FOLEAGE: $\mathbb{F}_4$OLE-Based Multi-Party Computation for Boolean Circuits

Maxime Bombar[1], Dung Bui[2], Geoffroy Couteau[3], Alain Couvreur[4], Clément Ducros[5], and Sacha Servan-Schreiber[6]

[1] Cryptology Group, CWI
maxime.bombar@cwi.nl
[2] IRIF, Université Paris Cité
bui@irif.fr
[3] CNRS, IRIF, Université Paris Cité
couteau@irif.fr
[4] INRIA, Laboratoire LIX, École Polytechnique, Institut Polytechnique de Paris
alain.couvreur@inria.fr
[5] IRIF, Université Paris Cité, INRIA
cducros@irif.fr
[6] MIT
3s@mit.edu

Abstract. Secure Multi-party Computation (MPC) allows two or more parties to compute any public function over their privately-held inputs, without revealing any information beyond the result of the computation. The main efficiency bottleneck in MPC protocols comes from *interaction* between parties. To limit the interaction, modern protocols for MPC generate a large amount of input-independent preprocessing material called *multiplication triples*, in an offline phase. This preprocessing can later be used by the parties to efficiently instantiate an input-dependent online phase computing the function.

To date, the state-of-the-art secure multi-party computation protocols in the preprocessing model are tailored to secure computation of *arithmetic* circuits over large fields and require little communication in the preprocessing phase. In contrast, when it comes to computing preprocessing for computations that are naturally represented as *Boolean* circuits, the state-of-the-art techniques have not evolved since the 1980s, and in particular, require every pair of parties to interactively generate a large number of *oblivious transfers*, which induces an $\Omega(N^2 \cdot m)$ communication overhead.

In this paper, we introduce $\mathbb{F}_4$OLEAGE, which addresses this gap by introducing an efficient preprocessing protocol tailored to Boolean circuits. $\mathbb{F}_4$OLEAGE exhibits excellent performance: it generates m triples using only $N \cdot m + O(N^2 \cdot \log m)$ bits of communication for N-parties, and can concretely produce over 11 million triples per second on one core of a commodity machine. Our result builds upon an efficient Pseudorandom Correlation Generator (PCG) for multiplications triples over the field $\mathbb{F}_4$. Roughly speaking, a PCG enables parties to stretch a short seed into a large number of pseudorandom correlations *non-interactively*, which greatly improves the efficiency of the offline phase in MPC protocols. Our construction significantly outperforms the state-of-the-art, which we demonstrate via a prototype implementation. This is achieved by introducing a number of protocol-level, algorithmic-level, and implementation-level optimizations on the recent PCG construction of Bombar et al. (Crypto 2023) from the Quasi-Abelian Syndrome Decoding assumption supporting any field size $q \geq 3$.

Table of Contents

1 Introduction

A secure multiparty computation (MPC) protocol for a public functionality f allows N parties with private inputs $(x_1, \cdots, x_N)$ to securely compute $f(x_1, \cdots, x_N)$, while concealing all other information about their private inputs to coalitions of corrupted parties. MPC was introduced in the seminal work of Goldreich, Micali, and Wigderson [GMW87] (GMW), and has since led to the creation of a rich body of work developing the foundations of MPC, and even practical open-source libraries [Kel20].

Two of the leading paradigms in secure computation are garbled circuits [Yao86] and secret-sharing-based secure computation [GMW87]. The seminal GMW protocol is of the latter type. In a secret-sharing-based MPC protocol, the parties hold shares of the inputs and iteratively compute the circuit representing the function, gate-by-gate. Because addition gates can be computed locally by the parties holding the input shares, only multiplication gates require interaction between the parties to evaluate. As such, the major bottleneck of MPC protocols is due to the communication required to evaluate the multiplication gates in a circuit. (Note that this is also true of the garbled circuit approach where addition gates are "free" and only multiplication gates need to be garbled [KS08].)

However, a core advantage of secret-sharing-based MPC, first identified in the work of Beaver [Bea92], is that secure multiplications can be *preprocessed* in an *input-independent* precomputation phase. In particular, the parties can securely generate additive shares of many "Beaver triples" $(a, b, a \cdot b) \in \mathbb{F}^3$. Then, for each multiplication gate that needs to be computed in the online phase, the parties can run a fast information-theoretically secure multiplication protocol that consumes one Beaver triple and involves communicating just two elements of $\mathbb{F}$ per party. This model of secure computation with preprocessing forms the basis for modern MPC protocols due to the efficiency of the online phase. However, this preprocessing paradigm only serves to push the inefficiency bottleneck of MPC to the offline phase that consists of generating many Beaver triples. We briefly survey the different techniques that have been developed in the last couple of decades for the efficient generation of Beaver triples in an MPC setting.

Modern secure computation protocols. The traditional approach for securely generating Beaver triples relies on Oblivious Transfers (OT) [Rab81, EGL82]: an N-party Beaver triple over $\mathbb{F}$ can be generated by letting each *pair* of parties execute $\log |\mathbb{F}|$ oblivious transfers [Gil99], and thanks to *OT extension* protocols [Bea96, IKNP03], generating a large number of OTs requires only cheap symmetric-key operations. This OT-based approach is very competitive in a setting where there are only a small number of parties but becomes very inefficient with many parties. Specifically, because *each pair* of parties needs to perform OTs, the communication and computation costs are on the order of $\Omega(N^2)$, which quickly becomes impractical as N grows large.

Over the past decade, the practicality of secure computation has increased tremendously [DPSZ12, KOS16, Kel20, DNNR17, HOSS18a, KPR18]. This is especially true in the setting of secure computation of *arithmetic circuits over large fields*. Starting with the celebrated SPDZ protocol [DPSZ12], a sequence of works has developed fast protocols that use Ring-LWE-based somewhat homomorphic encryption, or even linearly homomorphic encryption, to generate m Beaver triples with only $O(m \cdot N)$ communication and computation per triple. These approaches significantly improve over the "naïve" $\Omega(m \cdot N^2)$ cost of the OT-based approach. Over sufficiently large fields, and amortization that comes from generating many triples, state-of-the-art protocols such as Overdrive [KPR18] achieve very good concrete efficiency.

More recently, following the line of work on silent secure computation initiated in [BCGI18, BCG$^+$19b, BCG$^+$19a], Boyle et al. [BCG$^+$20b] have shown how to generate a large number m of *pseudorandom* (as opposed to truly random) Beaver triples under the Ring-LPN assumption. Their approach uses $O(\log m \cdot N^2)$ communication, followed solely by *local* computation, with good concrete efficiency (the authors estimated a throughput of around 10^5 triples per second on one core of a standard laptop). For sufficiently large values of m, this is highly competitive with Overdrive. However, both Overdrive and the existing PCG-based approach share a common restriction: they are only usable over large fields.

Secure computation of Boolean circuits. In contrast to the secure computation of arithmetic circuits over large fields, the fastest way to run N-party MPC protocols for Boolean circuits remains the "naïve" method of generating many pairwise OTs, at a cost of $\Omega(m \cdot N^2)$ bits for m Beaver triples. This is in contrast to the *two-party* setting, where two-party Beaver triples can be generated very efficiently (with $O(\log m)$ communication) thanks to a recent line of work [BCGI18, BCG$^+$19b, BCG+19a] on silent OT extension. In silent OT extension, two parties can generate m Beaver triples using only $O(\log m)$ communication. The state-of-the-art protocols in this area [CRR21, BCG+22, RRT23] achieve impressive throughputs of several million Beaver triples per second on one core of a standard laptop. Furthermore, the recent SoftSpoken OT extension protocol [Roy22] yields even faster OTs at the cost of increasing communication. For example, SoftSpoken can generate nearly 30M OT/s on `localhost` at the cost of increasing the communication to $64m$ bits to generate m Beaver triples; other communication/computation tradeoffs are possible [Roy22, Table 1].[7]

The situation, however, is much less satisfying for the setting of secure computation of Boolean circuits with a larger number of parties. Protocols such as SPDZ [DPSZ12] and Overdrive [KPR18] do not perform well when generating Beaver triples for Boolean circuits, even in the passive setting. This is due to the high overhead of embedding $\mathbb{F}_2$ in an extension field compatible with the number theoretic-transform used in efficient instantiations of the BGV encryption scheme [BGV12]. Furthermore, silent OT extension techniques build on Pseudorandom Correlation Generators (PCGs), which typically work only in the two-party setting [BCG+19b]. To handle more parties, one needs the stronger notion of *programmable* PCG [BCG+20b], which, informally, allows partially specifying parts of the generated correlation. Unfortunately, while efficient programmable PCGs over large fields were introduced in [BCG+20b], building efficient programmable PCGs over $\mathbb{F}_2$ has remained elusive thus far. The state-of-the-art is the recent work of Bombar et al. [BCCD23], which generates Beaver triples over any field $\mathbb{F}_q$ with $q \geq 3$. However, Bombar et al. [BCCD23] leave analyzing the concrete efficiency for future work.

In light of this state of affairs, to the best of our knowledge, the current most efficient approach for N-party secure computation of Boolean circuits remains the classical OT-based approach. In a little more detail, to generate each Beaver triple, each party P_i samples a random pair (a_i, b_i) of bits, and each pair (P_i, P_j) of parties executes two oblivious transfer protocols to generate additive shares of $a_i b_j$ and $a_j b_i$. Then, all parties aggregate their shares to obtain shares of $\sum_{i,j} a_i b_j = (\sum_i a_i) \cdot (\sum_j b_j)$. When generating m Beaver triples, this approach requires $N \cdot (N-1) \cdot m$ oblivious transfers in total (to be compared with the $O(N^2 \cdot \log m)$ communication of [BCG+20b], or the $O(N \cdot m)$ communication of Overdrive [KPR18], for the case of arithmetic circuits over large fields). While there has been tremendous progress in constructing efficient OT protocols, starting with the work of IKNP [IKNP03], even using silent OT extension (which has the lower communication overhead) requires $3N \cdot (N-1) \cdot m$ bits of communication (ignoring some $o(m)$ terms). Using SoftSpoken OT [Roy22] instead, which appears to be the most computationally efficient solution, and setting the "communication/computation tradeoff" parameter k to $k = 5$, the communication increases to $32N \cdot (N-1) \cdot m$ bits. When the number of parties grows, this soon becomes very inefficient.[8]

1.1 Our focus and contributions

In this paper, we focus on secure computation of general Boolean circuits with multiple parties in the semi-honest setting. Our main contribution is $\mathbb{F}_4$OLEAGE, a novel $\mathbb{F}_4$-OLE-based protocol for secure computation *in the preprocessing model* that significantly outperforms the state-of-the-art approach in both the *two-party* and *multi-party* setting. In particular, $\mathbb{F}_4$OLEAGE enjoys much lower communication in the preprocessing phase than all known alternatives and has a very low computational overhead. We expect $\mathbb{F}_4$OLEAGE to be the fastest alternative for large enough circuits on almost any realistic network setting, for any number of parties between two and several hundred. $\mathbb{F}_4$OLEAGE builds upon recent results constructing efficient PCGs and introduces several protocol-level, algorithmic-level, and implementation-level optimizations to make these PCG constructions blazing fast (see Section 7 for a performance evaluation).

In the two-party setting ($N = 2$), $\mathbb{F}_4$OLEAGE enjoys a silent preprocessing (generating m multiplication triples requires $O(\log m)$ communication), and significantly outperforms all previous silent protocols. In particular, our implementation generates around 11.3 million Beaver triples *per second* on one core of an Amazon `c5.metal` server. Compare this to the state-of-the-art silent OT protocol

[7] Note that we need two calls to the OT functionality to generate one Beaver triple.

[8] For a very large number of parties, the linear scaling in N of Overdrive should become favorable. However, after private communication with the authors of Overdrive, the break-even point for communication seems to happen only for values of N in the range of 400+, due to the high overhead of using BGV and embedding $\mathbb{F}_2$ elements.

RRT [RRT23] which generates 3.4 million Beaver triples per second with the same setup, which is more than three times slower. The fastest *non-silent* OT protocol, SoftSpoken OT, generates around 26 million multiplication triples per second on `localhost` in its fastest regime (using $k = 2$ [Roy22, Table 1]), while requiring around $128 \cdot m$ bits of total communication. However, while our approach does achieve a blazing-fast throughput, it has some limitations. In particular, the preprocessing phase of $\mathbb{F}_4$OLEAGE requires more rounds (16 rounds instead of 3 for generating 26M triples compared to [Roy22]). Additionally, our seed size is roughly 130× larger compared to [RRT23], and 2× larger compared to [BCG$^+$20b]. This makes $\mathbb{F}_4$OLEAGE less suitable for generating a small number of triples over a WAN network.

In the multi-party setting ($N > 2$), $\mathbb{F}_4$OLEAGE achieves *almost-silent* preprocessing: To securely compute a circuit with m AND gates, following a silent phase with $O(N^2 \cdot \log m)$ communication, our preprocessing phase requires a single broadcast of $N \cdot m$ bits (one bit per AND gate and per party), and the online phase is as efficient as the standard GMW protocol. As N grows, this represents a drastic reduction in communication compared to the $\sim 3 \cdot N^2 m$ communication obtained when using silent OT extension, or the $\sim 32 \cdot N^2 m$ communication obtained with SoftSpoken OT, while remaining highly competitive in terms of computation.

Comparison with the state of the art. In Table 1, we provide a comparison between $\mathbb{F}_4$OLEAGE, SoftSpoken, and RRT, for $N = 10$ and $N = 2$ parties. In the multiparty setting, due to the very low bandwidth requirement of $\mathbb{F}_4$OLEAGE, we observe that computation is systematically the bottleneck when evaluated on one core of a commodity server. This indicates that $\mathbb{F}_4$OLEAGE is likely to stand out even more whenever more computational power is available, e.g., when evaluated in parallel on multiple cores.

The numbers in Table 1 have been computed using the running time T measured for generating 3^{16} OLEs (Table 6, using the noise parameter $t = 13$) on one core of AWS `c5.metal`, and estimating the per-party cost to generate 10^9 N-party Beaver triples as $2 \cdot (N-1) \cdot T \cdot (10^9/3^{16})$. When $N = 2$, the cost is estimated as $T \cdot (10^9/3^{16})$, accounting for the factor-2 saving tailored to the 2-party setting. For communication, we computed an estimate of $C = 13$MB of communication for our distributed protocol for generating a seed for 3^{18} OLEs. While one could in principle directly generate a seed that stretches to 10^9 OLEs, this would significantly slow down the computation as the 10^9 OLEs must be expanded all at once, and would not fit in memory. Hence, we estimated the communication as $2 \cdot (N-1) \cdot (3^{18}/10^9) \cdot C$ for generating 10^9 N-party Beaver triples (as 3^{18} OLEs is the maximum expansion size we could fit in the memory), and an additional 10^9 bits of communication per party (in the setting $N > 2$).

Security. The security of $\mathbb{F}_4$OLEAGE relies on the Quasi-Abelian Syndrome Decoding (QA-SD) assumption, a variant of the syndrome decoding assumption that was recently introduced in [BCCD23]. QA-SD is a generalization of the standard quasi-cyclic syndrome decoding assumption (used in many previous works [ABD$^+$16,AMBD$^+$18,BCG$^+$19a,AAB$^+$22b]) which was shown to asymptotically resist all known attacks against LPN and syndrome decoding in [BCCD23]. As a contribution of independent interest, we complement their preliminary analysis with thorough concrete cryptanalysis of the security of QA-SD against *all* state-of-the-art attacks. Our analysis covers in full detail the distribution of the noisy coordinates under folding attacks and the cost of attacking folded QA-SD instances using tailored Information Set Decoding (ISD) algorithms over $\mathbb{F}_4$. We include the SageMath script used to select our parameters from this analysis. As a byproduct, our precise analysis yields an attack undermining the claimed security of the parameters from [BCCD23]. Specifically, with our attack and a set of parameters $c = 4, t = 16$ (see Section 6.4.1 for details), [BCCD23] can only achieve a security level of 103 bits instead of 128 bits. We provide an updated set of parameters in Section 6.4.1.

Implementation. Finally, we provide an open-source prototype implementation of our PCG construction in C. Our implementation is covered in detail in Section 7. It includes, in particular, a new implementation of distributed point functions that work with a ternary input domain (providing faster evaluation at a slightly increased key size), and optimized FFT over $\mathbb{F}_4$. We cover all these contributions in more detail in Section 2.

1.2 Organization

We provide a detailed technical overview of our results in Section 2. We introduce necessary preliminaries in Section 3, and describe our optimized PCG for OLEs over $\mathbb{F}_4$ in Section 4. In Section 5,

	Communication	`localhost`	LAN	WAN
Multi-party setting ($N = 10$)				
SoftSpoken ($k = 2$)	134 GB	342s	1192s	12207s
SoftSpoken ($k = 4$)	67 GB	405s	596s	6104s
SoftSpoken ($k = 8$)	34 GB	1900s	1900s	3052s
			*298s	
RRT	6.3 GB	2619s	2619s	2619s
			*50.3s	*515s
$\mathbb{F}_4$OLEAGE	0.7 GB	1589s	1589s	1589s
			*5.6s	*57.9s
Two-party setting ($N = 2$)				
SoftSpoken ($k = 2$)	15 GB	38s	119s	1221s
SoftSpoken ($k = 4$)	7.5 GB	45s	60s	610s
SoftSpoken ($k = 8$)	3.7 GB	211s	211s	211s
RRT	258 KB	292s	292s	292s
$\mathbb{F}_4$OLEAGE	33.5 MB	88s	88s	88s

Table 1: Comparison of state-of-the-art protocols to generate N-party Beaver triples over $\mathbb{F}_2$ for $N = 10$ and $N = 2$ parties. The `localhost` column reports the runtimes (ignoring communication) for generating 10^9 triples. All protocols run on one core of AWS `c5.metal` (3.4GHz CPU); all runtimes averaged across ten trials. "Communication" denotes the number of bits communicated per party for 10^9 triples. LAN and WAN refer to the theoretical time required to generate 10^9 triples over a 1 Gbps and 100 Mbps network respectively, with respective delays 1ms and 40ms. Numbers in red indicate that the bottleneck cost is the local computation. *maximum theoretical throughput with more computational power (e.g. using multiple cores). Since each party computes $2 \cdot (N - 1)$ expansions for the PCG in parallel for an N-party Beaver triple, the running time is divided by C when using C cores whenever $C \leq 2 \cdot (N - 1)$.

we describe our distributed seed generation protocol. In Section 6, we describe our cryptanalysis of the QA-SD assumption and our parameter selection algorithm. In Section 7, we report on our implementation and evaluate the performance of our scheme.

Appendices. We defer most of the technical details to the appendices sections: Appendix A contains additional preliminaries on PCGs. Appendix B covers the formal statement of our information-theoretic MPC protocols given access to a functionality for $\mathbb{F}_4$-OLEs. Appendix C further overviews a general framework for ISD algorithms over fields $\mathbb{F}_q$. Eventually, Appendix D covers an even faster PCG for generating OLEs using Cuckoo hashing, suitable in contexts where a trusted entity generates and distributes the PCG shares, and Appendix E contains all deferred proofs.

2 Technical Overview

In this section, we provide a detailed description of our results and the main technical ideas underlying them. In Section 2.1, we provide background on secure multi-party computation realized from PCGs for OLE correlations. In Section 2.2 we describe the PCG construction of [BCCD23], which forms the basis for our preprocessing protocol. In Section 2.3, we describe our idea for converting $\mathbb{F}_4$ triples into $\mathbb{F}_2$ triples, which we tailor to the two-party case in Section 2.4. In Section 2.5, we describe our optimized PCG construction. In Section 2.6, we explain how we can obtain an efficient distributed seed generation protocol for our PCG construction. Finally, in Section 2.7, we overview our improved analysis of the QA-SD assumption.

Notation. Unless otherwise stated, an N-party linear secret shares of a value v is denoted $[\![v]\!] = ([\![v]\!]_1, \dots, [\![v]\!]_N)$, where the i-th party obtains share $[\![v]\!]_i$. To disambiguate shares over $\mathbb{F}_4$ and shares over $\mathbb{F}_2$, we denote the field size with a superscript, i.e., $[\![\cdot]\!]^4$ and $[\![\cdot]\!]^2$, respectively. We identify $\mathbb{F}_4$ with $\mathbb{F}_2[X]/(X^2+X+1)$ and let θ denote a primitive root of X^2+X+1. Given an element $x \in \mathbb{F}_4$, we write $x(0)$ and $x(1)$ to denote the $\mathbb{F}_2$-coefficients of x viewed as a polynomial over $\mathbb{F}_2[X]/(X^2+X+1)$; that is, $x = x(0) + \theta \cdot x(1)$. Additional notation can be found in Section 3.

2.1 Background: Secure MPC from PCGs

We start by describing prior approaches to realizing MPC in the preprocessing model from PCGs for OLE correlations.

PCGs for the OLE correlation. Our starting point is the template for generating N-party pseudorandom Beaver triples put forth by Boyle et al. [BCG+20b]. At the heart of their framework is the use of a programmable PCG [BCG+20b] for the OLE correlation. Concretely, a PCG for a target correlation C (i.e., a distribution over pairs of strings) is a pair of algorithms $(\mathsf{PCG.Gen}, \mathsf{PCG.Eval})$ such that

- $\mathsf{PCG.Gen}$ generates a pair of *succinct* keys $(\mathsf{k}_0, \mathsf{k}_1)$ jointly encoding the target correlation, and
- $\mathsf{PCG.Expand}(\sigma, \mathsf{k}_\sigma)$ produces a string R_σ corresponding to party σ's secret share of the target correlation.

At a high level, a PCG must satisfy two properties: (1) *pseudorandomness* (or correctness) which states that (R_0, R_1) must be indistinguishable from a random sample from C, and (2) *security* which states that R_σ should appear random conditioned on satisfying the target correlation with $R_{1-\sigma} = \mathsf{PCG.Expand}(1-\sigma, \mathsf{k}_{1-\sigma})$ even given $\mathsf{k}_{1-\sigma}$, for $\sigma \in \{0,1\}$.

We focus on the OLE correlation over a finite field $\mathbb{F}$. For a length-m OLE correlation, the string R_0 (which we call the *sender* output) is a list of m tuples $(u_i, v_i)_{i \le m} \in (\mathbb{F}^2)^m$, and the string R_1 (which we call the *receiver* output) is a list of m pairs $(x_i, w_i)_{i \le m} \in (\mathbb{F}^2)^m$ such that $w_i = u_i \cdot x_i + v_i$ for every i. Observe that, we can equivalently view v_i and $-w_i$ as additive shares of $u_i \cdot x_i$, which we will denote as $[\![u_i \cdot x_i]\!]$. Informally, security for the OLE correlation amounts to showing that the following two properties hold:

- **Sender security:** from the viewpoint of the receiver (who has k_1 and generates (x_i, w_i)), the distribution of (u_i, v_i) is computationally indistinguishable from the distribution of $(u_i, w_i - u_i \cdot x_i)$, for a uniformly random $u_i \leftarrow_R \mathbb{F}$.
- **Receiver security:** from the viewpoint of the sender (who has k_0), the distribution of each x_i is computationally indistinguishable from a random field element.

Going from OLE to Beaver triples. As shown in [BCG+19b], given a PCG for the OLE correlation (or a PCG for OLE for short), two parties can generate many pseudorandom Beaver triples over $\mathbb{F}$ as follows. First, the parties compute $\mathsf{PCG.Gen}$ via a two-party secure computation protocol to obtain PCG keys k_0 and k_1, respectively. Then, using $\mathsf{PCG.Expand}$, the two parties locally obtain many correlations of the form $(u_i, [\![u_i x_i]\!]_0)$ and $(x_i, [\![u_i x_i]\!]_1)$, respectively. Given two such OLE correlations, where one party has $(u_0, u_1, [\![u_0 x_0]\!]_0, [\![u_1 x_1]\!]_0)$ and the other party has $(x_0, x_1, [\![u_0 x_0]\!]_1, [\![u_1 x_1]\!]_1)$, the two parties can *locally* derive one Beaver triple of the form $([\![a]\!], [\![b]\!], [\![ab]\!])$ by computing:

$$(\underbrace{[\![u_0 + x_1]\!]}_{[\![a]\!]},\ \underbrace{[\![u_1 + x_0]\!]}_{[\![b]\!]},\ [\![u_0 x_0 + u_1 x_1]\!] + u_0 u_1 + x_0 x_1 = \underbrace{[\![(u_0 + x_1) \cdot (u_1 + x_0)]\!]}_{[\![ab]\!]}).$$

In a little more detail, the sender computes their share of the Beaver triple as $(u_0, u_1, [\![u_0 x_0]\!]_0 + [\![u_1 x_1]\!]_0 + u_0 u_1)$ and the receiver computes their share as $(x_1, x_0, [\![u_0 x_0]\!]_1 + [\![u_1 x_1]\!]_1 + x_0 x_1)$. While this technique works well in the two-party setting, in the *multi*-party setting, things are not so simple.

Going from two parties to many parties. As first discussed by Boyle et al. [BCG+20b], to generate N-party Beaver triples using a PCG for OLE, the parties need to ensure *consistency* among the OLE correlations generated by *each pair of parties.* That is, to generate one multiplication triple $([\![a]\!], [\![b]\!], [\![ab]\!])$, we need each pair of parties (P_i, P_j) to hold respective values (a_i, b_i) and (a_j, b_j) (viewed as an individual share of a and b), together with two-party shares $[\![a_i b_j]\!]$ and $[\![a_j b_i]\!]$. Then, all parties can combine their shares to get

$$[\![(\textstyle\sum_i a_i) \cdot (\sum_j b_j)]\!] = \sum_{i \neq j} [\![a_i b_j]\!] + \sum_i a_i b_i.$$

Observe that this requires party P_i to have OLEs of the form $(a_i, \llbracket a_i a_j \rrbracket_i)$, with every other party P_j (who in turn has share $(a_j, \llbracket a_i a_j \rrbracket_j)$), *where P_j's value a_i remains the same across all OLEs.* This is precisely what the notion of a *programmable* PCG for OLE achieves: it allows the parties to specify seeds (ρ_0, ρ_1) such that $\mathsf{PCG.Gen}(\rho_0, \rho_1)$ outputs keys $\mathsf{k}_0, \mathsf{k}_1$ that, informally speaking, have all the pseudorandom (a_i, b_i) deterministically generated from the seeds ρ_0 and ρ_1 respectively (while still maintaining the required security properties). By reusing the same seeds across executions with multiple parties, the parties can ensure the required consistency across their outputs.

2.2 Constructing programmable PCGs

In addition to defining the notion of programmable PCGs, the work of Boyle et al. [BCG+20b] introduced a construction from a variant of the LPN assumption over rings. At a high level, the ring-LPN assumption they introduce states that $(a, as + e)$ is hard to distinguish from (a, b), where a, b are random polynomials from a suitable ring $\mathcal{R} = \mathbb{F}_q[X]/(P(X))$, where P splits into $\deg(P)$ linear factors and s, e are random *sparse* polynomials from $\mathcal{R}$. The construction of Boyle et al. proceeds by generating a single large pseudorandom OLE correlation over a polynomial ring $\mathcal{R} = \mathbb{F}_q[X]/(P(X))$, assuming the hardness of the ring-LPN assumption over $\mathcal{R}$. When P splits into $D = \deg(P)$ linear factors, the Chinese Remainder Theorem makes it possible to convert this large OLE correlation over $\mathcal{R}$ into D OLE correlations over $\mathbb{F}_q$ (by reducing it modulo each of the factors of P). Unfortunately, the condition that P splits requires $|\mathbb{F}_q| \geq D$, which restricts the construction to only work over large fields. This makes the resulting OLE correlations only suitable for generating Beaver triples over $\mathbb{F}_q$, which limits their applications. Moreover, other existing efficient (non-PCG-based) protocols for generating Beaver triples are also restricted to large fields [DPSZ12, KPR18]. However, for the Boolean circuit case, the state-of-the-art remains the basic OT-based approach originally proposed in the GMW protocol.

A programmable PCG for $\mathbb{F}_4$-OLE. The large-field restriction of the Boyle et al.'s PCG construction was recently overcome by Bombar et al. [BCCD23]. At a high-level, the authors of [BCCD23] manage to replace the polynomial ring $\mathcal{R}$ by a suitable *Abelian group algebra* $\mathbb{F}[\mathbb{G}]$ (that is, the set of formal sums $\sum_{g \in \mathbb{G}} a_g g$ for $a_g \in \mathbb{F}$, where $\mathbb{G}$ is an Abelian group; endowed with the convolution product), which identifies to some ring of multivariate polynomials. Moreover, they show that an appropriate choice of Abelian group algebra can simultaneously satisfy the following properties, for almost every choice of finite field $\mathbb{F}$:

1. $\mathbb{F}[\mathbb{G}]$ is isomorphic to many copies of $\mathbb{F}$ (note that this property is necessary to convert an OLE correlation over $\mathbb{F}[\mathbb{G}]$ into many OLEs over $\mathbb{F}$),
2. The assumption that $(a, as+e)$ is indistinguishable from random over $\mathbb{F}[\mathbb{G}] \times \mathbb{F}[\mathbb{G}]$, with $a \overset{\$}{\leftarrow} \mathbb{F}[\mathbb{G}]$ and (s, e) two random *sparse* elements of $\mathbb{F}[\mathbb{G}]$ (with respect to the canonical notion of sparsity over the group algebra, i.e., sparse formal sums $\sum_{g \in \mathbb{G}} a_g g$) is a plausible assumption,
3. Operations over $\mathbb{F}[\mathbb{G}]$ can be computed efficiently using a Fast Fourier Transform (FFT) algorithm [Obe07, BCCD23].

The second property is a new variant of the syndrome decoding (or LPN) assumption which the authors called *quasi-abelian syndrome decoding.* It naturally extends to a "module"-variant, i.e., the indistinguishability of pairs $(\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e)$ where $\mathbf{s}$ and e are drawn from a sparse distribution, and generalizes both the quasi-cyclic syndrome decoding (when $\mathbb{G}$ is a cyclic group), and the LPN or syndrome decoding assumption (when $\mathbb{G} = \{1\}$). The work of Bombar et al. [BCCD23] also provides extensive support for this assumption by showing that it resists all *linear attacks*, a class of attacks capturing the most known attacks on the LPN assumption and its variants, and proposes a set of parameters resisting all concrete attacks known at that time. The combination of these three properties allowed them to build an efficient programmable PCG for generating OLEs over $\mathbb{F}$.

Despite the progress made in the work of Bombar et al. [BCCD23], their programmable PCG construction is limited in that it applies only to generating OLE correlations over all finite fields $\mathbb{F}$ *except for* $\mathbb{F}_2$. This stems from the fact that there does not exist any group $\mathbb{G}$ such that $\mathbb{F}_2[\mathbb{G}]$ is isomorphic $\mathbb{F}_2^n$ for $n > 1$ (see [BCCD23, Theorem 47]). In contrast, the case of $\mathbb{F}_2$, is *precisely* the case that we are interested in when considering Boolean circuits, which require generating Beaver triples over $\mathbb{F}_2$.

Additionally, the *concrete* efficiency of an FFT computed over the group algebra remained unclear. Bombar et al. left estimating the performance of an FFTs on $\mathbb{F}[\mathbb{G}]$ as future work. As such, the concrete efficiency of their programmable PCG construction is unknown, making it difficult to determine whether or not it is sufficiently efficient to be applied in practical applications (all other components of their construction consist of standard tools used in the PCG literature, which are known to have concretely efficient implementations).

Our contribution. Looking ahead, our main contribution is to build upon the work of Bombar et al. through a number of simple, yet surprisingly powerful observations that allow us to arrive at an efficient PCG for Beaver triples, suitable for use in secure multi-party computation of Boolean circuits.

- First, we show that we can use their programmable PCG for generating OLEs *over* $\mathbb{F}_4$ to generate multiplication triples *over* $\mathbb{F}_2$, sidestepping the "$\mathbb{F}_2$ barrier" of their PCG construction, at the cost of a *single bit of communication* per triple and per party in the preprocessing phase.
- Second, we introduce a number of concrete optimizations to the PCG construction of Bombar et al. [BCCD23] that are tailored to the special case of $\mathbb{F} = \mathbb{F}_4$, which gives us an incredibly efficient programmable PCG over $\mathbb{F}_4$. Specifically, compared with the fastest previous programmable PCGs of [BCG⁺20b], our construction is *two orders of magnitude* faster.
- Third, we give a much more in-depth cryptanalysis of the QA-SD assumption, closely analyzing all known attacks in the literature, and showing that the set of parameters proposed in [BCCD23] actually only achieves 100 bits of security (as opposed to 128). To facilitate future cryptanalysis of the QA-SD assumption, in Section 6 we carefully overview all known attacks and assumptions and implement a script for automatically calculating parameters.

In the next few subsections, we provide more details on the above contributions.

2.3 $\mathbb{F}_2$-triples from $\mathbb{F}_4$-triples

Since $\mathbb{F}_4$ is an extension field of $\mathbb{F}_2$, a Boolean circuit can be viewed as an $\mathbb{F}_4$-arithmetic circuit. Hence, using an OLE correlation over $\mathbb{F}_4$ to construct N-party Beaver triples over $\mathbb{F}_4$ directly yields an MPC protocol for *Boolean* circuits in the preprocessing model via the GMW template [GMW87]. Unfortunately, compared to using $\mathbb{F}_2$-Beaver triples, the communication in the *online* phase is doubled, because each party has to send two elements of $\mathbb{F}_4$ per AND gate, hence 4 bits instead of 2 with GMW.

Our core observation is that one can make much better use of these N-party multiplication triples over $\mathbb{F}_4$: we show how to convert an $\mathbb{F}_4$-multiplication triple into an $\mathbb{F}_2$-multiplication triple using *a single bit of communication* per party. Once converted into $\mathbb{F}_2$-triples, these triples can be used within the standard GMW protocol that communicates two bits per party and per AND gate in the online phase. To explain the observation, let $(\llbracket a \rrbracket^4, \llbracket b \rrbracket^4, \llbracket ab \rrbracket^4)$ be a Beaver triple over $\mathbb{F}_4$. Writing $x = x(0) + \theta \cdot x(1)$ for any $x \in \mathbb{F}_4$, with θ a root of the polynomial $X^2 + X + 1$ (hence $\theta^2 = \theta + 1$), we have

$$\begin{aligned} a \cdot b =& a(0)b(0) + a(1)b(1) + \theta \cdot (a(0)b(1) + a(1)b(0) + a(1)b(1)) \\ &\implies (ab)(0) = a(0)b(0) + a(1)b(1). \end{aligned}$$

Now, assume that the parties reconstruct $b(1)$, which can be done using a single bit of communication per party from their shares $\llbracket b \rrbracket^4 = \llbracket b(0) \rrbracket^2 + \theta \cdot \llbracket b(1) \rrbracket^2$. Given $b(1)$, the parties can locally compute shares of $a(0)b(0)$ as follows:

$$\llbracket a(0)b(0) \rrbracket^2 = \llbracket ab \rrbracket^4(0) + b(1) \cdot \llbracket a \rrbracket^4(1).$$

Therefore, all parties output $(\llbracket a(0) \rrbracket^2, \llbracket b(0) \rrbracket^2, \llbracket ab \rrbracket^4(0) + b(1) \cdot \llbracket a \rrbracket^4(1))$, which forms a valid Beaver triple over $\mathbb{F}_2$. Security is straightforward: the only communication between the parties is the reconstruction of $b(1)$, which is a uniformly random bit independent of $a(0), b(0)$. From there, one immediately gets an improved protocol in the preprocessing model: in the preprocessing phase, given one $\mathbb{F}_4$-Beaver triple for each AND gate of the circuit, the parties broadcast one bit per gate, and then locally derive the $\mathbb{F}_2$-Beaver triples. In the online phase, the parties run the standard GMW protocol.

2.4 An improved protocol from $\mathbb{F}_4$-OLEs for $N = 2$

In the setting of $N = 2$ parties, we obtain a much more efficient alternative: we observe that two parties can directly convert a single OLE over $\mathbb{F}_4$ into a Beaver triple over $\mathbb{F}_2$. (In contrast, recall that the standard approach requires two oblivious transfers for each triple.) We consider two parties, Alice and Bob, holding respectively $(a, [\![ab]\!]^4_A)$ and $(b, [\![ab]\!]^4_B)$ for a and $b \in \mathbb{F}_4$. We have

$$\begin{aligned} a \cdot b &= [\![ab]\!]^4_A(0) + [\![ab]\!]^4_B(0) + \theta \cdot ([\![ab]\!]^4_A(1) + [\![ab]\!]^4_B(1)) \\ &= (a(0)b(0) + a(1)b(1)) + \theta \cdot (a(0)b(1) + a(1)b(0) + a(1)b(1)), \end{aligned}$$

where θ is the primitive root of $X^2 + X + 1$. Considering only the $(a \cdot b)(0)$ term from the above equation (i.e., the parts not multiplied by θ), we get that

$$(a \cdot b)(0) = [\![ab]\!]^4_A(0) + [\![ab]\!]^4_B(0) = a(0)b(0) + a(1)b(1),$$

and therefore,

$$\underbrace{a(0)a(1) + [\![ab]\!]^4_A(0)}_{\text{known by } A} + \underbrace{b(0)b(1) + [\![ab]\!]^4_B(0)}_{\text{known by } B} = \underbrace{(a(0) + b(1))}_{\text{shared by } A,B} \cdot \underbrace{(a(1) + b(0))}_{\text{shared by } A,B}.$$

Above, the values $a(0)a(1) + [\![ab]\!]^4_A(0)$ (known by Alice) and $b(0)b(1) + [\![ab]\!]^4_B(0)$ (known by Bob) form additive shares of the product $(a(0)+b(1))\cdot(a(1)+b(0))$, which Alice and Bob hold additive shares of. It is also easy to check that if the input is a random $\mathbb{F}_4$-OLE, the output is a random multiplication triple over $\mathbb{F}_2$. Therefore, following the local conversion procedure outlined above, Alice and Bob can transform a random $\mathbb{F}_4$-OLE instance into a random Beaver over $\mathbb{F}_2$ without having to communicate.

2.5 A fast programmable PCG for $\mathbb{F}_4$-OLEs

In light of the above observations, the only missing piece of the puzzle is an efficient way of generating a large number of $\mathbb{F}_4$-OLEs. In the $N > 2$ setting, if the OLEs are additionally programmable, the parties can afterward locally convert $N \cdot (N - 1)$ $\mathbb{F}_4$-OLE instances into an $\mathbb{F}_4$-Beaver triples.

Here, we build on the recent general programmable PCG construction of [BCCD23]. Because we are targeting OLEs over $\mathbb{F}_4$, we set the group $\mathbb{G}$ to $\mathbb{F}_3^n$, and the underlying group algebra becomes isomorphic to

$$\mathbb{F}_4[\mathbb{G}] \simeq \mathbb{F}_4[X_1, \ldots, X_n]/(X_1^3 - 1, \ldots, X_n^3 - 1) \simeq \mathbb{F}_4^{3^n}.$$

Before delving into the optimizations we develop for their construction, we describe the high-level ideas and main building blocks behind the PCG construction of Bombar et al. [BCCD23] when instantiated over $\mathbb{F}_4$.

The PCG construction of Bombar et al. As with previous constructions of PCGs [BCGI18, BCG$^+$19b], the construction of Bombar et al. uses Distributed Point Functions (DPF) [BGI15,BGI16, GI14] as a core building block. Informally, a DPF with domain $[D]$ allows a dealer to succinctly secret share a unit vector over $[D]$. The most efficient DPFs have shares of size roughly $\lambda \cdot \log D$ [BGI16], for some security parameter λ, and the cost of decompressing the shares is dominated by D calls to a length-doubling pseudorandom generator.

Public parameters. For a fixed *compression factor* c (typically a small constant, e.g., $c = 5$) and *noise parameter* t (e.g., $t = 14$), the public parameters contain a length-c vector $\mathbf{a}$ of n-variate polynomials.

Distributing PCG seeds. In their construction, $\mathsf{PCG.Gen}$ does the following:

- it samples two length-c vectors $(\mathbf{e}_0, \mathbf{e}_1)$ of t-sparse polynomials over $\mathbb{F}_4[\mathbb{G}]$;
- outputs keys $(\mathsf{k}_0, \mathsf{k}_1)$ that contain $\mathbf{e}_0$ and $\mathbf{e}_1$, respectively, as well as *succinct* shares of $\mathbf{e}_0 \otimes \mathbf{e}_1$, encoded using a DPF.

The tensor product $\mathbf{e}_0 \otimes \mathbf{e}_1$ contains c^2 polynomials, each with at most t^2 nonzero coordinates. Hence, the vectors of coefficients of all polynomials in $\mathbf{e}_0 \otimes \mathbf{e}_1$ can be succinctly secret shared using $(ct)^2$ DPFs with domain 3^n, which requires roughly $(ct)^2 \cdot \lambda \log(3^n)$ bits using the state-of-the-art DPF constructions [BGI15, BGI16].[9]

Generating correlations. To output a vector of OLE correlations, $\mathsf{PCG.Eval}$ proceeds as follows for party 0 (the evaluation for party 1 is similar):

[9] Using noise vector with a regular structure, the domain size of the DPFs can be reduced to $3^n/t$.

- evaluate all the DPFs to obtain a secret share of $[\![\mathbf{e}_0 \otimes \mathbf{e}_1]\!]_0$;
- set $x_0 \leftarrow \langle \mathbf{a}, \mathbf{e}_0 \rangle$ and $z_0 \leftarrow \langle \mathbf{a} \otimes \mathbf{a}, [\![\mathbf{e}_0 \otimes \mathbf{e}_1]\!]_0 \rangle$; $\triangleright$ Note: $z_0 = [\![\langle \mathbf{a} \otimes \mathbf{a}, \mathbf{e}_0 \otimes \mathbf{e}_1 \rangle]\!]_0$
- using the isomorphism $\mathbb{F}_4[\mathbb{G}] \simeq \mathbb{F}_4^{3^n}$, project $(x_0, z_0) \in \mathbb{F}_4[\mathbb{G}]^2$ onto 3^n pairs (x_0^i, z_0^i) of elements of $\mathbb{F}_4$.

Above, the projection amounts to evaluating the multivariate polynomials over $\mathbb{F}_4[X_1, \ldots, X_n]/(X_1^3 - 1, \ldots, X_n^3 - 1)$ on the 3^n tuples of elements of $(\mathbb{F}_4^\times)^n$. Observe that

$$\begin{aligned} z_0 + z_1 &= \langle \mathbf{a} \otimes \mathbf{a}, [\![\mathbf{e}_0 \otimes \mathbf{e}_1]\!]_0 \rangle + \langle \mathbf{a} \otimes \mathbf{a}, [\![\mathbf{e}_0 \otimes \mathbf{e}_1]\!]_1 \rangle \\ &= \langle \mathbf{a} \otimes \mathbf{a}, \mathbf{e}_0 \otimes \mathbf{e}_1 \rangle = \langle \mathbf{a}, \mathbf{e}_0 \rangle \cdot \langle \mathbf{a}, \mathbf{e}_1 \rangle \\ &= x_0 \cdot x_1. \end{aligned}$$

Since the isomorphism preserves additions and multiplications, it follows that all pairs (x_0^i, z_0^i) and (x_1^i, z_1^i) form OLEs over $\mathbb{F}_4$. Security follows from the Quasi-Abelian Syndrome Decoding assumption (QA-SD) [BCCD23], which states (informally) that given the random vector $\mathbf{a}$, $\langle \mathbf{a}, \mathbf{e} \rangle + e_0$ (where $(e_0, \mathbf{e})$ is a vector of random sparse polynomials) is indistinguishable from a random element of $\mathbb{F}_4[\mathbb{G}]$.

We now turn to describe several observations that we make about their construction and how these observations allow us to significantly optimize the concrete efficiency of the PCG. While simple in retrospect, these observations turn out to be surprisingly powerful, and yield an extremely efficient PCG for generating $\mathbb{F}_4$-OLEs (see Section 7 for our implementation and evaluation).

Early termination. The DPF construction of [BGI16] generates (pseudorandom) shares of a unit vector using a construction *à la* GGM [GGM86], generating a full binary tree of PRG evaluations starting from a root seed. The children of each node are computed by evaluating a length-doubling PRG on the node, and then adding some correction words. In this construction, each leaf of the tree is a λ-bit string (where typically $\lambda = 128$). In contrast, we wish to share unit vectors over $\mathbb{F}_4$. Hence, we can apply the *early termination* technique from [BGI16] that shaves several levels of PRG expansions. With early termination, to obtain a $D = 2^d$-long vector over $\mathbb{F}_4$, we use a tree of depth $2D/\lambda = 2^{d-6}$ (using $\lambda = 128$) and parse each of the 128-bit leaves as a 64-tuple of $\mathbb{F}_4$-elements. This immediately yields a 64-fold runtime improvement for each of the DPFs required in the PCG construction.

We note that while other constructions share a similar blueprint to the construction of Bombar et al., and in particular also require evaluating many DPFs under-the-hood, this early termination technique does not apply to them. The reason is that in silent OT extension protocols [BCG+19b, BCG+19a, CRR21, BCG+22, RRT23], the DPFs are used to compress secret shares of $\varDelta \cdot \mathbf{e}$, where $\varDelta$ is a 128-bit element from a suitable extension field, and in the previous PCG construction of [BCG+20b], the OLEs can only be generated over a large field $\mathbb{F}$ (chosen equal to $|\mathbb{F}| \approx 2^\lambda$ in their implementation). As such, early termination optimization appears to apply exclusively when specializing the PCG of [BCCD23] to work over small fields.

Using a single multi-evaluation step. Computing $\langle \mathbf{a} \otimes \mathbf{a}, [\![\mathbf{e}_0 \otimes \mathbf{e}_1]\!]_b \rangle$ (for $b = 0, 1$) requires c^2 polynomial multiplications. Fast polynomial multiplication is typically done using a multi-evaluation (i.e., an FFT) followed by a local product and an interpolation (i.e., an inverse FFT).

The above produces a single OLE over $\mathbb{F}_4[\mathbb{G}]$. When the end goal is to obtain OLEs over $\mathbb{F}_4$, the result is projected back onto $\mathbb{F}_4^{3^n}$ using a multi-evaluation. In this case, we show that we can reduce the sequence multi-evaluation → interpolation → multi-evaluation down to just a single multi-evaluation step. Concretely:

- Given that $\mathbf{a}$ is a random vector of polynomials (and part of the public parameters), it can directly be generated as c random length-3^n vectors over $\mathbb{F}_4^n$, corresponding to the vectors of the multi-evaluations of $\mathbf{a}$ over all n-tuples in $(\mathbb{F}_4^\times)^n$.
- The multi-evaluation of $\mathbf{a} \otimes \mathbf{a}$ can be computed once for all using pairwise products of elements of (the multi-evaluation of) $\mathbf{a}$, and included in the public parameters.
- Computing the multi-evaluation of $\langle \mathbf{a} \otimes \mathbf{a}, [\![\mathbf{e}_0 \otimes \mathbf{e}_1]\!]_b \rangle$ amounts to computing the multi-evaluation of $[\![\mathbf{e}_0 \otimes \mathbf{e}_1]\!]_b$ followed by component-wise inner products.

It follows that after expanding the shares $[\![\mathbf{e}_0 \otimes \mathbf{e}_1]\!]_b$, the cost of PCG.Expand is then dominated by c^2 instances of a multi-evaluation (i.e., an FFT). However, upon slightly closer inspection, we observe that it actually suffices to compute $c(c+1)/2$ FFTs (since the terms $e_0^i e_1^j$ and $e_0^j e_1^i$ share the same "coefficient" $a_i a_j$ in $\langle \mathbf{a} \otimes \mathbf{a}, \mathbf{e}_0 \otimes \mathbf{e}_1 \rangle$, hence the FFT can be evaluated on terms $e_0^i e_1^j + e_0^j e_1^i$ directly).

Blazing fast FFT. Our next observation is that the FFT over the group algebra $\mathbb{F}_4[\mathbb{G}]$ is actually *extremely* efficient. Indeed, given a polynomial $P(X_1, \cdots, X_n)$, one can rewrite P as

$$P_0(X_1, \cdots, X_{n-1}) \;+\; X_n P_1(X_1, \cdots, X_{n-1}) \;+\; X_n^2 P_2(X_1, \cdots, X_{n-1}).$$

Let us denote $\mathsf{FFT}(P, n)$ the functionality that evaluates P on all n-tuples over $(\mathbb{F}_4^\times)^n$, and outputs a multi-evaluation vector $\mathbf{v}$. By the above formula, computing $\mathsf{FFT}(P, n)$ reduces to

- computing $\mathbf{v}_i \leftarrow \mathsf{FFT}(P_i, n-1)$ for each $i \in \{0, 1, 2\}$, and
- setting $\mathbf{v} \leftarrow (\mathbf{v}_0 + \mathbf{v}_1 + \mathbf{v}_2 \mid\mid \mathbf{v}_0 + \theta\mathbf{v}_1 + (\theta+1)\mathbf{v}_2 \mid\mid \mathbf{v}_0 + (\theta+1)\mathbf{v}_1 + \theta\mathbf{v}_2)$.

Denoting $C(n)$ the cost of running $\mathsf{FFT}(P, n)$, we therefore have $C(n) = 3 \cdot C(n-1) + \ell \cdot 3^{n-1}$, where ℓ denotes the number of vector operations (naïvely, 6 additions of vectors and 4 scalar-vector products—but some additions and products can be reused). This yields a cost of $C(n) = n \cdot \ell \cdot 3^{n-1}$, where all operations are very fast: either additions of $\mathbb{F}_4$-vectors or multiplications by θ. Looking ahead, our implementation and evaluation (Section 7) confirm that, even with the straightforward recursive algorithm, the FFT results in minimal overhead compared to the cost of the DPFs.[10]

Stepping back: comparison with silent OT extension. To give an intuition about the efficiency of this construction, we provide a brief comparison with constructions of silent OT extension. In short, to get (say) 3^n OTs, these constructions run $c \cdot t$ DPFs on a domain of size $3^n/t$, followed by a multiplication with a compressive mapping. In the most efficient silent OT extension protocol to date [RRT23], this compressive mapping requires computing $21 \cdot c \cdot 3^n$ XORs, followed by 3^n XORs of random size-21 subsets of the bits of the resulting vector. Due to the overhead of many random memory accesses, the cost of computing this mapping dominates the overall runtime.

In contrast, we need $(c \cdot t)^2$ DPFs with domain size $3^n/t$, but get a 64× speedup from the early termination optimization. Using the parameters from our cryptanalysis (Section 6), we will have $c \cdot t = 60$. Hence, the cost of our DPFs should be essentially on par with that of [RRT23]. However, the FFT cost in our construction is largely dominated by the cost of the DPFs. Therefore, we expect (and this is confirmed by our implementation) that this PCG should produce $\mathbb{F}_4$-OLEs at a much faster rate compared with the rate at which [RRT23] produces OTs. In the two-party setting, when the goal is to generate Beaver triples over $\mathbb{F}_2$, we get an additional 2× speedup from the technique of Section 2.4, as we generate one triple from *one* $\mathbb{F}_4$-OLE (whereas [RRT23] requires two OTs). We provide an optimized implementation of our scheme and evaluate how it compares to previous works in Section 7. Our implementation is about 6× faster than the state of the art [RRT23].

2.6 Distributed seed generation

So far, we have only discussed the cost of expanding the PCG keys $(\mathsf{k}_0, \mathsf{k}_1)$. To obtain a full-fledged secure computation protocol, we need an efficient way for the parties to securely evaluate $\mathsf{PCG.Gen}$ procedure in a distributed fashion. In the following, as in all previous works on PCGs [BCG+19b, BCG+19a, BCG+20b, BCG+20a, CRR21, BCG+22, BCCD23, CD23], we assume that the noise follows a regular distribution. That is, a noise vector $\mathbf{e}$ is a vector of c polynomials $(e^1, \cdots e^c)$, where each polynomial e^i is *regular*: its coordinates are divided into t block of (approximately) equal length $3^n/t$, and it has a single nonzero coefficient in each block. For any integer h, let $[h]$ denote the set $\{1, \cdots, h\}$. The previous work of [BCG+20b] outlined the following methodology to securely distribute PCG seeds for generating D OLEs (in our context, $D = 3^n$):

- **Sampling the noise vectors.** Each party P_b generates its noise vector $\mathbf{e}_b$ locally, by sampling c t-sparse regular polynomials. We write $\mathbf{e}_b = (e_b^1, \cdots, e_b^c)$. For each $i \in [c]$, we let $(\mathsf{p}_{b,1}^i, \cdots, \mathsf{p}_{b,t}^i) \in [3^n/t]^t$ denote the t positions of the nonzero entries in e_b^i, and $(v_{b,1}^i, \cdots, v_{b,t}^i) \in \mathbb{F}_4^t$ denote the value of these nonzero coefficients.
- **Sharing the positions and values.** For every $i_0, i_1 \in [c]$, for every $j_0, j_1 \in [t]$, the parties run a distributed protocol with respective inputs $\mathsf{p}_{0,j_0}^{i_0}$ and $\mathsf{p}_{1,j_1}^{i_1}$ (i.e., the position of the j_0-th and j_1-th nonzero coefficients in $e_0^{i_0}$ and $e_1^{i_1}$, respectively) which securely computes bitwise shares of the $(j_0 + j_1)$-th nonzero coefficient of $e_0^{i_0} e_1^{i_1}$. In parallel, they also run a distributed protocol with

[10] Our implementation also exploits vectorized operations to perform a batch of multiple FFTs for essentially the cost of one, which further reduces the impact of FFTs on the overall runtime.

respective inputs $v^{i_0}_{0,j_0}$ and $v^{i_1}_{1,j_1}$ (the corresponding values of the nonzero coefficients) and securely compute bitwise shares of $v^{i_0}_{0,j_0} \cdot v^{i_1}_{1,j_1}$ (the value of the $(j_0 + j_1)$-th nonzero coefficient of $e^{i_0}_0 e^{i_1}_1$).

– **Distributing the DPF keys.** For every $i_0, i_1 \in [c]$, for every $j_0, j_1 \in [t]$, the parties run the Doerner-shelat protocol [Ds17] with their bitwise shares of the position and value to securely obtain DPF keys forming succinct shares of the point function $f_{\alpha,\beta}$ which evaluates to $\beta := v^{i_0}_{0,j_0} \cdot v^{i_1}_{1,j_1}$ on the index α of the $(j_0 + j_1)$-th nonzero coefficient of $e^{i_0}_0 e^{i_1}_1$, and to 0 on all other inputs.

Communication-wise, the Doerner-shelat protocol requires $2 \cdot \log(D/t)$ oblivious transfers for each DPF, for a total of $2(ct)^2 \log(D/t)$ oblivious transfers. Distributing the shares of the coefficients $v^{i_0}_{0,j_0} \cdot v^{i_1}_{1,j_1}$ is relatively straightforward: it involves two OLEs over $\mathbb{F}_4$ for each of the (ct^2) coefficients. As in [BCG$^+$20b], these OLEs can be obtained at a minimal cost by running the PCG in a "bootstrapping mode": whenever two parties use the PCG to generate D $\mathbb{F}_4$-OLEs, they can instead use a marginally larger instance to generate $D + (ct)^2$ $\mathbb{F}_4$-OLE, and store the $(ct)^2$ extra OLEs for use in the next distributed PCG seed generation.

In the work of Boyle et al. [BCG$^+$20b], an important overhead comes from the $(ct)^2$ instances of a distributed protocol to generate bitwise shares of the noise positions: each such instance requires securely running a Boolean adder to compute, from the bit decomposition of $\mathsf{p}^{i_0}_{0,j_0}$ and $\mathsf{p}^{i_1}_{1,j_1}$, the bit decomposition of the position of the corresponding entry in $e^{i_0}_0 e^{i_1}_1$. In the construction of [BCG$^+$20b], this contributes a large portion of the (communication and computation) overhead of the seed distribution procedure: about half of the communication, computation, and rounds of the full protocol.

An improved seed distribution from ternary DPFs. We now introduce an optimization that removes the need to distribute shares of noise positions altogether by working *directly* in the ternary basis. Our improved protocol is tailored to the setting of noise vectors with components over $\mathbb{F}_4[\mathbb{G}] = \mathbb{F}_4[X_1, \ldots, X_n]/(X_1^3 - 1, \ldots, X_n^3 - 1)$. Observe that every monomial over $\mathbb{F}_4[\mathbb{G}]$ can be written as $\mathbf{X}^{\mathsf{p}} := \prod_{i=1}^n X_i^{p_i}$, where $\mathsf{p} = (p_1, \ldots, p_n) \in \mathbb{F}_3^n$. Therefore, we can uniquely identify the position of the coefficient c_{p} of a monomial $\mathbf{X}^{\mathsf{p}}$ with the $\mathbb{F}_3$-vector $\mathsf{p} \in \mathbb{F}_3^n$. Now, consider the product of two polynomials e_0, e_1 known by P_0 and P_1, respectively. Let $\mathsf{p}_0 \in \mathbb{F}_3^n$ be the position of a nonzero entry in e_0, and $\mathsf{p}_1 \in \mathbb{F}_3^n$ be the position of a nonzero entry in e_1. Then, the corresponding nonzero entry in $e_0 e_1$ is the coefficient of the monomial $\mathbf{X}^{\mathsf{p}_0} \cdot \mathbf{X}^{\mathsf{p}_1} = \mathbf{X}^{\mathsf{p}_0 + \mathsf{p}_1 \bmod 3}$. That is, the corresponding nonzero position in $e_0 e_1$ is exactly $\mathsf{p}_0 + \mathsf{p}_1$ (where the sum is taken modulo 3). In other words, the two parties *already hold shares* of the noise position in $e_0 e_1$—but over the ternary basis!

Unfortunately, the Doerner-shelat protocol requires the parties to hold binary shares of the position, because its binary decomposition corresponds to the path from the root to the leaf in the (binary) GGM tree underlying the DPF construction of [BGI16, BGI15]. To remedy this situation, we modify the underlying DPF construction to use a *ternary tree*. That is, the full tree is obtained by computing the three children of a node by evaluating a length-tripling PRG $G : \{0,1\}^\lambda \to \{0,1\}^{3\lambda}$ on the node value. Adapting the DPF construction of [BGI15, BGI16] to this setting is relatively simple (though the security analysis becomes slightly more tedious, especially when adapting the Doerner-shelat protocol to work over a ternary basis), and requires increasing the number of correction words from 1 to 3 per level of the tree.[11] With this change, the path to a leaf is given directly by the leaf position written as a $\mathbb{F}_3$-vector. To securely generate the keys of this modified DPF, we adapt the Doerner-shelat protocol. Our adaptation requires two 1-out-of-3 oblivious transfers per level (instead of two 1-out-of-2 OTs as in [Ds17]), for the $\log_3(D/t)$ levels of the ternary DPF tree. In summary, we obtain a distributed seed generation protocol with the following pros-and-cons when compared to the original approach of [BCG$^+$20b]:

+ The parties "natively" hold shares of the nonzero positions and do not have to run a secure protocol to compute them. In the protocol of Boyle et al. [BCG$^+$20b], this step required $2(ct)^2 \cdot \log(D/t)$ oblivious transfers in $\log(D/t)$ rounds (i.e., half of the total number of rounds and OTs).

− The modified Doerner-shelat requires $2(ct)^2 \log_3(D/t)$ 1-out-of-3 OTs of 3λ-bit strings instead of $2(ct)^2 \log_2(D/t)$ 1-out-of-2 OTs of 2λ-bit strings, which represents slightly more communication and computation.

[11] Unfortunately, in the ternary tree construction, using the optimization described in [BGI16] for removing one extra correction word does not immediately apply. We leave open the problem of finding a similar optimization in the ternary case.

- − Due to the use of a ternary DPF, which has more correction words, the PCG seed size is slightly increased, by a factor ≈ 1.5.
- + Expanding the PCG seeds becomes about 20% faster because the total number of PRG evaluations is reduced when computing a full ternary tree compared to a full binary tree with a similar number of leaves.
- + The number of rounds of the Doerner-shelat protocol is also reduced, from $\log_2(D/t)$ to $\log_3(D/t)$, by having a shorter tree.

2.7 Concrete cryptanalysis of $\mathbb{F}_4$OLEAGE

The security of $\mathbb{F}_4$OLEAGE is based on the QA-SD assumption, as explained in Section 2.2. To derive our parameters, we conduct a precise analysis of attacks on $\mathsf{QA\text{-}SD}(c, t, \mathbb{G})$ over the finite field $\mathbb{F}_q$. The goal of the QA-SD assumption is to distinguish pairs $(a, \langle a, e\rangle + e_0) = ((a_1, \ldots, a_{c-1}), e_0 + a_1e_1 + \cdots + a_{c-1}e_{c-1})$ from random elements of $\mathbb{F}_q[\mathbb{G}]^c$ where $a_i \leftarrow \mathbb{F}_q[\mathbb{G}]$, and e_i are *sparse* elements of $\mathbb{F}_q[\mathbb{G}]$, with Hamming weight t. In its search version, QA-SD is equivalent to solving the following decoding problem:

$$\left(\mathbf{I}\ \mathbf{A}_1\ \ldots\ \mathbf{A}_{c-1}\right)\begin{pmatrix}\mathbf{e}_0\\ \vdots\\ \mathbf{e}_{c-1}\end{pmatrix} = \mathbf{s}$$

where the matrix $\mathbf{A}_i$ represents the multiplication by an element a_i, with respect to the basis given by $\mathbb{G}$ (for any ordering), and $\mathbf{e}_i$ are sparse vectors of weight t. To attack our instance, we consider state-of-the-art generic decoding algorithms, as well as structural attacks.

Specificity of our instance. In $\mathbb{F}_4$OLEAGE, we use very particular parameters: we consider the parity-check matrix of a code of length $c|\mathbb{G}|$, and dimension $(c-1)|\mathbb{G}|$, with $|\mathbb{G}| = (q-1)^n = 3^n$ in the millions, together with a sparse vector of errors (typically, there are $ct < 100$ errors). The problem of decoding when the number of errors is sublinear in the size of the code was studied in [CTS16], and the authors showed that in this case, the costs of the best generic algorithm for decoding, namely the ISD algorithm, are all of the form $2^{a(ct)(1+o(1))}$, with a depending only of the code rate $1 - 1/c$, even the seminal ISD algorithm by Prange [Pra62]. Consequently, any additional optimizations developed based on the Prange algorithm proved to be futile within this specific context.

However, this instance of the decoding problem is far from being generic, and there are known ways in which an attacker can exploit the internal structure. First, since the code is by definition kept invariant under the action of $\mathbb{G}$, they can use the *Decoding One Out of Many* (DOOM) strategy from [Sen11] to get a $\sqrt{|\mathbb{G}|}$ speed-up in *any* decoding algorithm. Beyond this approach, an attacker can try to reduce their sample modulo some ideal of $\mathbb{F}_q[\mathbb{G}]$, yielding a new instance of the decoding problem with smaller parameters. In general, this action can significantly increase the noise rate which may render the problem intractable. Nevertheless, this growth can be mitigated by considering reductions arising from quotients of $\mathbb{G}$ by some subgroup $\mathbb{H}$. The key observation here is that this operation, known as *folding* with respect to $\mathbb{H}$, maps a code of length $c|\mathbb{G}|$ onto a code of length $c|\mathbb{G}/\mathbb{H}|$, while keeping the code-rate invariant. It was first introduced in the context of the cryptanalysis of highly structured variants of McEliece's code-based encryption scheme [FOP+16, CT19], and later considered in [BCCD23] for the analysis of QA-SD. It works as follows: upon receiving a sample of QA-SD, an attacker can pick some sufficiently large subgroup $\mathbb{H}$ and compute the folding of both the received code and the syndrome to get a smaller instance of QA-SD, with respect to the subgroup $\mathbb{G}/\mathbb{H}$. On the folded instance, the noise is upper bounded by the original noise, but might be slightly decreased due to the existence of some collisions. The attacker can then run a generic decoder on this smaller instance. If they want to actually decode the original instance, they need to lift all the putative solutions back to the initial decoding problem. In short, this folding process may be compared to a sub-ISD routine inside a generic decoding algorithm. However, $\mathbb{F}_4$OLEAGE only relies on the *decisional* variant of QA-SD, and therefore getting a solution on the folded instance is enough to break the security. There is a caveat, though. Indeed, for the folded instance to be meaningful, its length should not be too small relative to the noise level. More precisely, if the target decoding distance is beyond the Gilbert-Varshamov (GV) bound, the folded instance will typically have exponentially many solutions, *even* when the original sample came from the uniform distribution [Deb23]. This does not provide the attacker with any distinguishing advantage.

Improvement over the previous approach. In Section 6, we provide a precise analysis of the probability distribution of the error weight on the folded instance. This allows us to give a more powerful attack than that of [BCCD23], taking more advantage of our specific QA-SD instance. Indeed, the group used in our construction, namely $\mathbb{G} := (\mathbb{Z}/(q-1)\mathbb{Z})^n$, has a huge number of subgroups of any given order. For example, in $\mathbb{F}_4$OLEAGE we set $q = 4$, and the subgroups of $\mathbb{G}$ of order 3^k are in bijection with the k-dimensional vector subspaces of $\mathbb{F}_3^n$. Their number is given by the Gaussian binomial coefficient $\binom{n}{k}_3$. At a high level, our attack strategy that makes use of this fact is the following:

1. Pick some subgroup $\mathbb{H}$ and compute the folding with respect to $\mathbb{H}$;
2. Guess that the folded error actually has a Hamming weight ω much smaller than expected;
3. Apply the best decoding algorithm corresponding to the size and level of noise;
4. Abort if the number of operations is greater than would be required for the pair (level of noise, algorithm), and return to Step 1.

The success probability of this procedure is given by the probability that the weight of the error meets the attacker's guess. In other words, this attack would cost, on average over the uniform choice of subgroups $\mathbb{H}$ of a given order,

$$\frac{\mathrm{Cost}_{\mathrm{Decoding}}(\omega) + \mathrm{Cost}_{\mathrm{Folding}}}{\Pr\left[\, w_H(\mathrm{Fold}_{\mathbb{H}}(x)) = \omega \,\right]},$$

where $\mathrm{Cost}_{\mathrm{Decoding}}(\omega)$ indicates the complexity of the best decoding algorithm, for a noise level of ω, and $\mathrm{Cost}_{\mathrm{Folding}}$ is the cost of the folding operator (which is in fact the length of the original code). In order to get the most out of this attack, it suffices to choose ω which minimizes this ratio.

Concrete Parameters. We provide a SageMath [S+24] script to determine a set of concrete parameters. It is available on GitHub.[12] Using the script, we estimate that taking $c = 5, t = 14$ offers more than 128 bits of security, while being quite conservative on the power afforded to the adversary.

	(c, t)	**Security level**
Conservative	$(5, 14)$	139
	$(6, 11)$	142
	$(6, 10)$	122
Medium	$(5, 13)$	128
Aggressive	$(6, 9)$	128
	$(5, 12)$	128

Table 2: Parameter estimation for $\mathbb{F}_4$OLEAGE for different values of c, t, and the corresponding security level estimated with our provided script. Those parameters are valid for any $n \in \{14, 15, 16, 17, 18\}$. Conservative parameters take into account the c-split structure and potential improvement on some polynomial factors in decoding algorithms. Aggressive parameters do not take into account the c-split structure and precise count of the polynomial factors described in Appendix C. Some improvements in ISD algorithms might yield a decrease in security.

Update on the parameters given by Bombar et al. [BCCD23]. In [BCCD23], the authors propose choosing $c = 4, t = 16$ when generating 2^{25} OLEs. We find that these parameters are too optimistic, as the attack described above manages to downgrade the security to about 100 bits as opposed to the claimed 128 bits of security. Refer to Section 6.4 for the full attack description. To use the same compression factor $c = 4$ (as suggested in [BCCD23]), we need to increase the noise to $t = 20$, leading to an overall noise equal to $ct = 80$. Such a noise level is somewhat larger than the $ct = 64$ given in the previous work. Taking $c = 5$ leads to a total weight of 70, notably lowering this gap.

[12] The code is available at `https://github.com/mbombar/estimator_folding`.

3 Preliminaries

In this section, we cover the basic notation and cryptographic preliminaries that we use throughout the paper.

3.1 Notation

We use $\mathbb{F}_4$ to denote the Galois field of order 4. For $\mathbb{G}$ an abelian group, we denote by $\mathbb{F}_q[\mathbb{G}]$ the corresponding group algebra. For an integer n and a polynomial $f \in \mathbb{F}_q[\mathbb{G}]$ with n variables, $\mathsf{Eval}_n(f)$ denote the full evaluations of f over $(\mathbb{F}_q^\times)^n$. We let $[N]$ denote the set $\{1, 2, \ldots, N\}$. Divisibility is denoted as $a \mid b$, to mean a divides b. The number of elements in a list L is denoted as $|L|$. For a vector $\mathbf{e} \in \mathbb{F}_q^n$, we denote by $w_H(\mathbf{e})$ the associated Hamming weight. More generally, the Hamming weight of an element of a finite-dimensional $\mathbb{F}_q$-algebra $\mathcal{R}$ is the weight of the vector formed by its coefficients in some basis (in general, $\mathcal{R} = \mathbb{F}_q[\mathbb{G}]$ and we consider a basis given by an arbitrary ordering of the elements of $\mathbb{G}$.[13]). For an integer t, we denote by $\mathcal{R}_t$ the subset of elements of Hamming weight t. We denote by $\mathsf{poly}(\cdot)$ any polynomial and by $\mathsf{negl}(\cdot)$ any negligible function. We use $x \leftarrow_R S$ to denote a uniformly random sample drawn from S, and $x \leftarrow \mathcal{A}$ to denote assignment from a possibly randomized algorithm Adv. We use $x := y$ to denote the initialization of a value x to the value of y. We use $A \simeq B$ to indicate that two sets are isomorphic. By an *efficient* algorithm $\mathcal{A}$ we mean that Adv is modeled by a (possibly non-uniform) Turing Machine that runs in probabilistic polynomial time. We write $D_0 \approx_c D_1$ to mean that two distributions D_0 and D_1 are *computationally* indistinguishable to all efficient distinguishers $\mathcal{D}$ and $D_0 \approx_s D_1$ to mean that D_0 and D_1 are *statistically* indistinguishable.

Vectors and tensor products. We denote vectors using bold lowercase letters. For two vector $\mathbf{u} = (u_1, \ldots, u_t), \mathbf{v} = (v_1, \ldots, v_t) \in R^t$ for some ring R, their tensor product $\mathbf{u} \otimes \mathbf{v}$ is defined by $\mathbf{u} \otimes \mathbf{v} = (u_i \cdot v_j)_{i,j \leq t} = (v_1 \cdot \mathbf{u}, \ldots, v_t \cdot \mathbf{u})$ and we denote by $\langle \mathbf{u}, \mathbf{v} \rangle$ their inner product. Similarly, we write $\mathbf{u} \boxplus \mathbf{v}$ to denote the outer sum of a vector, equal to $\mathbf{u} \boxplus \mathbf{v} = (u_i + v_j)_{i,j \leq t} = (v_1 + \mathbf{u}, \ldots, v_t + \mathbf{u})$. We let $\mathbf{u}[i]$ denote the value of index i in $\mathbf{u}$.

3.2 Function secret sharing

Function secret sharing (FSS), introduced in [BGI15,BGI16], allows a dealer to succinctly secret share a function with two parties. An FSS scheme splits a secret function $f : \mathcal{D} \to \mathbb{G}$, where $\mathbb{G}$ is some Abelian group into *keys* K_0, K_1 that can be used by party $\sigma \in \{0, 1\}$ to evaluate the function on an input $x \in \mathcal{D}$ and obtain the share $[\![f(x)]\!]_\sigma$ of the result. We focus on FSS for *point functions* which are known as Distributed Point Functions (DPFs).

Distributed Point Functions. Let $\mathcal{D}$ be an input domain and $\mathbb{G}$ be an Abelian group. A *point function* $P_{\alpha,\beta} : \mathcal{D} \to \mathbb{G}$ is a function that evaluates to message $\beta \in \mathbb{G}$ on a single input $\alpha \in \mathcal{D}$, and evaluates to $0 \in \mathbb{G}$ on all other inputs $x \neq \alpha \in \mathcal{D}$. A *distributed* point function (Definition 1) is a point function that is encoded into a pair of keys. Each key can be used to obtain an additive *secret-share* of the point function $P_\alpha(x)$, for any input $x \in \mathcal{D}$.

Definition 1 (Distributed Point Function (DPF) [GI14,BGI16]). *Let λ be the security parameter, $\mathcal{D}$ be an input domain, and $\mathbb{G}$ be an Abelian group. A* DPF *scheme (with a full-domain evaluation procedure) consists of a tuple of efficient algorithms* $\mathsf{DPF} = (\mathsf{Gen}, \mathsf{FullEval})$ *with the following syntax.*

- $\mathsf{DPF.Gen}(1^\lambda, 1^n, \alpha, \beta) \to (K_0, K_1)$*. Takes as input a security parameter, a domain size n, and index $\alpha \in \mathcal{D}$ and a payload $\beta \in \mathbb{G}$. Outputs two evaluation keys K_0 and K_1.*
- $\mathsf{DPF.FullEval}(\sigma, K_\sigma) \to \mathbf{v}_\sigma$*. Takes as input the party index σ and an evaluation key K_σ. Outputs a vector $\mathbf{v}_\sigma$.*

These algorithms must satisfy correctness, security, and efficiency:

Correctness. *A DPF is said to be* correct *if for all $\alpha \in \mathcal{D}$, all $\beta \in \mathbb{G}$, and all pairs of keys generated according to* $\mathsf{DPF.Gen}(1^\lambda, 1^n, \alpha, \beta)$*, the sum of the individual outputs from* $\mathsf{DPF.FullEval}$ *result in the one-hot basis vector scaled by the message β,*

$$\Pr\left[\ \mathsf{FullEval}(0, K_0) + \mathsf{FullEval}(1, K_1) = \beta \cdot \mathbf{e}_{\boldsymbol{\alpha}}\ \right] = 1,$$

[13] The Hamming weight does not depend on the ordering of the elements of $\mathbb{G}$.

where $\mathbf{e}_{\boldsymbol{\alpha}} \in \mathbb{G}^{|\mathcal{D}|}$ *is the* α*-th basis vector.*

Security. *A DPF is said to be* secure *if each individual evaluation key output by* $\mathsf{DPF.Gen}$ *leaks nothing about* (α, β) *to a computationally bounded adversary. Formally, there exists an efficient simulator* $\mathcal{S}$ *such that* $K_\sigma \approx_c \mathcal{S}(1^\lambda, 1^n, \sigma)$*, where* $\approx_c$ *denotes the computational indistinguishability of distributions.*

Efficiency. *A DPF is said to be* efficient *if the size of each key is sublinear in the domain size. That is, for all* $\sigma \in \{0,1\}$*,* $|K_\sigma| = |\mathcal{D}|^\epsilon$ *for some* $\epsilon < 1$*.*

FSS for the sum of point functions. We let SPFSS be an FSS scheme for the class of *sums of point functions*: functions of the form $f(x) = \sum_i f_{s_i,y_i}(x)$ where each $f_{s_i,y_i}(\cdot)$ evaluates to y_i on s_i, and to 0 everywhere else. As in previous works, we will use efficient constructions of SPFSS in our constructions of PCGs.

3.3 The Quasi-Abelian Syndrome Decoding Problem

In this section, we recall the *Quasi-Abelian Syndrome Decoding* assumption ($\mathsf{QA\text{-}SD}$) which was introduced in [BCCD23]. It can be seen as a generalization of both the plain *Syndrome Decoding* (SD) assumption, when the group is $\{1\}$, as well as the quasi-cyclic syndrome decoding assumption. Let $\mathbb{G}$ be a finite abelian group. The group algebra of $\mathbb{G}$ with coefficients in the finite field $\mathbb{F}_q$ is the set of formal linear combinations $\left\{\sum_{g\in\mathbb{G}} a_g g \mid a_g \in \mathbb{F}_q\right\}$, which is an $\mathbb{F}_q$-vector space of dimension $|\mathbb{G}|$, endowed with the convolution product:

$$\left(\sum_{g\in\mathbb{G}} a_g g\right)\left(\sum_{g\in\mathbb{G}} b_g g\right) := \sum_{g\in G}\left(\sum_{h\in\mathbb{G}} a_h b_{hg^{-1}}\right) g.$$

It can be seen that this product is commutative. The *Hamming* weight $w_H(a)$ of an element $a \in \mathbb{F}_q[\mathbb{G}]$ is the number of its non zero coordinates in the basis $(g)_{g\in\mathbb{G}}$. This is a well-defined notion since it does not depend on the ordering of the elements of $\mathbb{G}$.

Recall that a finite abelian group is nothing more than a direct product of cyclic groups:

$$\mathbb{G} \simeq \mathbb{Z}/d_1\mathbb{Z} \times \cdots \times \mathbb{Z}/d_r\mathbb{Z},$$

where the d_i's can be equal. Then, the group algebra $\mathbb{F}_q[\mathbb{G}]$ admits an explicit description as some particular multivariate polynomial ring:

$$\mathbb{F}_q[\mathbb{G}] \simeq \mathbb{F}_q[X_1, \ldots, X_r]/(X_1^{d_1} - 1, \ldots, X_r^{d_r} - 1),$$

where the isomorphism is given by $(k_1, \ldots, k_r) \mapsto X_1^{k_1} \cdots X_r^{k_r}$, and extended by linearity. We are now ready to define the main hard problem, which can be stated as a search and a decisional variant.

Definition 2 ((Search) $\mathsf{QA\text{-}SD}(q, c, t, \mathbb{G})$). *Let* $\mathbb{G}$ *be a finite abelian group,* $\mathbb{F}_q[\mathbb{G}]$ *its algebra with coefficients in the finite field* $\mathbb{F}_q$*, and let* $c \geq 2$ *be some constant integer called the compression factor. Given a target Hamming weight* $t \in \{1, \ldots, |\mathbb{G}|\}$ *and a probability distribution* Φ_t *which outputs elements* $x \in \mathbb{F}_q[\mathbb{G}]$ *such that* $\mathbb{E}(w_H(x)) = t$*. Given access to a pair of the form* $(\mathbf{a}, \langle \mathbf{a}, \mathbf{e}\rangle + e_0)$ *where* $\mathbf{a}$ *is uniformly distributed over* $\mathbb{F}_q[\mathbb{G}]^{c-1}$ *and* $\mathbf{e}' := (e_0, \mathbf{e}) \in \mathbb{F}_q[\mathbb{G}]^c$ *is formed by independent elements distributed according to* Φ_t*, the goal is to recover the error term* $\mathbf{e}'$*.*

Definition 3 ((Decisional) $\mathsf{QA\text{-}SD}(q, c, t, \mathbb{G})$). *Let* $\mathbb{G}$ *be a finite abelian group,* $\mathbb{F}_q[\mathbb{G}]$ *its algebra with coefficients in the finite field* $\mathbb{F}_q$*, and let* $c \geq 2$ *be some constant integer called the compression factor. Given a target Hamming weight* $t \in \{1, \ldots, |\mathbb{G}|\}$ *and a probability distribution* Φ_t *which outputs elements* $x \in \mathbb{F}_q[\mathbb{G}]$ *such that* $\mathbb{E}(w_H(x)) = t$*, the* Quasi-Abelian Syndrome Decoding *problem asks to distinguish, with a non-negligible advantage, between the distributions:*

$$\mathcal{D}_0 : \quad \left((a^{(i)})_{i\in\{1,\ldots,c-1\}}, u\right) \quad \textit{where } a^{(i)}, u \xleftarrow{\$} \mathbb{F}_q[\mathbb{G}]$$

$$\mathcal{D}_1 : \left((a^{(i)})_{i\in\{1,\ldots,c-1\}}, \sum_{i=1}^{c-1} a^{(i)} e_i + e_0\right) \quad \textit{where } a^{(i)} \xleftarrow{\$} \mathbb{F}_q[\mathbb{G}] \textit{ and } e_i \xleftarrow{\$} \Phi_t.$$

We say that the $\mathsf{QA\text{-}SD}(q, c, t, \mathbb{G})$ *assumption holds when this problem is hard for every non-uniform polynomial time distinguisher.*

Remark 4. When it is clear from the context, we might drop the dependency in q and simply write $\mathsf{QA\text{-}SD}(c, t, \mathbb{G})$.

In general we consider Φ_t to output uniform elements of Hamming weight t, or a *regular* variant where the basis $(g)_{g\in\mathbb{G}}$ is split into t blocks of size $|\mathbb{G}|/t$ (except maybe the last one) such that each block contains exactly one t.

Relation to linear codes. Fix an ordering of the elements of $\mathbb{G}$ and for every element $a \in \mathbb{F}_q[\mathbb{G}]$, denote by M_a the $|\mathbb{G}| \times |\mathbb{G}|$ matrix representing the multiplication by a. A code having a parity-check matrix formed by multiple blocks of the form M_a is known as a *quasi-abelian* code of group $\mathbb{G}$ (or a quasi-$\mathbb{G}$ code). Formally, it is an $\mathbb{F}_q[\mathbb{G}]$-submodule of the free module $\mathbb{F}_q[\mathbb{G}]^\ell$ for some integer $\ell > 0$. When $\mathbb{G} = \{1\}$, then $\mathbb{F}_q[\mathbb{G}]$ is nothing but the finite field $\mathbb{F}_q$, and therefore a quasi-$\{1\}$ code is simply an $\mathbb{F}_q$-linear code. On the other hand, when $\mathbb{G} = \mathbb{Z}/n\mathbb{Z}$ is cyclic, then $\mathbb{F}_q[\mathbb{G}] \simeq \mathbb{F}_q[X]/(X^n-1)$. Therefore, quasi-$\mathbb{Z}/n\mathbb{Z}$ codes are exactly *quasi-cyclic* codes, at the core of many code-based cryptosystems such as Nist Round 4 candidates BIKE [AAB+22a] and HQC [AAB+22b]. We refer to [BCCD23, Section 4] for more information on quasi-abelian codes.

Now, a sample $(\mathbf{a}, \langle \mathbf{a}, \mathbf{e}\rangle + e_0)$ corresponds to a pair $(\mathbf{H}, \mathbf{He'})$ where

$$\mathbf{H} = \left[\mathbf{I}_{|\mathbb{G}|}\ \mathbf{M}_{a_1} \cdots \mathbf{M}_{a_{c-1}}\right]$$

is by definition, a parity-check matrix of some random quasi-Abelian code of rate $1-1/c$, in systematic form, and $\mathbf{e'} = (e_0, \mathbf{e})$ is an error vector of length $c|\mathbb{G}|$ and weight ct, with a c-split structure (which is standard when dealing with structured variants of the decoding problem). In other words, the search version of $\mathsf{QA\text{-}SD}$ corresponds to a structured variant of the decoding problem of random quasi-abelian codes.

4 A Fast PCG for $\mathbb{F}_4$-OLEs

In this section, we present the construction of $\mathsf{QA\text{-}SD}_{\mathsf{OLE}}$ over $\mathbb{F}_4$ (Fig. 1) following optimizations via early termination and fast evaluation of polynomials for FFT.

4.1 PCGs over $\mathbb{F}_4$ from the QA-SD assumption

The general description of the framework based on it can be found in Appendix A. In [BCCD23], the authors point out that their $\mathsf{QA\text{-}SD}_{\mathsf{OLE}}$ construction is the first to produce a large number of OLE correlations over $\mathbb{F}_q$, for any $q \geq 3$. They propose using $\mathbb{G} = \prod_{i=1}^n \mathbb{Z}/(q-1)\mathbb{Z}$, $q \geq 3$. The direct consequence of this is that $\mathbb{F}_q[\mathbb{G}] \simeq \mathbb{F}_q[X_1, \dots, X_n]/(X_1^{q-1} - 1, \dots, X_n^{q-1} - 1) \simeq \prod_{i=1}^D \mathbb{F}_q$, where the last isomorphism equivalence comes from the Chinese Remainder Theorem. Above, $D = (q-1)^n$ is the number of elements in the group, and the number of OLE we can get over $\mathbb{F}_q$ by applying this isomorphism. Looking closely, we instantiate our particular PCG over $\mathcal{R} \simeq \mathbb{F}_4[X_1, \dots, X_n]/(X_1^3 - 1, \dots, X_n^3 - 1)$, by setting $q = 4$.

At the end of the protocol $\mathsf{QA\text{-}SD}_{\mathsf{OLE}}$, the parties obtain one OLE over $\mathcal{R}$; a general description of the construction is given in Fig. 14. Let us denote (x_σ, z_σ) the output of party σ. To obtain many OLE's over $\mathbb{F}_4$, the parties have to evaluate $x_\sigma, z_\sigma \in \mathbb{F}_4[X_1, \dots, X_n]/(X_1^3 - 1, \dots, X_n^3 - 1)$ over the full domain $(\mathbb{F}_4^\times)^n$. The standard approach is to use a Fast Fourier Evaluation to efficiently obtain this result. Here, we remark that, in our group algebra, fast multiplication *also* requires FFT, first in a multi-evaluation form, and then in the interpolation form. Therefore, doing the interpolation again is wasteful as in the end we will evaluate again after interpolating. As such, we can avoid the intermediate steps of multi-evaluation-then-interpolation and work directly with the evaluations, without coming back to $\mathcal{R}$. That is, we do not construct the polynomials x_σ, z_σ over $\mathbb{F}_4[X_1, \dots, X_n]/(X_1^3 - 1, \dots, X_n^3 - 1)$ but instead, we focus directly on the polynomials evaluations.

Let $\mathsf{Eval}_n(f) = \{f(x_1, \dots, x_n), (x_1, \dots, x_n) \in \{1, \theta, \theta + 1\}^n\}$ be the set of all the possible evaluations. Instead of giving the parties the description of the coefficients of the polynomials $a_i \in \mathbf{a}$, we can give them the vectors of all the evaluations of all the polynomials, that is giving them $\mathsf{Eval}_n(a_i)$, for all i. Because we can write $x_\sigma = e_\sigma^0 + e_\sigma^1 a_1 + \cdots + e_\sigma^{c-1} a_{c-1}$, it follows that all the evaluations of

x_σ can be obtained from $\mathsf{Eval}_n(e^i_\sigma)$ and $\mathsf{Eval}_n(a_i)$. All that remains is to evaluate the e^i_σ polynomials. They are sparse polynomials, and therefore their evaluations can be computed very efficiently i.e., if the polynomials have t non-zero coefficients, then the cost of the evaluation is linear in $t \cdot 3^n$. As a result, we can obtain $\mathsf{Eval}_n(x_\sigma)$ for a cost linear in 3^n.

The computation of $\mathsf{Eval}_n(z_\sigma)$ is a little trickier. As mentioned above, $x_0 \cdot x_1$ can be seen as a function of degree 2 in $(\mathbf{e_0}, \mathbf{e_1})$, with constant coefficients depending solely from $\mathbf{a} \otimes \mathbf{a}$. Because $\mathsf{Eval}_n(a_i)$ is already given to the parties, the evaluation of the coefficient from $\mathbf{a} \otimes \mathbf{a}$ can be obtained using only c^2 multiplications. It remains to compute the evaluations of the additive shares of the polynomials $e^i_0 \cdot e^j_1$. There are c^2 such polynomials shared among the parties, and we can view each share as a random polynomial. Therefore, each party has to compute the evaluation of c^2 random polynomials. This is a crucial part of the scheme and we devote the next section to it. Fig. 1 represents the PCG framework tailored to our setting, its correctness and security are followed by Theorem 19 and is stated in the below proposition.

Proposition 5. *Let* $\mathcal{R} = \mathbb{F}_4[\mathbb{G}] = \mathbb{F}_4[X_1, \ldots, X_n]/(X_1^3 - 1, \ldots, X_n^3 - 1)$ *where* $\mathbb{G} = \prod_{i=1}^n \mathbb{Z}/3\mathbb{Z}$ *is an Abelian group. Assume that* SPFSS *is a secure* FSS *scheme for sums of point functions and that the* $\mathsf{QA\text{-}SD}(q, c, t, \mathbb{G})$ *assumption holds for regular noise distribution. Then there exists a generic scheme to construct a* PCG *to produce one* OLE *correlation (described on Fig. 1). If the* SPFSS *is based on a* $\mathsf{PRG} : \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}$ *via the* PRG*-based construction from [BGI16], we obtain:*

- *Each party's seed has maximum size around :* $(c \cdot t)^2 \cdot ((n \cdot \log(3) - \log t + 1) \cdot (\lambda + 2) + \lambda + 2) + c \cdot t \cdot (n \cdot \log(3) + 2)$ *bits.*
- *The computation of* Expand *can be done with at most* $\log(3) \cdot (2 + \lfloor (2)/\lambda \rfloor) \cdot n \cdot c^2 \cdot t$ PRG *operations, and* $O(n \cdot \log(3) \cdot c^2 \cdot 3^n)$ *operations in* $\mathbb{F}_4$.

Proof. The proof follows immediately from Theorem 19 (in Appendix A) and the analysis of [BCCD23]. □

4.2 Optimizing the FSS evaluation via early termination

We remark that we can use a very simple trick that enables the parties to obtain the evaluation of their MPFSS shares 64 times faster than with the standard construction (*and* at a slight reduction in communication). The trick comes from the fact that the standard construction of the DPF based on the GGM tree implies that each leaf is of size $\lambda = 128$ bits. It was pointed out in [BGI16] that we can consider *early termination* in the case of small outputs. In our case, we would like a single leaf to encode a value in $\mathbb{F}_4$. This only requires 2 bits instead of the 128 bits we get as output, making the naïve evaluation "waste" 126 bits of the output. Instead, we can avoid wasting computation by truncating the tree 6 levels earlier and setting the value of the new 128-bit leaf on the special path to encode a unit vector consisting of zeroes except on the exact 2 bits where it equals to the correct value of $\mathbb{F}_4$ element. This essentially involves "hard-coding" the end of the path into the leaf directly, as illustrated in Section 4.2. Using this idea, we reduce the computational cost of evaluating the DPF by 64× and reduce the communication costs (key size of the DPF) by roughly $6 \cdot 128$ bits [BGI16]. This simple trick was initially introduced in the context of PIR applications [BGI16], but could not be applied to prior PCG constructions until now since all PCG constructions (except for the recent PCG construction of Bombar et al. [BCCD23]) required the DPF output to be encode elements of a large field. Similarly, in silent OT extension protocols [BCG+19b, BCG+19a, CRR21, BCG+22, RRT23], which are also bottlenecked by DPF evaluations, this optimization could not be applied because there, the DPF is used to output "authenticated" shares of a (potentially small) field element with a (large) MAC, which requires the leaves to encode 128 bit output value.

4.3 Fast evaluation over $\mathbb{F}_4[X_1, \ldots, X_n]/(X_1^3 - 1, \ldots, X_n^3 - 1)$

The high-level idea. Given a polynomial P with n variables, the party wants to compute $\mathsf{Eval}_n(P)$, that is, to evaluate P over $\left(\mathbb{F}_4^\times\right)^n$ where $\mathbb{F}_4^\times = \{1, \theta, \theta + 1\}$. Here, we adapt the standard divide-and-conquer style algorithm to our case, see for example the seminal work of [CT65]. Remark that

$$P(X_1, \ldots, X_n) = P_0(X_1, \ldots, X_{n-1}) + X_n P_1(X_1, \ldots, X_{n-1}) + X_n^2 P_2(X_1, \ldots, X_{n-1}).$$

Specific construction of $\mathsf{QA\text{-}SD_{OLE}}$ for $\mathbb{F}_4$OLEAGE

Parameters: Noise weight $t = t(\lambda)$, compression factor c, ring $\mathcal{R} = \mathbb{F}_4[X_1, \ldots, X_n]/(X_1^3 - 1, \ldots, X_n^3 - 1)$. A SPFSS scheme SPFSS = (SPFSS.Gen, SPFSS.FullEval) for sums of t^2 point functions, with domain $[0 \ldots 3^n)$ and range $\mathbb{F}_4$.

Public Input: $c - 1$ vectors of length 3^n over $\mathbb{F}_4$, corresponding to the result of $\mathsf{Eval}_n(a_i)$, for uniformly random $a_1, \cdots, a_{c-1} \in \mathcal{R}$, therefore the full evaluation of the c elements a_i.

```
PCG.Gen($1^\lambda$):
foreach $\sigma \in \{0,1\},\ i \in [0 \ldots c)$:
    Sample random $\mathbf{p_\sigma^i} \leftarrow \{(\mathbf{p^i_{\sigma,1}}, \ldots, \mathbf{p^i_{\sigma,t}}) \mid \mathbf{p^i_{\sigma,j}} \in \mathbb{F}_3^n\}$, and $\mathbf{v_\sigma^i} \leftarrow (\mathbb{F}_4^\times)^t$.
        ▷ $\mathbf{p}_\sigma^i$ can be sampled from regular noise distribution.
foreach $i, j \in [0 \ldots c)$:
    Sample FSS keys $(K_0^{i,j}, K_1^{i,j}) \xleftarrow{\$}$ SPFSS.Gen$(1^\lambda, 1^n, \mathbf{p_0^i} \boxplus \mathbf{p_1^j}, \mathbf{v_0^i} \otimes \mathbf{v_1^j})$.
        ▷ SPFSS is for sum of only $t$ point functions with domain $[0, \ldots, 3^{n/t})$ under regular noise.
Let $\mathsf{k}_\sigma = ((K_\sigma^{i,j})_{i,j \in [0 \ldots c)}, (\mathbf{p_\sigma^i}, \mathbf{v_\sigma^i})_{i \in [0 \ldots c)})$.
Output $(\mathsf{k}_0, \mathsf{k}_1)$.

PCG.Expand($\sigma, \mathrm{K}_\sigma$):
Parse $\mathsf{k}_\sigma$ as $((K_\sigma^{i,j})_{i,j \in [0 \ldots c)}, (\mathbf{p_\sigma^i}, \mathbf{v_\sigma^i})_{i \in [0 \ldots c)})$.
foreach $i \in [0 \ldots c)$:
    Define over $\mathbb{F}_4$ the polynomial:
        $e_\sigma^i(X) = \sum_{j \in [0 \ldots t)} \mathbf{v_\sigma^i}[j] \cdot \mathbf{X}^{\mathbf{p_\sigma^i}[j]}.$
    Compute $\mathsf{Eval}_n(e_\sigma^i)$.
Compute $x_\sigma = \langle \mathbf{a}, \mathbf{e_\sigma} \rangle$, where $\mathbf{a} = (1, a_1, \cdots, a_{c-1}), \mathbf{e_\sigma} = (e_\sigma^0, \cdots, e_\sigma^{c-1})$.
From $\mathsf{Eval}_n(e_\sigma^i)$ and $\mathsf{Eval}_n(a_i)$, compute $\mathsf{Eval}_n(x_\sigma)$.
foreach $i, j \in [0 \ldots c)$,
    Compute $u_{\sigma, i+cj} \leftarrow$ SPFSS.FullEval$(\sigma, K_\sigma^{i,j})$ and view it as a $c^2$ vector $\mathbf{u_\sigma}$ of elements in $\mathcal{R}$.
foreach $j \in [0 \ldots c^2)$:
    Compute $\mathsf{Eval}_n(u_{\sigma,j})$.    ▷ Optimization: only need to perform $c(c+1)/2$ FFTs, see Section 2.5.
Compute $\mathsf{Eval}_n(z_\sigma)$, with $z_\sigma = \langle \mathbf{a} \otimes \mathbf{a}, \mathbf{u_\sigma} \rangle$.
Output $(\mathsf{Eval}_n(x_\sigma), \mathsf{Eval}_n(z_\sigma))$.
```

Fig. 1: QA-SD-based PCG for OLE over $\mathcal{R}$ from evaluations of functions.

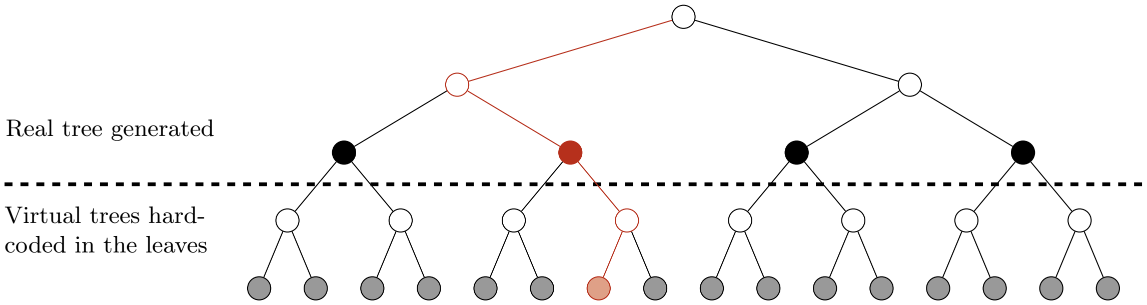

Fig. 2: Early termination example in the case we truncate only two steps earlier. Solid black nodes represent "zero" leaves, whereas solid red leaves can take on any value.

Instead of classically dividing our problem into 2 sub-problems, we divide it into 3 sub-problems. This is a ternary generalization of a standard FFT algorithm adapted to our case. Then,

$$\mathsf{Eval}_n(P) = \mathsf{Eval}_{n-1}(P_0) \cup X_n \mathsf{Eval}_{n-1}(P_1) \cup X_n^2 \mathsf{Eval}_{n-1}(P_2). \tag{1}$$

Denote by $\mathcal{C}(\mathsf{Eval}_n(P))$ the number of operations carried out to obtain all the 3^n evaluations on the set $\mathbb{F}_4^\times$. Then we have $\mathcal{C}(\mathsf{Eval}_n(P)) = 3\mathcal{C}(\mathsf{Eval}_n(P)) + 2 \cdot 3^n$, which leads us to $\mathcal{C}(\mathsf{Eval}_n(P)) = 4 \cdot n \cdot 3^n$. The concrete number of additions or multiplications is $2 \cdot n \cdot 3^n$. This quick back-of-the-envelope calculation captures the essence of the technique, even if it does not accurately count the cost of the various operations and does not take into account what is implemented in practice. We now turn to a concrete implementation of this idea:

Concrete implementation. An element of $\mathbb{F}_4$ has a direct canonical representation using 2 bits. Given an element $x \in \mathbb{F}_4$, we write $x(0)$ and $x(1)$ to denote the $\mathbb{F}_2$-coefficients of x viewed as a polynomial over $\mathbb{F}_2[X]/(X^2 + X + 1)$; that is, $x = x(0) + \theta \cdot x(1)$. Using a given machine word of 64 bits we represent a vector of size 32 over $\mathbb{F}_4$, such that the even indexed bits are high order and the odd indexed bits are low order. This is illustrated in Fig. 3.

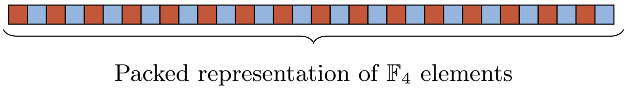

Fig. 3: Representation of a vector of $\mathbb{F}_4$ elements. Red blocks represent the high-order bits while the blue blocks represent low-order bits.

As stated before, we use a recursive algorithm to compute all the evaluations, displayed in algorithm Fig. 4. We considered using a non-recursive approach but no significant efficiency gains were observed, so we instead decided to use the recursive algorithm due to its conceptual simplicity.

Actual cost of the computation. A step in the algorithm of Fig. 4 is to evaluate a polynomial of degree 2, with coefficient in $\mathbb{F}_4$, for the values $\{1, \theta, \theta + 1\}$. Let the polynomial be $a + bX_i + cX_i^2$.

- in the case $X_i = 1$, then the evaluation of the polynomial becomes $a + b + c$.
- in the case $X_i = \theta$, the evaluation becomes $(a + c) + \theta \cdot (b + c)$.
- in the case $X_i = \theta + 1$, the evaluation becomes $(a + b) + \theta \cdot (b + c)$.

Note that we want to compute all the different evaluations, and therefore we can try to reduce the overall costs by reusing several of the intermediate calculations. We can obtain the three evaluations via the following steps: (1)compute $a + b, a + c, b + c$; (2) compute $\theta \cdot (b + c)$; (3) compute $a + b + c$; (4) Compute $(a + c) + \theta \cdot (b + c))$, and $(a + b) + \theta \cdot (b + c))$. Therefore, we count 12 classical bit-by-bit XOR over $\mathbb{F}_2$, and a multiplication by θ to obtain the three needed evaluations of the polynomial.

4.3.1 Taking advantage of the computer words. Today's processors offer XOR operations for machine words of size 64 bits. We take advantage of this parallelism to run multiple FFTs in parallel with a small overhead compared to running a single FFT. With 64-bit machine words, we can perform

```
Fast-Evaluation algorithm

PARAMETERS: $n > 0$ an integer, $P \in \mathbb{F}_3[X_1, \cdots, X_n]/(X_1^3 - 1, \cdots, X_n^3 - 1)$, a polynomial with $n$ variables.
FastEval$(n, P)$:
if $n = 1$ then
    return $\{P(1), P(\theta), P(\theta + 1)\}$
else
    Write $P(X_1, \cdots, X_n) = P_0(X_1, \cdots, X_{n-1}) + X_n P_1(X_1, \cdots, X_{n-1}) + X_n^2 P_2(X_1, \cdots, X_{n-1})$.
    $S \leftarrow \varnothing$.
    $\forall i \in \{0, 1, 2\}, S_i \leftarrow$ FastEval$(n - 1, P_i)$
    foreach $i \in [|S_0|]$:
        $f_j(X) := S_0[j] + S_1[j]X + S_2[j]X^2$.
        $S \leftarrow S \cup \{f_j(1), f_j(\theta), f_j(\theta + 1)\}$.
    return $S$.
```

Fig. 4: Fast evaluation of a polynomial in n variables.

up to 32 FFT in parallel. We pack the c^2 FFTs required by our PCG as follows: we let each machine word contain a single coefficient of the same monomial for each of the c^2 polynomials that we are trying to compute. This saves a factor of c^2, at no extra cost.[14] Therefore, the cost of the evaluation of a single polynomial being of $16n \cdot 3^{n-1}$ XOR, the optimization entails the cost of obtaining the full evaluation of the c^2 polynomials to be $16\lceil c^2/64 \rceil n \cdot 3^{n-1}$.

5 Distributed Seed Generation

In this section, we build a distributed point function that works over ternary indices. This generalization of the standard DPF construction allows us to cleanly work on a ternary basis. In particular, this makes the distributed seed generation protocol for our PCG construction in Fig. 14 much more efficient by avoiding the use of expensive secure binary decomposition protocols when working over ternary secret shares.

5.1 A ternary distributed point function

In prior constructions [GI14, BGI15, BGI16], the domain of DPF was set to $\mathcal{D} = \{0, 1\}^n$. In contrast, we will use a ternary domain $\{0, 1, 2\}^n$. While this change may appear conceptually straightforward, the constructions of [GI14, BGI15] do not immediately generalize to non-binary input domains. We therefore construct a *ternary* DPF using the main ideas behind the two state-of-the-art constructions [BGI15, BGI16].

Definition 6 (Random Distributed Point Function (rDPF)). *We say a* DPF *scheme is a* random *DPF* (rDPF) *scheme if* DPF.Gen *does not take the parameter β as input, and the output value at index α is a secret share of a value $(s \| 1) \in \{0, 1\}^{\lambda + 1}$, where s pseudorandom conditioned on K_σ, for $\sigma \in \{0, 1\}$.*

Lemma 7 (Adapted from [BGI16]). *Any random DPF scheme with output group $\{0, 1\}^{\lambda + 1}$ can be transformed into a DPF scheme for any choice of $\beta \in \mathbb{G}$ a the cost of increasing the key size by $\log |\mathbb{G}|$ bits.*

Remark 8. Looking ahead to Section 5.2, using a random DPF makes our protocols and analysis simpler. In particular, describing a distributed key generation protocol for a random DPF eliminates edge cases associated with the output value β. Separately, we show how to generate an "output correction word" that can be used to go from the $s \| 1$ output of an rDPF to an arbitrary output β.

[14] In practice, using larger machine words has an impact by increasing stack usage, but this is only observed when performing an FFT over very large polynomials.

Construction of Ternary rDPF

```
PARAMETERS: Pseudorandom generator $G\colon \{0,1\}^\lambda \to \{0,1\}^{3(\lambda+1)}$.

$\mathsf{rDPF.Gen}(1^\lambda, 1^n, \alpha)$:
parse $\alpha = \alpha_1 \| \cdots \| \alpha_n$ where $\alpha_i \in \{0,1,2\}$ for all $i \in [n]$.
$s_0^0, s_0^1 \leftarrow_R \{0,1\}^\lambda$, $t_0^0 \leftarrow 0$, $t_0^1 \leftarrow 1$.
foreach $i \in [n]$:
    $(s_{i,0}^0 \| t_{i,0}^0 \| s_{i,1}^0 \| t_{i,1}^0 \| s_{i,2}^0 \| t_{i,2}^0) \leftarrow G(s_{i-1}^0)$.
    $(s_{i,0}^1 \| t_{i,0}^1 \| s_{i,1}^1 \| t_{i,1}^1 \| s_{i,2}^1 \| t_{i,2}^1) \leftarrow G(s_{i-1}^1)$.
    $s_{i,0} \| t_{i,0} \leftarrow (s_{i,0}^0 \| t_{i,0}^0) \oplus (s_{i,0}^1 \| t_{i,0}^1)$.
    $s_{i,1} \| t_{i,1} \leftarrow (s_{i,1}^0 \| t_{i,1}^0) \oplus (s_{i,1}^1 \| t_{i,1}^1)$.
    $s_{i,2} \| t_{i,2} \leftarrow (s_{i,2}^0 \| t_{i,2}^0) \oplus (s_{i,2}^1 \| t_{i,2}^1)$.
    $\mathsf{CW}_{i,j} \leftarrow s_{i,j} \| t_{i,j}$ for all $j \in \{0,1,2\} \setminus \{\alpha_i\}$.
    $r_i \leftarrow_R \{0,1\}^\lambda$.
    $\mathsf{CW}_{i,\alpha_i} \leftarrow (s_{i,\alpha_i} \oplus r_i) \| (t_{i,\alpha_i} \oplus 1)$.
    $\mathsf{CW}_i \leftarrow \mathsf{CW}_{i,0} \| \mathsf{CW}_{i,1} \| \mathsf{CW}_{i,2}$.
    $s_i^0 \| t_i^0 \leftarrow s_{i,\alpha_i}^0 \| t_{i,\alpha_i}^0 \oplus (t_{i-1}^0 \cdot \mathsf{CW}_{i,\alpha_i})$.
    $s_i^1 \| t_i^1 \leftarrow s_{i,\alpha_i}^1 \| t_{i,\alpha_i}^1 \oplus (t_{i-1}^1 \cdot \mathsf{CW}_{i,\alpha_i})$.
$\mathsf{pp} \leftarrow (\mathsf{CW}_1, \ldots, \mathsf{CW}_n)$.
$K_A \leftarrow (\mathsf{pp}, s_0^0 \| t_0^0)$, $K_B \leftarrow (\mathsf{pp}, s_0^1 \| t_0^1)$.
return $(K_A, K_B)$.

$\mathsf{rDPF.FullEval}(\sigma, K_\sigma)$:
parse $K_\sigma = (\mathsf{pp}, s_0^\sigma \| t_0^\sigma)$.
$\mathsf{out} \leftarrow \mathsf{Traverse}(\sigma, \mathsf{pp}, s_0^\sigma, t_0^\sigma, 1)$, $\gamma \leftarrow 3^n$.
parse $\mathsf{out} = (v_1, \ldots, v_\gamma) \in (\{0,1\}^\lambda)^\gamma$.
return $(v_1, \ldots, v_\gamma)$.

$\mathsf{Traverse}(\sigma, \mathsf{pp}, s_{i-1}^\sigma, t_{i-1}^\sigma, i)$:
parse $\mathsf{pp} = (\mathsf{CW}_1, \ldots, \mathsf{CW}_n)$.
if $i = n+1$ then
    return $s_{i-1}^\sigma \| t_{i-1}^\sigma$.
else
    $\tau_i^\sigma \leftarrow G(s_{i-1}^\sigma)$, $\gamma_i^\sigma \leftarrow \tau_i^\sigma \oplus (t_{i-1}^\sigma \cdot \mathsf{CW}_i)$.
    parse $\gamma_i^\sigma = s_{i,0}^\sigma \| t_{i,0}^\sigma \| s_{i,1}^\sigma \| t_{i,1}^\sigma \| s_{i,2}^\sigma \| t_{i,2}^\sigma \in \{0,1\}^{3(\lambda+1)}$.
    return $\mathsf{Traverse}(\sigma, \mathsf{pp}, s_0^\sigma, t_0^\sigma, i+1) \| \mathsf{Traverse}(\sigma, \mathsf{pp}, s_1^\sigma, t_1^\sigma, i+1) \| \mathsf{Traverse}(\sigma, \mathsf{pp}, s_2^\sigma, t_2^\sigma, i+1)$.
```

Fig. 5: Ternary DPF construction with full-evaluation optimization. All s values are $\{0,1\}^\lambda$ bit strings and t values are bits. Superscripts 0 and 1 represent a party identifier which we write as $\sigma \in \{0,1\}$ when referring to a value held by party $\sigma \in \{0,1\}$.

We present our construction for a ternary rDPF in Fig. 5 and analyze security in Proposition 9. The construction follows a similar template to the DPF construction of [BGI15].

Proposition 9 (Ternary rDPF security). *Fig. 5 satisfies the correctness and security properties of Definition 6.*

Proof. Deferred to Appendix E.1. □

5.2 Distributed DPF key generation

In this section, we describe how two parties can generate DPF keys using secret-shares of the index α (i.e., the index at which the point function evaluates to a non-zero value). Our approach is inspired by the protocol of Doerner-shelat [Ds17], which makes only black-box use of OT to select the appropriate correction word at each level. While formally constructing such a protocol requires multiple functionalities and is quite tedious, conceptually, the core idea is very simple. At a high level, each party evaluates the DPF tree, layer-by-layer, and computes the correction word CW_i for layer i using a secure protocol that takes as input shares of the i-th trit $\alpha_i \in \{0,1,2\}$ and the "shares" of the left, middle, and right node labels $(s_{i,0}\|t_{i,0},\ s_{i,1}\|t_{i,1},\ s_{i,2}\|t_{i,2})$. The protocol then outputs CW_i exactly as computed in Fig. 5 to both parties. However, this only results in the parties getting rDPF keys. To make the output consist of a chosen value $\beta \in \mathbb{F}_4$, we construct a separate protocol that outputs a special "output" correction word, denoted $\mathsf{CW}_{\mathsf{out}}$, that can be used to go from an rDPF output to a chosen output β (which the parties hold shares of). Conceptually, $\mathsf{CW}_{\mathsf{out}}$ is the last-layer correction word that encodes the "early termination" output in addition to β.

Overview of functionalities and instantiations. The main ideal functionality, $\mathcal{F}_{\mathsf{rDPF\text{-}DKG}}$, for computing the full rDPF keys (matching the distribution of $\mathsf{rDPF.Gen}$ in Fig. 5) is presented in Fig. 8. It is followed by an instantiation, $\Pi_{\mathsf{rDPF\text{-}DKG}}$, in Fig. 9 where we show how to (1) compute the correction words in each i-th layer by executing the sub-protocol $\Pi_{\mathsf{rDPF\text{-}CW}}$, (2) define the input of $\Pi_{\mathsf{rDPF\text{-}CW}}$ for each i-th layer that maintains the correctness of $\mathsf{rDPF.FullEval}$ (indicator bits and constraints between correction words) and the $\mathsf{rDPF.Gen}$ can be distributed recursively. Since we are using a sub-protocol $\Pi_{\mathsf{rDPF\text{-}CW}}$, we construct its instantiation in Fig. 7 and define its ideal functionality in $\mathcal{F}_{\mathsf{rDPF\text{-}CW}}$ Fig. 6. $\Pi_{\mathsf{rDPF\text{-}CW}}$ shows how to securely compute the correction words for each i-th layer based on $\binom{1}{3}$-OT (note that our protocol $\Pi_{\mathsf{rDPF\text{-}CW}}$ only outputs CW_i, it does not handle the correctness of indicator bits and the constraints between correction words in two consecutive layers).

Then, in $\Pi_{\mathsf{Output\text{-}CW}}$ Fig. 11, we show how to handle computing the "output" correction word that allows us to go from a random output (as computed by the rDPF) to a chosen output, by computing a final correction word $\mathsf{CW}_{\mathsf{out}}$ and satisfies the ideal functionality $\mathcal{F}_{\mathsf{Output\text{-}CW}}$ defined in Fig. 10. The output of PCG-OLE is formed by the multiple shares of each party so an extra OLE over $\mathbb{F}_4$ is used to convert from multiple shares to additive shares before being the input of Fig. 11.

We show that all of our instantiations are secure in the semi-honest setting and we prove security in the UC model, where we only make use of the standard ideal functionality 1-out-of-3 chosen OT $\binom{1}{3}$-OT (Fig. 22). Due to limited space, the security proofs for security of $\Pi_{\mathsf{rDPF\text{-}CW}}, \Pi_{\mathsf{rDPF\text{-}DKG}}, \Pi_{\mathsf{Output\text{-}CW}}$ are provided in Appendix E.2, Appendix E.3, and Appendix E.4, respectively.

Ideal Functionality $\mathcal{F}_{\mathsf{rDPF\text{-}CW}}$

The functionality interacts with a party P_σ and an adversary $\mathcal{A}$.

PARAMETERS: Pseudorandom generator $G\colon \{0,1\}^\lambda \to \{0,1\}^{3(\lambda+1)}$.

FUNCTIONALITY:

```
Wait for input $(\llbracket\alpha_i\rrbracket_{\bar\sigma}, r_i^{\bar\sigma}, (s_{i,j}^{\bar\sigma}\|t_{i,j}^{\bar\sigma})_{j\in\{0,1,2\}}) \in \mathbb{F}_3 \times \{0,1\}^\lambda \times \{0,1\}^{3(\lambda+1)}$ from $\mathcal{A}$.
Wait for input $(\llbracket\alpha_i\rrbracket_{\sigma}, r_i^{\sigma}, (s_{i,j}^{\sigma}\|t_{i,j}^{\sigma})_{j\in\{0,1,2\}}) \in \mathbb{F}_3 \times \{0,1\}^\lambda \times \{0,1\}^{3(\lambda+1)}$ from party $P_\sigma$.
Set $\alpha_i := \llbracket\alpha_i\rrbracket_0 + \llbracket\alpha_i\rrbracket_1 \in \mathbb{F}_3$, $r_i := r_i^0 \oplus r_i^1$.
Compute $s_{i,j}\|t_{i,j} := (s_{i,j}^0\|t_{i,j}^0) \oplus (s_{i,j}^1\|t_{i,j}^1)$ for $j \in \{0,1,2\}$.
$\mathsf{CW}_{i,j} := s_{i,j}\|t_{i,j}$ for all $j \in \{0,1,2\} \setminus \{\alpha_i\}$, $\mathsf{CW}_{i,\alpha_i} := (s_{i,\alpha_i} \oplus r_i)\|(t_{i,\alpha_i} \oplus 1)$.
$\mathsf{CW}_i := \mathsf{CW}_{i,0}\|\mathsf{CW}_{i,1}\|\mathsf{CW}_{i,2}$.
Output $\mathsf{CW}_i$ to both $P_\sigma$ and $\mathcal{A}$.
```

Fig. 6: Ideal functionality $\mathcal{F}_{\mathsf{rDPF\text{-}CW}}$ for computing the correction words

Lemma 10 (Ternary $\mathsf{rDPF\text{-}CW}$ security). *The construction $\Pi_{\mathsf{rDPF\text{-}CW}}$ in Fig. 7 securely realizes the ideal functionality $\mathcal{F}_{\mathsf{rDPF\text{-}CW}}$ (Fig. 6) against semi-honest adversaries in the $\binom{1}{3}$-OT hybrid model.*

Protocol $\Pi_{\mathsf{rDPF\text{-}CW}}$

PARAMETERS:

- Party $\sigma \in \{0,1\}$ has input $[\![\alpha_i]\!]_\sigma \in \mathbb{F}_3, r_i^\sigma \in \{0,1\}^\lambda, (s_{i,j}^\sigma \| t_{i,j}^\sigma)_{j\in\{0,1,2\}} \in \{0,1\}^{3(\lambda+1)}$.
- An instantiation of chosen $\binom{1}{3}$-OT.

PROTOCOL:

For each party $\sigma \in \{0,1\}$:

1: Sample $z^\sigma \leftarrow_R \{0,1\}^{3(\lambda+1)}$.

2: Define

$$\begin{aligned}
\mathbf{C}_0^\sigma &:= (r_i^\sigma \oplus s_{i,0}^\sigma \| (t_{i,0}^\sigma \oplus \sigma),\ s_{i,1}^\sigma \| t_{i,1}^\sigma,\ s_{i,2}^\sigma \| t_{i,2}^\sigma) \oplus z^\sigma && \triangleright\ [\![\mathsf{CW}_i]\!]_\sigma \text{ when } \alpha_i = 0\\
\mathbf{C}_1^\sigma &:= (s_{i,0}^\sigma \| t_{i,0}^\sigma,\ r_i^\sigma \oplus s_{i,1}^\sigma \| (t_{i,1}^\sigma \oplus \sigma),\ s_{i,2}^\sigma \| t_{i,2}^\sigma) \oplus z^\sigma && \triangleright\ [\![\mathsf{CW}_i]\!]_\sigma \text{ when } \alpha_i = 1\\
\mathbf{C}_2^\sigma &:= (s_{i,0}^\sigma \| t_{i,0}^\sigma,\ s_{i,1}^\sigma \| t_{i,1}^\sigma,\ r_i^\sigma \oplus s_{i,2}^\sigma \| (t_{i,2}^\sigma \oplus \sigma)) \oplus z^\sigma && \triangleright\ [\![\mathsf{CW}_i]\!]_\sigma \text{ when } \alpha_i = 2\\
\mathbf{M}_0^\sigma &:= (\mathbf{C}_0^\sigma, \mathbf{C}_1^\sigma, \mathbf{C}_2^\sigma),\ \mathbf{M}_1^\sigma := (\mathbf{C}_1^\sigma, \mathbf{C}_2^\sigma, \mathbf{C}_0^\sigma),\ \mathbf{M}_2^\sigma := (\mathbf{C}_2^\sigma, \mathbf{C}_0^\sigma, \mathbf{C}_1^\sigma)
\end{aligned}$$

3: Invoke $\binom{1}{3}$-OT with party $\bar{\sigma}$ as follows:

- Party $\bar{\sigma}$ plays the role of the sender with inputs $\mathbf{M}^{\bar{\sigma}}_{[\![\alpha_i]\!]_{\bar{\sigma}}}$.
- Party σ plays the role of the receiver and inputs $[\![\alpha_i]\!]_\sigma \in \mathbb{F}_3$.
- Party σ gets $\mathbf{M}^{\bar{\sigma}}_{[\![\alpha_i]\!]_{\bar{\sigma}}}[[\![\alpha_i]\!]_\sigma] \in \{0,1\}^{3(\lambda+1)}$ while party $\bar{\sigma}$ gets nothing.

4: Define $[\![\mathsf{CW}_i]\!]_\sigma := \mathbf{M}^{\bar{\sigma}}_{[\![\alpha_i]\!]_{\bar{\sigma}}}[[\![\alpha_i]\!]_\sigma] \oplus z^\sigma$ and broadcast $[\![\mathsf{CW}_i]\!]_\sigma$.

5: Construct $\mathsf{CW}_i := [\![\mathsf{CW}_i]\!]_\sigma \oplus [\![\mathsf{CW}_i]\!]_{\bar{\sigma}} \in \{0,1\}^{3(\lambda+1)}$.

6: Output $(\mathsf{CW}_{i,0}, \mathsf{CW}_{i,1}, \mathsf{CW}_{i,2})$.

Fig. 7: Instantiation of $\mathcal{F}_{\mathsf{rDPF\text{-}CW}}$ for computing the correction words.

Ideal Functionality $\mathcal{F}_{\mathsf{rDPF\text{-}DKG}}$

The functionality interacts with a party P_σ and an adversary $\mathcal{A}$.

PARAMETERS: $\mathsf{rDPF} = (\mathsf{rDPF.Gen}, \mathsf{rDPF.FullEval})$ as constructed in Fig. 5.

FUNCTIONALITY:

1: Wait for input $([\![\alpha_i]\!]_{\bar{\sigma}})_{i\in[n]} \in \mathbb{F}_3^n$, $(r_i^{\bar{\sigma}})_{i\in[n]} \in (\{0,1\}^\lambda)^n$, and $s_0^{\bar{\sigma}} \in \{0,1\}^\lambda$ from $\mathcal{A}$.

2: Wait for input $([\![\alpha_i]\!]_\sigma)_{i\in[n]} \in \mathbb{F}_3^n$, $(r_i^\sigma)_{i\in[n]} \in (\{0,1\}^\lambda)^n$, and $s_0^\sigma \in \{0,1\}^\lambda$ from party P_σ.

3: Set $t_0^0 = 0, t_0^1 = 1$.

4: Set $\alpha_i := [\![\alpha_i]\!]_0 + [\![\alpha_i]\!]_1 \in \mathbb{F}_3$ and $r_i := r_i^0 + r_i^1 \in \{0,1\}^\lambda$ for each $i \in [n]$.

5: For each $i \in [n]$:
Compute CW_i as done in Step 3 of $\mathsf{rDPF.Gen}(1^\lambda, 1^n, \alpha)$ in Fig. 5.

6: Set $\mathsf{pp} := (\mathsf{CW}_1, \ldots, \mathsf{CW}_n)$, $K_0 := (\mathsf{pp}, s_0^0 \| t_0^0)$, and $K_1 := (\mathsf{pp}, s_0^1 \| t_0^1)$.

7: Output K_σ to P_σ and $K_{\bar{\sigma}}$ to $\mathcal{A}$.

Fig. 8: Ideal functionality $\mathcal{F}_{\mathsf{rDPF\text{-}DKG}}$ for distributed key generation of the ternary rDPF construction from Fig. 5.

Proposition 11 (Ternary rDPF-DKG security). *The construction in Fig. 9 securely realizes the ideal functionality $\mathcal{F}_{\mathsf{rDPF\text{-}DKG}}$ (Fig. 8) against semi-honest adversaries in the $\mathcal{F}_{\mathsf{rDPF\text{-}CW}}$ hybrid model.*

Proposition 12 (Ternary Output-CW security). *The construction $\Pi_{\mathsf{Output\text{-}CW}}$ in Fig. 11 securely realizes the ideal functionality $\mathcal{F}_{\mathsf{Output\text{-}CW}}$ (Fig. 10) against semi-honest adversaries in the $\binom{1}{3}$-OT hybrid model.*

Protocol $\mathit{\Pi}_{\mathsf{rDPF\text{-}DKG}}$

Parameters:

- Pseudorandom generator $G \colon \{0,1\}^\lambda \to \{0,1\}^{3(\lambda+1)}$.
- There are two parties $\sigma, \bar{\sigma} \in \{0,1\}$ with input $([\![\alpha_i]\!]_\sigma)_{i\in[n]} \in \mathbb{F}_3^n$, $(r_i^\sigma)_{i\in[n]} \in (\{0,1\}^\lambda)^n$, $s_{0,0}^\sigma \in \{0,1\}^\lambda$.

Protocol:

For each party $\sigma \in \{0,1\}$:

```
1: Set $\hat{t}_{0,1}^\sigma := \sigma$ and $\hat{s}_{0,1} := s_{0,0}$.
2: foreach $i \in [n]$:
    3.1: Set $d := 3^i$.
    3.2: foreach $j \in [d-1]$:
        3.1.1: $(s_{i,3j}^\sigma \| t_{i,3j}^\sigma \| s_{i,3j+1}^\sigma \| t_{i,3j+1}^\sigma \| s_{i,3j+2}^\sigma \| t_{i,3j+2}^\sigma) \leftarrow G(\hat{s}_{i-1,j}^\sigma)$.
    3.3: $s_{i,0}^\sigma \| t_{i,0}^\sigma := \bigoplus_{j=1}^d (s_{i,3j}^\sigma \| t_{i,3j}^\sigma)$.
    3.4: $s_{i,1}^\sigma \| t_{i,1}^\sigma := \bigoplus_{j=1}^d (s_{i,3j+1}^\sigma \| t_{i,3j+1}^\sigma)$.
    3.5: $s_{i,2}^\sigma \| t_{i,2}^\sigma := \bigoplus_{j=1}^d (s_{i,3j+2}^\sigma \| t_{i,3j+2}^\sigma)$.
    3.6: Invoke $\mathit{\Pi}_{\mathsf{rDPF\text{-}CW}}$ with party $\bar{\sigma}$:
         $\mathsf{CW}_i \leftarrow \mathit{\Pi}_{\mathsf{rDPF\text{-}CW}}(i, [\![\alpha_i]\!]_\sigma, r_i^\sigma, (s_{i,j}^\sigma \| t_{i,j}^\sigma)_{j\in\{0,1,2\}})$.
    3.7: foreach $j \in [d-1]$:
        3.7.1: $(\hat{s}_{i,3j}^\sigma \| \hat{t}_{i,3j}^\sigma \| \hat{s}_{i,3j+1}^\sigma \| \hat{t}_{i,3j+1}^\sigma \| \hat{s}_{i,3j+2}^\sigma \| \hat{t}_{i,3j+2}^\sigma) :=$
               $(s_{i,3j}^\sigma \| t_{i,3j}^\sigma \| s_{i,3j+1}^\sigma \| t_{i,3j+1}^\sigma \| s_{i,3j+2}^\sigma \| t_{i,3j+2}^\sigma) \oplus (\hat{t}_{i-1,j}^\sigma \cdot \mathsf{CW}_i)$.
3: $\mathsf{pp} := (\mathsf{CW}_1, \ldots, \mathsf{CW}_n)$.
4: $K_A := (\mathsf{pp}, s_{0,0}^0 \| 0)$, $K_B := (\mathsf{pp}, s_{0,0}^1 \| 1)$.
5: return $(K_A, K_B)$.
```

Fig. 9: Instantiation of rDPF Distributed Key Generation Protocol.

6 Cryptanalysis and Parameter Selection

In this section, we discuss attacks against our instantiation of $\mathsf{QA\text{-}SD}(c,t,\mathbb{G})$. We provide a SageMath [S+24] script to help select a concrete choice of parameters based on our analysis. In practice, we considered $c = 5, t = 14$ to achieve over 128 bits of security. The spirit of the attack which we describe here was already described in [BCCD23], but we give a more in-depth analysis. As a result of this deeper analysis, we show that the parameters considered in [BCCD23] do not achieve the claimed 128 bits of security, and instead achieve closer to 100 bits of security in practice.

6.1 Model of attack

In the following, we focus on attacks on $\mathsf{QA\text{-}SD}(c,t,\mathbb{G})$ over the finite field $\mathbb{F}_q$. Everything that we describe applies to any q, but we will focus on $q = 4$ when deriving parameters. Our goal is to distinguish pairs $((a_1, \ldots, a_{c-1}), s := e_0 + a_1 e_1 + \cdots + a_c e_c)$ from the uniform distribution over $\mathbb{F}_q[\mathbb{G}]^c \times \mathbb{F}_q[\mathbb{G}]$, where $a_i \xleftarrow{\$} \mathbb{F}_q[\mathbb{G}]$, and e_i are sparse elements of $\mathbb{F}_q[G]$ with Hamming weight t. As mentioned in Section 3.3, the search version of QA-SD is equivalent to solving a decoding problem of the form

$$\begin{pmatrix}\mathbf{I} \ \mathbf{A}_1 \ \ldots \ \mathbf{A}_{c-1}\end{pmatrix} \begin{pmatrix}\mathbf{e}_0 \\ \vdots \\ \mathbf{e}_{c-1}\end{pmatrix} = \mathbf{s},$$

where $\mathbf{A}_i$ is the matrix representation of the multiplication by element a_i with respect to the basis given by $\mathbb{G}$, for some arbitrary ordering of $\mathbb{G}$, and $\mathbf{e}_i$ are sparse vectors of Hamming weight t. In particular, the error vector is split into c blocks of weight t. This is standard when dealing with

Ideal Functionality $\mathcal{F}_{\textsf{Output-CW}}$

Parameters:

- The functionality interacts with a party P_σ and an adversary $\mathcal{A}$.
- Pseudorandom function $G\colon \{0,1\}^\lambda \to (\mathbb{F}_4)^{3^t}$.

Functionality:

```
1: Wait for input $(([\![\alpha_i]\!]_{\bar\sigma})_{i\in[t]}, [\![\beta]\!]_{\bar\sigma}, s^{\bar\sigma}) \in (\mathbb{F}_3)^t \times \mathbb{F}_4 \times \{0,1\}^\lambda$ from $\mathcal{A}$.
2: Wait for input $(([\![\alpha_i]\!]_{\sigma})_{i\in[t]}, [\![\beta]\!]_{\sigma}, s^{\sigma}) \in (\mathbb{F}_3)^t \times \mathbb{F}_4 \times \{0,1\}^\lambda$ from party $P_\sigma$.
3: Set $\alpha_i := [\![\alpha_i]\!]_0 + [\![\alpha_i]\!]_1 \in \mathbb{F}_3$, $\alpha := \sum_{i=1}^{t} \alpha_i 3^{i-1} \in [3^t]$,
   $\beta := [\![\beta]\!]^0 + [\![\beta]\!]^1 \in \mathbb{F}_4$.
4: $\mathsf{CW}_t \leftarrow e_\alpha \cdot \beta \oplus G(s^0) \oplus G(s^1)$, where $e_\alpha \in (\mathbb{F}_4)^{3^t}$ is the $\alpha$-th indicator vector.
5: Output $\mathsf{CW}_t$ to $P_\sigma$ and $\mathcal{A}$.
```

Fig. 10: Ideal functionality $\mathcal{F}_{\textsf{Output-CW}}$ for computing the output correct word with message β.

Protocol $\Pi_{\textsf{Output-CW}}$

Parameters:

- There are two parties $\sigma, \bar\sigma \in \{0,1\}$ with input $([\![\alpha_i]\!]_\sigma)_{i\in[t]} \in (\mathbb{F}_3)^t, [\![\beta]\!]_\sigma \in \mathbb{F}_4, s^\sigma \in \{0,1\}^\lambda$.
- An instantiation of chosen $\binom{1}{3}$-OT.
- Pseudorandom function $G\colon \{0,1\}^\lambda \to (\mathbb{F}_4)^{3^t}$.

Protocol:

```
For each party $\sigma \in \{0,1\}$, for $i \in [t]$:
1: Sample $z_i^\sigma \leftarrow_R (\mathbb{F}_4)^{3^i}$.
2: Define
       $\mathbf{C}^\sigma_{i,0} = ([\![\beta]\!]_\sigma, 0, 0) \oplus z_i^\sigma \in (\mathbb{F}_4)^{3^i}$,
       $\mathbf{C}^\sigma_{i,1} = (0, [\![\beta]\!]_\sigma, 0) \oplus z_i^\sigma \in (\mathbb{F}_4)^{3^i}$,
       $\mathbf{C}^\sigma_{i,2} = (0, 0, [\![\beta]\!]_\sigma) \oplus z_i^\sigma \in (\mathbb{F}_4)^{3^i}$,
       $\mathbf{M}^\sigma_0 = (\mathbf{C}^\sigma_{i,0}, \mathbf{C}^\sigma_{i,1}, \mathbf{C}^\sigma_{i,2}),\ \mathbf{M}^\sigma_1 = (\mathbf{C}^\sigma_{i,1}, \mathbf{C}^\sigma_{i,2}, \mathbf{C}^\sigma_{0}),\ \mathbf{M}^\sigma_2 = (\mathbf{C}^\sigma_{i,2}, \mathbf{C}^\sigma_{i,0}, \mathbf{C}^\sigma_{i,1})$
3: Invoke $\binom{1}{3}$-OT with party $\bar\sigma$ as follows:
   - Party $\bar\sigma$ plays the role of the sender with inputs $\mathbf{M}^{\bar\sigma}_{[\![\alpha_i]\!]_{\bar\sigma}}$.
   - Party $\sigma$ plays the role of the receiver and inputs $[\![\alpha_i]\!]_\sigma \in \mathbb{F}_3$.
   - Party $\sigma$ gets $\mathbf{M}^{\bar\sigma}_{[\![\alpha_i]\!]_{\bar\sigma}}[[\![\alpha_i]\!]_\sigma] \in (\mathbb{F}_4)^{3^i}$ while party $\bar\sigma$ gets nothing.
4: Define $[\![\beta]\!]_\sigma := \mathbf{M}^{\bar\sigma}_i[[\![\alpha_i]\!]_\sigma] \oplus z_i^\sigma \in (\mathbb{F}_4)^{3^i}$.
Output $[\![\mathsf{CW}]\!]_t := [\![\beta]\!]_\sigma \oplus G(s^\sigma)$.
```

Fig. 11: Instantiation of $\mathcal{F}_{\textsf{Output-CW}}$ for computing the last correction words with constraint β.

structured variants of the decoding problem. This c-split structure has recently received renewed attention since it is also used in the NIST submission SDitH [AMFG⁺23] (for unstructured codes). The hardness of this variant can be estimated using the following theorem from [FJR22].

Theorem 13 ([FJR22, Theorem 1]). *Let n, k, w be positive integers such that $n > k$, $n > w$, and c divides both w and n. If there is an algorithm solving a random instance of the c-split decoding problem with code length n, dimension k and w errors, in time T and with probability ε_c, then there* *exists an algorithm which solves a random instance of the standard decoding problem with the same parameters, in time T, and with probability ε_1 with*

$$\varepsilon_1 \geq \frac{\binom{n/c}{w/c}^c}{\binom{n}{w}} \cdot \varepsilon_c.$$

With Theorem 13 in hand, one might contend that the complexity of solving a c-split variant of the decoding problem cannot be lower than the best complexity for solving the usual syndrome decoding problem, times $\binom{n/c}{w/c}^c / \binom{n}{w}$. When choosing parameters, we make sure that this lower bound is beyond the targeted security level.

Remark 14. This lower bound has been updated in [CHT23] when taking the number of solutions into account. However, in our low noise regime, there is only one solution to the decoding problem with overwhelming probability, and in this situation, the two bounds coincide.

Remark 15. Adding the penalty given by Theorem 13 gives conservative parameters since no decoding algorithm currently makes use of the c-split structure. However, this assumption on the noise has been less thoroughly studied than general instances of the decoding problem, and it may be possible that better attacks appear shortly. By taking this result into account, we remain conservative with respect to all current attack strategies.

In our setting, the code rate is $1 - 1/c$ which is always bounded away from $1/2$. Therefore, generic attacks on LPN such as Arora-Ge [AG11] or BKW [BKW03], which require a very low code-rate, do not apply. In other words, we will only consider attacks against the decoding problem. Nevertheless, our instance is far from being generic. Indeed, the *quasi-abelian* structure gives more power to the adversary. First, the adversary can apply the DOOM strategy from [Sen11] to get a $\sqrt{|\mathbb{G}|}$ speed-up over *any* decoding algorithm. Second, the additional structure allows an attacker to reduce their sample modulo some ideal of $\mathbb{F}_q[\mathbb{G}]$ to obtain a *smaller* instance of the decoding problem, with the hope that the noise rate does not grow too much. As already observed in [BCCD23], the best possible choices arise from quotients of $\mathbb{G}$ and are known as *folding attacks.* They will be precisely analyzed in Section 6.3. In short, the folding operator with respect to a subgroup $\mathbb{H}$ will send a code of length $c|\mathbb{G}|$ to a code of length $c|\mathbb{G}/\mathbb{H}|$, while keeping the code-rate constant. On the other hand, the noisy vector will still be c-split (with blocks of size $|\mathbb{G}/\mathbb{H}|$), but the number of error coordinates in each block might decrease due to collisions. In other words, the noise is *upper bounded* by t on each block.

Attack Strategy. The general idea of folding attacks to solve $\mathsf{QA\text{-}SD}(c, t, \mathbb{G})$ consists in picking some subgroup $\mathbb{H}$, applying the folding operator with respect to $\mathbb{H}$ and run a generic decoding algorithm on this smaller instance. Therefore, we need to choose the parameters such that generic decoding algorithms against all folded instances remain beyond the targeted security parameter, even considering the loss induced by the c-split structure, and the $\sqrt{|\mathbb{G}|/|\mathbb{H}|}$ speed-up from the DOOM strategy. In general, this folding strategy will only be helpful for large enough subgroups $\mathbb{H}$ (yielding small instances of the decoding problem). Nevertheless, for this approach to be relevant, we also need the number of errors to remain below the Gilbert-Varshamov (GV) bound of the folded instance, which ensures on average the uniqueness of the solution. Recall that for a code of rate R, the GV bound is defined as $\delta_{GV} = h_q^{-1}(1 - R)$ where $h_q(x) := -x \log_q \left(\frac{x}{q-1} \right) - (1 - x) \log_q(1 - x)$ for $x \in (0, 1 - 1/q)$, is the q-ary entropy function. It is well-known that this provides a threshold for the number of solutions on a random instance $(\mathbf{H}, \mathbf{s})$ of the decoding problem of a code of length n and at distance w: when $w < n\delta_{GV}$, then there will be at most one solution with overwhelming probability over the choices of $\mathbf{H}$, while when $w > n\delta_{GV}$ there will be exponentially many solutions, even when $\mathbf{s}$ was picked uniformly at random (see [Deb23, Chapter 2]). In particular, if the number of errors in the folded instance is beyond the GV bound, then we need to lift each of those exponentially many putative solutions back to the original decoding problem to see whether it is an actual solution, or not. Doing so is equivalent to specifying a small number of unknowns in the linear system coming from the original syndrome decoding instance (see [CT19, Section IV]), and this strategy is far less efficient than solving a single decoding problem for a larger code-length. Note that finding a solution to a folded instance below the GV bound is enough to provide a distinguisher between an instance of $\mathsf{QA\text{-}SD}$ and a uniformly random vector, thus breaking the security of $\mathbb{F}_4\mathsf{OLEAGE}$, since we only need to solve the decisional variant of $\mathsf{QA\text{-}SD}$.

Improvement. In Section 6.3 we give a precise analysis of the probability distribution of the weight of the folded error. Since the initial error is extremely sparse, the weight of the folded error will be very close to that of the initial error with overwhelming probability (since there will be very few collisions). However, we show how an attacker can bet that the folded error has a much smaller weight w_0 than expected, which happens with some exponentially small probability p_{w_0}. They can then benefit from the fact that our instantiation has an extremely large number of potential subgroups $\mathbb{H}$ to run $\frac{1}{p_{w_0}}$ algorithms tailored for decoding w_0 errors, resulting in an attack which brings the security of the parameters chosen in [BCCD23] down to roughly 100 bits instead of the targeted 128.

Remark 16 (Regular noise). For the optimized variants of our PCG, we also put an additional structure on the error term. Beyond being c-split, it is in fact *regular*, namely it is the concatenation of t unit vectors. This variant has been introduced for building the hash function FSB [AFS03] which was submitted to the SHA-3 competition and was subsequently analyzed in [FGS07, BLPS11]. In general, it is not known to induce a significant weakness, however, it has not undergone the same thorough analysis that the standard syndrome decoding problem has gone through. This regular variant has recently become more popular due to many works on signatures and secure computation [HOSS18b, BCGI18, BCG$^+$19b, BCG$^+$20b, BCG$^+$22, CCJ23, BCCD23]. We note that Theorem 13 is not tight in the regular case, and it would be overly conservative to consider this loss here. More precisely, every work that really targets the regular variant of syndrome decoding shows that the code rate needs to be rather small for this variant to have an impact on the security [BØ23, ES23]. The latter reference even argues that regular syndrome decoding is *harder* than standard syndrome decoding for in the high code-rate regime (see [ES23, Figure 1]), which is precisely our setting.

6.2 Generic decoding algorithms

Generic attacks against the decoding problem are divided into two families, which we summarize here.

Information Set Decoders (ISD). Starting from Prange's seminal work [Pra62], Information Set Decoding represents the main technique to solve the decoding problem. They have been extensively studied over the past 60 years [Pra62, Ste89, Wag02, MMT11, BJMM12, MO15, BM18, DEEK24], and in general their complexity is exponential in the number of error coordinates. However, most analyses were conducted over the binary field, in order to improve on the exponent. In particular, as far as we know, there is no precise analysis of the concrete, non asymptotic, efficiency over larger fields $\mathbb{F}_q$. We note though that the most recent algorithms (starting from [MO15]), have prohibitive hidden constants, which make them less practical [Hir16]. We give an overview of ISD algorithms in Appendix C.

Dual attacks. Up until recently, Information Set Decoding algorithms were considered to be the best algorithms to solve the generic decoding problem. However, another strategy known as *statistical decoding* or *dual attacks*, has been introduced in [Jab01, Ove06] and were greatly improved in the past few years [DT17, CDMT22, MT23, CDMT24]. Most recent dual attacks even outperform ISD algorithms in some regimes. Nevertheless, those improvements are only apparent for codes of the rate below 1/2 [CDMT24, Figure 1]. In particular, we will not consider them in our high code-rate regime.

6.3 Analysis of Folding attacks

In general, there are very few known ways in which an attacker can exploit the structure of variants of the decoding problem to get a significant advantage. In fact, it is a longstanding open problem in algebraic coding theory to design a generic decoder for quasi-abelian codes [Wil21]. Moreover, the authors of [BCCD23] proved that our particular instantiation of QA-SD comes with a search-to-decision reduction. All of that argues in favor of the hardness of QA-SD$(c, t, \mathbb{G})$. Nevertheless, in the case of QA-SD, an attacker can use one of the many subgroups of $\mathbb{G}$ to get a smaller instance, with the hope of getting an easier decoding problem. In this section, we give a precise analysis of such folding attacks.

6.3.1 The Folding Operation

Let $\mathbb{H}$ be a subgroup of $\mathbb{G}$. The canonical projection $\mathbb{G} \to \mathbb{G}/\mathbb{H}$ induces a morphism of algebras

$$\pi_{\mathbb{H}} : \left\{\begin{array}{ccc}\mathbb{F}_q[\mathbb{G}] & \longrightarrow & \mathbb{F}_q[\mathbb{G}/\mathbb{H}] \\ \displaystyle\sum_{g\in\mathbb{G}} a_g g & \longmapsto & \displaystyle\sum_{\bar{g}\in\mathbb{G}/\mathbb{H}} \left(\sum_{h\in\mathbb{H}} a_{g+h}\right)\bar{g}.\end{array}\right.$$

Represented as a vector of length $|\mathbb{G}|$, an element $x \in \mathbb{F}_q[\mathbb{G}]$ can also be seen as a collection of $|\mathbb{G}/\mathbb{H}|$ interleaved subvectors of length $|\mathbb{H}|$ indexed by each coset of $\mathbb{H}$. The operation π_H simply consists of summing up all the entries of each subvector. An illustration is given on Fig. 12.

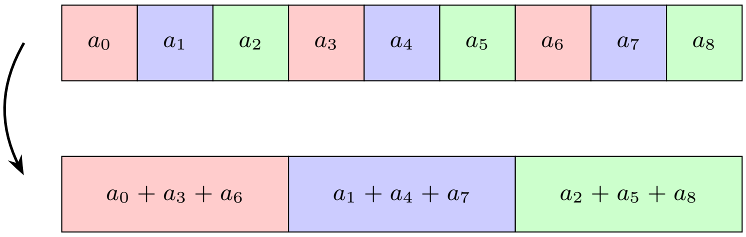

Fig. 12: Example representation of $\pi_{\mathbb{H}}$ for a group $\mathbb{G}$ of size 9 and a subgroup $\mathbb{H}$ of size 3. Each coset of $\mathbb{H}$ is represented by a different color. Note that in general, there is no reason to have this regular pattern.

Given an element $\mathbf{y} := (y_1, \ldots, y_c) \in \mathbb{F}_q[\mathbb{G}]^c$, we define the folding with respect to $\mathbb{H}$ as $\mathrm{Fold}_{\mathbb{H}}(\mathbf{y}) := (\pi_{\mathbb{H}}(y_1), \ldots, \pi_{\mathbb{H}}(y_c)) \in \mathbb{F}_q[\mathbb{G}/\mathbb{H}]^c$. Now, consider an instance $(\mathbf{a}, \langle \mathbf{a}, \mathbf{e}\rangle + e_0)$ of $\mathsf{QA\text{-}SD}(c, t, \mathbb{G})$. Applying $\pi_{\mathbb{H}}$ on the syndrome yields

$$\pi_{\mathbb{H}}\left(e_0 + \sum_{i=1}^{c-1} a_i e_i\right) = \pi_{\mathbb{H}}(e_0) + \sum_{i=1}^{c-1} \pi_{\mathbb{H}}(a_i)\pi_{\mathbb{H}}(e_i) \in \mathbb{F}_q[\mathbb{G}/\mathbb{H}],$$

which is nothing but the syndrome of $\mathrm{Fold}_{\mathbb{H}}(\mathbf{e})$ with respect to the matrix represented by $\mathrm{Fold}_{\mathbb{H}}(1, \mathbf{a})$. In other words, starting from a decoding problem with code-length $c|\mathbb{G}|$ and dimension $(c-1)|\mathbb{G}|$, we build a smaller decoding problem with code-length $c|\mathbb{G}/\mathbb{H}|$. The main observation is that, doing so, we *did not* change the code-rate, which remains equal to $1 - 1/c$. On the other hand, the Hamming weight of the folded error might be slightly below the original one. The goal of this section is to carefully estimate this weight. To do so, we will give a precise analysis of the probability distribution of $w_H(\mathrm{Fold}_{\mathbb{H}}(\mathbf{x}))$ when $\mathbf{x} := (x_1, \ldots, x_c) \in \mathbb{F}_q[\mathbb{G}]^c$ is such that the x_i's are independent and uniformly distributed over the elements of Hamming weight t. This distribution is nothing but the c-fold convolution of the probability distribution of the weights of a single component, which we derive in Section 6.3.2. Our analysis is similar to that of [CT19], written in the language of group algebras.[15]

6.3.2 Weight distribution of the image of the folding

For $u \in \{0, \ldots, t\}$ and $\mathbb{H}$ a subgroup of $\mathbb{G}$, denote by

$$A_{\mathbb{H}}(t, u) := \left|\left\{x \in \mathbb{F}_q[\mathbb{G}] \mid w_H(x) = t \text{ and } w_H(\pi_{\mathbb{H}}(x)) = u\right\}\right|.$$

When x is uniformly chosen amongst the elements of the group algebra $\mathbb{F}_q[\mathbb{G}]$ of Hamming weight t, then $\Pr_x\left[\, w_H(\pi_{\mathbb{H}}(x)) = u \,\right] = \dfrac{A_{\mathbb{H}}(t,u)}{\binom{|\mathbb{G}|}{t}(q-1)^t}$. In other words, the goal is to compute $A_{\mathbb{H}}(t, u)$.

Let $x \in \mathbb{F}_q[\mathbb{G}]$ of Hamming weight t and let $\ell := |\mathbb{H}|$. We can view x as an interleaving of $|\mathbb{G}/\mathbb{H}|$ subvectors of length ℓ corresponding to the different cosets of $\mathbb{H}$, such that the weights of the subvectors form a partition of t into parts of size at most ℓ. For a given element x of weight t, we call the corresponding partition of t the *signature* of x. By definition, a coordinate of $\pi_{\mathbb{H}}(x)$, indexed by

[15] In [CT19, Proposition 3], the authors forgot a $q-1$ factor, which we fix in our analysis. Note that this does not impact the results of [CT19] since their actual setting was for $q = 2$.

some coset $g_0 + \mathbb{H}$, is zero if and only if the corresponding entries of x sum up to 0, i.e., if and only if the word $(x_{g_0+h})_{h\in\mathbb{H}} \in \mathbb{F}_q^\ell$ belongs to the $[\ell, \ell-1]_q$-parity code denoted by $\mathscr{C}_\ell$. In particular, $A_{\mathbb{H}}(t,u)$ does not depend on the specific subgroup $\mathbb{H}$, but rather on its order ℓ.

For $\omega \in \{0,\ldots,\ell\}$, let $\nu(\omega,\ell)$ be the number of codewords of $\mathscr{C}_\ell$ of Hamming weight ω and let $P_{\nu,\ell} := \sum_{\omega=0}^{\ell} \nu(\omega,\ell)X^\omega$ be the weight enumerator of $\mathscr{C}_\ell$. Similarly, we denote by $P_{\theta,\ell} := \sum_{\omega=0}^{\ell} \theta(\omega,\ell)X^\omega$ the weight enumerator of the complement $\mathbb{F}_q^\ell \setminus \mathscr{C}_\ell$. Let $a \in \mathbb{F}_q[\mathbb{G}/\mathbb{H}]$ be of weight u. Without loss of generality, we assume that the u nonzero positions of a correspond to its first u entries. Then, given a partition $(i_1,\ldots,i_{|\mathbb{G}/\mathbb{H}|})$ of t into parts of size at most ℓ, the number of elements $x \in \mathbb{F}_q[\mathbb{G}]$ having this signature and which are mapped to a is exactly

$$\prod_{j=1}^{|\mathbb{G}/\mathbb{H}|} \left(\theta(i_j,\ell)\mathbb{1}_{a_{i_j}\neq 0} + \nu(i_j,\ell)\mathbb{1}_{a_{i_j}=0}\right) = \prod_{j=1}^{u} \theta(i_j,\ell) \prod_{j=u+1}^{|\mathbb{G}/\mathbb{H}|} \nu(i_j,\ell).$$

In other words, we have

$$A_{\mathbb{H}}(t,u) = \binom{|\mathbb{G}/\mathbb{H}|}{u} \sum_{\substack{i_1+\cdots+i_{|G/\mathbb{H}|}=t \\ 0\le i_j\le \ell}} \left(\prod_{j=1}^{u} \theta(i_j,\ell) \prod_{j=u+1}^{|\mathbb{G}/\mathbb{H}|} \nu(i_j,\ell) \right),$$

which is nothing but the coefficient of X^t in the polynomial $\binom{|\mathbb{G}/\mathbb{H}|}{u} P_{\theta,\ell}^u(X) P_{\nu,\ell}^{|\mathbb{G}/\mathbb{H}|-u}(X)$, which we denote by $[X^t]\left(\binom{|\mathbb{G}/\mathbb{H}|}{u} P_{\theta,\ell}^u(X) P_{\nu,\ell}^{|\mathbb{G}/\mathbb{H}|-u}(X)\right)$. We can then deduce the following proposition.

Proposition 17. *Let $0 \le t \le |\mathbb{G}|$ and $0 \le u \le \min(t, |\mathbb{G}/\mathbb{H}|)$. When x is uniformly distributed over the elements of $\mathbb{F}_q[\mathbb{G}]$ of Hamming weight t, then*

$$\mathrm{Pr}_x\left[\, w_H(\pi_{\mathbb{H}}(x)) = u \,\right] = \frac{\binom{|\mathbb{G}/\mathbb{H}|}{u}[X^t]\left(P_{\theta,\ell}^u(X) P_{\nu,\ell}^{|\mathbb{G}/\mathbb{H}|-u}(X)\right)}{\binom{|\mathbb{G}|}{t}(q-1)^t}.$$

The following lemma, a corollary of MacWilliam's identity (see [MS86]), gives the closed formulas of $P_{\nu,\ell}$ and $P_{\theta,\ell}$.

Lemma 18 ([CT19, Lemma 1]). *We have*

$$P_{\nu,\ell}(X) = \frac{1}{q}\Big((1+(q-1)X)^\ell + (q-1)(1-X)^\ell\Big),$$

and

$$P_{\theta,\ell}(X) = \frac{q-1}{q}\Big((1+(q-1)X)^\ell - (1-X)^\ell\Big).$$

6.3.3 Choice of the folding subgroup

The choice of the subgroup $\mathbb{H}$ is motivated by the targeted length $c|\mathbb{G}/\mathbb{H}|$ of the folded code (we discuss in Section 6.4 folding with respect to several subgroups). As mentioned above, we need $t < \delta_{GV}|\mathbb{G}/\mathbb{H}|$ for this procedure to be relevant, where δ_{GV} is the GV bound at rate $1-1/c$. In our setting, $\mathbb{G}$ is of the form $(\mathbb{Z}/3\mathbb{Z})^n$ and therefore $\mathbb{H}$ will be of the form $(\mathbb{Z}/3\mathbb{Z})^{n'}$. In particular, $|\mathbb{G}/\mathbb{H}| = 3^{n-n'}$ should be such that $c\cdot t \le c\cdot\delta_{GV}3^{n-n'}$. In practice, choosing $n' = n - \lceil \log_3(t/\delta_{GV})\rceil$, or $n - \lceil \log_3(t/\delta_{GV})\rceil - 1$ when t is particularly small, yields the best complexity.

6.4 Improving the attack: Folding for several subgroups

Since we chose $\mathbb{G}$ of the form $(\mathbb{Z}/3\mathbb{Z})^n$, there are many different subgroups of given order $3^{n'}$. More precisely, they are in one-to-one correspondence with the n'-dimensional subspaces of $\mathbb{F}_3^n$: there are $\binom{n}{n'}_3$ of them. Instead of picking one particular subgroup and running the best generic decoding algorithm on the folded instance, an attacker could bet that the error has a much smaller weight ω_0. They could then run the best decoding algorithm, aborting the procedure when the number of iterations exceeds that required to decode ω_0 errors, and then pick another subgroup. On average, this procedure will allow to decode in time $\frac{\text{Cost}_{\text{Decoding}}(\omega_0) + \text{Cost}_{\text{Folding}}}{\Pr\left[\, w_H(\text{Fold}_{\mathbb{H}}(x)) = \omega_0 \,\right]}$, where $\text{Cost}_{\text{Decoding}}$ is the complexity of the best decoding algorithm for the target weight ω_0 in the folded code, and $\text{Cost}_{\text{Folding}}$ is the cost of computing the folding operation, i.e., $\text{Cost}_{\text{Folding}}$ is exactly the length of the initial code. In order to run the attack, it suffices to choose the best ω_0 which minimizes this ratio to get the best complexity. This attack is much more efficient for small values of c, that is why we chose $c = 5$ in our setting. This allows us to maintain small parameters while achieving more than 128 bits of security (while still keeping some security margin).

6.4.1 Parameter Estimation and Revaluation of [BCCD23]

We provide a SageMath [S+24] script to help choose a set of concrete parameters for QA-SD. It computes the probability distribution of the weight of the folded error and computes the cost of the best attack. It also takes into account the previous attack.

As a by-product, we can use it to give a new estimation of the security of [BCCD23]. Results are given in Section 6.4.1. It shows that we would need to significantly increase the value of t in order to achieve 128 bits of security.

n	c	t	$(n_{\text{fold}}, k_{\text{fold}}, \omega_0)$	$(\text{N}_{\text{iter}}, \text{Cost}_{\text{Decoding}})$	Number of subgroups	Actual security	New t
25	4	16	$(2048, 1536, 54)$	$(2^{14}, 2^{89})$	2^{145}	103	20
30	4	16	$(2048, 1536, 54)$	$(2^{14}, 2^{89})$	2^{190}	103	20
35	4	16	$(2048, 1536, 54)$	$(2^{14}, 2^{89})$	2^{235}	103	20

Table 3: Reestimation of the security for the parameters given in [BCCD23]. They were considered to yield more than 128 bits of security, they were even considered to be conservative. Note that in [BCCD23], all the parameters were for $q = 3$. Here t is the number of errors per block, while in [BCCD23] it was the total number of errors. n_{fold} and k_{fold} are respectively the length and dimension of the folded code. N_{iter} is the number of different foldings necessary to run the attack, and ω_0 is the optimal target weight. We provide un updated value for t to achieve 128 bits of security.

7 Implementation and Evaluation

We implement $\mathbb{F}_4$OLEAGE in C (v15.0.0) as a library that consists of two main components: (1) an optimized implementation of the ternary DPF construction and (2) an implementation of the FFT over $\mathbb{F}_4$. The open-source code for our $\mathbb{F}_4$OLEAGE PCG benchmarks is available online.[16]

Implementation details. Our DPF implementation takes advantage of the `AES-NI` instruction to implement a fast PRG G using fixed-key AES (from the OpenSSL library [Ope]) and the Davies-Meyer transform to instantiate a PRF.

We implement the recursive FFT over $\mathbb{F}_4$ described in Section 4.3 and perform the FFT in parallel by packing all the coefficients into one machine word (for our parameters, we require fewer than 32 parallel FFTs, so we can perform 32 FFTs in parallel using a `uint32` type). While the FFT could possibly be optimized further using an iterative algorithm and taking advantage of AVX instructions, the simplicity of the recursive algorithm coupled with the parallel packing makes it sufficiently fast for $\mathbb{F}_4$OLEAGE, where the DPF evaluations end up being the dominant cost. The implementation can also be optimized further by taking advantage of AVX instructions when packing and unpacking the polynomial coefficients, as the operation amounts to performing a matrix transpose. We leave such optimizations for future work.

We do not implement the distributed seed generation protocol given that it consists of black-box invocations of any one-out-of-three OT protocol. However, we do benchmark the concrete performance and communication costs for running distributed seed generation using the libOTe library on state-of-the-art OT protocols [RR].

[16] `https://github.com/sachaservan/FOLEAGE-PCG`.

Benchmarks. We perform our benchmarks using AWS `c5.metal` (3.4GHz CPU) and `t2.large` instances. All experiments are averaged across ten trials and evaluated on a single core. To gain a better understanding of the overhead involved with each component, we start by benchmarking the DPF and FFT implementation separately and report the results in Tables 4 and 5. Concretely, if we are packing 3^n coefficients over $\mathbb{F}_4$, we want the output of the DPF to be close to a power of 3. To achieve this, we can terminate 5 levels early and pack 243 elements of $\mathbb{F}_4$ in the virtual leaves by having the DPF output be a 512 bit block. Therefore, the key size of each DPF is $3 \cdot 128 \cdot (n-5) + 128 + 2 \cdot 256$ when using AES with 128-bit keys.

Range	DPF.Gen	DPF.FullEval	AES	**Key Size**
(elements of $\mathbb{F}_4$)	(c5.metal | t2.large)	(c5.metal | t2.large)	(c5.metal | t2.large)	(per party)
3^{14}	9 μs | 20 μs	0.51 ms | 1.45 ms	0.36 ms | 0.95 ms	512 B
3^{16}	11 μs | 22 μs	9 ms | 12 ms	5 ms | 9 ms	608 B
3^{18}	12 μs | 26 μs	91 ms | 130 ms	51 ms | 86 ms	704 B

Table 4: Performance of our DPF implementation on two different EC2 instances. When evaluating the SPFSS (sum of many DPFs), we observe a roughly 1.8× improvement in computation due to better cache performance when evaluating many DPFs and working over the same memory allocation).

Number of Variables	Packed FFT (4×)	Packed FFT (16×)	Packed FFT (32×)
	(c5.metal | t2.large)	(c5.metal | t2.large)	(c5.metal | t2.large)
14	20 ms | 30 ms	21 ms | 33 ms	28 ms | 45 ms
16	180 ms | 280 ms	213 ms | 329 ms	312 ms | 475 ms
18	1682 ms | 2608 ms	2165 ms | 3280 ms	4913 ms | 7478 ms

Table 5: Performance of our FFT implementation for polynomials with coefficients over $\mathbb{F}_4$ on two different EC2 instances. Packing increases throughput almost linearly with the packing size. However, with a large number of variables (> 16), it is more efficient to use smaller packing values to avoid the increased memory usage on the stack due to the recursive FFT implementation.

Benchmarking our PCG. Next, we benchmark the performance of the PCG from Fig. 14 on various parameters. The parameter $D = 3^n$ determines the number of Beaver triples we generate in total. In contrast, the parameters c (compression factor) and t (noise weight) influence the size of the PCG key and evaluation time. Specifically, evaluating the PCG requires $(c \cdot t)^2$ calls to the DPF on domain size n/t (due to regular noise) and $(c+1)/2$ calls to the FFT (which we can parallelize by a factor of up to 32 using packing). We set $c = 5$, $t = 14$, and report the computational costs of the PCG for different values of D in Table 6. The choice of $(c = 5, t = 14)$ corresponds to a conservative parameter choice based on our calculations in Table 2. To show the influence of the noise weight t on the performance, we also evaluate our PCG construction on $t = 13$, which corresponds to a less conservative parameter choice. We observe slightly smaller PCG seeds and better concrete performance with $t = 13$.

(a) Parameters: $(c = 5,\ t = 14)$

	PCG.Expand	**Key Size**
D	(c5.metal | t2.large)	(per party)
3^{14}	417 ms | 695 ms	2.1 MB
3^{16}	4.1 s | 6.5 s	2.6 MB
3^{18}	37.2 s | –	3.1 MB

(b) Parameters: $(c = 5,\ t = 13)$

	PCG.Expand	**Key Size**
D	(c5.metal | t2.large)	(per party)
3^{14}	384 ms | 633 ms	1.8 MB
3^{16}	3.8 s | 6.2 s	2.2 MB
3^{18}	34.2 s | –	2.6 MB

Table 6: Performance of our PCG implementation on two different EC2 instances. We set the compression factor $c = 5$ and noise parameter $t = 14$ in the left table and $t = 13$ in the right table; these parameters are computed using Section 6. $D = 3^{18}$ ran out of memory on the `t2.large`.

Estimating setup costs. To estimate the setup costs, we use the libOTe library [RR] to benchmark the state-of-the-art OT protocols. We run libOTe on `localhost` and evaluated both SoftSpoken OT [Roy22] and the RRT silent OT [RRT23] protocols. For SoftSpoken, we measured roughly 50,000,000 OT/s on the `c5.metal` machine roughly 32,000,000 OT/s on the `t2.large`. For the RRT silent OT protocol, we measure a throughput of nearly 7,000,000 on `c5.metal` and 4,000,000 on the `t2.large`. To run our distributed DPF key generation protocol, we require $n = 14$ (at $D = 3^{14}$) and $n = 18$ (at $D = 3^{18}$) rounds per DPF (this corresponds to the depth of the DPF tree, including the rounds required to pack the output message). All the $(ct)^2$ DPF keys can be computed in parallel. Therefore, in total, using our parameters of $c = 5$ and $t = 14$, we require roughly 4,900 parallel calls to an OT functionality in n consecutive rounds.

Acknowledgements

We thank Peter Rindal for help with running the libOTe [RR] library, and Marcel Keller for answering our questions about Overdrive [KPR18]. Geoffroy Couteau, Clément Ducros, and Dung Bui were supported by the French Agence Nationale de la Recherche (ANR), under grant ANR-20-CE39-0001 (project SCENE), and by the France 2030 ANR Project ANR22-PECY-003 SecureCompute. Dung Bui was supported by DIM Math Innovation 2021 (N°IRIS: 21003816) from the Paris Mathematical Sciences Foundation (FSMP) funded by the Paris Île-de-France Region.

References

AAB$^+$22a. C. Aguilar Melchor, N. Aragon, P. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Ghosh, S. Gueron, T. Güneysu, R. Misoczki, E. Persichetti, J. Richter-Brockmann, N. Sendrier, J.-P. Tillich, V. Vasseur, and G. Zémor. BIKE. Round 4 Submission to the NIST Post-Quantum Cryptography Call, v. 5.1, October 2022.

AAB$^+$22b. C. Aguilar Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J. Bos, J.-C. Deneuville, A. Dion, P. Gaborit, J. Lacan, E. Persichetti, J.-M. Robert, P. Véron, G. Zémor, and J. Bos. HQC. Round 4 Submission to the NIST Post-Quantum Cryptography Call, October 2022. https://pqc-hqc.org/.

ABD$^+$16. C. Aguilar, O. Blazy, J.-C. Deneuville, P. Gaborit, and G. Zémor. Efficient encryption from random quasi-cyclic codes. Cryptology ePrint Archive, Report 2016/1194, 2016. https://eprint.iacr.org/2016/1194.

AFS03. D. Augot, M. Finiasz, and N. Sendrier. A fast provably secure cryptographic hash function. IACR Cryptology ePrint Archive, Report2003/230, 2003. http://eprint.iacr.org/.

AG11. S. Arora and R. Ge. New algorithms for learning in presence of errors. In *Automata, Languages and Programming*, *LNCS* 6755, pages 403–415. Springer Berlin Heidelberg, 2011.

AMBD$^+$18. C. Aguilar-Melchor, O. Blazy, J.-C. Deneuville, P. Gaborit, and G. Zémor. Efficient encryption from random quasi-cyclic codes. *IEEE Transactions on Information Theory*, 64(5):3927–3943, 2018.

AMFG$^+$23. C. Aguilar Melchor, T. Feneuil, N. Gama, S. Gueron, J. Howe, D. Joseph, A. Joux, E. Persichetti, T. Randrianarisoa, M. Rivain, and D. Yue. SDitH. Round 1 Additional Signatures to the NIST Post-Quantum Cryptography: Digital Signature Schemes Call, May 2023.

BC23. D. Bui and G. Couteau. Improved private set intersection for sets with small entries. In *PKC 2023, Part II*, *LNCS* 13941, pages 190–220. Springer, Heidelberg, May 2023.

BCCD23. M. Bombar, G. Couteau, A. Couvreur, and C. Ducros. Correlated pseudorandomness from the hardness of quasi-abelian decoding. In *CRYPTO 2023, Part IV*, LNCS, pages 567–601. Springer, Heidelberg, August 2023.

BCDL19. R. Bricout, A. Chailloux, T. Debris-Alazard, and M. Lequesne. Ternary syndrome decoding with large weight. In *SAC 2019*, *LNCS* 11959, pages 437–466. Springer, Heidelberg, August 2019.

BCG$^+$19a. E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, P. Rindal, and P. Scholl. Efficient two-round OT extension and silent non-interactive secure computation. In *ACM CCS 2019*, pages 291–308. ACM Press, November 2019.

BCG$^+$19b. E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In *CRYPTO 2019, Part III*, *LNCS* 11694, pages 489–518. Springer, Heidelberg, August 2019.

BCG$^+$20a. E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl. Correlated pseudorandom functions from variable-density LPN. In *61st FOCS*, pages 1069–1080. IEEE Computer Society Press, November 2020.

BCG$^+$20b. E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl. Efficient pseudorandom correlation generators from ring-LPN. In *CRYPTO 2020, Part II*, *LNCS* 12171, pages 387–416. Springer, Heidelberg, August 2020.

BCG$^+$22. E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, N. Resch, and P. Scholl. Correlated pseudorandomness from expand-accumulate codes. In *CRYPTO 2022, Part II*, *LNCS* 13508, pages 603–633. Springer, Heidelberg, August 2022.

BCGI18. E. Boyle, G. Couteau, N. Gilboa, and Y. Ishai. Compressing vector OLE. In *ACM CCS 2018*, pages 896–912. ACM Press, October 2018.

Bea92. D. Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO'91*, *LNCS* 576, pages 420–432. Springer, Heidelberg, August 1992.

Bea96. D. Beaver. Correlated pseudorandomness and the complexity of private computations. In *28th ACM STOC*, pages 479–488. ACM Press, May 1996.

BGI15. E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 337–367. Springer, 2015.

BGI16. E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing: Improvements and extensions. In *ACM CCS 2016*, pages 1292–1303. ACM Press, October 2016.

BGV12. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, pages 309–325. ACM, 2012.

BJMM12. A. Becker, A. Joux, A. May, and A. Meurer. Decoding random binary linear codes in $2^{n/20}$: How $1 + 1 = 0$ improves information set decoding. In *EUROCRYPT 2012*, *LNCS* 7237, pages 520–536. Springer, Heidelberg, April 2012.

BKW03. A. Blum, A. Kalai, and H. Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *Journal of the ACM (JACM)*, 50(4):506–519, 2003.

BLPS11. D. J. Bernstein, T. Lange, C. Peters, and P. Schwabe. Faster 2-regular information-set decoding. In *Coding and Cryptology, Proceedings of IWCC*, *LNCS* 6639, pages 81–98. Springer, 2011.

BM18. L. Both and A. May. Decoding linear codes with high error rate and its impact for LPN security. In *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018*, pages 25–46. Springer, Heidelberg, 2018.

BØ23. P. Briaud and M. Øygarden. A new algebraic approach to the regular syndrome decoding problem and implications for PCG constructions. In *EUROCRYPT 2023, Part V*, *LNCS* 14008, pages 391–422. Springer, Heidelberg, April 2023.

CCJ23. E. Carozza, G. Couteau, and A. Joux. Short signatures from regular syndrome decoding in the head. In *EUROCRYPT 2023, Part V*, *LNCS* 14008, pages 532–563. Springer, Heidelberg, April 2023.

CD23. G. Couteau and C. Ducros. Pseudorandom correlation functions from variable-density LPN, revisited. In *PKC 2023, Part II*, *LNCS* 13941, pages 221–250. Springer, Heidelberg, May 2023.

CDMT22. K. Carrier, T. Debris-Alazard, C. Meyer-Hilfiger, and J. Tillich. Statistical decoding 2.0: Reducing decoding to LPN. In *Advances in Cryptology - ASIACRYPT 2022*, LNCS. Springer, 2022.

CDMT24. K. Carrier, T. Debris-Alazard, C. Meyer-Hilfiger, and J. Tillich. Reduction from sparse lpn to lpn, dual attack 3.0. In *Advances in Cryptology - EUROCRYPT 2024*, LNCS. Springer, 2024.

CHT23. K. Carrier, V. Hatey, and J.-P. Tillich. Projective space stern decoding and application to sdith. *Cryptology ePrint Archive*, 2023.

CRR21. G. Couteau, P. Rindal, and S. Raghuraman. Silver: Silent VOLE and oblivious transfer from hardness of decoding structured LDPC codes. In *CRYPTO 2021, Part III*, *LNCS* 12827, pages 502–534, Virtual Event, August 2021. Springer, Heidelberg.

CT65. J. W. Cooley and J. W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comput.*, 19:297–301, 1965.

CT19. R. Canto-Torres and J. Tillich. Speeding up decoding a code with a non-trivial automorphism group up to an exponential factor. In *Proc. IEEE Int. Symposium Inf. Theory - ISIT 2019*, pages 1927–1931, 2019.

CTS16. R. Canto Torres and N. Sendrier. Analysis of Information Set Decoding for a Sub-linear Error Weight. In *Post-Quantum Cryptography - PQCrypto 2016*, Fukuoka, Japan, February 2016.

DA23. T. Debris-Alazard. Code-based cryptography: Lecture notes. arxiv:2304.03541,. 2023.

Deb23. T. Debris-Alazard. Code-based cryptography: Lecture notes, 2023. https://arxiv.org/abs/2304.03541.

DEEK24. L. Ducas, A. Esser, S. Etinski, and E. Kirshanova. Asymptotics and improvements of sieving for codes. 2024.

DNNR17. I. Damgård, J. B. Nielsen, M. Nielsen, and S. Ranellucci. The TinyTable protocol for 2-party secure computation, or: Gate-scrambling revisited. In *CRYPTO 2017, Part I*, *LNCS* 10401, pages 167–187. Springer, Heidelberg, August 2017.

DPSZ12. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO 2012*, *LNCS* 7417, pages 643–662. Springer, Heidelberg, August 2012.

Ds17. J. Doerner and a. shelat. Scaling ORAM for secure computation. In *ACM CCS 2017*, pages 523–535. ACM Press, October / November 2017.

DT17. T. Debris-Alazard and J.-P. Tillich. Statistical decoding. In *Proc. IEEE Int. Symposium Inf. Theory - ISIT 2017*, pages 1798–1802, Aachen, Germany, June 2017.

Dum89. I. Dumer. Two decoding algorithms for linear codes. *Probl. Inf. Transm.*, 25(1):17–23, 1989.

EGL82. S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. In *CRYPTO'82*, pages 205–210. Plenum Press, New York, USA, 1982.

ES23. A. Esser and P. Santini. Not just regular decoding: Asymptotics and improvements of regular syndrome decoding attacks. *Cryptology ePrint Archive*, 2023.

FGS07. M. Finiasz, P. Gaborit, and N. Sendrier. Improved Fast Syndrome Based Cryptographic Hash Functions. In *ECRYPT Hash Workshop 2007*, 2007.

FJR22. T. Feneuil, A. Joux, and M. Rivain. Syndrome decoding in the head: Shorter signatures from zero-knowledge proofs. In *CRYPTO 2022, Part II*, *LNCS* 13508, pages 541–572. Springer, Heidelberg, August 2022.

FOP$^+$16. J.-C. Faugère, A. Otmani, L. Perret, F. de Portzamparc, and J.-P. Tillich. Folding alternant and Goppa Codes with non-trivial automorphism groups. *IEEE Trans. Inform. Theory*, 62(1):184–198, 2016.

GGM86. O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM (JACM)*, 33(4):792–807, 1986.

GI14. N. Gilboa and Y. Ishai. Distributed point functions and their applications. In *Advances in Cryptology–EUROCRYPT 2014: 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings 33*, pages 640–658. Springer, 2014.

Gil99. N. Gilboa. Two party RSA key generation. In *CRYPTO'99*, *LNCS* 1666, pages 116–129. Springer, Heidelberg, August 1999.

GMW87. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

Hir16. S. Hirose. May-Ozerov Algorithm for Nearest-Neighbor Problem over Ʊq and Its Application to Information Set Decoding. In *Innovative Security Solutions for Information Technology and Communications - 9th International Conference, SECITC 2016, Bucharest, Romania, June 9-10, 2016, Revised Selected Papers*, *Lecture Notes in Computer Science* 10006, pages 115–126, 2016.

HOSS18a. C. Hazay, E. Orsini, P. Scholl, and E. Soria-Vazquez. Concretely efficient large-scale MPC with active security (or, TinyKeys for TinyOT). In *ASIACRYPT 2018, Part III*, *LNCS* 11274, pages 86–117. Springer, Heidelberg, December 2018.

HOSS18b. C. Hazay, E. Orsini, P. Scholl, and E. Soria-Vazquez. TinyKeys: A new approach to efficient multi-party computation. In *CRYPTO 2018, Part III*, *LNCS* 10993, pages 3–33. Springer, Heidelberg, August 2018.

IKNP03. Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *CRYPTO 2003*, *LNCS* 2729, pages 145–161. Springer, Heidelberg, August 2003.

Jab01. A. A. Jabri. A statistical decoding algorithm for general linear block codes. In *Cryptography and coding. Proceedings of the 8th IMA International Conference*, *LNCS* 2260, pages 1–8, Cirencester, UK, December 2001. Springer.

KD08. R. Kutzelnigg and M. Drmota. *Random bipartite graphs and their application to Cuckoo Hashing.* na, 2008.

Kel20. M. Keller. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 1575–1590, 2020.

KKRT16. V. Kolesnikov, R. Kumaresan, M. Rosulek, and N. Trieu. Efficient batched oblivious PRF with applications to private set intersection. In *ACM CCS 2016*, pages 818–829. ACM Press, October 2016.

KOS16. M. Keller, E. Orsini, and P. Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In *ACM CCS 2016*, pages 830–842. ACM Press, October 2016.

KPR18. M. Keller, V. Pastro, and D. Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT 2018, Part III*, *LNCS* 10822, pages 158–189. Springer, Heidelberg, April / May 2018.

KS08. V. Kolesnikov and T. Schneider. Improved garbled circuit: Free xor gates and applications. In *Automata, Languages and Programming: 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II 35*, pages 486–498. Springer, 2008.

LB88. P. J. Lee and E. F. Brickell. An observation on the security of McEliece's public-key cryptosystem. In *EUROCRYPT'88*, *LNCS* 330, pages 275–280. Springer, Heidelberg, May 1988.

MMT11. A. May, A. Meurer, and E. Thomae. Decoding random linear codes in $\tilde{\mathcal{O}}(2^{0.054n})$. In *ASIACRYPT 2011*, *LNCS* 7073, pages 107–124. Springer, Heidelberg, December 2011.

MO15. A. May and I. Ozerov. On computing nearest neighbors with applications to decoding of binary linear codes. In *Advances in Cryptology - EUROCRYPT 2015*, *LNCS* 9056, pages 203–228. Springer, 2015.

MS86. F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes.* North–Holland, Amsterdam, fifth edition, 1986.

MT23. C. Meyer-Hilfiger and J.-P. Tillich. Rigorous foundations for dual attacks in coding theory. In *Theory of Cryptography Conference, TCC 2023*, LNCS. Springer Verlag, December 2023. to appear.

Obe07. U. Oberst. The fast fourier transform. *SIAM journal on control and optimization*, 46(2):496–540, 2007.

Ope. OpenSSL Project. OpenSSL cryptography and SSL/TLS toolkit. https://www.openssl.org/. Accessed: 2024-02-12.

Ove06. R. Overbeck. Statistical decoding revisited. In *Information security and privacy : 11th Australasian conference, ACISP 2006*, *LNCS* 4058, pages 283–294. Springer, 2006.

Pet10. C. Peters. Information-set decoding for linear codes over F_q. In *The Third International Workshop on Post-Quantum Cryptography, PQCRYPTO 2010*, pages 81–94. Springer, Heidelberg, May 2010.

Pra62. E. Prange. The use of information sets in decoding cyclic codes. *IRE Transactions on Information Theory*, 8(5):5–9, 1962.

PSWW18. B. Pinkas, T. Schneider, C. Weinert, and U. Wieder. Efficient circuit-based PSI via cuckoo hashing. In *EUROCRYPT 2018, Part III*, *LNCS* 10822, pages 125–157. Springer, Heidelberg, April / May 2018.

Rab81. M. Rabin. How to exchange secrets by oblivious transfer. *Technical Report TR-81, Harvard University,*, 1981.

Roy22. L. Roy. SoftSpokenOT: Quieter OT extension from small-field silent VOLE in the minicrypt model. In *CRYPTO 2022, Part I*, *LNCS* 13507, pages 657–687. Springer, Heidelberg, August 2022.

RR. P. Rindal and L. Roy. libOTe: an efficient, portable, and easy to use oblivious transfer library. https://github.com/osu-crypto/libOTe.

RRT23. S. Raghuraman, P. Rindal, and T. Tanguy. Expand-convolute codes for pseudorandom correlation generators from LPN. In *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part IV*, *Lecture Notes in Computer Science* 14084, pages 602–632. Springer, 2023.

RS21. P. Rindal and P. Schoppmann. VOLE-PSI: Fast OPRF and circuit-PSI from vector-OLE. In *EUROCRYPT 2021, Part II*, *LNCS* 12697, pages 901–930. Springer, Heidelberg, October 2021.

S$^+$24. W. Stein et al. *Sage Mathematics Software (Version 10.2).* The Sage Development Team, 2024. http://www.sagemath.org.

Sen11. N. Sendrier. Decoding one out of many. In *Post-Quantum Cryptography 2011*, *LNCS* 7071, pages 51–67, 2011.

Sen23. N. Sendrier. Wave parameter selection. In *Post-Quantum Cryptography*, pages 91–110, Cham, 2023. Springer Nature Switzerland.

SGRR19. P. Schoppmann, A. Gascón, L. Reichert, and M. Raykova. Distributed vector-OLE: Improved constructions and implementation. In *ACM CCS 2019*, pages 1055–1072. ACM Press, November 2019.

Ste89. J. Stern. A method for finding codewords of small weight. *Coding Theory and Applications*, 388(10):106–113, 1989.

Tor17a. R. C. Torres. Asymptotic analysis of isd algorithms for the q-ary case. *Proceedings of the Tenth International Workshop on Coding and Cryptography WCC 2017*, 2017.

Tor17b. R. C. Torres. Optimizing bjmm with nearest neighbors:full decoding in $2^{2/21n}$ and mceliece security. *WCC Workshop on Coding and Cryptography WCC 2017*, 2017.

Wag02. D. Wagner. A generalized birthday problem. In *Advances in Cryptology - CRYPTO 2002*, *LNCS* 2442, pages 288–303. Springer, 2002.

Wil21. W. Willems. *Codes in group algebras*, chapter 16. Chapman and Hall/CRC, 2021.

Yao86. A. C.-C. Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

Appendices

A PCGs from the QA-SD assumption

In this section, we recall the construction of PCGs from the *Quasi-Abelian Syndrome Decoding* assumption ($\mathsf{QA\text{-}SD}$) which was introduced in [BCCD23], and properly defined in Appendix A. We start with a short overview of the PCG construction over $\mathbb{F}_q$ of Bombar et al. [BCCD23]. Let $\mathcal{R} = \mathbb{F}_q[\mathbb{G}] = \left\{\sum_{g\in\mathbb{G}} a_g g \mid a_g \in \mathbb{F}_q\right\}$, with $\mathbb{G}$ an abelian group. We refer to $\mathcal{R}_t$ as the set of ring elements of $\mathcal{R}$ of weight at most t. The goal is to construct a PCG that would achieve the functionality described in Fig. 13.

Ideal Functionality $\mathsf{QA\text{-}SD}_{\mathsf{OLE-Setup}}$

Parameters: Security parameter 1^λ, $\mathsf{PCG}_{\mathsf{OLE}} = (\mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Gen}, \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Expand})$ as per Fig. 14.

Functionality:

```
Sample $(k_0, k_1) \leftarrow \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Gen}(1^\lambda)$.
Output $k_\sigma$ to party $P_\sigma$ for $\sigma \in \{0,1\}$.
```

Fig. 13: Functionality $\mathsf{QA\text{-}SD}_{\mathsf{OLE-Setup}}$.

The protocol is described in Fig. 14. The goal of the OLE correlation is to give the two parties a pseudorandom $x_\sigma \in \mathcal{R}$, as well as an additive sharing of the product $x_0 \cdot x_1$. To achieve this, the authors constructed the framework on the Quasi-Abelian Syndrome Decoding assumption. The players first have access to a vector $\mathbf{a} = (1, a_1, \cdots, a_{c-1})$ of elements in $\mathcal{R}$, publicly. Taking advantage of the canonical notion of the sparseness of $\mathcal{R}$, players can define $x_\sigma = \langle \mathbf{a}, \mathbf{e_\sigma} \rangle$, where $e_\sigma = (e_\sigma^0, \cdots, e_\sigma^{c-1})$ is a vector of t-sparse elements of $\mathcal{R}$, for a given t. Because of the QA-SD assumption, x_σ is pseudorandom. Giving the parties additive sharing of $x_0 \cdot x_1$ can be achieved via Function Secret Sharing. Indeed, $x_0 \cdot x_1 = \langle \mathbf{a}, \mathbf{e_0} \rangle + \langle \mathbf{a}, \mathbf{e_1} \rangle$ can be fully expressed via the elements in $(\mathbf{e_0} \otimes \mathbf{e_1})$, and the public elements in the expression $\mathbf{a} \otimes \mathbf{a}$. Therefore we want to obtain an additive sharing of $(\mathbf{e_0} \otimes \mathbf{e_1})$. Because each e_σ^i are t-sparse element in $\mathcal{R}$, all the products $e_0^i \cdot e_1^j$ are t^2-sparse element. Therefore they can easily be concisely shared using t^2 Single Point Function Secret Sharing (SPFSS). This enables the parties to obtain a short seed k_σ which contains their FSS keys to obtain the full evaluation and recover the additive sharing of $\mathbf{e_0} \otimes \mathbf{e_1}$, as well as the descriptions of $\mathbf{e_\sigma}$. Later on, the parties can use their seed to recover x_σ and their additive share z_σ of $x_0 \cdot x_1$.

Theorem 19 ([BCCD23]). *Let $\mathbb{G}$ be an Abelian group. Assume that* SPFSS *is a secure* FSS *scheme for sums of point functions and that the* $\mathsf{QA\text{-}SD}(q, c, t, \mathbb{G})$ *assumption holds. Then there exists a generic scheme to construct a* PCG *to produce one* OLE *correlation (described on Fig. 14). If the* SPFSS *is based on a* $\mathsf{PRG} : \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}$ *via the* PRG*-based construction from [BGI16], we obtain:*

- *Each party's seed has maximum size around :* $(ct)^2 \cdot ((\log|\mathbb{G}| - \log t + 1) \cdot (\lambda + 2) + \lambda + \log q) + ct(\log|\mathbb{G}| + \log q)$ *bits.*
- *The computation of* Expand *can be done with at most* $(2 + \lfloor(\log q)/\lambda\rfloor)|\mathbb{G}|c^2 t$ PRG *operations, and* $O(c^2|\mathbb{G}|\log|\mathbb{G}|)$ *operations in* $\mathbb{F}_q$.

B N-party MPC with Preprocessing from $\mathbb{F}_4$-OLEs

B.1 Secure computation in the $\mathcal{F}_{\mathsf{cBT}}$-hybrid model

In this section, we show how to securely compute arbitrary Boolean circuits in the preprocessing model, given access to an ideal functionality generating Beaver triples over $\mathbb{F}_4$. Because $\mathbb{F}_4$ is an

General construction of QA-SD$_{\mathsf{OLE}}$

Parameters: Security parameter λ, noise weight $t = t(\lambda)$, compression factor $c \geq 2$, $\mathbb{G}$ a finite abelian group, $\mathcal{R} = \mathbb{F}_q[\mathbb{G}]$. An FSS scheme (SPFSS.Gen,SPFSS.FullEval) for sums of t^2 point functions, with domain $[0 \ldots |\mathbb{G}|)$ and range $\mathbb{F}_q$.

Public Input: $c - 1$ random ring elements $a_1, \ldots, a_{c-1} \in \mathcal{R}$.

```
PCG.Gen($1^\lambda$):
1: foreach $\sigma \in \{0,1\}$, $i \in [0 \ldots c)$:
    1.1: $\mathbf{p}^{\boldsymbol{i}}_{\boldsymbol{\sigma}} \leftarrow (p^i_{\sigma,1}, \cdots, p^i_{\sigma,t})_{p^i_{\sigma,j} \in \mathbb{G}}$ and $\mathbf{v}^{\boldsymbol{i}}_{\boldsymbol{\sigma}} \leftarrow (\mathbb{F}_q^\times)^t$.
2: foreach $i, j \in [0 \ldots c)$:
    2.1: Sample FSS keys $(K_0^{i,j}, K_1^{i,j}) \xleftarrow{\$}$ SPFSS.Gen$(1^\lambda, 1^n, \mathbf{p}^{\boldsymbol{i}}_{\mathbf{0}} \otimes \mathbf{p}^{\boldsymbol{j}}_{\mathbf{1}}, \mathbf{v}^{\boldsymbol{i}}_{\mathbf{0}} \otimes \mathbf{v}^{\boldsymbol{j}}_{\mathbf{1}})$.
3: Let $\mathsf{k}_\sigma = ((K_\sigma^{i,j})_{i,j \in [0 \ldots c)}, (\mathbf{p}^{\boldsymbol{i}}_{\boldsymbol{\sigma}}, \mathbf{v}^{\boldsymbol{i}}_{\boldsymbol{\sigma}})_{i \in [0 \ldots c)})$.
4: Output $(\mathsf{k}_0, \mathsf{k}_1)$.

PCG.Expand($\sigma$, $\mathrm{K}_\sigma$):
1: Parse $\mathsf{k}_\sigma$ as $((K_\sigma^{i,j})_{i,j \in [0 \ldots c)}, (\mathbf{p}^{\boldsymbol{i}}_{\boldsymbol{\sigma}}, \mathbf{v}^{\boldsymbol{i}}_{\boldsymbol{\sigma}})_{i \in [0 \ldots c)})$.
2: foreach $i \in [0 \ldots c)$:
    2.1: Define the element of $\mathcal{R}_t$,
         $$e^i_\sigma = \sum_{j \in [0 \ldots t)} \mathbf{b}^{\boldsymbol{i}}_{\boldsymbol{\sigma}}[j] \cdot \mathbf{p}^{\boldsymbol{i}}_{\boldsymbol{\sigma}}[j].$$
3: Compute $x_\sigma = \langle \mathbf{a}, \mathbf{e}_{\boldsymbol{\sigma}} \rangle$, where $\mathbf{a} = (1, a_1, \cdots, a_{c-1}), \mathbf{e}_{\boldsymbol{\sigma}} = (e^0_\sigma, \cdots, e^{c-1}_\sigma)$.
4: foreach $i, j \in [0 \ldots c)$:
    4.1: Compute $u_{\sigma, i+cj} \leftarrow$ SPFSS.FullEval$(\sigma, K_\sigma^{i,j})$ and view it as a $c^2$ vector $\mathbf{u}_{\boldsymbol{\sigma}}$ of elements in $\mathcal{R}_{t^2}$.
5: Compute $z_\sigma = \langle \mathbf{a} \otimes \mathbf{a}, \mathbf{u}_{\boldsymbol{\sigma}} \rangle$.
6: Output $x_\sigma, z_\sigma$.
```

Fig. 14: PCG for OLE over $\mathcal{R}$, based on QA-SD.

Ideal Functionality $\mathcal{F}_{\mathsf{cBT}}(\mathbb{F})$

The functionality interacts with N parties $P_1, \ldots, P_N$ and an adversary $\mathcal{A}$.

FUNCTIONALITY:

1: Wait for the input $\mathsf{Corr} \subsetneq [N]$ from $\mathcal{A}$ consisting of the set of corrupted parties and a list $(\llbracket a \rrbracket_i, \llbracket b \rrbracket_i, \llbracket c \rrbracket_i)_{i \in \mathsf{Corr}}$ of triples in $\mathbb{F}$.

2: Wait for the command init from each party P_i for $i \notin \mathsf{Corr}$.

3: Sample $(\llbracket a \rrbracket_i, \llbracket b \rrbracket_i, \llbracket c \rrbracket_i)_{i \in [N] \setminus \mathsf{Corr}} \in \mathbb{F}^3$ uniformly at random conditioned on

$$a \cdot b = \left(\sum_{i=1}^{N} \llbracket a \rrbracket_i \right) \cdot \left(\sum_{i=1}^{N} \llbracket b \rrbracket_i \right) = \sum_{i=1}^{N} \llbracket c \rrbracket_i .$$

4: Output $(\llbracket a \rrbracket_i, \llbracket b \rrbracket_i, \llbracket c \rrbracket_i)$ to each party P_i for $i \notin \mathsf{Corr}$.

Fig. 15: Ideal corruptible functionality $\mathcal{F}_{\mathsf{cBT}}$ for sampling N-party Beaver triples over a field $\mathbb{F}$.

extension field of $\mathbb{F}_2$, we note that simply replacing the $\mathbb{F}_2$-Beaver triples with $\mathbb{F}_4$-Beaver triples in the classical instantiation of the GMW protocol in the preprocessing model works out-of-the-box. However, doing so naïvely *doubles* the communication during the online phase, from 2 bits per AND gate and per party to 4 bits per AND gate and per party (due to using masks over $\mathbb{F}_4$ instead of $\mathbb{F}_2$). In the technical overview, we introduced an improved strategy, which first converts each $\mathbb{F}_4$-triple into an $\mathbb{F}_2$-triple using one bit of communication per party, and then runs the standard GMW protocol over $\mathbb{F}_2$. To formally prove this result, we introduce on Fig. 15 a corruptible functionality for generating an N-party Beaver triple over a field $\mathbb{F}$. Here, corruptible means that the adversary can freely choose the shares obtained by the corrupted parties; it is known that this functionality suffices to securely instantiate GMW [BCG+19b] and can be securely instantiated given a programmable PCG for OLE over $\mathbb{F}$ [BCG+20b]. That is:

Theorem 20 ([BCG+19b, Theorem 19, Theorem 41]). *Assume that there is a programmable PCG for generating m OLEs over $\mathbb{F}$. Then there exists a protocol securely realizing m calls to the functionality $\mathcal{F}_{\mathsf{cBT}}(\mathbb{F})$ in Fig. 15 using $N \cdot (N-1)$ instances of a protocol to securely distribute the seeds of the PCG, and no further communication.*

We formally state our construction as a protocol $\Pi_{\mathsf{BT}}(\mathbb{F}_4 \to \mathbb{F}_2)$ that securely instantiates the $\mathcal{F}_{\mathsf{cBT}}(\mathbb{F}_2)$ functionality in the $\mathcal{F}_{\mathsf{cBT}}(\mathbb{F}_4)$-hybrid model, using a single call to $\mathcal{F}_{\mathsf{cBT}}(\mathbb{F}_4)$ and one bit of communication per party. The protocol is represented in Fig. 16.

Protocol $\Pi_{\mathsf{BT}}(\mathbb{F}_4 \to \mathbb{F}_2)$

PROTOCOL:

1: The parties invoke the functionality $\mathcal{F}_{\mathsf{cBT}}(\mathbb{F}_4)$ with init. Each party P_i receives a triple $(\llbracket a \rrbracket_i^4, \llbracket b \rrbracket_i^4, \llbracket c \rrbracket_i^4) \in \mathbb{F}_4^3$.

2: Each party P_i broadcasts $\llbracket b \rrbracket_i^4(1)$. All parties reconstruct $b(1) = \sum_{i=1}^{N} \llbracket b \rrbracket_i^4(1)$.

OUTPUT: Each party P_i outputs $(\llbracket a \rrbracket_i^4(0), \llbracket b \rrbracket_i^4(0), \llbracket c \rrbracket_i^4(0) + b(1) \cdot \llbracket a \rrbracket_i^4(1))$.

Fig. 16: An N-party protocol $\Pi_{\mathsf{BT}}(\mathbb{F}_4 \to \mathbb{F}_2)$ that securely realizes the $\mathcal{F}_{\mathsf{cBT}}(\mathbb{F}_2)$ functionality in the $\mathcal{F}_{\mathsf{cBT}}(\mathbb{F}_4)$-hybrid model.

Lemma 21. *The protocol $\Pi_{\mathsf{BT}}(\mathbb{F}_4 \to \mathbb{F}_2)$ of Fig. 16 securely realizes the $\mathcal{F}_{\mathsf{cBT}}(\mathbb{F}_2)$ corruptible functionality in the $\mathcal{F}_{\mathsf{cBT}}(\mathbb{F}_4)$-hybrid model, using one bit of communication per party and a single call to $\mathcal{F}_{\mathsf{cBT}}(\mathbb{F}_4)$.*

Combining this lemma with the GMW protocol yields:

Corollary 22. *There exists an N-party computation protocol that securely evaluates all Boolean circuits with m AND gates in the preprocessing model using m calls to the $\mathcal{F}_{\mathsf{cBT}}(\mathbb{F}_4)$ functionality. The protocol uses $N \cdot m$ bits of communication in the preprocessing phase, and two bits of communication per AND gate and per party in the online phase.*

Proof of Lemma 21.

Proof. Sim emulates the $\mathcal{F}_{\mathsf{cBT}}(\mathbb{F}_4)$ functionality, and receives from $\mathcal{A}$ the set $\mathsf{Corr} \subsetneq [N]$ of corrupted parties and the list $([\![a]\!]_i^4, [\![b]\!]_i^4, [\![c]\!]_i^4)_{i\in\mathsf{Corr}}$ of corrupted triples over $\mathbb{F}_4$. On behalf of each honest party P_i for $i \notin \mathsf{Corr}$, Sim broadcast a uniformly random bit $[\![b]\!]_i^4(1)$. Then, Sim reconstructs $b(1) \leftarrow \sum_{i=1}^N [\![b]\!]_i^4(1)$ and sends $([\![a]\!]_i^4(0), [\![b]\!]_i^4(0), [\![c]\!]_i^4(0) + b(1) \cdot [\![a]\!]_i^4(1))_{i\in\mathsf{Corr}}$ to the ideal functionality $\mathcal{F}_{\mathsf{cBT}}(\mathbb{F}_2)$ on behalf of the ideal adversary. As the $[\![b]\!]_i^4(1)$ are sampled uniformly and independently at random by $\mathcal{F}_{\mathsf{cBT}}(\mathbb{F}_4)$, it only remains to show that the honest parties output in an execution of $\mathit{\Pi}_{\mathsf{BT}}(\mathbb{F}_4 \to \mathbb{F}_2)$ is distributed as the output of the adversaries from $\mathcal{F}_{\mathsf{cBT}}(\mathbb{F}_2)$ in the simulated game. In turn, this follows immediately from the fact that from the viewpoint of $\mathcal{A}$, the shares $[\![a]\!]_i^4(0), [\![b]\!]_i^4(0)$ are uniformly distributed for every $i \notin \mathsf{Corr}$, and the values $[\![c]\!]_i^4(0) + b(1) \cdot [\![a]\!]_i^4(1)$ for $i \notin \mathsf{Corr}$ form uniformly random shares of

$$\begin{aligned}\sum_{i\notin\mathsf{Corr}} \left([\![c]\!]_i^4(0) + b(1) \cdot [\![a]\!]_i^4(1)\right) &= \sum_{i=1}^{N} \left([\![c]\!]_i^4(0) + b(1) \cdot [\![a]\!]_i^4(1)\right) - C \\ &= a(0)b(0) + a(1)b(1) + b(1) \cdot a(1) - C \\ &= a(0)b(0) - C,\end{aligned}$$

where $C \leftarrow \sum_{i\in\mathsf{Corr}} [\![c]\!]_i^4(0) + b(1) \cdot [\![a]\!]_i^4(1)$ denote the sum of the corrupted parties' last output. This concludes the proof. □

B.2 An improved protocol for $N = 2$ parties

In the previous section, we described how N parties can construct an $\mathbb{F}_2$-triple using one invocation to $\mathcal{F}_{\mathsf{cBT}}(\mathbb{F}_4)$ and N bits of communication. Typically, the functionality $\mathcal{F}_{\mathsf{cBT}}(\mathbb{F}_4)$ is realized by making $N \cdot (N-1)$ calls to a (programmable) OLE functionality over $\mathbb{F}_4$. When $N = 2$, this translates to using two calls to the OLE functionality, and two bits of communication. In this section, we introduce an improved construction, where $N = 2$ parties generate a Beaver triple over $\mathbb{F}_2$ using a *single* call to an OLE functionality over $\mathbb{F}_4$, *and no communication.* The corruptible functionality for generating OLEs over $\mathbb{F}_4$ is represented on Fig. 17.

Ideal Functionality $\mathcal{F}_{\mathsf{cOLE}}(\mathbb{F})$

The functionality interacts with 2 parties A, B and an adversary $\mathcal{A}$.

FUNCTIONALITY:

1: Wait for an input $(\mathsf{Corr}, u, v) \in \{A, B, \emptyset\} \times \mathbb{F} \times \mathbb{F}$ from $\mathcal{A}$, and for the command init from each party.

2: If $\mathsf{Corr} = \emptyset$, sample $(a, b, [\![ab]\!]_A) \xleftarrow{\$} \mathbb{F}^3$ and set $[\![ab]\!]_B \leftarrow a \cdot b - [\![ab]\!]_A$. If $\mathsf{Corr} = A$, sample $b \xleftarrow{\$} \mathbb{F}$, set $(a, [\![ab]\!]_A) \leftarrow (u, v)$ and $[\![ab]\!]_B \leftarrow a \cdot b - [\![ab]\!]_A$. If $\mathsf{Corr} = B$, sample $a \xleftarrow{\$} \mathbb{F}$, set $(b, [\![ab]\!]_B) \leftarrow (u, v)$ and $[\![ab]\!]_A \leftarrow a \cdot b - [\![ab]\!]_B$.

3: Output $(a, [\![ab]\!]_A)$ to A and $(b, [\![ab]\!]_B)$ to B.

Fig. 17: Ideal corruptible functionality $\mathcal{F}_{\mathsf{cOLE}}(\mathbb{F})$ for sampling an OLE correlation over a field $\mathbb{F}$.

In Fig. 18, we represent our protocol for realizing $\mathcal{F}_{\mathsf{cBT}}(\mathbb{F}_2)$ when $N = 2$ in the $\mathcal{F}_{\mathsf{cOLE}}(\mathbb{F}_4)$-hybrid model.

Protocol $\Pi(\mathbb{F}_4\mathsf{OLE} \to \mathbb{F}_2\mathsf{BT})$

PROTOCOL:

1: The parties invoke the functionality $\mathcal{F}_{\mathsf{cOLE}}(\mathbb{F}_4)$ with init, and receive $(a, \llbracket ab \rrbracket_A^4)$ and $(b, \llbracket ab \rrbracket_B^4)$, respectively.

OUTPUT:

Alice outputs $(a(0), a(1), a(0)a(1) + \llbracket ab \rrbracket_A^4(0))$ and Bob outputs $(b(1), b(0), b(0)b(1) + \llbracket ab \rrbracket_B^4(0))$.

Fig. 18: A 2-party protocol $\Pi(\mathbb{F}_4\mathsf{OLE} \to \mathbb{F}_2\mathsf{BT})$ that securely realizes the $\mathcal{F}_{\mathsf{cBT}}(\mathbb{F}_2)$ functionality in the $\mathcal{F}_{\mathsf{cOLE}}(\mathbb{F}_4)$-hybrid model.

Lemma 23. *The protocol $\Pi(\mathbb{F}_4\mathsf{OLE} \to \mathbb{F}_2\mathsf{BT})$ of Fig. 18 securely realizes the $\mathcal{F}_{\mathsf{cBT}}(\mathbb{F}_2)$ corruptible functionality for $N = 2$ parties in the $\mathcal{F}_{\mathsf{cOLE}}(\mathbb{F}_4)$-hybrid model, using no communication and a single call to $\mathcal{F}_{\mathsf{cOLE}}(\mathbb{F}_4)$.*

We refer the reader to the technical overview (Section 2.4) for the correctness analysis. The proof of security is straightforward and we omit it.

C Complexity of Informations Set Decoding algorithms over $\mathbb{F}_q$

In this section, we provide a detailed framework for the ISD algorithm, discussing its primary variations. We conduct a thorough analysis of the complexity associated with each variant, aiming to determine the most suitable choice for addressing our QA-SD assumption. Deep analysis has been conveyed on these algorithms, to improve their asymptotic complexity, but this has mainly been done for ISD over $\mathbb{F}_2$. This is all the more true in our context: due to the significant size of the involved matrices, we have to take into account the polynomial factors coming from linear algebra which are usually neglected when doing asymptotic analysis. Hence, we present a comprehensive analysis that considers polynomial factors, particularly over $\mathbb{F}_q$. It's worth noting that the general case of $\mathbb{F}_q$ has been explored in some previous works [Pet10, Tor17a, BCDL19]. We will primarily describe the overarching framework provided by [DA23], along with relevant findings from [Sen23], tailored to our specific scenario.

Let us consider the syndrome decoding assumption in its search variant: given a matrix $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$ and $\mathbf{y} \in \mathbb{F}_q^{n-k}$, find $\mathbf{e} \in \mathbb{F}_q^n$ such that the weight of $\mathbf{e}$ achieves a certain target t. The general idea behind all ISD algorithms is to pick randomly what is called an *information set*. For a set $\mathcal{I} \subset [0, n-1]$, we define by $\overline{\mathcal{I}}$ it's complementary in $[0, n-1]$. For a matrix $\mathbf{H}$, we denote by $\mathbf{H}_{\mathcal{I}}$ the restriction of $\mathbf{H}$ to the columns indicated by $\mathcal{I}$, and for a vector $\mathbf{e}$, we write $e_{\mathcal{I}}$ for the restriction of $\mathbf{e}$ to the entries indexed by $\mathcal{I}$. Given a parity check matrix $\mathbf{H}$, an information set is defined by $\mathcal{I} \subset [0, n-1], |\mathcal{I}| = k + \ell, \ell \in \mathbb{N}$, and $\mathbf{H}_{\overline{\mathcal{I}}}$ is full rank. The upcoming discussion relies on the ISD framework outlined by [DA23]. While [Sen23] also offers a comparable general framework, it introduces a permutation matrix to denote an information set, a classic selection for clarity reasons. However, in practical applications, this approach introduced overheads due to matrix multiplication. Consequently, we have opted not to pursue it.

General ISD framework. Consider two parameters $0 \le \ell \le n - k$ and $0 \le p \le \min(t, k + \ell)$

1. Pick randomly a subset $\mathcal{I} \subset [0, n-1], |\mathcal{I}| = k + \ell$, until $\mathbf{H}_{\overline{\mathcal{I}}}$ is full rank.
2. Perform a Gaussian elimination to compute a non-singular matrix $\mathbf{U} \in \mathbb{F}_q^{(n-k)\times(n-k)}$ such that $\mathbf{U}\mathbf{H}_{\overline{\mathcal{I}}} = \begin{bmatrix} I_{n-k-\ell} \\ 0 \end{bmatrix}$, and compute $\tilde{\mathbf{s}} = \mathbf{U}\mathbf{s}$. Write $\mathbf{U}\mathbf{H}_{\mathcal{I}} = \begin{bmatrix} \mathbf{H_1} \\ \mathbf{H_2} \end{bmatrix}$.
3. Compute $\mathcal{L} \subset \{\mathbf{e_2} \in \mathbb{F}_q^{k+\ell}, w_H(\mathbf{e_2}) = p, \mathbf{H_2}\mathbf{e_2} = \tilde{\mathbf{s}}_{\mathcal{I}}\}$ according to a given ISD-subroutine.
4. Find a $\mathbf{e_2} \in \mathcal{L}$ such that $\mathbf{e_1} = \tilde{\mathbf{s}}_{\overline{\mathcal{I}}} - \mathbf{H_1}\mathbf{e_2}$ has weight $t - p$. If no such $\mathbf{e_2}$ is found, return to Step 1. If it is found, return the vector $\mathbf{e}$ such that $\mathbf{e} \in \mathbb{F}_q^n$ and $\mathbf{H}\mathbf{e} = \mathbf{s} \begin{bmatrix} \mathbf{e_1} \\ \mathbf{e_2} \end{bmatrix}$.

An ISD algorithm is an iterative algorithm that loops until it finds a solution. The algorithm is therefore probabilistic, with its iterations being independent, and its complexity is given by the complexity of Steps 2 and 3, multiplied by the number of iterations (i.e., the inverse of the probability of finding an information set for which the procedure terminates). An instance of the general ISD algorithm is given by the choice of the parameters (ℓ, p) and of the ISD subroutine to compute $\mathcal{L}$. Next, based on the work of Sendrier [Sen23], we expose the practical complexity of the different variants of the ISD algorithm defined above and take into account some polynomial factors while being conservative.

Remark 24. In this analysis, we do not consider the modern and more recent attacks on syndrome decoding using nearest neighbors search [Tor17b, BM18], as they are not considered to be practical, even in the binary field case. To the best of our knowledge, there has been no work on extending these algorithms to the general case of $\mathbb{F}_q$, and we believe that it would not be practical in our range of parameters.

We denote by $T(n,k,q,*)$ the complexity of a given ISD, $*$ indicating here the different parameters of the variant considered to be tuned. As stated before an ISD algorithm is by essence probabilistic, and its complexity can be formulated like this:

$$T(n,k,q,t,*) = \frac{T_G(n,k,q,*) + C_S(n,k,q,*)}{\mathcal{P}_S(n,k,q,t,*)},$$

where $T_G(n,k,q,*)$ counts the costs of the Gaussian elimination done at each iteration, $C_S(n,k,q,*)$ is the complexity of the subroutine chosen for this ISD, and $\mathcal{P}_S(n,k,q,t,*)$ the probability of randomly selecting an Information Set that led to a solution.

Remark 25 (Gaussian Elimination Cost). For our estimations, we consider the following cost

$$T_G(n,k,q,p,\ell) = (n-k-\ell)n.$$

In other words, we consider here that Gaussian elimination costs just as much as the size of the matrix, which is a very conservative lower bound.

C.1 Stern/Dumer

The algorithm proposed by Stern [Ste89] and independently by Dummer [Dum89] corresponds to the case where we consider the following subroutine:

1. We create two lists

$$\mathcal{L}_1 = \left\{ (\mathbf{e_0} = \begin{bmatrix} \mathbf{e'} \\ \mathbf{0} \end{bmatrix}, \mathbf{H_2 e_0}), \mathbf{e'} \in \mathbb{F}_q^{\frac{k+\ell}{2}}, w_H(\mathbf{e'}) = \frac{p}{2} \right\}$$
$$\mathcal{L}_2 = \left\{ (\mathbf{e'_0} = \begin{bmatrix} \mathbf{0} \\ \mathbf{e'} \end{bmatrix}, (\tilde{\mathbf{s}}_{\mathcal{I}} - \mathbf{H_2 e'_0}), \mathbf{e'} \in \mathbb{F}_q^{\frac{k+\ell}{2}}, w_H(\mathbf{e'}) = \frac{p}{2} \right\}$$

   each list is of size $|\mathcal{L}_1| = |\mathcal{L}_2| = \binom{(k+\ell)/2}{p/2}(q-1)^{p/2}$.

   Complexity: $O(|\mathcal{L}_1|)$.

2. Search for all pairs $((\mathbf{e_0}, \mathbf{H_2 e_0}), (\tilde{\mathbf{s}}_{\mathcal{I}} - \mathbf{H_2 e'_0})) \in \times \mathcal{L}_2$ such that $\mathbf{H_2} \cdot \mathbf{e_0} = \tilde{\mathbf{s}}_{\mathcal{I}} - \mathbf{H_2 e'_0}$, and construct the list

$$\mathcal{L} = \{\mathbf{e_0} + \mathbf{e'_0}, ((\mathbf{e_0}, \mathbf{H_2 e_0}), (\mathbf{e'_0}, \tilde{\mathbf{s}}_{\mathcal{I}} - \mathbf{H_2 e'_0}) \in \mathcal{L}_1 \times \mathcal{L}_2, \mathbf{H_2} \cdot \mathbf{e_0} = \tilde{\mathbf{s}}_{\mathcal{I}} - \mathbf{H_2 e'_0}\}.$$

   The list $\mathcal{L}$ created like this satisfies that

$$\mathcal{L} \subset \{\mathbf{e_2} \in \mathbb{F}_q^{k+\ell}, w_H(\mathbf{e_2}) = p, \mathbf{H_2 e_2} = \tilde{\mathbf{s}}_{\mathcal{I}}\}.$$

   The size of the list is $|\mathcal{L}| = \binom{(k+\ell)/2}{p/2}^2 (q-1)^p q^{-l}$.

   Complexity: $O(\max(|\mathcal{L}_1|, |\mathcal{L}|))$.

3. We perform Step 4 of the general ISD framework to find e_1 of weight $t-p$.

Complexity: $O(|\mathcal{L}|(n-k-\ell)(k+\ell))$.

Proposition 26 (Stern Algorithm). *The complexity of an iteration of the Stern algorithm is given by*

$$C_S(n,k,q,p,l) = O(|\mathcal{L}|(n-k-\ell)(k+\ell) + \max(|\mathcal{L}_1|,|\mathcal{L}|) + |\mathcal{L}_1|)$$

with

$$|\mathcal{L}_1| = \binom{(k+\ell)/2}{p/2}(q-1)^{p/2} \qquad\qquad |\mathcal{L}| = \binom{(k+\ell)/2}{p/2}^2 (q-1)^p q^{-l}$$

and the probability of finding a solution is

$$\mathcal{P}_S(n,k,q,t,p,l) = \frac{\binom{k+\ell}{p}\binom{n-k-\ell}{t-p}}{\binom{n}{t}} \cdot \frac{\binom{(k+\ell)/2}{p/2}^2}{\binom{k+\ell}{p}} = \frac{\binom{(k+\ell)/2}{p/2}^2\binom{n-k-\ell}{t-p}}{\binom{n}{t}}.$$

Remark 27. The probability of success is obtained as the probability of having an error vector that has exactly p errors on the chosen information set (first fraction) multiplied by the probability that the error on the information set is equally distributed into the two halves of the corresponding vector $e_{\mathcal{I}}$.

Remark 28. The complexity given at Step 1 and 2 is voluntarily given without any polynomial coefficient, for conservative purposes. In fact, for each element of the lists, we have to perform the computation of $\tilde{\mathbf{s}}_{\mathcal{I}} - \mathbf{H_2 e'_0}$. This costs naively $\ell(\ell+k)$ per element in the list (for a total cost of $O(\ell(\ell+k)|\mathcal{L}_1|)$. Second, we have to check for equality among vectors of size l, and this costs l for each element in the list (for a total cost of $O(\max(|\mathcal{L}_1|,|\mathcal{L}|)\cdot\ell)$. Nevertheless, some optimization exists for these Steps (see [Pet10]). Therefore, we chose to stick with $O(|\mathcal{L}_1|)$ and $O(\max(|\mathcal{L}_1|,\mathcal{L}|))$, as conservative lower bounds.

Corollary 29 (Prange and Lee Brickell complexities). *The Prange [Pra62] algorithm is a very particular case of the Stern algorithm with* $p=0, \ell=0$ *and no subroutine. In the case of Prange, we have*

$$C(n,k,q,t) = 0 \qquad\qquad \mathcal{P}_S(n,k,q,t) = \frac{\binom{n-k}{t}}{\binom{n}{t}}.$$

The same goes for the Lee-Brickell algorithm [LB88], by taking $\ell = 0$ *with only* p *to optimize, and no subroutine. In the case of Lee-Brickell, we therefore have:*

$$C(n,k,q,t) = 0 \qquad\qquad \mathcal{P}_S(n,k,q,t) = \frac{\binom{k}{p}\binom{n-k}{t-p}}{\binom{n}{t}}.$$

C.2 MMT

The MMT algorithm [MMT11] of May *et al.* introduced the *representation technique*. They remark that we can split a given vector $\mathbf{e}$ of weight p into R different sums of two vectors $\mathbf{e_1}$ and $\mathbf{e_2}$, of the same size but weight $p/2$, and disjoint support.

$$\mathbf{e} = \mathbf{e_1} + \mathbf{e_2}, \quad w_H(\mathbf{e_1}) = p/2, \quad w_H(\mathbf{e_2}) = p/2.$$

Remark that $\mathbf{H_2 e} = \tilde{\mathbf{s}}_{\mathcal{I}}$ implies $\mathbf{H_2 e_1} = \tilde{\mathbf{s}}_{\mathcal{I}} - \mathbf{H_2 e_2}$. Therefore they create $\mathcal{L}_{1,0} \subset \{\mathbf{H_2 e_1}, w_H(\mathbf{e_1}) = p/2\}$ and $\mathcal{L}_{1,1} \subset \{\tilde{\mathbf{s}}_{\mathcal{I}} - \mathbf{H_2 e_2}, w_H(\mathbf{e_2}) = p/2\}$ and look for possible collisions. R is called the number of representations, and we have that

$$R = \binom{p}{p/2}.$$

Let $0 \le r = \log_q(R) \le l$. The subroutine is as follows:

1. For $0 \leq i \leq 1$, construct the following lists
$$\mathcal{L}_{i,0} = \left\{(\mathbf{e_0} = \begin{bmatrix}\mathbf{e'}\\ \mathbf{0}\end{bmatrix}, \mathbf{H_2 e_0}), \mathbf{e'} \in \mathbb{F}_q^{\frac{k+\ell}{2}}, w_H(\mathbf{e'}) = \frac{p}{4}\right\},$$
$$\mathcal{L}_{i,1} = \left\{(\mathbf{e'_0} = \begin{bmatrix}\mathbf{0}\\ \mathbf{e'}\end{bmatrix}, \tilde{\mathbf{s}}_{\mathcal{I}} - \mathbf{H_2 e'_0}), \mathbf{e'} \in \mathbb{F}_q^{\frac{k+\ell}{2}}, w_H(\mathbf{e'}) = \frac{p}{4}\right\}.$$
The size of each list is $|\mathcal{L}_{0,0}| = |\mathcal{L}_{0,1}| = |\mathcal{L}_{1,0}| = |\mathcal{L}_{1,1}| = \binom{(k+\ell)/2}{p/4}(q-1)^{p/4}$.

Complexity: $O(|\mathcal{L}_{0,0}|)$.

2. For $0 \leq i \leq 1$, and given a fix set of r entries, search for all pairs $((\mathbf{e_0}, \mathbf{H_2 e_0}),$ $(\mathbf{e'_0}, \tilde{\mathbf{s}}_{\mathcal{I}} - \mathbf{H_2 e'_0})) \in \mathcal{L}_{i,0} \times \mathcal{L}_{i,1}$ such that $\mathbf{H_2} \cdot \mathbf{e_0} = \tilde{\mathbf{s}}_{\mathcal{I}} - \mathbf{H_2 e'_0}$ on the r entries. Construct the list
$$\mathcal{L}_{i,2} = \{\mathbf{e_0} + \mathbf{e'_0}, (\mathbf{e_0}, \mathbf{H_2 e_0}), (\mathbf{e'_0}, \tilde{\mathbf{s}}_{\mathcal{I}} - \mathbf{H_2 e'_0})) \in \mathcal{L}_{i,0} \times \mathcal{L}_{i,1}, \mathbf{H_2 e_0} = \tilde{\mathbf{s}}_{\mathcal{I}} - \mathbf{H_2 e'_0}\}.$$
We have that
$$\mathcal{L}_{i,2} \subset \{\mathbf{e_2} \in \mathbb{F}_q^{k+\ell}, w_H(\mathbf{e_2}) = p/2, \mathbf{H_2 e_2} = \tilde{\mathbf{s}}_{\mathcal{I}}\}.$$
The size of the lists resulting from the merge is $|\mathcal{L}_{0,2}| = |\mathcal{L}_{1,2}| = |\mathcal{L}_{0,0}|^2/q^r$.

Complexity: $O(\max(|\mathcal{L}_{0,0}|, |\mathcal{L}_{0,2}|))$.

3. Merge the two previous lists again, to obtain $\mathcal{L}$ which contains vectors of size $k+\ell$, weight p, and with the appropriate fixed value for the remaining $\ell - r$ coordinates of their syndromes. The size of the resulting list is $|\mathcal{L}| = |\mathcal{L}_{0,0}|^4/q^{l+r}$.

Complexity: $O(\max(|\mathcal{L}_{0,2}|, |\mathcal{L}|))$.

4. We perform Step 4 of the general ISD framework to find $\mathbf{e_1}$ of weight $t - p$.

Complexity: $O(|\mathcal{L}|(n-k-\ell)(k+\ell))$.

Proposition 30 (General MMT algorithm). *The complexity of an iteration of the MMT algorithm is given by*
$$C_S(n,k,q,p,l) = O(|\mathcal{L}|(n-k-\ell)(k+\ell) + \max(|\mathcal{L}_{0,0}|, |\mathcal{L}_{0,2}|, |\mathcal{L}|) + \mathcal{L}_{0,0})$$
with
$$|\mathcal{L}_{0,0}| = \binom{(k+\ell)/2}{p/4}(q-1)^{p/4} \qquad |\mathcal{L}_{0,2}| = \binom{(k+\ell)/2}{p/4}^2 (q-1)^{p/2} q^{-r}$$
$$|\mathcal{L}| = \binom{(k+\ell)/2}{p/4}^4 (q-1)^p q^{-\ell-r} \qquad R = \binom{p}{p/2}$$
$$r = \log_q(R)$$
and the probability of finding a solution is
$$\mathcal{P}_S(n,k,q,t,p,l) = \frac{\binom{k+\ell}{p}\binom{n-k-\ell}{t-p}}{\binom{n}{t}} \cdot \frac{\binom{(k+\ell)/2}{p/4}^4}{\binom{k+\ell}{p/2}^2}.$$

Remark 31. The probability of success is obtained as the probability of having an error vector that has exactly p errors on the chosen Information Set (first fraction) multiplied by the probability that $\mathbf{e_1}$ has $p/2$ errors equally distributed into its two halves, multiplied again by the probability that $\mathbf{e_1}$ has $p/2$ errors equally distributed into its two halves.

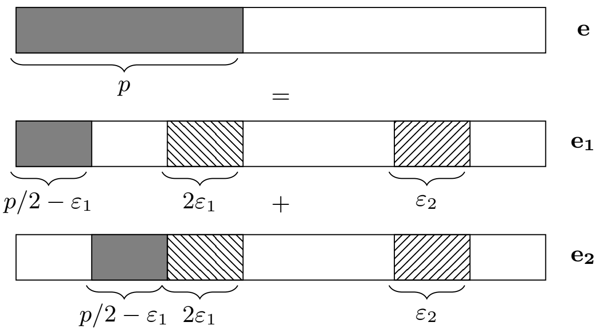

Fig. 19: BJMM representations

C.3 BJMM

The algorithm introduced by Becker et al. [BJMM12], proposes an improvement of the MMT algorithm, by increasing the number of representations one can get. Their idea follows the approach of MMT, except that an additional parameter ε is introduced. Let $p_1 = p/2 + \varepsilon$. Their idea is still to write a vector $\mathbf{e} \in \mathbb{F}_q^{k+\ell}$, of weight p as a sum.

$$\mathbf{e} = \mathbf{e_1} + \mathbf{e_2}$$

but now we want to allow their support to coincide on a proportion ε, and we ask that $w_H(\mathbf{e_1}) = w_H(\mathbf{e_2}) = p_1$.

The support of $\mathbf{e_1}$ and $\mathbf{e_2}$ can coincide in two different ways: either the sum of the entries vanishes, or they do not vanish. For this reason, we split $\varepsilon = \varepsilon_1 + \varepsilon_2$. For $\mathbf{e}$ to have a support of size p, the support of $\mathbf{e_1}$ and $\mathbf{e_2}$ should coincide on the same $2\varepsilon_1 + \varepsilon_2$ entries, and be distinct on the other $p/2 - \varepsilon_1$ entries, as represented in Figure 19. The number of representations is therefore given by

$$R = \sum_{\varepsilon_1+\varepsilon_2=\varepsilon} \binom{p}{p/2-\varepsilon_1}\binom{p/2+\varepsilon_1}{2\varepsilon_1}\binom{k+\ell-p}{\varepsilon_2}(q-1)^{2\varepsilon_1+\varepsilon_2}.$$

Let $r = \log_q(R)$. The subroutine is as follows:

1. For $0 \leq i \leq 1$ construct the following lists

$$\mathcal{L}_{i,0} = \left\{ (\mathbf{e_0} = \begin{bmatrix} \mathbf{e'} \\ \mathbf{0} \end{bmatrix}, \mathbf{H_2 e_0}), \mathbf{e'} \in \mathbb{F}_q^{\frac{k+\ell}{2}}, w_H(\mathbf{e'}) = \frac{p_1}{2} \right\},$$
$$\mathcal{L}_{i,1} = \left\{ (\mathbf{e'_0} = \begin{bmatrix} \mathbf{0} \\ \mathbf{e'} \end{bmatrix}, \tilde{\mathbf{s}}_{\mathcal{I}} - \mathbf{H_2 e'_0}), \mathbf{e'} \in \mathbb{F}_q^{\frac{k+\ell}{2}}, w_H(\mathbf{e'}) = \frac{p_1}{2} \right\}.$$

   The size of each list is $|\mathcal{L}_{0,0}| = |\mathcal{L}_{0,1}| = |\mathcal{L}_{1,0}| = |\mathcal{L}_{1,1}| = \binom{(k+\ell)/2}{p_1/2}(q-1)^{p_1/2}$.

   Complexity: $O(|\mathcal{L}_{0,0}|)$.

2. For $0 \leq i \leq 1$, and given a fix set of r entries, search for all pairs $(\mathbf{e_0}, \mathbf{e'_0}) \in \mathcal{L}_{i,0} \times \mathcal{L}_{i,1}$ such that $\mathbf{H_2} \cdot \mathbf{e_0} = \tilde{\mathbf{s}}_{\mathcal{I}} - \mathbf{H_2 e'_0}$ on the r entries. Construct the list

$$\mathcal{L}_{i,2} = \{\mathbf{e_0} + \mathbf{e'_0}, (\mathbf{e_0}, \mathbf{H_2 e_0}), (\mathbf{e'_0}, \tilde{\mathbf{s}}_{\mathcal{I}} - \mathbf{H_2 e'_0})) \in \mathcal{L}_{i,0} \times \mathcal{L}_{i,1}, \mathbf{H_2} \cdot \mathbf{e_0} = \tilde{\mathbf{s}}_{\mathcal{I}} - \mathbf{H_2 e'_0}\}.$$

   We have that

$$\mathcal{L}_{i,2} \subset \{\mathbf{e_2} \in \mathbb{F}_q^{k+\ell}, w_H(\mathbf{e_2}) = p_1/2, \mathbf{H_2 e_2} = \tilde{\mathbf{s}}_{\mathcal{I}}\}$$

   The size of the lists resulting from the merge is $|\mathcal{L}_{0,2}| = |\mathcal{L}_{1,2}| = |\mathcal{L}_{0,0}|^2/q^r$.

   Complexity: $O(\max(|\mathcal{L}_{0,0}|, |\mathcal{L}_{0,2}|))$.

3. Merge the two previous lists again, to obtain $\mathcal{L}$ which contains vectors of size $k+\ell$, weight p, and with the appropriate fixed value for the remaining $\ell - r$ coordinates of their syndromes. The size of the resulting list is $|\mathcal{L}| = |\mathcal{L}_{0,0}|^4/q^{l+r}$.

Complexity: $O(\max(|\mathcal{L}_{0,2}|, |\mathcal{L}|))$.

4. We perform Step 4 of the general ISD framework to find $\mathbf{e_1}$ of weight $t-p$.

Complexity: $O(|\mathcal{L}|(n-k-\ell)(k+\ell))$.

Proposition 32 (BJMM algorithm). *The complexity of an iteration of the BJMM algorithm is given by*

$$C_S(n,k,q,p,l,\varepsilon) = O(|\mathcal{L}|(n-k-\ell)(k+\ell) + \max(|\mathcal{L}_{0,0}|, |\mathcal{L}_{0,2}|, |\mathcal{L}|) + \mathcal{L}_{0,0})$$

with

$$|\mathcal{L}_{0,0}| = \binom{(k+\ell)/2}{p_1/2}(q-1)^{p_1/2} \qquad |\mathcal{L}_{0,2}| = \binom{(k+\ell)/2}{p_1/2}^2 (q-1)^{p_1} q^{-r}$$

$$|\mathcal{L}| = \binom{(k+\ell)/2}{p_1/2}^4 (q-1)^{2p_1} q^{-\ell-r} \qquad p_1 = p/2+\varepsilon$$

and

$$R = \sum_{\varepsilon_1+\varepsilon_2=\varepsilon} \binom{p}{p/2-\varepsilon_1}\binom{p/2+\varepsilon_1}{2\varepsilon_1}\binom{k+\ell-p}{\varepsilon_2}(q-1)^{2\varepsilon_1+\varepsilon_2}$$

$$r = \log(R)$$

and the probability of finding a solution is

$$\mathcal{P}_S(n,k,q,t,p,l,\varepsilon) = \frac{\binom{k+\ell}{p}\binom{n-k-\ell}{t-p}}{\binom{n}{t}} \cdot \frac{\binom{(k+\ell)/2}{p_1/2}^4}{\binom{k+\ell}{p_1}^2}.$$

Remark 33. The probability of success is obtained as the probability of having an error vector that has exactly p errors on the chosen Information Set (first fraction) multiplied by the probability that $\mathbf{e_1}$ has p_1 errors equally distributed into its two halves, multiplied again by the probability that $\mathbf{e_2}$ has p_1 errors equally distributed into its two halves.

Remark 34. We want to stress the fact that in the case of the MMT and BJMM-like attacks, our estimations are quite conservative, as we do not take into account the costs implied by the memory calls. In our range of parameters, they should not be discarded in practice. Indeed, the memory size is about the size of the biggest lists computed, therefore

$$M(n,k,q,p,l) \approx \max(|\mathcal{L}_{0,2}|, |\mathcal{L}_{1,1}|, |\mathcal{L}|),$$

which can be be over 2^{50}, and entails an overhead in the computation time. This is often neglected in the literature.

D Faster seed expansion from hashing (application to silent OT)

In this section, we describe how to combine hashing techniques with our PCG construction to reduce the number of DPFs required to share the noise vector. This has the advantage of making seed expansion faster with the number of PRGs operations being $O(2|\mathbb{G}|)$, but comes at the cost of requiring a trusted setup. As such, this optimization is primarily suited to silent-OT applications where a trusted setup process is assumed.

D.1 Faster seed expansion

We propose a better $\mathsf{fQA\text{-}SD_{OLE}}$ scheme in terms of seed expansion efficiency that is based on the DPF scheme and hashing techniques. First, we take advantage of Cuckoo hashing and simple hashing [KKRT16, PSWW18, BC23, RS21] to distribute the noise positions. After that, we use a DPF scheme for each bin to obtain the shares of value in each noise position by using a DPF for each bin. In general, using our Cuckoo hashing trick, we obtain a faster seed expansion but need to assume a trusted dealer because the Doerner-shelat protocol does not apply out-of-the-box. However, our $\mathsf{fQA\text{-}SD_{OLE}}$ construction has an advantage compared to the $\mathsf{QA\text{-}SD_{OLE}}$ protocol, i.e., the computation cost of Expand is smaller and independent of t (number of noisy coordinates). Concretely, the number of PRG operations is reduced from $O(t|\mathbb{G}|)$ to $O(2|\mathbb{G}|)$. Just as with the $\mathsf{QA\text{-}SD_{OLE}}$ construction, our $\mathsf{fQA\text{-}SD_{OLE}}$ construction can be used to obtain OLE correlation over any finite field $\mathbb{F}$ (except for $\mathbb{F}_2$), however, to be consistent with the concept of our main contribution, we cast our $\mathsf{fQA\text{-}SD_{OLE}}$ over $\mathbb{F}_3$ for concrete efficiency.

D.1.1 Hashing technique. We fix K random hash functions $h_1, \cdots, h_K$, where $h_i \colon \mathbb{G} \to [m]$ and $m = O(n)$. Using these K hash functions, the formal definition and parameter choices of Cuckoo hashing and simple hashing schemes (similarly as in PSI works [KD08, KKRT16, BC23, RS21]) are as follows:

1. **Cuckoo hashing schemes** with parameters $(\mathbb{G}, K, m, n)$ enable mapping a set of n item into a table T of size m using K hash functions $(h_i)_{i \le K}$ such that each bin in T has at most one item. The algorithm takes an item $x \in \mathbb{G}$ and inserts it into the bin $T[h_1(x)]$, if this bin is occupied then evicts the item in this bin and relocates it using h_2, this process continues until all items are inserted in table T. The hashing algorithm can fail if a cycle eviction is found or a threshold number of relocations has been performed, this failure can be avoided with high probability by choosing appropriate parameters (K, m, n) or using a stash to store the last item in each cycle eviction if it exists [PSWW18]. Here, we choose the parameters such that $K = 2, m = 2n$ [BC23, RS21], and do not use a stash. Looking ahead, this will imply that insertion will have a noticeable failure probability; however, we will show that in this case, the trusted dealer can simply resample the noise vector until insertion succeeds. This induces a small bias on the noise, but a straightforward reduction shows that it does not harm the security of the underlying syndrome decoding assumption (if the insertion fails with probability α, then the reduction loses a factor α in the advantage). For $K = 2$, $m = 2n$, the failure probability of Cuckoo hashing is known to be $\sqrt{2/3} + o(1)$ [KD08], which translates to a small constant security loss. We note in passing that the same observation applies to the use of Cuckoo hashing in a previous work [SGRR19] and allows them to reduce the computational overhead of their LPN-based construction from 3 to 2 compared to their regular LPN-based construction.
2. **Simple hashing** with parameters $(\mathbb{G}, K, m)$ uses K hash functions $(h_i)_{i \le K}$ to insert each item $x \in \mathbb{G}$ to the bin $B[h_i(x)]$ of a table B of size m. With very high probability, for $m = O(n \log n)$ bins, the maximum possible items per bin is $O(\log m)$. In particular, by the randomness of hash functions, with high probability, the maximum number of items per bin (denoted as $\mathsf{max\_load}$) is bounded by $\frac{3 \ln m}{\ln \ln m}$. That is, $\Pr\left[\mathsf{max\_load} \ge \frac{3 \ln m}{\ln \ln m}\right] \le \frac{1}{m}$. To estimate our concrete efficiency, we highlight the total number of items in all bins always is $K \cdot |\mathbb{G}|$ since each item in $\mathbb{G}$ is mapped to K bins using different K hash functions.

D.1.2 OLE from hashing techniques. In our construction we have $\mathbb{G} = \prod_{i=1}^{n} \mathbb{Z}/(q-1)\mathbb{Z}$. Because we are working over $\mathbb{F}_4$, $|\mathbb{G}| = 3^n$. We later work on "balls and bins" where each bin has a different number of balls and is associated with a DPF to distributed shares of a point vector then we make use of notation DPF_n for DPF scheme with arbitrary domain $[n]$.

We reuse all notations defined in Section 2.5, the intuition of the construction is the same as in Section 2.5 except that we provide a more efficient way to distribute the position of noise coordinates before using the DPF to give each party the shares of point vector. Briefly, to construct an OLE correlation over the ring $\mathcal{R}$, we want to give the parties shares of $x_0 \cdot x_1$. Note that $x_0 \cdot x_1$ is a degree-2 function in $(\mathbf{e_0}, \mathbf{e_1})$; therefore, it suffices to share $\mathbf{e_0} \otimes \mathbf{e_1}$. Since $\mathbf{e_0}, \mathbf{e_0} \in \mathcal{R}_t^c$ both are sparse vectors of weight t over $\mathcal{R}_t$, where the product of two sparse vectors is a sparse vector with sparsity t^2. So the goal here is to securely distribute the tensor $\mathbf{e_0} \otimes \mathbf{e_1}$ to both parties as in other existing PCGs

(Figure 1 and Figure 14). Note that $\mathbf{e}_{\boldsymbol{\sigma}} = (e^0_\sigma, \cdots, e^{c-1}_\sigma)$, $\sigma \in \{0,1\}$ and each random $e^i_\sigma \in \mathbb{R}_t$ is defined by a pair of vectors $\mathbf{p}^{\boldsymbol{i}}_{\boldsymbol{\sigma}} \in |\mathbb{G}|^t, \mathbf{v}^{\boldsymbol{i}}_{\boldsymbol{\sigma}} \in \mathbb{F}_3^t$ (can be considered as set of positions of non-zero entries of a vector over $\mathbb{R}_t$ and the corresponding values of these entries). Then, $\mathbf{e_0} \otimes \mathbf{e_1}$ is defined as a vector over $\mathbb{R}_{t^2}$ where $(\mathbf{v}^{\boldsymbol{i}}_{\mathbf{0}} \otimes \mathbf{v}^{\boldsymbol{j}}_{\mathbf{1}})[k],\ k \in [t^2]$ is the value of entry in position $(\mathbf{p}^{\boldsymbol{i}}_{\mathbf{0}} \otimes \mathbf{p}^{\boldsymbol{j}}_{\mathbf{1}})[k]$.

Now, the seed generation and seed expansion are processed as follows. Simple hashing is applied to all elements in the group $\mathbb{G}$ to get a table $\mathbf{B}$ of size m where each bin in this table contains elements of $\mathbb{G}$ and items in all bins are sorted in some canonical order. Then Gen uses Cuckoo hashing and distributes each entry $(\mathbf{p}^{\boldsymbol{i}}_{\mathbf{0}} \otimes \mathbf{p}^{\boldsymbol{j}}_{\mathbf{1}})[k]$ of $\mathbf{p}^{\boldsymbol{i}}_{\mathbf{0}} \otimes \mathbf{p}^{\boldsymbol{j}}_{\mathbf{1}}$ to only one bin (denote l_k) of a table $\mathbf{T}$ of size $m = O(t^2)$ while in simple hashing this entry is inserted to K bins $\{i_1, \cdots, i_K\}$ instead of one (note that $l_k \in \{i_1, \cdots, i_K\}$), then

1. For bin l_k, using $\mathsf{DPF}_{|\mathbf{B}_{l_k}|}.\mathsf{Gen}$ to generate keys such that the point function is defined by position in bin l_k where $(\mathbf{p}^{\boldsymbol{i}}_{\mathbf{0}} \otimes \mathbf{p}^{\boldsymbol{j}}_{\mathbf{1}})[k]$ is inserted and the value output shared is $(\mathbf{v}^{\boldsymbol{i}}_{\mathbf{0}} \otimes \mathbf{v}^{\boldsymbol{j}}_{\mathbf{1}})[k]$,
2. For other bins $\{i_1, \cdots, i_K\} \setminus \{l_k\}$, in the position where $(\mathbf{p}^{\boldsymbol{i}}_{\mathbf{0}} \otimes \mathbf{p}^{\boldsymbol{j}}_{\mathbf{1}})[k]$ is inserted, the value shared is 0.

The formal constructions of key generation and key expansion are shown in Figure 20 and Figure 21 respectively.

Theorem 35. *Let $\mathcal{R} = \mathbb{F}_4[\mathbb{G}] = \mathbb{F}_4[X_1, \ldots, X_n]/(X_1^3 - 1, \ldots, X_n^3 - 1)$ where $\mathbb{G} = \prod_{i=1}^n \mathbb{Z}/3\mathbb{Z}$. Assume that DPF_n is a secure FSS scheme for point function with domain n and that the $\mathsf{QA\text{-}SD}(c,t,\mathbb{G})$ assumption holds. Then, there exists a generic construction scheme to construct a PCG to produce one OLE correlation (described in Figure 20 and Figure 21). Using the DPF [BGI16] based on a PRG $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}$ and Cuckoo hashing scheme parameters $(\mathbb{G}, K, m, t^2)$ then we obtain:*

- *Each party's seed has maximum size: $c^2 \cdot (0.9m) \cdot ((\log(3)n - \log t + 1) \cdot (\lambda + 2) + \lambda + 2) + 4t \cdot (\log(3)n + 2)$ bits.*
- *The computation of Expand can be done with at most $(2 + \lfloor 2/\lambda \rfloor) \cdot (K3^n) \cdot c^2$ PRG operations, and $O(c^2 \cdot (\log(3)n) \cdot 3^n)$ operations in $\mathbb{F}_4$.*

Proof sketch. The security of our construction is based on the security of FSS scheme and is followed the same as in [BCG+20b, BCCD23] since (1) The parameter of the simple hashing scheme is public then both parties can self-compute the table $\mathbf{B}$ obtained from this scheme (2) and $\mathbf{p}_0^i \otimes \mathbf{p}_1^j$ and $\mathbf{v}_0^i \otimes \mathbf{v}_1^j$ are mapped to the table $\mathbf{T}_{i,j}$ by using Cuckoo hashing scheme later they are inputs of DPF.

Now we argue the correctness and efficiency in turn.

Correctness. First, note that

$$\begin{aligned} e_0^i \cdot e_1^j &= \left(\sum_{k \in [0 \ldots t)} \mathbf{v}^{\boldsymbol{i}}_{\boldsymbol{\sigma}}[k] \cdot \mathbf{p}^{\boldsymbol{i}}_{\boldsymbol{\sigma}}[k] \right) \cdot \left(\sum_{l \in [0 \ldots t)} \mathbf{v}^{\boldsymbol{j}}_{\boldsymbol{\sigma}}[l] \cdot \mathbf{p}^{\boldsymbol{j}}_{\boldsymbol{\sigma}}[l] \right) \\ &= \sum_{k,l \in [0 \ldots t)} \left(\mathbf{v}^{\boldsymbol{i}}_{\mathbf{0}}[k] \cdot \mathbf{v}^{\boldsymbol{j}}_{\mathbf{1}}[l] \right) \cdot \left(\mathbf{p}^{\boldsymbol{i}}_{\mathbf{0}}[k] \cdot \mathbf{p}^{\boldsymbol{j}}_{\mathbf{1}}[l] \right) = \sum_{k \in [0 \ldots t^2)} (\mathbf{v}^{\boldsymbol{i}}_{\mathbf{0}} \otimes \mathbf{v}^{\boldsymbol{j}}_{\mathbf{1}})[k] \cdot (\mathbf{p}^{\boldsymbol{i}}_{\mathbf{0}} \otimes \mathbf{p}^{\boldsymbol{j}}_{\mathbf{1}})[k]. \end{aligned}$$

Observe that $\mathbf{u}_{0,i+cj}[k] + \mathbf{u}_{1,i+cj}[k] = \sum_{l=1}^K (\mathbf{w}_{0,l_k}[r_k] + \mathbf{w}_{1,l_k}[r_k])$ then from the correctness of DPF:

1. If $k \in \mathbf{A}_{i,j}$ i.e., k is inserted to table $\mathbf{T}_{i,j}$ using Cuckoo hashing scheme, denote t as the bin of $\mathbf{T}_{i,j}$ where k is inserted then:
$$\mathbf{u}_{0,i+cj}[k] + \mathbf{u}_{1,i+cj}[k] = \mathbf{w}_{0,t}[r_k] + \mathbf{w}_{1,t}[r_k] = (\mathbf{v}^{\boldsymbol{i}}_{\mathbf{0}} \otimes \mathbf{v}^{\boldsymbol{j}}_{\mathbf{1}})[k].$$
2. Otherwise, $k \notin \mathbf{A}_{i,j}$ then $\mathbf{u}_{0,i+cj}[k] + \mathbf{u}_{1,i+cj}[k] = 0$.

Therefore,

$$\mathbf{u}_0 + \mathbf{u}_1 = \langle \mathbf{a} \otimes \mathbf{a}, \mathbf{e}_0 \otimes \mathbf{e}_1 \rangle = \langle \mathbf{a}, \mathbf{e}_0 \rangle \cdot \langle \mathbf{a}, \mathbf{e}_1 \rangle = x_0 \cdot x_1.$$

Efficiency. We show how to obtain the party's seed size and the computation cost of Expand in Theorem 35 by using two optimizations which are tailored on the choice of Cuckoo's parameters (to reduce the number of bins) and regular noise distribution (to reduce a factor t in the number of PRG calls).

The formulas we get are adapted from [BCG+20b, BCCD23] for $|\mathbb{G}| = 3^n$ and followed by our optimizations.

Construction $\mathsf{fQA\text{-}SD_{OLE}}$

Parameters: Security parameter λ, noise weight $t = t(\lambda)$, compression factor $c = 4$, ring $\mathcal{R} = \mathbb{F}_4[X_1, \ldots, X_n]/(X_1^3 - 1, \ldots, X_n^3 - 1)$.

- An DPF_n scheme $(\mathsf{DPF}_n.\mathsf{Gen}, \mathsf{DPF}_n.\mathsf{FullEval})$ with an arbitrary domain $[n]$ and range $\mathbb{F}_4$.
- K hash functions $h_1, \cdots, h_K : \mathbb{G} \to [m]$.
- Cuckoo hashing and simple hashing schemes with parameters $(\mathbb{G}, K, m, t^2)$ and $(\mathbb{G}, K, m)$ respectively where $m = O(t^2)$.

Public Input: $c - 1$ vectors of length 3^n over $\mathbb{F}_4$, for $a_1, \cdots, a_{c-1} \in \mathcal{R}$.

```
fQA-SD.Gen $(1^\lambda)$:
1: foreach $\sigma \in \{0,1\},\ i \in [0 \ldots c)$:
    1.1: Sample random vectors $\mathbf{p}_{\boldsymbol{\sigma}}^{\boldsymbol{i}} \leftarrow (p_{\sigma,1}^i, \cdots, p_{\sigma,t}^i)_{p_{\sigma,j}^i \in \mathbb{G}}$ and $\mathbf{v}_{\boldsymbol{\sigma}}^{\boldsymbol{i}} \leftarrow (\mathbb{F}_3)^t$.
2: Hashing algorithm:
    2.1: Use simple hashing scheme $(\mathbb{G}, K, m)$ to get a table $\mathbf{B}$ having $m$ bins $(\mathbf{B}_i)_{i \le m}$ such that:
             $$\mathbf{B}_i = \{x \in \mathbb{G} \mid \exists j \in [K] \wedge h_j(x) = i\}.$$
         Each bin is sorted in some canonical order.
    2.2: foreach $i, j \in [0 \ldots c)$:
        2.2.1: Use the Cuckoo hashing scheme $(\mathbb{G}, K, m, t^2)$ to insert $\mathbf{p}_{i,j} := \mathbf{p}_{\mathbf{0}}^{\boldsymbol{i}} \otimes \mathbf{p}_{\mathbf{1}}^{\boldsymbol{j}}$ to table $\mathbf{T}_{i,j}$.
               We denote each $k$-th entry value $\mathbf{p}_{i,j}[k]$ of vector $\mathbf{p}_{i,j}$ is inserted to the bin $l_k \in [m]$ of table $\mathbf{T}_{i,j}$ i.e. $\mathbf{T}_{i,j}[l_k] = \mathbf{p}_{i,j}[k]$ and when considering in bin $\mathbf{B}_{l_k}$ of table $\mathbf{B}$ (obtained from simple hashing), denote the position of $\mathbf{p}_{i,j}[k]$ as $r_k$ i.e. $\mathbf{B}_{l_k}[r_k] = \mathbf{p}_{i,j}[k]$.
3: foreach $i, j \in [0 \ldots c),\ l_k \in [m]$:
    3.1: If $\mathbf{p}_{i,j}[k] = \mathbf{T}_{i,j}[l_k]$ and $\mathbf{p}_{i,j}[k] = \mathbf{B}_{l_k}[r_k]$, sample DPF keys for each bin $\mathbf{B}_{l_k}$:
             $$(K_{0,l_k}^{i,j}, K_{1,l_k}^{i,j}) \leftarrow \mathsf{DPF}_{|\mathbf{B}_{l_k}|}.\mathsf{Gen}(1^\lambda, r_k, \mathbf{v}_{\mathbf{0}}^{\boldsymbol{i}} \otimes \mathbf{v}_{\mathbf{1}}^{\boldsymbol{j}}[k]).$$
    3.2: Otherwise, generate randomly $r_k \leftarrow_R |\mathbf{B}_{l_k}|$ and sample DPF keys:
             $$(K_{0,l_k}^{i,j}, K_{1,l_k}^{i,j}) \leftarrow \mathsf{DPF}_{|\mathbf{B}_{l_k}|}.\mathsf{Gen}(1^\lambda, r_k, 0).$$
4: Let $\mathsf{k}_\sigma = ((K_{\sigma,l_k}^{i,j})_{i,j \in [0 \ldots c), l_k \in [m]}, ((\mathbf{p}_{\boldsymbol{\sigma}}^i, \mathbf{v}_{\boldsymbol{\sigma}}^{\boldsymbol{i}})_{i \in [0 \ldots c)})$.
5: Output $(\mathsf{k}_0, \mathsf{k}_1)$.
```

Fig. 20: Seed generation of fQA-SD over $\mathcal{R}$, based on QA-SD and hashing techniques

Construction $\mathsf{fQA\text{-}SD_{OLE}}$

```
$\mathsf{fQA\text{-}SD.Expand}\ (\sigma, \mathrm{K}_\sigma)$:
Parse $\mathsf{k}_\sigma$ as $((K^{i,j}_{\sigma,l_k})_{i,j\in[0\dots c),l_k\in[m]}, ((\mathbf{p}^i_{\boldsymbol{\sigma}}, \mathbf{v}^{\boldsymbol{i}}_{\boldsymbol{\sigma}})_{i\in[0\dots c)})$.
foreach $i \in [0\dots c)$:
    Define the element of $\mathcal{R}_t$:
        $e^i_\sigma = \sum_{j\in[0\dots t)} \mathbf{v}^{\boldsymbol{i}}_{\boldsymbol{\sigma}}[j] \cdot \mathbf{p}^{\boldsymbol{i}}_{\boldsymbol{\sigma}}[j].$
Compute $x_\sigma = \langle \mathbf{a}, \mathbf{e}_{\boldsymbol{\sigma}} \rangle$, where $\mathbf{a} = (1, a_1, \cdots, a_{c-1}), \mathbf{e}_{\boldsymbol{\sigma}} = (e^0_\sigma, \cdots, e^{c-1}_\sigma)$.
foreach $i, j \in [0\dots c)$:
    $\forall k \in [m]$, compute $\mathbf{w}_{\sigma,k} \leftarrow \mathsf{DPF}_{|\mathbf{B}_k|}.\mathsf{FullEval}(\sigma, K^{i,j}_{\sigma,k})$.
    foreach $k \in |\mathbb{G}|$:
        $\forall l \in [1, K]$, compute $h_l(k) = l_k$, then find the bin $r_k$ of $k$ in bin $\mathbf{B}_{l_k}$ i.e., $k = \mathbf{B}_{l_k}[r_k]$.
        Define a vector $\mathbf{u}_{\sigma,i+cj}$ over $\mathcal{R}$ such that $\mathbf{u}_{\sigma,i+cj}[k] = \sum_{l=1}^{K} \mathbf{w}_{\sigma,l_k}[r_k]$.
    View the set of $\mathbf{u}_{\sigma,i+cj}$ as a $c^2$ vector $\mathbf{u}_{\boldsymbol{\sigma}}$ of element in $\mathcal{R}$.
Compute $z_\sigma = \langle \mathbf{a} \otimes \mathbf{a}, \mathbf{u}_{\boldsymbol{\sigma}} \rangle$.
Output $x_\sigma, z_\sigma$.
```

Fig. 21: Seed expansion of $\mathsf{fQA\text{-}SD}$ over $\mathcal{R}$, based on $\mathsf{QA\text{-}SD}$ and hashing techniques.

1. Party's seed size, since we have m bins and each bin $(\mathbf{B}_i)_{i\in[m]}$ needs to use a $\mathsf{DPF}_{|\mathbf{B}_i|}$ scheme so in total, we have $(m \cdot c^2)$ pairs of keys $(K^{i,j}_{\sigma,l_k})_{i,j\in[0\dots c),l_k\in[m]}$ having a size of

$$c^2 \cdot m \cdot ((\log(3)n - \log t + 1) \cdot (\lambda + 2) + \lambda + 2) + 4t \cdot (\log(3)n + 2).$$

We make an observation that since the number of bins m is defined to avoid the failure of the Cuckoo hashing scheme. However, because the distributed key generation phase is honestly executed, we allow the Cuckoo hashing scheme to fail with an acceptable probability (say 90% of standard experimental failure probability [PSWW18]). In the case of failure, then we repeat the Cuckoo hashing scheme (with a new set of functions) until it succeeds (the number of repetitions is very small in expectation). This leads to an optimization for the number of bins m to be reduced to $0.9m$ while the trade-off in security is reasonable (the adversary knows the distribution of noise sampled has to make sure the Cuckoo hashing succeeds while the number of bins is only $0.9m$). Then, the seed size is reduced roughly to

$$c^2 \cdot (0.9m) \cdot ((\log(3)n - \log t + 1) \cdot (\lambda + 2) + \lambda + 2) + 4t \cdot (\log(3)n + 2).$$

2. For the number of PRG calls in seed expansion, in each bin $(\mathbf{B}_i)_{i\in[m]}$, we make use of a $\mathsf{DPF}_{|\mathbf{B}_i|}$ with domain $|\mathbf{B}_i|$ and from the property of simple hashing scheme $\sum_{i\leq m} |\mathbf{B}_i| = K \cdot |\mathbb{G}|$ then the number of PRG operations is at most $(2 + \lfloor 2/\lambda \rfloor) \cdot (K3^n) \cdot c^2$ (reduced by a factor t from the regular noise distribution).

□

D.2 Application of OLE over $\mathbb{F}_4$ to silent OT extension

In this section, we show how to convert an $\mathbb{F}_4$-OLE into a random 1-out-of-2 OT in $\mathbb{F}_2$ using a single bit of communication. To explain the observation, let us consider two parties, Alice and Bob, holding respectively $(a, [\![ab]\!]^4_A)$ and $(b, [\![ab]\!]^4_B)$ for a and $b \in \mathbb{F}_4$. We have

$$\begin{aligned} a \cdot b &= [\![a \cdot b]\!]^4_A(0) + [\![a \cdot b]\!]^4_B(0) + ([\![a \cdot b]\!]^4_A(1) + [\![a \cdot b]\!]^4_B(1))\theta \\ &= (a(0) \cdot b(0) + a(1) \cdot b(1)) + (a(0) \cdot b(1) + a(1) \cdot b(0) + a(1) \cdot b(1)) \cdot \theta, \end{aligned}$$

where θ is the primitive root of $X^2 + X + 1$. Considering only the $(a \cdot b)(1)$ term from the above equation (i.e., the parts multiplied by θ, while in conversion from single OLE to Beaver triple over $\mathbb{F}_2$ the part taken is without θ), we get that

$$(a \cdot b)(1) = [\![a \cdot b]\!]^4_A(1) + [\![a \cdot b]\!]^4_B(1) = a(0) \cdot b(1) + a(1) \cdot b(0) + a(1) \cdot b(1),$$

and therefore,

$$\underbrace{[\![a \cdot b]\!]^4_A(1)}_{\text{known by } A} + a(1) \cdot (b(0) + b(1)) = a(0) \cdot b(1) + \underbrace{[\![a \cdot b]\!]^4_B(1)}_{\text{known by } B}.$$

If Bob sends $(b(0) + b(1))$ to Alice then the equation becomes

$$\underbrace{[\![a \cdot b]\!]^4_A(1) + a(1) \cdot (b(0) + b(1))}_{\text{known by } A} = a(0) \cdot b(1) + \underbrace{[\![a \cdot b]\!]^4_B(1)}_{\text{known by } B}.$$

It turns out that if Alice as a receiver in OT sets $t = a(0), m_t = [\![a \cdot b]\!]^4_A(1) + a(1) \cdot (b(0) + b(1))$ and Bob as a sender in OT sets $m_0 = [\![a \cdot b]\!]^4_B(1), m_1 := b(1) + [\![a \cdot b]\!]^4_B(1)$ then we have an instantiation of 1-out-of-2 OT over $\mathbb{F}_2$. The correctness is followed by the above equation and security is straightforward: the only communication between the parties is sending $b(0) + b(1)$ from Bob to Alice, which is a uniform random bit in the view of Alice (from the randomness of OLE).

Efficiency. To get 3^n random OT, our main $\mathsf{QA\text{-}SD_{OLE}}$ PCG needs $2(c^2 \cdot t)$ PRG calls (omitting some common factors) along with a factor-64 speedup from the early termination optimization while $\mathsf{fQA\text{-}SD}$ only needs $(K \cdot c^2)$ PRG calls where $K = 2, t = 14$ (see Section 6) and can be optimized by the same optimizations. Hence, we estimate to get 3^N-OT over $\mathbb{F}_2$, $\mathsf{fQA\text{-}SD}$ can be about 30× faster in terms of computation compared to [RRT23], since the cost of $\mathsf{QA\text{-}SD_{OLE}}$ is essentially on par with that of [RRT23] (see Section 2 for detail).

E Deferred Proofs

E.1 Proof of Proposition 9

Proof. We prove correctness and security in turn.

Correctness. Fix an index $\alpha = (\alpha_1, \ldots, \alpha_n)$ in a ternary basis. Consider the ternary tree consisting of 3^n nodes. Let $L^\sigma_{i,j}$ be any node label at depth i and index $j \in [3^i]$ as computed by party σ. Define $L_{i,j} = L^0_{i,j} \oplus L^1_{i,j}$ and $\ell_i = 3^i\alpha_i + 3^{i-1}\alpha_{i-1} + \cdots + \alpha_1$. To prove correctness, we start by showing that the following two invariants are maintained throughout the tree:

1. For all node labels $L_{i,j}$ where $j \neq \ell_i$, $L_{i,j} = 0^\lambda \| 0$.
2. For all node labels $L_{i,j}$ where $j = \ell_i$, $L_{i,j} = L' \| 1$ where $L' \in \{0,1\}^\lambda \setminus \{0^\lambda\}$.

If we can show that the two invariants are maintained, it immediately follows that only one path along the tree contains non-zero labels. Once we've shown this, we can argue why the output at the non-zero label at the leaf is equal to β.

We first prove that all child node labels $(L_{i,3j}, L_{i,3j+1}, L_{i,3j+2})$ of parent nodes with labels $L_{i-1,j} = 0^\lambda \| 0$ are also zero. That is, $(L_{i,3j}, L_{i,3j+1}, L_{i,3j+2}) = (0^{\lambda+1})^3$. By definition, $L_{i-1,j} = L^0_{i-1,j} \oplus L^1_{i-1,j}$, where $L^\sigma_{i-1,j} = s^\sigma_{i-1} \| t^\sigma_{i-1}$, for $\sigma \in \{0,1\}$. Since $L_{i-1,j} = 0^\lambda \| 0$, it holds that $L^0_{i-1,j} = L^1_{i-1,j}$. Hence, $G(s^0_{i-1,j}) = G(s^1_{i-1,j})$, which implies that both parties compute the same τ^σ_i in Line 3.1 of $\mathsf{Traverse}$. Moreover, because $L_{i-1,j} = s_{i-1} \| t_{i-1} = 0^\lambda \| 0$, it follows that $t^0_{i-1} \oplus t^1_{i-1} = 0$. In turn, we have that $\gamma^0_i \oplus \gamma^1_i = \tau^0_i \oplus \tau^1_i$, since the correction word is not applied (it is multiplied by t^σ_{i-1}—an XOR share of zero). This implies that (1) $L_{i,3j} = s^0_{i,0} \| t^0_{i,0} \oplus s^1_{i,0} \| t^1_{i,0} = 0^\lambda \| 0$, (2) $L_{i,3j+1} = s^0_{i,1} \| t^0_{i,1} \oplus s^1_{i,1} \| t^1_{i,1} = 0^\lambda \| 0$, and (3) $L_{i,3j+2} = s^0_{i,2} \| t^0_{i,2} \oplus s^1_{i,2} \| t^1_{i,2} = 0^\lambda \| 0$, as required.

In other words, the above proves that all child node labels that are off the "special path" described by α remain zero. We must now prove that all child node labels that are on the "special path" (i.e., child nodes where the parent node is non-zero) only have one non-zero sibling after the correction word is applied.

We prove this by induction starting with the root of the tree. Note that $L_0 = s^0_0 \| t^0_0 \oplus s^1_0 \| t^1_0$ and that $t^0_0 \oplus t^1_0 = 1$ by definition (Line 2 of $\mathsf{DPF.Gen}$). Since the root label has no siblings, the base case is trivially satisfied. Now consider any parent node label $L_{i-1,j}$ of the form $L_{i-1,j} = s\|1$ for some $s \neq 0^\lambda$. Consider the child node labels $(L_{i,3j}, L_{i,3j+1}, L_{i,3j+2})$ of parent node label $L_{i-1,j}$. Since $L_{i-1,j} \neq 0^\lambda\|1$, it holds that $L^0_{i-1,j} \neq L^1_{i-1,j}$. Hence, with overwhelming probability, it holds that $G(s^0_{i-1,j}) \neq G(s^1_{i-1,j})$, which implies that both parties compute a different τ_i^σ in Line 3.1 of $\mathsf{Traverse}$. Moreover, because $t_i = t_i^0 \oplus t_i^1 = 1$, it holds that $\gamma_i = \gamma_i^0 \oplus \gamma_i^1 = (\tau_i^0 \oplus \tau_i^1) \oplus (\mathsf{CW}_{i,0}\|\mathsf{CW}_{i,1}\|\mathsf{CW}_{i,2})$. Then, by construction of $\mathsf{CW}_{i,0}$, $\mathsf{CW}_{i,1}$, $\mathsf{CW}_{i,2}$ (Lines 4.6–4.9 of $\mathsf{DPF.Gen}$), we have that $\gamma_i = s_{i,0}\|t_{i,0}\|s_{i,1}\|t_{i,1}\|s_{i,2}\|t_{i,2}$ where $s_{i,k}\|t_{i,k} = 0^\lambda\|0$ for all $k \in \{0,1,2\} \setminus \{\alpha_i\}$. In turn, we have that two (out of the three) child labels are zero. In words, this proves that at each level of the tree, only one label is non-zero, and the non-zero label is at index $\ell_i = 3^i\alpha_i + 3^{i-1}\alpha_{i-1} + \cdots + \alpha_1$ because the α_{i-1}-st child label of each non-zero parent label is always non-zero. This proves the second invariant.

To satisfy the rDPF definition, it remains to show that the non-zero leaf label of the tree is of the form $L_n = s\|1$ where s is a pseudorandom value (from the viewpoint of either party). To see this, consider the base case of $\mathsf{Traverse}$ (when $i = n+1$) for the non-zero leaf label. The output of $\mathsf{Traverse}$ is just $s_n\|t_n$. Therefore, for the non-zero child, $(s_n^0\|t_n^0 \oplus s_n^1\|t_n^1) = s_n\|t_n \oplus \mathsf{CW}_{n,\alpha_n} = s_n \oplus (s_n \oplus r_n)\|1 = r_n\|1$ by definition of CW_{n,α_n} in Line 3.8 of $\mathsf{DPF.Gen}$. It follows that r_n is pseudorandom since it is chosen uniformly at random and is masked by s_n in CW_{n,α_n}, which is pseudorandom conditioned on either key.

Security. Informally, security holds because (1) the starting labels s_0^0 and s_0^1 given to party 0 and 1, respectively, are uniformly random and (2) the correction words (i.e., pp) consist of the expanded shares of party σ masked by the pseudorandom shares of party $1-\sigma$ which makes them computationally indistinguishable from random from the viewpoint of party σ given K_σ.

Formally, we prove security via a sequence of hybrid distributions and reducing to the pseudorandomness of a GGM tree construction [GGM86] (generalized to have a branching factor of 3 to match our ternary tree). Fix $\alpha = \alpha_1\|\ldots\|\alpha_n$.

Hybrid $\mathcal{H}_0$. This hybrid consists of the DPF key K_σ as defined in Figure 5. Specifically,

$$\mathcal{H}_0 = \{\underbrace{(\overbrace{\mathsf{CW}_{1,0}\|\mathsf{CW}_{1,1}\|\mathsf{CW}_{1,2}}^{\mathsf{CW}_1}, \ldots, \overbrace{\mathsf{CW}_{n,0}\|\mathsf{CW}_{n,1}\|\mathsf{CW}_{n,2}}^{\mathsf{CW}_n})}_{\mathsf{pp}}, s_0^\sigma\|t_0^\sigma\},$$

where $\mathsf{CW}_{i,j}$, for $i \in [n]$ and $j \in \{0,1,2\}$, is defined as $\mathsf{CW}_{i,j} = s^0_{i,j}\|t^0_{i,j} \oplus s^1_{i,j}\|t^1_{i,j}$ if $j \neq \alpha_i$, $\mathsf{CW}_{i,j} = (s_{i,\alpha_i} \oplus r_i)\|(t^0_{i,j} \oplus t^1_{i,j} \oplus 1)$ if $j = \alpha_i \wedge i \neq n$, and $\mathsf{CW}_{n,j} = s_n \oplus \beta\|0$ if $j = \alpha_i \wedge i = n$.

Hybrid $\mathcal{H}_1$. In this hybrid, we define each correction word as being party σ's share masked with a uniformly random mask (as opposed to being masked by a value computed by party $1-\sigma$). Specifically,

$$\mathcal{H}_1 = \{(\overline{\mathsf{CW}}_{1,0}\|\overline{\mathsf{CW}}_{1,1}\|\overline{\mathsf{CW}}_{1,2}, \ldots, \overline{\mathsf{CW}}_{n,0}\|\overline{\mathsf{CW}}_{n,1}\|\overline{\mathsf{CW}}_{n,2}), s_0^\sigma\|t_0^\sigma\},$$

where $\overline{\mathsf{CW}}_{i,j}$, for $i \in [n]$ and $j \in \{0,1,2\}$, is defined as $\overline{\mathsf{CW}}_{i,j} = s^\sigma_{i,j}\|t^\sigma_{i,j} \oplus \mathsf{mask}_{i,j}$ if $j \neq \alpha_i$, $\overline{\mathsf{CW}}_{i,j} = (s^\sigma_{i,\alpha_i} \oplus r_i)\|(t^\sigma_{i,j} \oplus 1) \oplus \mathsf{mask}_{i,j}$ if $j = \alpha_i \wedge i \neq n$, and $\overline{\mathsf{CW}}_{n,j} = (s^\sigma_n \oplus \beta\|0) \oplus \mathsf{mask}_{n,j}$ if $j = \alpha_i \wedge i = n$. Where for each i, j, $\mathsf{mask}_{i,j} \leftarrow_R \{0,1\}^{\lambda+1}$.

Hybrid $\mathcal{H}_2$. In this hybrid, we define each correction word as being a uniformly random string of appropriate length. That is,

$$\mathcal{H}_2 = \{(\overline{\mathsf{CW}}_{1,0}, \overline{\mathsf{CW}}_{1,1}\|\overline{\mathsf{CW}}_{1,2}\| \ldots, \overline{\mathsf{CW}}_{n,0}\|\overline{\mathsf{CW}}_{n,1}\|\overline{\mathsf{CW}}_{n,2}), s_0^\sigma\|t_0^\sigma\},$$

where $\overline{\mathsf{CW}}_{i,j}$, for all $i \in [n]$ and $j \in \{0,1,2\}$, is defined as $\overline{\mathsf{CW}}_{i,j} \leftarrow_R \{0,1\}^{\lambda+1}$.

Claim. $\mathcal{H}_0 \approx_c \mathcal{H}_1$

Proof. The claim follows from the pseudorandomness of the GGM construction [GGM86] (generalized to the ternary-arity case). Specifically, each $s^{1-\sigma}_{i,j}\|t^{1-\sigma}_{i,j}$ is computed as the output of a PRG applied to a pseudorandom seed $s^{1-\sigma}_{i-1}\|t^{1-\sigma}_{i-1}$, where the starting seed $s_0^{1-\sigma}$ is sampled independently of s_0^σ. Therefore, by the pseudorandomness of the GGM construction [GGM86, Theorem 3], for all $i \in [n], j \in \{0,1,2\}$, it follows that $s^{1-\sigma}_{i,j}\|t^{1-\sigma}_{i,j}$ is also pseudorandom. We can therefore replace the pseudorandom strings with *uniformly random* strings $\mathsf{mask}_{i,j}$, which proves the claim.

Claim. $\mathcal{H}_1 \equiv \mathcal{H}_2$

Proof. The claim follows immediately by noticing that the uniformly random strings $M_{i,j}$ and $\mathsf{mask}_{i,j}$ in $\mathcal{H}_1$ act as information-theoretic masks, making the distribution identical to $\mathcal{H}_2$.

Finally, to prove security we must prove the existence of an efficient simulator $\mathcal{S}$ that generates a computationally indistinguishable DPF key K_σ on input $(1^\lambda, 1^n, \sigma)$. The existence of $\mathcal{S}$ follows trivially by the fact that $\mathcal{H}_3$ consists of a uniformly random string of length $\{0,1\}^{3(\lambda+1)+\lambda+1}$.
□

E.2 Proof of Lemma 10

Ideal Functionality $\binom{1}{3}$-OT

There are two parties, a sender and a receiver. The sender has input $(m_0, m_1, m_2) \in \{0,1\}^*$ while the receiver has input $b \in \{0,1,2\}$.

FUNCTIONALITY:

1: Wait for input $(m_0, m_1, m_2) \in \{0,1\}^*$ from the sender.
2: Wait for input $b \in \{0,1,2\}$ from the receiver.
3: Output m_b to the receiver and $\perp$ to the sender.

Fig. 22: Ideal functionality $\binom{1}{3}$-OT in the semi-honest setting.

Proof. We show correctness and security in turn.

Correctness. To prove the correctness, we show that in our construction $\mathit{\Pi}_{\mathsf{rDPF\text{-}CW}}$, $\mathsf{CW}_i = (\mathsf{CW}_{i,0}, \mathsf{CW}_{i,1}, \mathsf{CW}_{i,2})$ satisfies the following two conditions:

1. For all $j \in \{0,1,2\} \setminus \{\alpha_i\}$ then $\mathsf{CW}_{i,j} = (s^0_{i,j} \| t^0_{i,j}) \oplus (s^1_{i,j} \| t^1_{i,j})$.
2. Otherwise, $\mathsf{CW}_{i,\alpha_i} = (s^0_{i,\alpha_i} \| t^0_{i,\alpha_i}) \oplus (s^1_{i,\alpha_i} \| t^1_{i,\alpha_i}) \oplus (r_i \| 1)$.

Observe that in our construction from the definition of $\{\mathbf{C}^0_j, \mathbf{C}^1_j\}_{j\in\{0,1,2\}}$, we have $\mathbf{C}^0_{\alpha_i} \oplus \mathbf{C}^1_{\alpha_i} = \mathsf{CW}_i \oplus z^0 \oplus z^1$. Note that $\alpha_i = [\![\alpha_i]\!]_0 + [\![\alpha_i]\!]_1$. Consider,

$$\mathbf{M}^\sigma_0 = (\mathbf{C}^\sigma_0, \mathbf{C}^\sigma_1, \mathbf{C}^\sigma_2),\ \mathbf{M}^\sigma_1 = (\mathbf{C}^\sigma_1, \mathbf{C}^\sigma_2, \mathbf{C}^\sigma_0),\ \mathbf{M}^\sigma_2 = (\mathbf{C}^\sigma_2, \mathbf{C}^\sigma_0, \mathbf{C}^\sigma_1),$$

for $\sigma \in \{0,1\}$. Observe that each $\mathbf{M}^\sigma_j = (\mathbf{C}^\sigma_j, \mathbf{C}^\sigma_{j+1}, \mathbf{C}^\sigma_{j+2})$, for $j \in \{0,1,2\}$, is a cyclically shifted vector defined by shifting $(\mathbf{C}^\sigma_0, \mathbf{C}^\sigma_1, \mathbf{C}^\sigma_2)$ to the left j times.

We show that party $\sigma = 0$ obtains the correct correction word (the case where $\sigma = 1$ is symmetric). Party 0 invokes the $\binom{1}{3}$-OT as the receiver with input $[\![\alpha_i]\!]_0$, and party 1 plays the role of the sender with input

$$\mathbf{M}^1_{[\![\alpha_i]\!]_1} = (\mathbf{C}^1_{[\![\alpha_i]\!]_1}, \mathbf{C}^1_{[\![\alpha_i]\!]_1+1}, \mathbf{C}^1_{[\![\alpha_i]\!]_1+2}).$$

After invoking $\binom{1}{3}$-OT, party 0 obtains $\mathbf{C}^1_{[\![\alpha_i]\!]_1+[\![\alpha_i]\!]_0} = \mathbf{C}^1_{\alpha_i}$. (By a symmetric argument, party 1 gets $\mathbf{C}^0_{\alpha_i}$.) It is easy to see that $\mathbf{C}^1_{\alpha_i} \oplus z^0$ and $\mathbf{C}^0_{\alpha_i} \oplus z^1$ form shares of CW_i, since $(\mathbf{C}^0_{\alpha_i} \oplus z^1) \oplus (\mathbf{C}^1_{\alpha_i} \oplus z^0) = (\mathsf{CW}_i \oplus z^0 \oplus z^1) \oplus (z^0 \oplus z^1)$, where CW_i is defined identically to the i-th iteration of Fig. 5. It follows that the output of the protocol (opening of the shares of CW_i) is correct.

Security. We prove our security in the UC model. In a nutshell, the proof boils down to the security of the 1-out-of-3 oblivious transfer functionality $\binom{1}{3}$-OT. We assume $\binom{1}{3}$-OT securely realizes the ideal functionality Fig. 22 in a semi-honest setting. Since the role of the two parties is symmetric, we can build a simulator that simulates the view of both parties when one of them is corrupted. Without loss of generality, assume σ is the corrupted party, Sim interacts with $\mathcal{A}$ as in the hybrid model below.

Hybrid $\mathcal{H}_0$. This hybrid is identical to the real protocol, both parties are honest, and $\binom{1}{3}$-OT is executed honestly.

Hybrid $\mathcal{H}_1$. This hybrid is identical to $\mathcal{H}_0$ except that now Sim plays the role of $\mathcal{A}$ and inputs $\{[\![\alpha]\!]_\sigma, r_i^\sigma, (s^\sigma_{i,j} \| t^\sigma_{i,j})_{j\in\{0,1,2\}}\}$ to the ideal functionality $\mathcal{F}_{\mathsf{rDPF\text{-}CW}}$ and receives CW_i as output.

The distribution of this hybrid is identical to that of the previous hybrid since in this hybrid, Sim does not interact with $\mathcal{A}$.

Hybrid $\mathcal{H}_2$. This hybrid distribution is identical to $\mathcal{H}_1$ except now Sim emulates $\binom{1}{3}$-OT as follows.

- When $\mathcal{A}$ is the sender, learns the input vector $\mathbf{M}^\sigma_{[\![\alpha_i]\!]_\sigma}$ of $\mathcal{A}$. Since Sim knows $(s^\sigma_{i,j} \| t^\sigma_{i,j})_{j\in\{0,1,2\}}$ then from $\mathbf{M}^\sigma_{[\![\alpha_i]\!]_\sigma}$, $[\![\alpha_i]\!]_\sigma$, Sim recomputes z^σ.
- When $\mathcal{A}$ is the receiver with input $[\![\alpha_i]\!]_\sigma$, Sim plays the role of $\binom{1}{3}$-OT and gives $\mathcal{A}$ an random string $\mathbf{M}^{\bar\sigma}_{[\![\alpha_i]\!]_{\bar\sigma}}[[\![\alpha_i]\!]_\sigma]$.

This hybrid is indistinguishable from $\mathcal{H}_1$ since by assumption we assume that $\binom{1}{3}$-OT realizes the ideal functionality of one-out-of-three OTs.

Hybrid $\mathcal{H}_3$. In this hybrid, Sim defines $[\![\mathsf{CW}_i]\!]_{\bar\sigma} := \mathbf{M}^{\bar\sigma}_{[\![\alpha_i]\!]_{\bar\sigma}}[[\![\alpha_i]\!]_\sigma] \oplus z^\sigma$ and $[\![\mathsf{CW}_i]\!]_{\bar\sigma} := \mathsf{CW}_i - [\![\mathsf{CW}_i]\!]_\sigma \in \{0,1\}^{3(\lambda+1)}$. Sim plays the role of party σ, sends $[\![\mathsf{CW}_i]\!]_{\bar\sigma}$ to $\mathcal{A}$.

This hybrid is indistinguishable from the previous hybrid. First, from the definition of $[\![\mathsf{CW}_i]\!]_\sigma$ and $[\![\mathsf{CW}_i]\!]_{\bar\sigma}$, the output CW_i of ideal world and real world are identically distributed. Second, in the view of $\mathcal{A}$ after invoking the $\binom{1}{3}$-OT, the message it gets $\mathbf{M}^{\bar\sigma}_{[\![\alpha_i]\!]_{\bar\sigma}}[[\![\alpha_i]\!]_\sigma]$ is uniformly random since $z^{\bar\sigma}$ used as a mask is random over $\{0,1\}^{3(\lambda+1)}$. This concludes the proof. □

E.3 Proof of Proposition 11

Proof. **Correctness.** Fix an index $\alpha^i = (\alpha_1, \ldots, \alpha_i)$ in a ternary tree of i-th layer. The correctness follows by the correctness of $\Pi_{\mathsf{rDPF\text{-}CW}}$ Lemma 10. Since both parties invoke $\Pi_{\mathsf{rDPF\text{-}CW}}$ and get $\mathsf{CW}_i \leftarrow \Pi_{\mathsf{rDPF\text{-}CW}}(i, [\![\alpha_i]\!]_\sigma, r_i^\sigma, (s^\sigma_{i,j} \| t^\sigma_{i,j})_{j\in\{0,1,2\}})$. where $\mathsf{CW}_i = (\mathsf{CW}_{i,0}, \mathsf{CW}_{i,1}, \mathsf{CW}_{i,2})$ such that:

1. For all $j \in \{0,1,2\} \setminus \{\alpha_i\}$ then $\mathsf{CW}_{i,j} = (s^0_{i,j} \| t^0_{i,j}) \oplus (s^1_{i,j} \| t^1_{i,j})$.
2. Otherwise, $\mathsf{CW}_{i,\alpha_i} = (s^0_{i,\alpha_i} \| t^0_{i,\alpha_i}) \oplus (s^1_{i,\alpha_i} \| t^1_{i,\alpha_i}) \oplus (r_i \| 1)$.

Therefore, CW_i defined in $\Pi_{\mathsf{rDPF\text{-}CW}}$ has the same properties as CW_i defined in Fig. 5. Let $s = [\![s]\!]^0 \oplus [\![s]\!]^1$. To prove correctness of $\mathsf{rDPF.FullEval}$ when using CW_i computed via $\Pi_{\mathsf{rDPF\text{-}CW}}$, we show that for each $i \in [n]$,

$$\hat{t}_{i-1,\alpha^i} := \hat{t}^0_{i-1,\alpha^i} \oplus \hat{t}^1_{i-1,\alpha^i} = 1,$$

otherwise $\hat{t}_{i-1,\alpha^i} = 0$ for $i \in [3^i - 1] \setminus \{\alpha^i\}$. This is done by induction for $i \in [n]$. For $i = 1$, this follows immediately. For $i \geq 1$, we have

$$\begin{aligned} &(\hat{s}_{i,3j} \| \hat{t}_{i,3j} \| \hat{s}_{i,3j+1} \| \hat{t}_{i,3j+1} \| \hat{s}_{i,3j+2} \| \hat{t}_{i,3j+2}) \\ &= (s_{i,3j} \| t_{i,3j} \| s_{i,3j+1} \| t_{i,3j+1} \| s_{i,3j+2} \| t_{i,3j+2}) \oplus (\hat{t}_{i-1,j} \cdot \mathsf{CW}_i). \end{aligned}$$

Assume $\alpha^i \in \{3j^*, 3j^*+1, 3j^*+2\}$ for some $j^* \in [3^i - 1]$, from the definition of $(s^\sigma_{i,j} \| t^\sigma_{i,j})_{j\in\{0,1,2\},\sigma\in\{0,1\}}$ then we have:

1. $(s_{i,3j} \| t_{i,3j} \| s_{i,3j+1} \| t_{i,3j+1} \| s_{i,3j+2} \| t_{i,3j+2}) = (0,0,0)$ for $j \neq j^*$.
2. $\mathsf{CW}_{i,j} = s_{i,3j*+j} \| t_{i,3j*+j}$ for $j \neq \alpha_i$,
   otherwise $\mathsf{CW}_{i,\alpha_i} = (s_{i,3j*+\alpha_i} \| t_{i,3j*+\alpha_i}) \oplus (r_1 \| 1)$.

then by induction $\hat{t}_{i-1,j^*} = 1 \implies \hat{t}_{i,3j^*+\alpha_{i+1}} = 1$ otherwise, $\hat{t}_{i,j} = 1$ for $i \in [3^{i+1} - 1]$.

Security. Since our instantiation does not have any interaction between two parties except both parties invoke to $\Pi_{\mathsf{rDPF\text{-}CW}}$. So our security against semi-honest setting is directly achieved from the security of $\Pi_{\mathsf{rDPF\text{-}CW}}$ Lemma 10. Note that the distribution of our CW_i is indistinguishable from the distribution of CW_i defined in Fig. 5 and it is proved to be secure (Appendix E.1) in the view of party σ given K_σ. □

E.4 Proof of Proposition 12

Proof. Informally, the construction $\mathit{\Pi}_{\mathsf{Output\text{-}CW}}$ is very similar to the way we compute the intermediate CW_i in Fig. 7, except that (1) the inputs of the sender in each $\binom{1}{3}$-OT execution of the i-th iteration are always the share of a vector $(\mathbf{C}^1_{i,\alpha_i}\oplus z^0_i)\oplus(\mathbf{C}^0_{i,\alpha_i}\oplus z^1_i) \in (\mathbb{F}_4)^{3^i}$ (the entry of position $\alpha := \sum_{k=0}^{i} \alpha_k 3^{k-1}$ is β, all others entries are 0), and (2) $\mathit{\Pi}_{\mathsf{Output\text{-}CW}}$ is computed iteratively such that the value that each party obtains (seen as $\llbracket \mathsf{CW}_i \rrbracket_\sigma$) is kept secret.

Correctness. We show that for each $i \in [t]$,

$$\llbracket \beta \rrbracket_0 \oplus \llbracket \beta \rrbracket_1 = \beta, \text{ where, } \llbracket \beta \rrbracket_i \in (\mathbb{F}_4)^{3^t}$$

As shown in the correctness proof of Lemma 10, $\mathbf{M}^\sigma_j = (\mathbf{C}^\sigma_j, \mathbf{C}^\sigma_{j+1}, \mathbf{C}^\sigma_{j+2})$ for $j \in \{0,1,2\}$ and $\sigma \in \{0,1\}$ is obtained by cyclically shifting the vector $(\mathbf{C}^\sigma_0, \mathbf{C}^\sigma_1, \mathbf{C}^\sigma_2)$ to the left j times. Moreover, we have that $\llbracket \alpha_i \rrbracket_1 + \llbracket \alpha_i \rrbracket_0 = \alpha_i$.

Consider party 0 (the case for party 1 is symmetric). Party 0 invokes the $\binom{1}{3}$-OT functionality as the receiver with input $\llbracket \alpha_i \rrbracket_0$ and party 1 plays the role of the sender with input

$$\mathbf{M}^1_{\llbracket \alpha_i \rrbracket_1} = (\mathbf{C}^1_{i,\llbracket \alpha_i \rrbracket_1}, \mathbf{C}^1_{\llbracket \alpha_i \rrbracket_1+1}, \mathbf{C}^1_{\llbracket \alpha_i \rrbracket_1+2}),$$

(viewed as a vector of vectors). Then, it holds that party 0 obtains as output

$$\mathbf{M}^1_{\llbracket \alpha_i \rrbracket_1}[\llbracket \alpha_i \rrbracket_0] = \mathbf{C}^1_{i,\llbracket \alpha_i \rrbracket_1+\llbracket \alpha_i \rrbracket_0} = \mathbf{C}^1_{i,\alpha_i}.$$

By symmetry, party 1 then obtains $\mathbf{C}^0_{i,\alpha_i}$ by invoking the $\binom{1}{3}$-OT protocol with party 0 now playing the role of the sender. Then, letting $\llbracket \beta \rrbracket_1 = \mathbf{C}^0_{i,\alpha_i} \oplus z^1_i$ and $\llbracket \beta \rrbracket_0 = \mathbf{C}^1_{i,\alpha_i} \oplus z^0_i$ be the shares of β, because the z^0_i and z^1_i terms cancel out the masking terms added by each party, we get that

$$\llbracket \beta \rrbracket_0 \oplus \llbracket \beta \rrbracket_1 = \mathbf{C}^1_{i,\alpha_i} \oplus z^1_i \oplus \mathbf{C}^0_{i,\alpha_i} \oplus z^0_i = \begin{cases} (\beta,0,0)), & \text{if } \alpha_i = 0. \\ (0,\beta,0)), & \text{if } \alpha_i = 1. \\ (0,0,\beta)), & \text{if } \alpha_i = 2. \end{cases}$$

or in other words, $\llbracket \beta \rrbracket_0 \oplus \llbracket \beta \rrbracket_1 = \mathbf{e}_{\alpha_i} \cdot \beta$, where $\mathbf{e}_{\alpha_i}$ is the α_i-th standard basis vector over $\mathbb{F}_4^{3^i}$.

To see that correctness still holds after the t-th iteration, $\{\mathbf{C}^\sigma_{i+1,j}\}_{\{j\in\{0,1,2\}}$ in $(i+1)$-th iteration is defined by $\llbracket \beta \rrbracket_\sigma \in (\mathbb{F}_4)^{3^i}$ from the i-th iteration and adding 0's to make sure that β and the position of share value in vector of size $(\mathbb{F}_4)^{3^i}$ is $\sum_{k=0}^{i} \alpha_k 3^{k-1}$. This leads to $\mathsf{CW} := e_\alpha \cdot \beta \oplus G(s^0) \oplus G(s^1) \in (\mathbb{F}_4)^{3^t}$ where $\alpha = \sum_{i=0}^{t} \alpha_i 3^{i-1}$, as required.

Security Analysis. The simulator Sim is constructed similarly to the simulator in the proof of Lemma 10, except that Sim does not play the role of party σ to output $\llbracket \beta \rrbracket_\sigma$ for each iteration. Instead, it only outputs the share for the last iteration, which allows both parties to construct the output CW. Sim emulates $\binom{1}{3}$-OT to learn the mask $z^{\bar{\sigma}}_i$ and inputs of $\mathcal{A}$ that plays the role of the sender to $\binom{1}{3}$-OT in the i-th iteration. Then, using this information, Sim simulates the $(i+1)$-th iteration. And for the last step, to simulate the output of the honest party, Sim defines $\llbracket \mathsf{CW} \rrbracket_\sigma$ from $\llbracket \mathsf{CW} \rrbracket_\sigma$ that is constructed by emulating $\binom{1}{3}$-OT and CW the output of ideal functionality $\mathcal{F}_{\mathsf{Output\text{-}CW}}$. □