial performance gains through parallelization. Our implementation utilizes a worker pool with one worker per available thread. We queue individual DPFs within the DSPF for processing by these workers. For $N = 2^{20}$, we observe speedups of 3.5x for $l = 16$ and 4.6x for $l = 256$. The higher speedup for $l = 256$ is expected due to a longer overall runtime, reducing the relative overhead of the parallelization setup. This effect also explains why parallelization is less beneficial for smaller values of N (e.g., $N = 2^{10}$). Furthermore, the advantage of parallelization with $l = 256$ over $l = 16$ increases with larger N. This stems from the suboptimal thread utilization with $l = 16$: from our 14 available threads, 12 are left idle after the initial 14 DPFs are processed. The negative impact becomes more apparent for larger values of N, as individual tasks take longer to complete, resulting in the unused threads idling for longer periods. However, this issue is mitigated for $l = 256$ since more tasks are in the queue, resulting in more efficient hardware utilization and greater speedup.

5.1.2 Operations on Polynomials of High Degree

In the following, we evaluate the runtime of our polynomial arithmetic implementation, specifically optimized for the PCG use case, where support for high-degree polynomial operations is needed. We evaluate the techniques proposed in Section 4.2 for polynomials of up to degree 2^{20}. These include the Fast Fourier Transform (FFT), sparse polynomial multiplication, and Horner's method. The results indicate that the PCG construction benefits significantly from the strategic use of the techniques used. Without these optimizations, the PCG would not be practical within reasonable time constraints.

Dense Multiplication

Dense degree N polynomials possess non-zero coefficients for all terms ranging from the constant to x^N. As detailed in Section 4.2.1, their naive multiplication has a computational complexity of $O(N^2)$. On the contrary, the Fast Fourier Transform (FFT) optimizes this operation to $O(N \log N)$. In Figure 5.2 we present benchmarks for both approaches for $N \in \{2^{10}, ..., 2^{20}\}$. The results confirm the stark difference in scaling: The naive approach demonstrates significantly worse performance compared to the quasilinear runtime achieved by the FFT. The difference between both approaches underlines

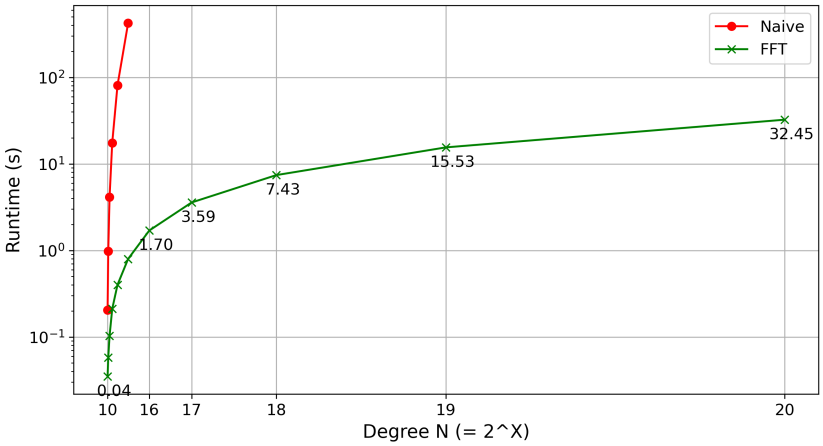
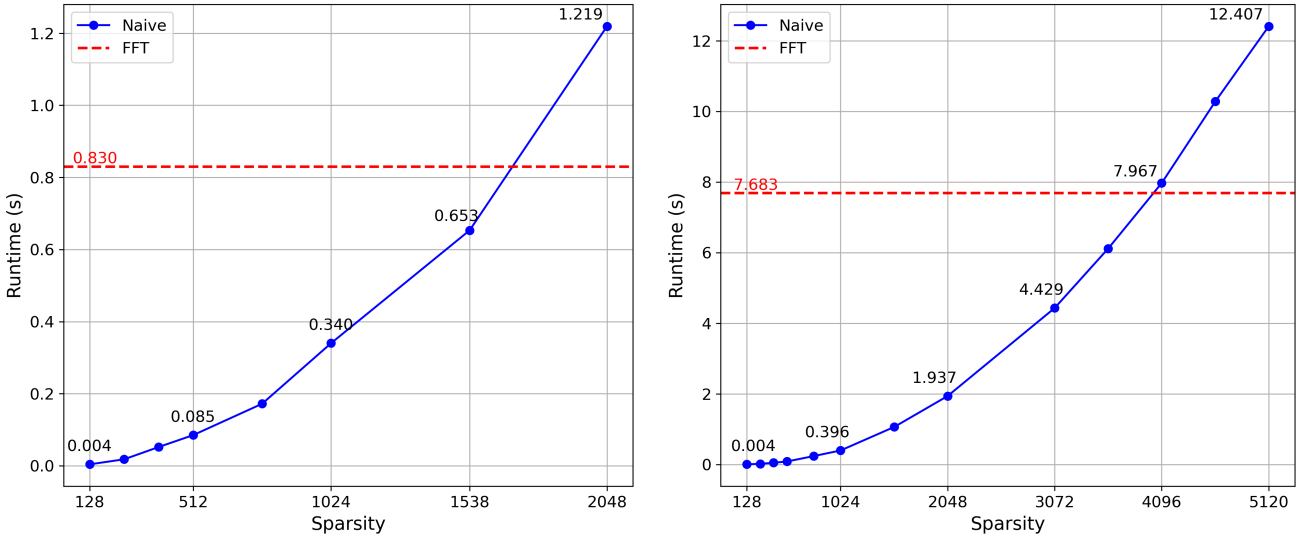


Figure 5.2: Comparing approaches to polynomial multiplication of two dense degree-N polynomials



(a) $N = 2^{15}$ (b) $N = 2^{18}$

Figure 5.3: Comparing approaches to polynomial multiplication of two sparse degree-N polynomials

the importance of using FFT for high-degree polynomial multiplications and, therefore, the PCG implementation.

Sparse Multiplication

Sparse polynomials of degree N possess only a few non-zero coefficients within the terms ranging from the constant to x^N. In Section 4.2.3, we claim that, for certain scenarios, naive multiplication of sparse polynomials could outperform the FFT when zero-coefficient multiplications are efficiently handled. Figure 5.3 validates this claim by benchmarking the multiplication of polynomials with varying sparsity for degrees $N = 2^{15}$ and $N = 2^{18}$. As the complexity of FFT depends solely on the degree of the result, its runtime remains constant for a fixed N. Conversely, the naive approach scales quadratically with sparsity but exhibits significantly lower runtimes for very sparse polynomials. Importantly, the higher the degree N, the wider the sparsity range where the naive approach outperforms FFT. In practice, accounting for this in the PCG implementation poses a significant performance increase as the LPN error polynomials are chosen τ-sparse. When we account for $\tau = 16$ and $N = 2^{18}$, a naive multiplication takes only a few ms instead of multiple seconds using FFT.

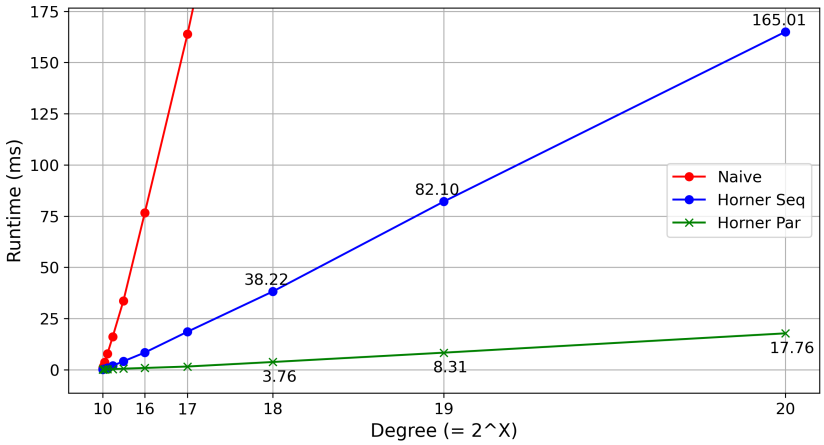


Figure 5.4: Comparing approaches to polynomial evaluation of a dense degree-N polynomial

Polynomial Evaluation

In Section 4.2.2, we outline how Horner's method optimizes polynomial evaluation to achieve linear $O(N)$ complexity, a significant improvement over the naive quadratic approach. To validate this, we present benchmarks for evaluating polynomials of degree $N \in \{2^{11}, ..., 2^{20}\}$ in Figure 5.4. As expected, the naive approach shows worse performance, especially as N increases. Horner's method demonstrates a clear linear runtime increase, enabling the evaluation of a degree $N = 2^{20}$ polynomial at a specific position in approximately 165ms. However, after PCG expansion, N individual position needs to be evaluated on (multiple) polynomials to extract all (V)OLE correlations. This underscores the importance of polynomial evaluation for the PCG.

Parallelization. To address a potential bottleneck, we explore parallelization. Unlike parallelizing multiple evaluations, our strategy directly enhances performance for single-point evaluations – potentially beneficial if storage complexity is a concern and parties split the ring on demand (cf. Section 4.3). Our parallelized implementation of Horner's method achieves a near-10x speedup, demonstrating good hardware utilization. This reduces the evaluation time for a degree $N = 2^{20}$ polynomial to 17.7ms, down from 165ms.

5.1.3 Computing Roots of Unity

In the following, we evaluate our implementation for computing all roots of unity for the $2N$-th cyclotomic polynomial. As outlined in Section 4.3, we implemented two approaches: the naive method directly iterates over Equation 3.15 to generate all roots of unity, while the optimized approach leverages a Horner-inspired technique to reuse exponentiations for improved efficiency. The benchmarks for $N = \{2^{15}, ..., 2^{20}\}$ (Figure 5.5) show that the optimized approach scales linearly. This is beneficial for PCG construction, as it does not negatively affect the time required per (V)OLE correlation for higher PCG domains. Note also that we achieve a significant performance improvement of about 11x over the naive method.

5.2 PCG Construction

In this section, we evaluate the PCG construction presented in Section 3.2.3 for different domains N. We choose the LPN parameters to be $(c, \tau) = (4, 16)$, which achieves 128-bit security, as reported by Boyle et al. [Boy+20].

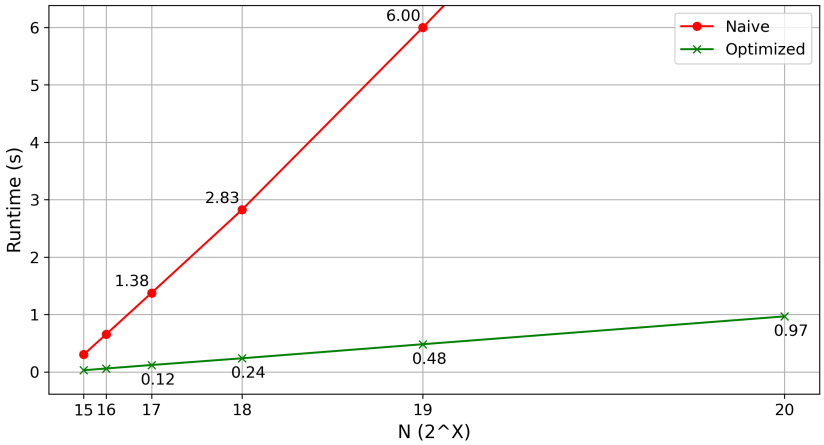


Figure 5.5: Comparing approaches to generate all *roots of unity* for the $2N$-th cyclotomic polynomial

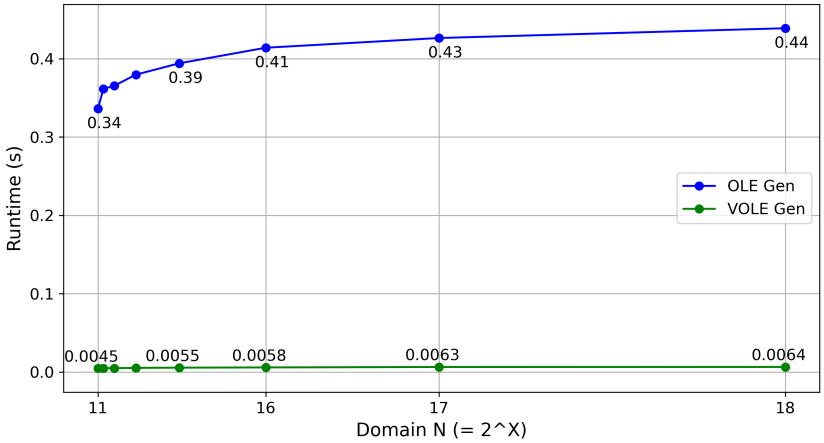


Figure 5.6: Runtime comparison of the PCG seed generation for OLE and VOLE over N

5.2.1 Generation

Figure 5.6 presents the runtime of PCG seed generation, particularly when using a trusted dealer. This contrasts with the higher overhead of distributed generation algorithms, which are not part of this work. It is worth noting that distributed generation alternatives exist, but they introduce significant performance tradeoffs.

The VOLE construction requires four calls to $\texttt{DSPF}_N^{\tau}\texttt{.Gen}$. With $\tau = 16$, this translates into 64 underlying DPF calculations (cf. Section 4.1.2). On the contrary, the OLE construction performs 16 calls to $\texttt{DSPF}_{2N}^{\tau^2}\texttt{.Gen}$, demanding a significantly higher 4096 DPF computations. This 64x increase in DPFs aligns well with the benchmark results. For $N = 2^{18}$, the OLE runtime of 0.44s is approximately 68 times longer than the VOLE's 0.0064s. Interestingly, the domain size N has a relatively minor impact on generation time. This comes from the tree-based approach used within the underlying DPFs. Doubling N leads to a roughly constant increase in operations (proportional to τ or τ^2) due to the fixed value of τ.

5.2.2 Expansion

We present the runtimes of the PCG constructions relative to the amount of (V)OLE correlation they generate in Figure 5.7. This figure also separately visualizes the added cost of extracting correlations (splitting the ring elements) via polynomial evaluation on a root of unity. Observe the quasilinear scaling for both OLE and VOLE constructions with respect to N. As discussed previously, this pri-

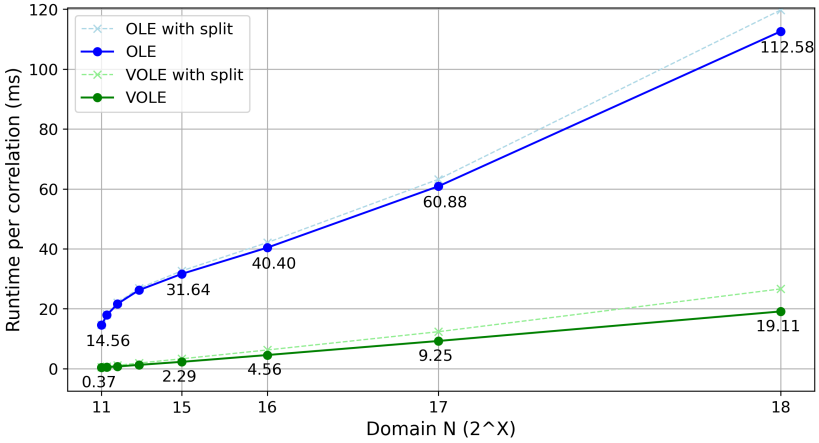


Figure 5.7: Runtime comparison of the PCG expansion for OLE and VOLE over N

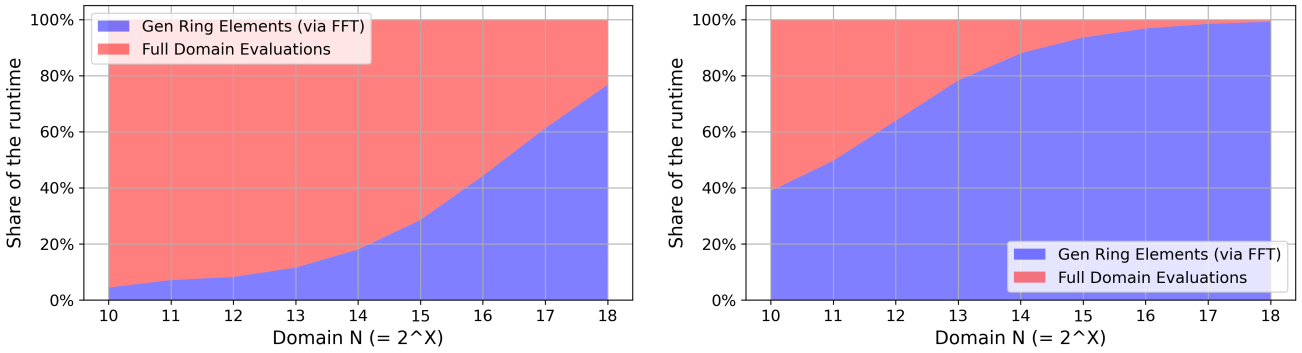
marily stems from the quasilinear nature of polynomial multiplication using FFT despite achieving linearity in other building blocks. It is worth emphasizing that compared to the expansion itself, our optimized computation of roots of unity adds negligible overhead. For example, with $N = 2^{18}$, this step adds only 0.24 ms per correlation. Therefore, this is not part of our evaluation.

We observe a nearly exponential runtime increase in OLE construction for small N, transitioning to the expected quasilinear increase as N grows. We identify this behavior as an effect of the parallelization overhead that dominates in small domains. As N increases, the parallelization setup cost amortizes, revealing the construction's expected quasilinear runtime. The same applies to the VOLE construction, but is less visible due to the figures scaling. Notably, the VOLE construction scales more favorably than the OLE construction, exhibiting a near-linear runtime. Several factors explain this:

- **Reduced Multiplications:** With $c = 4$, the VOLE construction requires only 8 polynomial multiplications compared to OLE's 24. Since multiplications (via FFT) drive the quasilinear runtime, their reduced usage in the VOLE construction yields a runtime closer to the complexity of other primitives.
- **Lower Degrees:** The multiplications of the VOLE construction operate on degree-N polynomials, while a significant portion of OLE constructions multiplications involve degree-$2N$ polynomials. As FFT complexity becomes closer to linear for smaller degrees, this further reduces its overall impact on the VOLE construction.
- **Exploiting Sparsity:** The VOLE construction involves more sparse polynomial multiplications. Here, our optimized approach strategically selects the more efficient naive method, further improving performance.

Splitting the Ring: Note that the (V)OLE constructions do not include the splitting of the ring elements. However, since this is a crucial practical step, we include the overhead it introduces in Figure 5.7 separately. Our parallelized optimizations of Horner's method for polynomial evaluation yield noticeable benefits, resulting in a moderate overhead for the constructions. At $N = 2^{18}$, the increase is 39.35% for VOLE and only 6.34% for OLE. While the absolute runtime for this operation is identical in both cases, the proportional increase is higher for the VOLE construction due to its lower overall runtime. Therefore, we find that further optimizations would be valuable, particularly in reducing the overhead associated with ring splitting in the VOLE case.

Full Domain Evaluation: Processing DSPF full domain evaluations contributes significantly to the overall runtime of the OLE construction, with less of an impact on the VOLE construction. This



(a) OLE construction

(b) VOLE construction

Figure 5.8: Comparing runtime allocation of `PCG.Expand` over N

is analogous to the PCG seed generation described in Section 5.2.1. Observe the following for LPN parameters $(c, \tau) = (4, 16)$:

- OLE Construction: Employs 16x $\texttt{DSPF}_{2N}^{\tau^2}\texttt{.FullEval}$, which implies 4096 DPF full domain evaluation on domain $2N$.
- VOLE Construction: Utilizing 4x $\texttt{DSPF}_{N}^{\tau}\texttt{.FullEval}$, which implies only 64 DPF full domain evaluation on (smaller) domain N.

This disparity, coupled with the higher frequency of polynomial multiplication, provides a clear explanation for the longer runtime and worse scaling of the OLE construction. Although we observe these attributes, we believe that the quasilinear performance of both constructions is practical and present an application in Chapter 6.

Profiling: We suggested that FFT-based polynomial multiplications are the primary runtime bottleneck of the PCG constructions. Although this is indeed the case for large N, we found that the (relative) impact of the FFT varies and applies differently to the OLE and VOLE construction. To quantify this behavior, we profile the runtime distribution between the DSPF full domain evaluations (step 3) and generating ring elements via FFT (step 4/5) of Construction 2 and 3 in Figure 6.4. We observe that for the OLE construction, the quasilinearity of the FFT becomes dominant only for larger N. This is also due to the fact that the runtime required for the (large) DSPF full domain evaluations is quite high to begin with, which is less the case for the smaller DSPF domains in the VOLE case, as described before. Simultaneously, this is the reason why the FFT is more dominant for smaller N in the VOLE construction. Recall from Section 5.1.1 that the advantage of parallelizing the full domain evaluation for the OLE case increases with larger N compared to the VOLE case. We recognize this effect here, as the increase in the dominance of the FFT increases steadily for OLE but flattens out for VOLE. Ultimately, the quasilinear nature of the FFT implies that its relative share of the runtime approaches 100% in both constructions if N is large enough.

6 PCG for Threshold BBS+

In this chapter, we recall and evaluate an application for PCGs in regard to realizing a threshold signature scheme (cf. Definition 1) based on BBS+ (cf. Construction 1). Recent threshold BBS+ schemes [GGI19; Doe+23] necessitate interaction among parties during signing, resulting in communication overhead that leads to latency, especially in scenarios where the signing parties are geographically dispersed. Faust et al. [Fau+23] address this limitation with a threshold BBS+ scheme that strategically divides the signing process into two phases: an interactive *offline* preprocessing phase generating message-independent presignatures and a non-interactive *online* signing phase where presignatures are used without further communication between signers. Besides achieving sublinear communication complexity in the offline phase, the essential advantage of this approach is the trade-off it offers. Shifting computationally demanding operations to the offline phase enables a highly efficient online signing process. This flexibility is beneficial for real-world applications dealing with variable utilization patterns and/or geographically dispersed parties, allowing presignature generation during low-demand periods to enable rapid signature creation during peak system load.

We begin by recalling the threshold BBS+ scheme by Faust et al. [Fau+23]. We explain how the scheme thresholdizes standard BBS+ to achieve applicability of PCGs, such that their expansion enables the offline preprocessing phase. Then, we present a complete construction for the n-out-of-n case based on the PCG for (V)OLE from Section 3.3. Further, we show the necessary adaptions to realize the threshold t-out-of-n case. Finally, we implement both cases, provide benchmarks to evaluate the construction, and derive implications for the efficiency of the online phase.

6.1 Thresholdization

Refer to Construction 1 for the standard BBS+ signature scheme. For simplicity, we focus on the n-out-of-n scenario. Therefore, the key generation (`KeyGen`) can be distributed via additive secret sharing. For an t-out-of-n scenario, Shamir's Secret Sharing [Sha79] (which employs Lagrange interpolation) would be utilized. Notice that for adapting the signing (`Sign`), the main difficulty lies in distributing the computation of Equation 6.1.

$$A := \left(g_1 \cdot h_0^s \cdot \prod_{\ell \in [k]} h_\ell^{m_\ell} \right)^{\frac{1}{x+e}} . \tag{6.1}$$

For realizing the non-interactive online phase, the thresholdization must be able to create message-independent presignatures for A while preventing the disclosure of the secret key (shares of) x during the computation of the inverse of $(x + e)$. This challenge is similar to those faced in other signature schemes that depend on exponentiation, such as ECDSA. Commonly, solving this involves calculating and revealing a value $B = M^a$ and $\delta = a \cdot x$ for a randomly chosen blinding value a. The signature of the massage $M^{1/x}$ can then be computed through B without revealing x, since $M^{1/x} = B^{1/\delta}$. This is suitable for message-independent presignatures, as δ remains independent of M. This idea is applied in the following:

BBS+ presignature tuples. For a n-out-of-n setting, secret key x and random blinding factor a, we define a BBS+ presignature φ_i held by party P_i as a tuple $(a_i, e_i, s_i, \delta_i, \alpha_i) \in \mathbb{Z}_q^5$, such that the following correlations hold

$$\begin{aligned} \delta = \sum_{i\in[n]} \delta_i = a(x+e), \quad \sum_{i\in[n]} \alpha_i = as, \\ a = \sum_{i\in[n]} a_i, \quad e = \sum_{i\in[n]} e_i, \quad s = \sum_{i\in[n]} s_i. \end{aligned} \tag{6.2}$$

Assuming the existence of BBS+ presignature tuples φ_i, we recall the following n-out-of-n TSS:

Construction 4 (n**-out-of-**n **Threshold BBS+**). *Extending from Construction 1, let $P_0, \ldots, P_{n-1}$ denote the parties involved, each holding an array of N independent presignatures $\varphi_i[1], \ldots, \varphi_i[N]$ for party P_i. The n-out-of-n BBS+ TSS includes the following polynomial-time algorithms:*

- `ThreshKeyGen`(1^λ)*: For a security parameter λ, sample a secret $x \xleftarrow{\$} \mathbb{Z}_p^*$, compute $y = g_2^x$, and split x as shares $sk_0, \ldots, sk_{n-1}$ via a secret sharing scheme. Each party i receives (pk, sk_i).*
- `ThreshSig`$_{\varphi_i[j]}(\{m_\ell\}_{\ell\in[k]})$*: Given secret share sk_i and the j-th pre-signature $\varphi_i[j] = (a_i, e_i, s_i, \delta_i, \alpha_i)$, compute $A_i = h_0^{\alpha_i} \cdot \left(g_1 \prod_{\ell\in[k]} h_\ell^{m_\ell}\right)^{a_i}$ and return partial signature $\sigma_i = (A_i, \delta_i, e_i, s_i)$.*
- `CombineSig`$(\{\sigma_i\}_{i\in[n]})$*: Combine all partial parital signatures $\{\sigma_i\}_{i\in[n]}$ by reconstructing δ, e, s additively and computing $A = (\prod_{i\in[n]} A_i)^{\frac{1}{\delta}}$ to output $\delta = (A, e, s)$.*
- `Verify`$_{pk}(\{m_\ell\}_{\ell\in[k]}, \sigma)$*: Parse the signature $\sigma = (A, e, s)$ and public key $pk = y$ to output 1 iff $e(A, y \cdot g_2^e) = e(g_1 \cdot h_0^s \cdot \prod_{\ell\in[k]} h_\ell^{m_\ell}, g_2)$.*

We observe that `CombineSig` produces a tuple (A, e, s) that represents a valid BBS+ signature as shown in Equation 6.3. Notice that the blinding factor a cancels out. Therefore, the original `Verify` functionality of BBS+ outputs 1.

$$A = (\prod_{i\in[n]} A_i)^{\frac{1}{\delta}} = \left(h_0^{as} \cdot (g_1 \cdot \prod_{\ell\in[k]} h_\ell^{m_\ell})^a\right)^{\frac{1}{a(x+e)}} = \left(g_1 \cdot h_0^s \cdot \prod_{\ell\in[k]} h_\ell^{m_\ell}\right)^{\frac{1}{x+e}} \tag{6.3}$$

Intuitively, the scheme fulfills the *unforgability* property of a TSS (cf. Section 2.2) as long as no j-th round of presignatures $\varphi_0[j], \ldots, \varphi_{n-1}[j]$ for $j \in [N]$ is used more than once. On the one hand, a and therefore also α and secret x cannot be derived from any (set of) partial signature σ_i. On the other hand, if each j-th round of presignatures is independent of any other round r for $r \neq j$, an adversary cannot learn anything from observing valid (partial) signatures.

6.2 Offline Preprocessing Phase

We recalled the construction of a non-interactive n-out-of-n BBS+ TSS from [Fau+23], assuming the existence of a message-independent presignature database that satisfies two criteria: (1) each set of presignatures satisfies the specified correlations (cf. Equation 6.2), and (2) each set is computationally indistinguishable from all others so that forgery of unused (partial) presignatures is not possible. These are attributes a PCG (Definition 3) can fulfill, as it can securely produce pseudorandom distributions that are correlated in some pre-defined way. Criteria (1) follows from the PCGs *correctness* property, and criteria (2) from the PCGs *security* property.

Correlations. Notice that the correlations of BBS+ presignature tuples can be decomposed into two OLE and one VOLE correlation. Specifically, the correlation $\alpha = as$ can be realized as an OLE correlation by assuming that a and s are already additively shared among the parties. Each party i, for $i, j \in [n]$, then receives shares of the cross-terms $a_i s_j$ (for $i \neq j$) represented as $a_i s_j = z^i_{(i,j)} + z^j_{(i,j)}$. Here, $z^i_{(i,j)}$ denotes the share held by party i and $z^j_{(i,j)}$ denotes the share held by party j. Note that party i can compute the term $a_i s_i$ locally. Finally, each party i can construct its additive share of $\alpha = \sum_{i\in[n]} \alpha_i$ as in Equation 6.4:

$$\alpha_i = a_i s_i + \sum_{j \in [n] \setminus \{i\}} z^i_{(i,j)} + \sum_{j \in [n] \setminus \{i\}} z^i_{(j,i)}. \tag{6.4}$$

This works for the other correlations as well. Notice that $\delta = a(x + e)$ can be split into an OLE correlation $\delta^0 = ae$ and a VOLE correlation $\delta^1 = ax$, where the secret x is invariant, allowing us to express δ as the sum $\delta = \delta^0 + \delta^1$. Since there are no other requirements, we can utilize the PCG constructions for (V)OLE correlations introduced in Section 3.3 to construct a PCG for BBS+ presignature tuples.

6.2.1 PCG Construction for BBS+

The basic idea behind combining the (V)OLE constructions of Section 3.3 for connecting multiple correlations is that we let the individual PCG instantiations share LPN error vectors strategically. Simultaneously, the LPN error vectors themselves can be utilized as additive shares of a, e, and s. We present the PCG construction for BBS+ presignature tuples (Equation 6.2) in Construction 5. Our construction is derived from the PCG proposed by Abram et al. [Abr+22] for their non-interactive threshold ECDSA scheme.

Seed Generation. For $\text{PCG.Gen}_{\texttt{BBS+}}$, we begin with sampling additive secret key shares sk_i. We also sample LPN error vectors for every party. Each set of error vectors represents a seed for an additive share of the targeted base parameter $a \sim (\boldsymbol{\omega}, \boldsymbol{\beta})$, $e \sim (\boldsymbol{\eta}, \boldsymbol{\gamma})$, $s \sim (\boldsymbol{\phi}, \boldsymbol{\epsilon})$. In step 3, we initiate the VOLE PCG, for which sk_i serves as the constant parameter. Notice that we multiply the j-th parties secret share sk_j with $\boldsymbol{\beta}_i$ as we realize a secret share for all cross-term $a_i \cdot sk_j$. For $a_i \cdot sk_i$, a distribution is unnecessary as party i holds all necessary parts. In step 4, we initiate the PCG for OLE for all cross-terms. Notice here how the LPN error vectors of a, $(\boldsymbol{\omega}, \boldsymbol{\beta})$ are used for both initiations. This embeds to the relation of the correlations ($\delta_0 = a \cdot e$ and $\alpha = a \cdot s$) the individual PCGs can be expanded for. Ultimately, the seeds that include the party's respective secret key share, the LPN error vectors, and the DSPF keys are returned.

Seed Expansion. For $\text{PCG.Expand}_{\texttt{BBS+}}$, we begin by reconstructing the LPN error polynomials from the seed. In step 2, each party computes its share of the VOLE correlation by adding its known adjacent term with all shares of the cross-terms. Notice that both directions of the cross-terms need to be evaluated from the DSPF. Similarly, in step 3, the OLE correlations are being evaluated. All parameters are then interpreted as vectors of polynomials. Analogously to the PCG construction for (V)OLE, computing the inner product of those polynomials with public parameter a then yields ring elements in R_p that adhere to the specified correlations. In order to extract individual BBS+ pairs in $\mathbb{Z}_p$, the ring elements need to be split via evaluation on a common root of unity as described in Section 3.2.2.

Construction 5: PCG for BBS+ Presignature Tuples

Let λ be the security parameter, (t, c) the parameters of the R^c-LPN_τ assumption, and p the modulus. We denote N as the domain of the PCG. Further, let $R_p := \mathbb{Z}_p[X]/(F(X))$ be a ring for a degree-N polynomial $F(X) \in \mathbb{Z}_p[X]$ and $\boldsymbol{a} = (1, a_2, ..., a_c)$ for $a_2, ..., a_c \in R_p$ be a public input.

$\text{PCG.Gen}_{\texttt{BBS+}}(1^\lambda)$:

1: Sample key shares $\text{sk}_i \xleftarrow{\$} \mathbb{Z}_p$ for every $i \in [n]$.
2: For every $i \in [n], r \in [c]$, sample $\boldsymbol{\omega}^r_i, \boldsymbol{\eta}^r_i, \boldsymbol{\phi}^r_i \xleftarrow{\$} [N]^t$ and $\boldsymbol{\beta}^r_i, \boldsymbol{\gamma}^r_i, \boldsymbol{\epsilon}^r_i \xleftarrow{\$} [\mathbb{Z}_p]^t$ uniformly at random.
3: For every $i, j \in [n]$ with $i \neq j, r \in [c]$, compute

$$\left(U^{r,0}_{i,j}, U^{r,1}_{i,j}\right) \xleftarrow{\$} \texttt{DSPF}^t_N.\texttt{Gen}\left(\mathbb{1}^\lambda, \boldsymbol{\omega}^r_i, \text{sk}_j \cdot \boldsymbol{\beta}^r_i\right).$$

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

$$\left(C_{i,j}^{r,s,0}, C_{i,j}^{r,s,1}\right) \xleftarrow{\$} \mathtt{DSPF}_{2N}^{t^2}.\mathtt{Gen}\left(\mathbb{1}^\lambda, \boldsymbol{\omega}_i^r \boxplus \boldsymbol{\eta}_j^s, \boldsymbol{\beta}_i^r \otimes \boldsymbol{\gamma}_j^s\right),$$
$$\left(V_{i,j}^{r,s,0}, V_{i,j}^{r,s,1}\right) \xleftarrow{\$} \mathtt{DSPF}_{2N}^{t^2}.\mathtt{Gen}\left(\mathbb{1}^\lambda, \boldsymbol{\omega}_i^r \boxplus \boldsymbol{\phi}_j^s, \boldsymbol{\beta}_i^r \otimes \boldsymbol{\epsilon}_j^s\right).$$

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

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

PCG.Expand$_{\mathtt{BBS+}}(\sigma, \kappa_\sigma)$:

1: For every $r \in [c]$, define the degree $< N$, t-sparse LPN polynomials:

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

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

$$\widetilde{u}_i^r \leftarrow \mathrm{sk}_i \cdot u_i^r + \sum_{j\neq i} \left(\mathtt{DSPF}_N^t.\mathtt{FullEval}\left(U_{i,j}^{r,0}\right) + \mathtt{DSPF}_N^t.\mathtt{FullEval}\left(U_{j,i}^{r,1}\right)\right).$$

3: For every $r, s \in [c]$, compute

$$w_i^{r,s} \leftarrow u_i^r \cdot v_i^s + \sum_{j\neq i} \left(\mathtt{DSPF}_{2N}^{t^2}.\mathtt{FullEval}\left(C_{i,j}^{r,s,0}\right) + \mathtt{DSPF}_{2N}^{t^2}.\mathtt{FullEval}\left(C_{j,i}^{r,s,1}\right)\right),$$
$$m_i^{r,s} \leftarrow u_i^r \cdot k_i^s + \sum_{j\neq i} \left(\mathtt{DSPF}_{2N}^{t^2}.\mathtt{FullEval}\left(V_{i,j}^{r,s,0}\right) + \mathtt{DSPF}_{2N}^{t^2}.\mathtt{FullEval}\left(V_{j,i}^{r,s,1}\right)\right).$$

4: Define the vectors of polynomials $\boldsymbol{u}_i := (u_i^0, \ldots, u_i^{c-1})$, and similarly for $\boldsymbol{v}_i$, $\boldsymbol{k}_i$, and $\widetilde{\boldsymbol{u}}_i$.
5: Let $\boldsymbol{w}_i := (w_i^{0,0}, \ldots, w_i^{c-1,0}, w_i^{0,1}, \ldots, w_i^{c-1,1}, \ldots, w_i^{c-1,c-1})$, and similarly for $\boldsymbol{m}_i$.
6: Compute the final polynomials

$$a_i \leftarrow \langle \boldsymbol{a}, \boldsymbol{u}_i \rangle, \qquad s_i \leftarrow \langle \boldsymbol{a}, \boldsymbol{v}_i \rangle, \qquad e_i \leftarrow \langle \boldsymbol{a}, \boldsymbol{k}_i \rangle,$$
$$\alpha_i \leftarrow \langle \boldsymbol{a} \otimes \boldsymbol{a}, \boldsymbol{w}_i \rangle, \qquad \delta_i^0 \leftarrow \langle \boldsymbol{a} \otimes \boldsymbol{a}, \boldsymbol{m}_i \rangle, \qquad \delta_i^1 \leftarrow \langle \boldsymbol{a}, \widetilde{\boldsymbol{u}}_i \rangle,$$
$$\delta_i \leftarrow \delta_i^0 + \delta_i^1$$

in $\mathbb{F}_q[X]/(F(X))$. Output $(\alpha_i, \mathrm{sk}_i, a_i, s_i, e_i, \delta_i)$.

Expansion Complexity

Although the BBS+ presignature tuples only contain two OLE and one VOLE correlation, the presented PCG primitives for (V)OLE (cf. Section 3.3) realize these correlations for two parties. Therefore, we would need to instantiate significantly more primitives for additional parties; each new party requires additional cross-terms. Naively, for n parties, the PCG for BBS+ would consist of $n^2 - 1$ individual VOLE PCGs and $2 \cdot (n^2 - 1)$ OLE PCGs. This suggests a potentially quadratic scaling. However, notice that Construction 5 intertwines the individual primitives. Recall from Section 5.2.2 that generating ring elements dominates the runtime in the (V)OLE PCG, especially for large N. By directly summing the full domain evaluations (step 2/3), we mitigate this to generate only five ring elements (step 6), independent of the number of parties (n). Since ring element generation involves quasilinear FFT while full domain evaluations are linear, our construction's overall complexity remains quasilinear with respect to the domain size N. For n, on the other hand, we notice that each additional

party adds six full domain evaluations to the expansion. Therefore, Construction 5's computational complexity is quasilinear for domain N and linear for the participating parties n. Section 6.3 provides a detailed evaluation of how this complexity transfers to the implementation.

6.2.2 Threshold Setting

In particular, the summation of the full domain evaluations in step 2/3 hinders compatibility with the t-out-of-n setting for threshold $t < n$. In this setting, parties require only specific cross-term subsets determined by the current signer set. Direct summation prevents this selective use. Furthermore, summing all cross-terms together impedes interpolation on secret key shares when employing methods like Shamir's Secret Sharing [Sha79].

Solution. We can solve this by individually computing the full domain evaluations and generating a ring element (via a) for each. This allows parties to evaluate only the needed ring elements and subsequently interpolate the secret shares according to the signer set. Finally, they can combine all parts to obtain a valid BBS+ tuple. Note that each full domain evaluation must be stored individually for the VOLE component (step 2). However, the forward and backward evaluations can be summed directly in the OLE case (step 3) since interpolation is not required there. This solution does come with drawbacks. First, there is increased space complexity since all intermediate results (in form or additional ring elements) need to be retained. Second, computational overhead arises during the expansion due to the need to generate significantly more ring elements (one for each cross-term). Specifically, the parties need to compute $(c \cdot 2 \cdot (n-1))$ ring elements for the VOLE correlation and $(c^2 \cdot (n-1))$ for each OLE correlation. Notice that the parties are left with two options on how to proceed with these ring elements:

1. **Split All in The Offline Phase:** Instead of splitting five ring elements on all N roots of unity (requiring N polynomial evaluations), the parties now need to split all $(c \cdot 2 \cdot (n-1) + 2 * c \cdot 2 \cdot (n-1)) = 6 \cdot c \cdot (n-1)$ ring elements in the offline phase. Even for the basic 2-out-of-3 setting and $c = 4$, this results in 48 ring elements that must be fully split. Although introducing a heavy overhead to the offline phase, the advantage of this approach is that there is no effect on the online phase, except for a few (negligible) additions determined by threshold t.

2. **Split Selection in The Online Phase:** By recombining the ring elements during the online phase dependent on the current signer set, the parties only need to split five ring elements. The drawback here is that the splitting cannot be done in advance, adding the overhead of five polynomial evaluations to the online phase.

This leaves the parties with a trade-off: either accept a significantly longer offline phase, especially when there are many participants or/and a high domain, or go with a comparably small overhead in the (time-critical) online phase. We analyze the severity of the computational drawbacks of the t-out-of-n approach and the trade-off it requires in Section 6.3.2.

6.3 Evaluation

In this section, we evaluate our implementation[1] of the PCG for BBS+ presignature tuples presented in Construction 5, which realizes the offline preprocessing phase of Faust et al.'s non-interactive threshold BBS+ scheme [Fau+23]. We start with the n-out-of-n setting and compare its performance over different amounts of parties n and domain choices N. Further, we evaluate our implementation of the t-out-of-n setting. We validate that both settings stay quasilinear (regarding domain N) driven through FFT for generating the ring elements and that an increase in participating parties only comes with a linear increase in runtime. This behavior implies that the PCG is practical for realizing an offline preprocessing phase for Construction 4. We find that the adaption to the t-out-of-n setting

[1] `https://github.com/leandro-ro/Threshold-BBS-Plus-PCG`

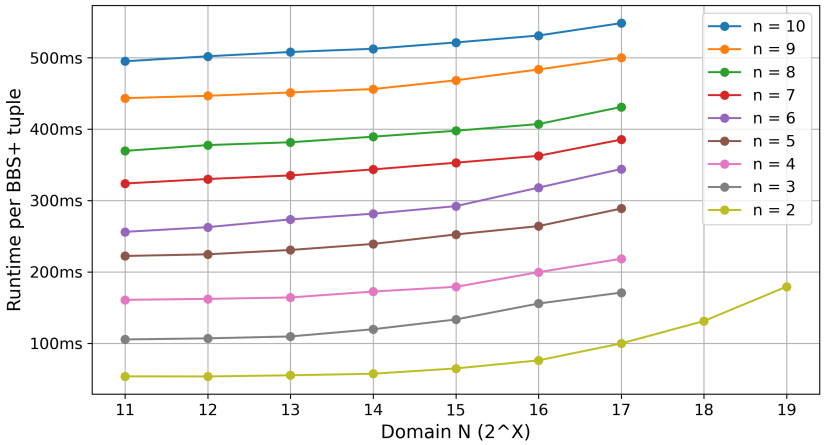


Figure 6.1: n-out-of-n BBS+ PCG expansion over N

strengthens the quasilinearity of the approach, therefore only introducing a manageable overhead. Furthermore, we identify areas of improvement and verify the efficiency of the implementation by comparing our runtimes to [Abr+22]. Finally, we recall the assessment of the online phase from [Fau+23] and apply the findings of our evaluation.

Setup. We employ the same benchmarking setup previously introduced in Chapter 5, utilizing a Xeon Gold 5120 CPU @ 2.20GHz with 14 cores and 64GB of RAM. We maintain the optimizations and parallel processing techniques discussed in Chapter 4 and again choose LPN parameters $(c, \tau) = (4, 16)$, as suggested by Boyle et al. [Boy+20]. Because of the complexity of the benchmarks and the large number of repetitions within the constructions, we limit our benchmark execution to a single run for each parameter configuration.

6.3.1 n-out-of-n

We evaluate our implementation of the n-out-of-n setting for parties $n \in \{2, ..., 10\}$ and domains $N \in \{2^{11}, ..., 2^{19}\}$ in Figure 6.1. As expected, we observe a quasilinear runtime increase with respect to the domain size N for all party counts. Concurrently, runtime increases linearly for n. This aligns with our complexity analysis:

- **Ring Element Generation:** The generation of ring elements is independent of n and fixed while contributing a quasilinear runtime increase for N due to the use of FFT. Therefore, we observe that the runtime increase over N is similar for all n.
- **Full Domain Evaluations:** The amount of full domain evaluations scales linearly with n, while each instantiation comes with linear complexity (cf. Section 5.1.1). Therefore, we observed a linear increase in runtime per additional participant.

Compared to the reported runtimes for the original PCG for OLE (recall Figure 5.7), our BBS+ construction achieves a disproportionately favorable runtime, even though it handles multiple correlations (2x OLE, 1x VOLE). Focusing on the two-party case for a fair comparison, a single BBS+ tuple requires 100ms at $N = 2^{17}$, while a single OLE correlation takes roughly 60ms. This highlights the effectiveness of intertwining the PCG primitives.

6.3.2 t-out-of-n

We evaluate our implementation of the t-out-of-n setting for $t = 2$, parties $n \in \{3, ..., 10\}$ and domains $N \in \{2^{11}, ..., 2^{18}\}$ in Figure 6.1. Notably, t does not influence the runtime as described in Section 6.2.2; therefore, the presented numbers hold for any choice of t. Again, we observe that the runtime

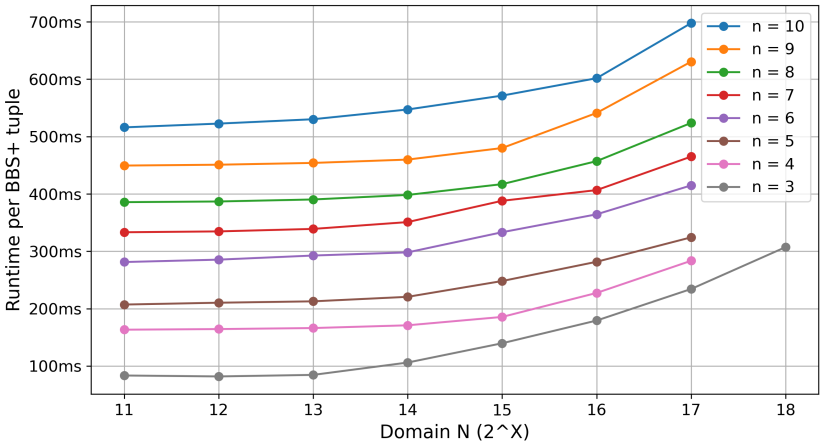


Figure 6.2: 2-out-of-n BBS+ PCG expansion over N

increases quasilinear for domain N. In contrast to the n-out-of-n setting before, the runtime increase over N is notably steeper. This behavior derives from the construction now needing to deal with more ring elements, leading to more frequent use of FFT. Since FFT is used more frequently, the primitive has a higher (non-linear) impact on the total runtime, resulting in a stronger runtime increase over N.

Comparison to n-out-of-n. Directly comparing runtimes, we find the t-out-of-n implementation takes roughly 37% longer for $N = 2^{17}$ and $n = 3$, with similar overhead existing for other values of n. This confirms that the presented adaptation to implement the t-out-of-n setting, while introducing some overhead, remains computationally manageable.

Introducing Overhead to the Offline or Online Phase. We observe that one major drawback of the t-out-of-n approach for Construction 5 is that we are forced to either introduce a large overhead to the offline phase or a small overhead to the online phase (cf. Section 6.2.2). For the first option, we present the additional runtime needed in Figure 6.3 with $N \in \{2^{16}, 2^{17}, 2^{18}\}$ and participating parties $n \in \{3, ..., 10\}$. We determine that the additional overhead in the offline phase is indeed significant: For $N = 2^{18}$ and $n = 3$, we observe an increase in runtime of 60%. Concerning a single presignature, this does seem acceptable. However, when looking at the overall time needed for preprocessing, we go from 21.84 hours to 34.95 hours, which could limit the practicality of the offline phase for real-world applications. Also, notice that the additional overhead is dependent on the amount of participating parties n and rises linearly.

This significant overhead in the offline phase is omitted when opting for option two (cf. Section 6.2.2). Instead, a smaller overhead is introduced in the online phase. Although the amount of ring element evaluations is fixed (to 5), the degree of their polynomial representation is determined by N. This implies that choosing larger domains N for the PCG in the offline phase introduces an increasing overhead during the online signing phase. This increase can be quantified as five degree-N polynomial evaluations, therefore by recalling numbers from Figure 5.4, the overhead for $N = 2^{10}$ amounts to 0.3ms and $N = 2^{18}$ to 18.8ms. Notice that the overhead is much smaller than opting for option one and, most importantly, is independent of the amount of participating parties. Although the overhead is smaller, we want to stress that the online phase is usually more time-critical; therefore, choosing the right approach depends heavily on the specific environment. We want to add that since the point at which the polynomials need to be evaluated is the same, there may be room for improvement since exponentiations can potentially be reused. In any case, some overhead remains at this stage. We decide to leave this potential optimization for future work.

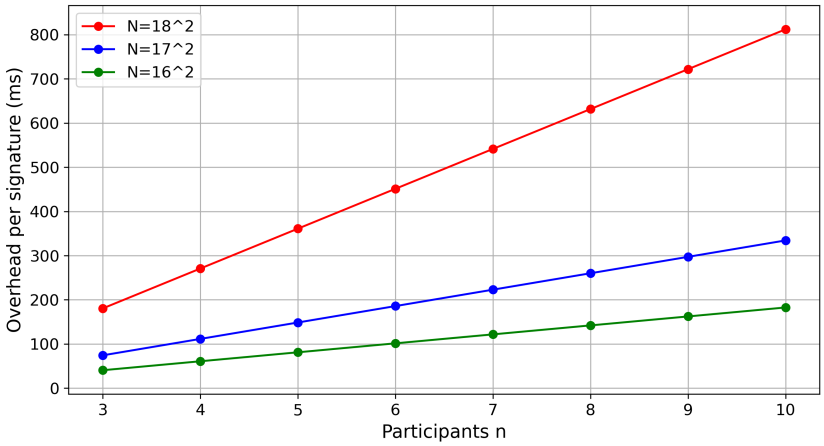


Figure 6.3: Overhead for the t-out-of-n setting when splitting all ring elements in the offline phase

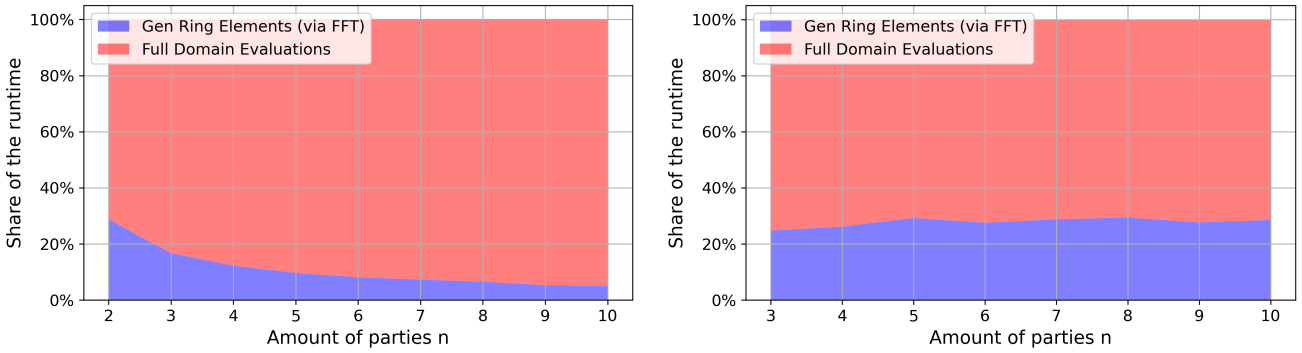
6.3.3 Identifying Bottlenecks

Recall that in the PCG for (V)OLE, generating ring elements using FFT was the primary performance bottleneck (cf. Figure 6.4), especially for large domains. However, let us analyze how this changes within the context of the BBS+ PCG. Figure 6.4 breaks down the construction's runtime for different party counts (n) on $N = 2^{17}$.

In the t-out-of-n setting, the proportion of time spent on full domain evaluations (step 2/3) versus ring element generation (step 6) remains roughly constant. This is because as n becomes larger, more full domain evaluations are needed for each of which additional ring elements need to be created. Therefore, the runtime increases similarly for both building blocks, resulting in an allocation that remains close to constant. Interestingly, in the n-out-of-n setting, ring element generation becomes less dominant. This trend is expected as the number of ring elements stays fixed while full domain evaluations increase with n. Surprisingly, in contrast to FFT being the main cost within the PCG for (V)OLE, our analysis reveals that full domain evaluations of DSPFs represent the primary bottleneck in the PCG for BBS+ tuples. Therefore, further optimizing this building block would significantly improve the overall performance.

Comparison to [Abr+22]. For evaluating the overall efficiency gains introduced by our practical considerations, we compare our runtimes with the *rust* implementation[2] of the n-out-of-n PCG for threshold ECDSA proposed by Abram et al. in [Abr+22]. Since our construction is derived from theirs, the PCGs are very similar. The main difference is that their PCG incorporates only one OLE and one VOLE correlation, while ours incorporates two OLE and one VOLE. Similar to our implementation, the DSPF building block is parallelized and operates on Boyle et al.'s tree-based approach [BGI16]. Therefore, we argue that comparing their implementation to ours is meaningful, although we want to emphasize that *rust* generally offers a performance advantage compared to *golang* in most scenarios[3]. For technical reasons, their selection for the PCG's domain N is not a power of two, but from a given set of optimized values that we adhere to. For a fair comparison, we run their implementation on our test bench and find that for the 2-out-of-2 setting, the PCG requires per presignature 247.79ms for $N = 14304$ and 871.83ms for $N = 94019$. These runtimes align (proportionately) with the numbers reported by the authors. From our measurements (Figure 6.1), we derive that our implementation significantly outperforms theirs by approximately 6x-10x for the same settings despite generating an additional OLE correlation. In particular, our advantage increases for larger N. Consequently, we conclude that the careful optimizations concerning the building blocks as presented in Chapter 4 contribute significant performance improvements to the overall runtime of `Module-LPN` based PCGs as

[2] `https://github.com/ZenGo-X/silent-ecdsa`
[3] `https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust-go.html`



(a) n-out-of-n (b) 2-out-of-n

Figure 6.4: Comparing runtime allocation of n-out-of-n with 2-out-of-n for $N = 2^{17}$

introduced by Boyle et al. [Boy+20], therefore positively affecting their practicality.

6.3.4 Implications for Non-Interactive Threshold BBS+

The presented benchmarks show that the implemented PCG, while resource intensive, is practical for the offline preprocessing phase of Faust et al.'s threshold BBS+ scheme [Fau+23]. For the 2-out-of-2 setting, 2^{17} signatures can be preprocessed in around 9.5 hours (under 132ms per signature), which is certainly practical for some real-world applications, e.g., when preprocessing is performed overnight. Having generated the presignatures, Faust et al. report for the online phase that their implementation of `ThreshSign` then requires around 0.3ms per included message $l \in [k]$, with notably no communication needed between the signing parties. Note that communication between the signing parties and the requester must still be accounted for, which makes the server with the highest latency determine the total runtime of the signing protocol. In that regard, Faust et al. compare their online signing phase with an interactive threshold BBS+ protocol by Doerner et al. [Doe+23] that does not employ the preprocessing model. The authors find that the non-interactive approach pays off, especially in the WAN setting, as it achieves between 2x and 3x superior runtimes. In the LAN setting, the speedup is increasing linearly over the participating parties, with around 0.5x for ten parties to 2x for 15 parties.

While the above evaluation of the online phase holds for the n-out-of-n setting, it disregards the potential overhead the t-out-of-n setting introduces when opting for option two, as described in Section 6.2.2 and evaluated in Section 6.3.2. In this case, the parties recombine and evaluate their ring elements on-demand, dependent on the given signer set. Applying the overhead to the reported numbers reveals that this potentially impacts the protocol's advantage in lower latency settings, dependent on the chosen N in the offline phase. Faust et al. report a runtime of around 8ms for $k = 1$ in the LAN setting, constant for the amount of participating parties. As reported, assuming $N = 2^{18}$, additional 18.8ms need to be accounted for. The overhead can be reduced by making N smaller, but this also results in fewer presignatures available. Depending on the number of participating parties (that drive up the runtime/latency in the interactive scheme) and the chosen N, the non-interactive approach may lose its advantage in the LAN scenario. Note that this is heavily dependent on the chosen parameters and only affects low-latency settings. The advantage of the non-interactive online phase still holds for high latency settings (like WAN), regardless of the number of participating parties.

We conclude that in applications where high latency is a dominant factor and/or many participants are present, computing the ring elements on-demand (option two) pays off, as it significantly reduces the complexity of the offline phase while keeping the overhead in the online phase small. Notably, in low-latency environments, this approach makes the threshold BBS+ signature scheme slower than other schemes that do not rely on preprocessing. Therefore, a longer preprocessing phase is to be accepted here to maintain the signature scheme's superiority in the online phase.

7 Conclusion

The central contribution of this thesis is the presentation of practical considerations for implementing Boyle et al.'s `Ring-LPN` based PCG primitive [Boy+20]. These considerations include the optimization of the underlying DSPF building block for reduced space complexity (Section 4.1) and the strategic use of different approaches for handling sparse polynomials (Section 4.2). We also presented a formula for iteratively computing roots of unity (Section 3.2.2) and show how the principles of Horner's method [Hor19] can be used to compute all roots of unity within linear time complexity (Section 4.3). Evaluations within Chapter 5 highlight the impact of these optimizations on the practicality of the PCG construction. The PCG implementation derived from the optimized building blocks exhibits quasilinear scaling as the number of correlations generated increases (Section 5.2.2). This is very appealing for real-world applications that utilize this PCG for preprocessing since the setup phase amortizes faster, and the need for repeating the offline phase is mitigated.

To further demonstrate the value of our optimizations, we incorporate the building blocks for implementing the preprocessing phase of Faust et al.'s threshold BBS+ scheme [Fau+23]. Our proposed PCG is optimized for the n-out-of-n case (Section 6.2.1) and can easily adapt to the threshold setting (Section 6.2.2). Implementations for both cases validate the quasilinear scaling concerning the number of preprocessed correlations (Section 6.3). Notably, the preprocessing phase exhibits linear scaling concerning the number of parties, demonstrating the scheme's suitability for including many signers. Compared to the only other PCG implementation available [Abr+22] (which is limited to the n-out-of-n case), our implementation achieves a 6x to 10x performance improvement for $n = 2$ (Section 6.3.3), clearly highlighting the effectiveness of our practical considerations considering that our PCG includes an additional OLE correlation.

Finally, within the t-out-of-n threshold case, we acknowledge a limitation of the proposed PCG for the threshold BBS+ signature scheme, which introduces a linear overhead in the online phase proportional to the number of precomputed correlations (Section 6.3.4). We observe that this overhead significantly impacts the protocol runtime in low-latency environments, while its impact is insignificant in high-latency environments. We find that this overhead can be omitted by accepting an additional large overhead of around 60% in the offline phase (Section 6.3.2).

7.1 Future Work

We identify several promising directions for future research. Firstly, Boyle et al. [Boy+20] propose various suitable LPN parameter sets achieving 128-bit security equivalence. While we employed $(c, \tau) = (4, 16)$ for consistency with prior work [Abr+22], evaluating performance across different parameter choices for potentially higher security levels presents an interesting direction for future practical assessment. Secondly, our work primarily addressed PCG expansion, assuming a trusted seed generation phase. Implementing and evaluating the distributed setup phase outlined (but not implemented) by Abram et al. [Abr+22] would be a valuable extension. Furthermore, in Section 6.3.2, we briefly mention the possibility of reducing the overhead introduced within the t-out-of-n threshold setting. Optimizations here could mitigate the overhead that the offline phase introduces to the online phase, making the scheme more interesting for low-latency settings as well. Finally, Tessaro et al. [TZ23] recently proposed a more compact BBS+ signature scheme that eliminates the need for one OLE correlation within the BBS+ presignatures. Investigating the potential performance gains of applying this simplification to the PCG remains future work.

Bibliography

[Hor19] William George Horner. "XXI. A new method of solving numerical equations of all orders, by continuous approximation". In: *Philosophical Transactions of the Royal Society of London* 109 (1819), pp. 308–335.

[CT65] James W Cooley and John W Tukey. "An algorithm for the machine calculation of complex Fourier series". In: *Mathematics of computation* 19.90 (1965), pp. 297–301.

[Pol71] John M Pollard. "The fast Fourier transform in a finite field". In: *Mathematics of computation* 25.114 (1971), pp. 365–374.

[Pol74] John M Pollard. "Theorems on factorization and primality testing". In: *Mathematical Proceedings of the Cambridge Philosophical Society*. Vol. 76. 3. Cambridge University Press. 1974, pp. 521–528.

[Bla79] George Robert Blakley. "Safeguarding cryptographic keys". In: *Managing Requirements Knowledge, International Workshop on*. IEEE Computer Society. 1979, pp. 313–313.

[Sha79] Adi Shamir. "How to share a secret". In: *Communications of the ACM* 22.11 (1979), pp. 612–613.

[Des87] Yvo Desmedt. "Society and group oriented cryptography: A new concept". In: *Conference on the Theory and Application of Cryptographic Techniques*. Springer. 1987, pp. 120–127.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L Rivest. "A digital signature scheme secure against adaptive chosen-message attacks". In: *SIAM Journal on computing* 17.2 (1988), pp. 281–308.

[DF91] Yvo Desmedt and Yair Frankel. "Shared generation of authenticators and signatures". In: *Annual International Cryptology Conference*. Springer. 1991, pp. 457–469.

[Des92] Yvo Desmedt. "Threshold cryptosystems". In: *International Workshop on the Theory and Application of Cryptographic Techniques*. Springer. 1992, pp. 1–14.

[Gen96] Rosario Gennaro. "Theory and practice of verifiable secret sharing". PhD thesis. Massachusetts Institute of Technology, 1996.

[GI99] Niv Gilboa and Yuval Ishai. "Compressing cryptographic resources". In: *Annual International Cryptology Conference*. Springer. 1999, pp. 591–608.

[Sho99] Peter W Shor. "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer". In: *SIAM review* 41.2 (1999), pp. 303–332.

[BBS04] Dan Boneh, Xavier Boyen, and Hovav Shacham. "Short group signatures". In: *Annual international cryptology conference*. Springer. 2004, pp. 41–55.

[CDI05] Ronald Cramer, Ivan Damgård, and Yuval Ishai. "Share conversion, pseudorandom secret-sharing and applications to secure computation". In: *Theory of Cryptography: Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005. Proceedings 2*. Springer. 2005, pp. 342–362.

[ASM06] Man Ho Au, Willy Susilo, and Yi Mu. "Constant-size dynamic k-TAA". In: *Security and Cryptography for Networks: 5th International Conference, SCN 2006, Maiori, Italy, September 6-8, 2006. Proceedings 5*. Springer. 2006, pp. 111–125.

[IPS09] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. "Secure arithmetic computation with no honest majority". In: *Theory of Cryptography: 6th Theory of Cryptography Conference, TCC 2009, San Francisco, CA, USA, March 15-17, 2009. Proceedings 6*. Springer. 2009, pp. 294–314.

[Hey+12] Stefan Heyse et al. "Lapin: an efficient authentication protocol based on ring-LPN". In: *Fast Software Encryption: 19th International Workshop, FSE 2012, Washington, DC, USA, March 19-21, 2012. Revised Selected Papers*. Springer. 2012, pp. 346–365.

[Pie12] Krzysztof Pietrzak. "Cryptography from learning parity with noise". In: *International Conference on Current Trends in Theory and Practice of Computer Science*. Springer. 2012, pp. 99–114.

[GGM13] Christina Garman, Matthew Green, and Ian Miers. "Decentralized anonymous credentials". In: *Cryptology ePrint Archive* (2013).

[Ish+13] Yuval Ishai et al. "On the power of correlated randomness in secure computation". In: *Theory of Cryptography: 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*. Springer. 2013, pp. 600–620.

[SV14] Nigel P Smart and Frederik Vercauteren. "Fully homomorphic SIMD operations". In: *Designs, codes and cryptography* 71 (2014), pp. 57–81.

[BGI15] Elette Boyle, Niv Gilboa, and Yuval Ishai. "Function secret sharing". In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer. 2015, pp. 337–367.

[Cou+15] Emanuel Ferreira Coutinho et al. "Elasticity in cloud computing: a survey". In: *annals of telecommunications-annales des télécommunications* 70 (2015), pp. 289–309.

[BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. "Function secret sharing: Improvements and extensions". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 1292–1303.

[GGN16] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. "Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security". In: *Applied Cryptography and Network Security: 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings 14*. Springer. 2016, pp. 156–174.

[Hal+16] Shai Halevi et al. "Secure multiparty computation with general interaction patterns". In: *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*. 2016, pp. 157–168.

[YS16] Yu Yu and John Steinberger. "Pseudorandom functions in almost constant depth from low-noise LPN". In: *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*. Springer. 2016, pp. 154–183.

[Boy+17] Elette Boyle et al. "Homomorphic secret sharing: optimizations and applications". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 2105–2122.

[Liu+17] Hanlin Liu et al. "On the Hardness of Sparsely Learning Parity with Noise". In: *Provable Security: 11th International Conference, ProvSec 2017, Xi'an, China, October 23-25, 2017, Proceedings 11*. Springer. 2017, pp. 261–267.

[Boy+18] Elette Boyle et al. "Compressing vector OLE". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 896–912.

[ZYZ18] Shuoyao Zhao, Yu Yu, and Jiang Zhang. "On the Hardness of Learning Parity with Noise over Rings". In: *International Conference on Provable Security*. Springer. 2018, pp. 94–108.

[Boy+19] Elette Boyle et al. “Efficient pseudorandom correlation generators: Silent OT extension and more”. In: *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*. Springer. 2019, pp. 489–518.

[GGI19] Rosario Gennaro, Steven Goldfeder, and Bertrand Ithurburn. “Fully distributed group signatures”. In: *See orbs. com/white-papers/fully-distributed-group-signatures/website* (2019).

[GMW19] Oded Goldreich, Silvio Micali, and Avi Wigderson. “How to play any mental game, or a completeness theorem for protocols with honest majority”. In: *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*. 2019, pp. 307–328.

[Sch+19] Phillipp Schoppmann et al. “Distributed vector-OLE: Improved constructions and implementation”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 1055–1072.

[Boy+20] Elette Boyle et al. “Efficient pseudorandom correlation generators from ring-LPN”. In: *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part II 40*. Springer. 2020, pp. 387–416.

[CRR21] Geoffroy Couteau, Peter Rindal, and Srinivasan Raghuraman. “Silver: silent VOLE and oblivious transfer from hardness of decoding structured LDPC codes”. In: *Annual International Cryptology Conference*. Springer. 2021, pp. 502–534.

[Abr+22] Damiano Abram et al. “Low-bandwidth threshold ECDSA via pseudorandom correlation generators”. In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2022, pp. 2554–2572.

[BC22] Dung Bui and Geoffroy Couteau. “Private Set Intersection from Pseudorandom Correlation Generators.” In: *IACR Cryptol. ePrint Arch.* 2022 (2022), p. 334.

[Dit+22] Samuel Dittmer et al. “Authenticated garbling from simple correlations”. In: *Annual International Cryptology Conference*. Springer. 2022, pp. 57–87.

[Wag22] Sameer Wagh. “BarnOwl: Secure Comparisons using Silent Pseudorandom Correlation Generators”. In: *Cryptology ePrint Archive* (2022).

[BC23] Dung Bui and Geoffroy Couteau. “Improved private set intersection for sets with small entries”. In: *IACR International Conference on Public-Key Cryptography*. Springer. 2023, pp. 190–220.

[Doe+23] Jack Doerner et al. “Threshold bbs+ signatures for distributed anonymous credential issuance”. In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2023, pp. 773–789.

[Fau+23] Sebastian Faust et al. “Non-Interactive Threshold BBS+ From Pseudorandom Correlations”. In: *Cryptology ePrint Archive* (2023).

[KOR23] Yashvanth Kondi, Claudio Orlandi, and Lawrence Roy. “Two-Round Stateless Deterministic Two-Party Schnorr Signatures From Pseudorandom Correlation Functions”. In: *Annual International Cryptology Conference*. Springer. 2023, pp. 646–677.

[TZ23] Stefano Tessaro and Chenzhi Zhu. “Revisiting BBS Signatures”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2023, pp. 691–721.