Succinct Two-Round Two-Party Signing from PCFs*

Lennart Braun[1], Geoffroy Couteau[1], Kelsey Melissaris[1], Mahshid Riahinia[2**], and Elahe Sadeghi[3]

[1] Université Paris Cité, CNRS, IRIF, Paris, France
[2] DIENS, École normale supérieure, CNRS, Paris, France
[3] University of Texas at Austin
{lbraun,couteau,Kelsey.Melissaris}@irif.fr,mahshid.riahinia@ens.fr,elahesadeghi@utexas.edu

Abstract. We introduce new two-party threshold signature schemes with strong efficiency and security. The application of our methodology to the two most popular signatures, Schnorr and ECDSA, yields two-round, two-party, stateless and deterministic signing with 4–12 ms of computation on one core of a standard laptop, extremely low communication—96 B for Schnorr, and 128 B for ECDSA—and full concurrent simulatable security. At the heart of our approach is a new pseudorandom correlation function (PCF) for vector-OLE that admits an efficient key generation protocol; we design an end-to-end maliciously-secure and highly parallelizable DKG for this PCF and, using this DKG, we obtain an estimated runtime of 44 s for the (one-time) distributed setup of Schnorr and ECDSA on one core of a standard laptop.

* This article is the full version of an article with the same title [BCMRS26], © IACR 2026 https://doi.org/10.1007/978-3-032-35374-0_4.

** This work was done when visiting IRIF, Paris.

Table of Contents

Succinct Two-Round Two-Party Signing from PCFs . . . 1
Lennart Braun, Geoffroy Couteau, Kelsey Melissaris, Mahshid Riahinia, and Elahe Sadeghi
1 Introduction . . . 4
1.1 Contributions . . . 5
1.2 Techniques . . . 7
2 Technical Overview . . . 8
2.1 PCFs for Vector-OLE . . . 8
2.2 Two-Party Signing from VOLE-PCF . . . 9
2.3 Distributing Key Generation . . . 11
2.4 A Template for Efficient VOLE-PCFs . . . 12
2.5 On the Cost of Distributing VOLE-PCFs . . . 14
2.6 Our Approach . . . 15
2.7 First Contribution: Puncturable PRFs with Distributed Setup . . . 16
2.8 Second Contribution: from PPRFs to FSS via DKG-friendliness of the Correlation-Robust Hash . . . 18
2.9 A VOLE-PCF from DKG-friendly KI-DPFs . . . 20
2.10 Two-Round Distributed Signing from the VOLE-PCF . . . 21
3 Formalized Preliminaries . . . 23
3.1 Notation . . . 23
Cycle of Elliptic Curve Groups. . . . 23
EC-GGM. . . . 23
3.2 Signature Schemes . . . 23
3.3 Pseudorandom Generators, Functions and Correlation Robust Functions . . . 25
3.4 Non-Interactive Zero Knowledge Proofs of Knowledge . . . 27
4 LPN Variants . . . 29
4.1 Learning Parity with Noise . . . 29
4.2 Filtered LPN . . . 30
4.3 Sparse Learning Parity with Regular Noise and Static Leakage . . . 31
5 Puncturable PRFs . . . 34
5.1 Definition . . . 34
5.2 PPRF Constructions . . . 35
5.3 A Compound PPRF . . . 41
6 Distributed Point Functions from Puncturable PRFs . . . 43
6.1 Definition . . . 43
6.2 DPFs from PPRFs . . . 44
7 Pseudorandom Correlation Functions for VOLE . . . 48
7.1 Definition . . . 48
7.2 The Recursive LPN-Based PCF for VOLE of [BCMRS25] . . . 50
8 An Arithmetization-Friendly Correlation Robust Hash Function . . . 53
9 Auxiliary Functionalities for Distributed Setup . . . 61
9.1 VOLE . . . 61
9.2 Zero-Knowledge with Oblivious Opening . . . 61
Notation. . . . 61
Oblivious Opening. . . . 61
CheckDL. . . . 61
9.3 MPC . . . 63
Notation. . . . 63
Point Addition in MPC. . . . 63
Realizing CheckEC. . . . 64
10 Distributed Key Generation for Distributed Point Functions . . . 66
10.1 The Ideal Functionality $\mathcal{F}_{\mathsf{DPF}}$ . . . 66

10.2 The Protocol Π_{DPF} . . . 66
10.3 Macro Properties . . . 70
10.4 Security Proof – Setup . . . 71
Correctness . . . 71
Simulation. . . . 71
10.5 Security Proof – CHHK25-PPRF Keygen . . . 71
Correctness. . . . 71
Simulation. . . . 73
10.6 Security Proof – k-NR-PPRF Keygen . . . 76
Correctness. . . . 76
Simulation. . . . 76
11 Distributed PCF Key Generation . . . 78
11.1 The Ideal Functionality $\mathcal{F}_{\mathsf{PCF}}$ . . . 78
11.2 The Protocol Π_{PCF} . . . 78
11.3 Security Proof . . . 79
12 Two-Party Signing . . . 81
12.1 The $\mathcal{F}_{\mathsf{ro\text{-}VOLE}}$-Hybrid Model . . . 81
12.2 Deterministic Two Party ECDSA . . . 85
12.3 Deterministic Two Party Schnorr . . . 98
13 Parameters . . . 100
13.1 LPN Parameters . . . 100
Number Noise Coordinates t. . . . 100
Matrix Sparsity k. . . . 100
13.2 Choosing the DPF . . . 100
Estimating the Runtime of the Distributed Setup. . . . 101
Parameter Sets for Different Optimization Goals. . . . 101
References . . . 103

1 Introduction

Threshold cryptography enables a group of users to collectively execute cryptographic algorithms while concealing all secrets from every unqualified subset of users, i.e., subsets of size smaller than the threshold. Threshold signatures, a prominent area within threshold cryptography that has attracted growing and widespread interest from both academia and industry, distribute signing power among these users.

Arguably the most common application of threshold signatures at present is the case of cryptocurrency custody. Often a two-party signing algorithm is employed to protect transactions by distributing signing authority. By far the most common signature schemes in this setting are ECDSA [JMV01] (used by, e.g., Bitcoin and Ethereum) and Schnorr [Sch90] (enabled on, e.g., Bitcoin via the Taproot upgrade). Immense research effort has been directed towards meeting the industrial demand for increasingly efficient distributed protocols for threshold ECDSA and Schnorr signatures, leading to the recent surge in related publications at top tier cryptography and security venues. A large fraction of this effort has been devoted to achieving the following highly desirable efficiency and security features for distributed signatures.

Computational Overhead. The first, and perhaps most obvious, requirement is low computational overhead for threshold signing.

Communication Complexity. Another important metric is a protocol's communication complexity. Practically, achieving low communication is fundamental in a variety of scenarios, such as when participants have limited bandwidth, batch signing (where many signatures must be issued at once), or sign with an Android or Apple device (on which the natural communication API uses push notifications that are limited to 4–5 kB [Adj+24]). Theoretically, it is similarly fundamental to understand the bounds on, and quantify the gap between, the communication required for distributed signing and the signature size, as witnessed by recent work [Kon26].

Round Complexity. An often overlooked but crucial metric is the number of communication rounds. A majority of modern distributed ECDSA and Schnorr protocols require computation in the 1–50 ms range. In this range latency often dominates runtime; a single roundtrip between two machines can take between 50–100 ms for distant machines in the US [Adj+24; DKLs19], and between 100–400 ms for machines in distant countries, one work reporting as much as 348 ms for India-Belgium communication [DKLs19]. In such settings a single round significantly impacts end-to-end performance, further motivating two-round over three-round signing protocols.

Deterministic Signing. It is widely acknowledged that randomness can be a source of critical failures, as access to good sources of entropy or well-seeded PRNGs is notoriously hard, e.g., on cloud infrastructures (see [GKMN21; KOR23] for extended discussions). As a result, signatures schemes such as EdDSA derive nonces deterministically. However, in the distributed setting where the nonce is shared by the participants this criterion is rarely met; most existing protocols resort to probabilistic nonce generation. In theory signers could derandomize nonce generation by storing a seed as part of their long-term secret key and maintaining a counter to ensure "fresh" pseudorandomness in each signing session, but this approach requires state continuity (that the state of the device can be reliably updated and read on demand) which is similarly well-known to be highly non-trivial, particularly in the face of malicious attacks, software errors, and power interruptions. Existing deterministic protocols typically incur large computational overhead compared to their randomized counterparts due to the necessity of proving that a nonce contribution was generated via the prescribed pseudorandom process. As a result, practical deterministic distributed signing is a key area of research in threshold signatures.

Existing distributed signing protocols achieve a spectrum of tradeoffs between the listed features and other notable metrics, e.g., small setup runtime, minimal cryptographic assumptions, concurrent security, simulation-based security. Our work provides a new perspective on these tradeoffs by devising a methodology to achieve unmatched performances on almost all metrics, at the cost of a slower (but still feasible) maliciously-secure distributed key generation protocol. As setup is executed once during the initialization phase, this represents a useful tradeoff. On the theoretical front, our work demonstrates the possibility of simultaneously achieving optimal round and communication complexity, with communication approaching the output signature size, while meeting state-of-the-art performance on the other key metrics, and achieving highly desireable security guarantees.

1.1 Contributions

We construct new threshold signature schemes in the two-party setting. At the heart of our methodology is a new pseudorandom correlation function (PCF) for vector oblivious linear evaluation (VOLE) correlations over a large prime-order field $\mathbb{F}_p$, henceforth referred to as a VOLE-PCF, that admits an efficient maliciously secure distributed key generation (DKG) protocol. Our methodology yields succinct two-party threshold signatures with previously unmatched efficiency and security guarantees for various standard signature schemes, such as Schnorr [Sch90] and ECDSA [JMV01].[4] A typical instantiation in our framework yields:

- competitive computation, in the 4–12 ms range on one core of a laptop,
- extremely low communication (around a hundred bytes),
- two rounds of communication,
- deterministic stateless signing, and
- universally composable security assuming the security of the underlying VOLE-PCF.

This unique combination of extreme online efficiency and strong security comes at the cost of a slower one-time setup. The end-to-end maliciously secure DKG for our VOLE-PCF requires an estimated runtime of 22 s on one core of a standard laptop and distributes keys of size 3.9 MB and 1.7 MB, respectively, for one example parameter set. Depending on the underlying signature scheme and high-level approach distributed signing requires between one and three VOLE-PCF keys, entailing a setup time in the 22–66 s range. Notably, our DKG is highly paralellizable. Thousands of parallel executions of the same underlying procedure can easily be sped up by a factor c on c cores. For example, a full setup of three VOLE-PCFs can run in under 10 s on a modern laptop equipped with 8 performance cores.

In terms of cryptographic assumptions, ourVOLE-PCF relies on (variants of) the sparse-LPN and power-DDH assumptions. Our distributed setup further assumes a "DKG-friendly" hash function, and standard tools (such as multiplication triples, which can be generated via somewhat homomorphic encryption [KPR18] or pseudorandom correlation generators [Boy+20c]). The DKG-friendly hash can be instantiated by various arithmetization-friendly hash functions, but we introduce a new efficient instantiation from a variant of the power-DDH assumption which is proven in the (elliptic curve) generic group model.

Performance for Schnorr and ECDSA. Concretely, we obtain two-round two-party Schnorr signatures with 96 B total communication (two group elements and one $\mathbb{F}_p$-element, amounting to only 50% more than the signature size), an estimated per-party runtime of 8 ms in the deterministic case, and 4 ms in the stateful non-deterministic case. Compared with the previous best deterministic two-round Schnorr signature scheme from [KOR23] which communicates 1028 B with 188 ms of per-party computation, this represents 8- and 23−fold improvements in communication and computation, respectively. FROST [KG20], the best-known two-round threshold Schnorr protocol, requires 67% more communication, and neither achieves determinism, nor simulation security, and requires the one-more discrete logarithm assumption. Even "classic Schnorr", the standard commit-and-open three-round protocol [Mak22], requires 33% more communication than our scheme. In executions between potentially distant participants the computational overhead of our protocol is essentially negligible when compared with the latency-induced overhead.

In the ECDSA setting we obtain two-round two-party ECDSA with 128 B of total communication (two group elements and two $\mathbb{F}_p$-elements, amounting to only twice the signature size) and an estimated per-party runtime of 12 ms. Compared with the previous best deterministic two-round ECDSA signatures schemes from [BHLS25; ACGMY26], even when replacing the Paillier-based eVRF from [BHLS25] with the (more efficient) PCF from [KOR23], we get a 25× reduction in communication and a 100-fold reduction in computation. When comparing our protocol with all known two-party ECDSA protocols, only [DKLs24] and the recent work of [Kon26] (both of which are not deterministic) have lower computation (3 ms and 2.2 ms respectively), but the former has 900× more communication and one more round, and the latter has 33% more communication, one more round, and (more importantly) is not concurrently secure.

Table 1 provides a comparison of our Schnorr and ECDSA protocols, using one, two, and three VOLE-PCF instances, with the state of the art. Setup costs are omitted as a significant fraction of the listed works either do not describe a setup protocol or do not report setup benchmarks, e.g., [KOR23; Kon26; BHLS25]. Nevertheless, all these protocols do (or can) admit relatively efficient distributed setups,

[4] While our methodology applies similarly to other signature schemes we focus on ECDSA and Schnorr in this work, as these are the most used schemes.

usually in the range of a few hundred milliseconds to a few seconds (for example, the heavily-optimized setup of [Adj+24] takes 650 ms). For all our protocols, the setup cost can be approximated as (number of VOLE-PCFs)×22 s (on one core of a standard laptop) when choosing parameters for our VOLE-PCF that allow up to 2^{35} threshold signing sessions. All runtimes reported for our protocols (for setup and signing) are estimates computed via an exhaustive count of all operations, individual benchmarks of some operations, and benchmarks extracted from other papers (e.g., for zero-knowledge proofs based on QuickSilver [YSWW21] or for generating multiplication triples with Overdrive [KPR18]), sometimes with extrapolation (e.g., Overdrive reports numbers over 64-bit and 128-bit fields, showing a ×2 slowdown when doubling the field size, and we extrapolate a similar slowdown when working over a 256-bit field).

Table 1. Comparison of two-party Schnorr and ECDSA signature schemes. All estimates refer to the costs (rounds, communication, computation) of the signing phase, and ignore the (one-time) setup cost, but reflect the setup assumptions in the column "Assumptions". Det stands for "deterministic", Sim for "simulatable" (non-simulatable protocols, e.g., FROST [KG20] and [Adj+24], let the adversary bias the nonce and compensate by making a stronger assumption on the signature scheme), and Conc for "concurrent" (non-concurrent protocols must run sequentially and abort as soon as a failure occurs).

Construction	Assumptions	Det.	Sim.	Conc.	Rounds	Comm.	Comp.
Two-Party Schnorr							
Classic Schnorr	Commitments	✗	✓	✓	3	128B	< 1ms
[KOR23]	DCR	✓	✓	✓	2	1028B	188ms
[BHLS25]	ROM, coDDH, DCR	✓	✓	✓	2	∼15kB	160ms
[KG20]	OMDL	✗	✗	✓	2	160B	< 1ms
Ours, 1 VOLE	PCF[1]	✗	✓	✓	2	96B	4ms
Ours, 2 VOLE	PCF[1]	✓	✓	✓	2	96B	8ms
Two-Party ECDSA							
[Lin17]	ROM, DCR	✗	✓	✗	4	0.9kB	12ms
[DKLs24]	ROM, OT	✗	✓	✓	3	115kB	3ms
[Xue+23]	ROM, DCR	✗	✓	✓	3	4.1kB	209ms
[BHLS25] (DDH)[2]	ROM, coDDH, DCR	✓	✓	✓	2	3.8kB	∼800ms
[BHLS25] (DCR)	ROM, DCR	✓	✓	✓	2	17.3kB	1168ms
[Adj+24]	multiple[3]	✗	✗	✓	2	2kB	48ms
[Kon26]	ROM, OT	✗	✗	✗	2	170B	2.2ms
[Kon26]	ROM, OT	✗	✓	✗	3	282B	2.2ms
[Bou+]	ROM, HSM, DLOG	✗	✓	✓	4	0.8kB	35ms
Ours, 1 VOLE	PCF[1], KEA, deEUF-CMA	✗	✗	✓	2	128B	4ms
Ours, 2 VOLE	PCF[1], KEA	✓	✓	✓	2	128B	12ms
Ours, 3 VOLE	PCF[1]	✓	✓	✓	2	128B	12ms

[1] PCF refers to the assumptions used for our VOLE-PCF. This includes the power-DDH assumption and the correlation robustness of our new hash function (both hold in the generic group model) as well as the (filtered) sparse-LPN assumption.
[2] We use the numbers from the recent implementation from [DNP25].
[3] [Adj+24] assumes the ROM, the generic group model, the knowledge of exponent assumption, the short-exponent discrete logarithm assumption, the strong-RSA assumption, and the DCR assumption.

Other Signature Schemes. Our VOLE-PCF can also be used as a stand-alone primitive in the distributed computation of many other signatures, including, among others, BBS/BBS+ [BBS04; CL04; ASM06], Boneh-Boyen [BB04], the Russian standard GOST [DD13], the Chinese standard SM2 [ZLLZH24], and Pointcheval-Sanders [PS16]. In all cases our VOLE-PCF admits comparable efficiency benefits. We focus in this work on Schnorr and ECDSA, the most widely used schemes, but most of our contributions apply equally to all these schemes.

1.2 Techniques

Our framework is complex and involves technical contributions at each of the protocol, primitive, and algorithm level. We give an overview here, and provide a more detailed overview of our approach and the listed techniques in Section 2.

Protocol Optimizations. The observation that VOLE-PCFs can be used to construct distributed signatures has been made independently in several works. We provide a unified presentation of these approaches, and introduce the following improvements at the protocol level.

- We introduce a "hashing trick" reminiscent of implicit zero-knowledge [BCPW15] and conditional disclosure of secrets [GIKM98] with which the VOLE-PCF can be used to authenticate nonces with *zero communication overhead*.
- We observe that this trick similarly provides an "implicit" proof of knowledge (PoK) of an exponent with zero communication overhead for a specific PoK construction under the knowledge-of-exponent (KEA) assumption.
- We observe that secure multiplication between nonce and secret key shares, as required by threshold ECDSA and various other schemes, can be used as a nonce-authentication step; a secret key share can be used as a MAC key to authenticate the nonce given a PoK of the nonce contribution (and a verification value). Combined with the previous observation, this lets us remove one VOLE-PCF instance with zero communication overhead under KEA (this corresponds to the "Ours, 2 VOLE" row under Two-Party ECDSA in Table 1).

Primitive Optimizations. We start from a recent VOLE-PCF proposal from [BCMRS25], which can be used as-is for stateful distributed signing in scenarios where a trusted dealer distributes the key. However, we observe that it does not seem to admit an efficient DKG and has shortcomings in the stateless setting, and we introduce several modifications to address these major shortcomings:

- We introduce a (parametrized family of) new puncturable pseudorandom function(s) (PPRF) from the power-DDH assumption. PPRFs are a core component of the VOLE-PCF of [BCMRS25]. Our new PPRF strikes a much better tradeoff between public parameter sizes, secret key size, and evaluation time compared to the state of the art, and its algebraic nature makes it amenable to efficient, maliciously secure DKG.
- We convert this PPRF into a known-index distributed point function (KI-DPF) in a similar fashion as [Boy+19b; Boy+19a; BCMRS25]. We observe that this conversion requires modifying the DKG protocol to include a non-blackbox evaluation of a hash function satisfying correlation-robustness against correlations defined by scalar multiplications over an elliptic curve. To address this issue, we introduce a new candidate hash function under a variant of the power-DDH assumption. Roughly, the standard power-DDH assumption states that terms of the form g^{r^i} look random over a group $\mathbb{G}$ of order p, where r^i is computed over $\mathbb{F}_p$. Our assumption asserts that terms of the form $g^{r^i.\mathsf{x}}$ look random when r is a point over an elliptic curve defined over $\mathbb{F}_p$, r^i is the exponentiation over this curve, and $.\mathsf{x}$ denotes the projection to the x-coordinate of the resulting point. To provide support for our assumption, we formally prove that it holds in the generic group model.
- The VOLE-PCF of [BCMRS25] requires the sparse-LPN assumption [Ale03]. We identify a mismatch between the security guarantees provided by this PCF and the security notion required to achieve stateless, deterministic signing (in the stateful regime, this issue disappears and the standard sparse-LPN assumption can be used). Roughly, this stems from the fact that LPN fixes a random matrix, while using a VOLE-PCF in a stateless distributed signature lets an attacker pick the messages to be signed, and this translates to allowing a form of rejection sampling over the rows of the matrix. While this does not matter asymptotically (the resulting assumption still reduces to LPN), it has a significant impact on the concrete parameters. We remedy this issue by formally introducing the filtered (sparse-)LPN assumption, that captures the attacker's ability to reject-sample the rows of the matrix. We analyze how this ability changes the security by formally bounding the attackers advantage in the linear test framework of [Boy+20a], building upon the original analysis of [BCMRS25].

Parameter Optimization and Hybrid Construction. We observe that the VOLE-PCF of [BCMRS25] has a leveled structure, where KI-DPFs of different domain sizes are used at each level. This opens the

door to a significant optimization, albeit one that makes the full construction highly complex: given all known KI-DPF constructions, including the PRG-based construction of [Boy+19a], the power-DDH-based constructions from [BCMPR24; CHHK25] (enhanced with our new correlation-robust hash), and our new family of power-DDH-based KI-DPFs, we use a script to select independently for each level the best possible primitive when optimizing either for setup time, evaluation time, or key size. We additionally leverage whenever possible the common structure between some of the candidates to merge some of the public parameters they use. We formalize this optimization by building an "all-in-one" compound KI-DPF that generates reusable public parameters and, upon invocation for generating keys for a KI-DPF over a range, selects which algorithm to run from a a family of alternatives as a function of the range.

Distributed Key Generation. Eventually, we construct a complete, end-to-end, maliciously secure distributed protocol for our compound KI-DPF (including all its internal alternative algorithms). This DKG protocol leverage a large array of standard tools, such as vector-OLEs, oblivious transfers, QuickSilver-style zero-knowledge proofs, authenticated multiplication triples, and an optimized bucket-based cut-and-choose strategy from [WYKW21]. While all the individual components are standard, putting them together in our DKG construction is non-trivial and highly technical. The full security analysis also addresses many subtleties that arise from the interplay between values shared over different algebraic structures, such as finite fields and elliptic curves.

Open Questions. Our work leaves several intriguing questions open. Perhaps the core open question is whether the DKG runtime can be significantly reduced. While a few dozen seconds is feasible for a highly parallelizable one-time setup, it remains slow. As the runtime of our DKG scales linearly with the noise weight, one possible avenue would be the VOLE-PCF from Expand-Accumulate LPN of [Boy+22] which uses a much lower weight. However, this VOLE-PCF does not use KI-DPFs, but instead (known-index) distributed *interval* functions (KI-DCFs), which are difficult to make DKG-friendly and efficient. Our PPRF-based approach does not efficiently extend to this setting. A second open question is how to eliminate the non-blackbox use of the hash functions, as this represents the main efficiency bottleneck of our DKG accounting for more than 70% of the total runtime despite being confined to a single non-blackbox evaluation per KI-DPF.

Organization Due to the complexity of the construction we devote the published version of this work to a detailed technical overview that addresses each of our contributions. All formal details with full-fledged security analyses and efficiency estimates will appear online in a full version of the paper.

2 Technical Overview

Our starting point is an observation made in previous work: pseudorandom correlation functions (PCFs, [Boy+20a]) are a powerful primitive for threshold signature schemes with unprecendentedly low communication and round complexity. Our focus is on PCFs for the n-dimensional vector-OLE (VOLE) correlation over a large prime-order field $\mathbb{F}_p$ (VOLE-PCFs), where $p = p(\lambda)$ is assumed to be of length $\Theta(\lambda)$,[5] which output correlated strings $(\boldsymbol{u}, \boldsymbol{v}) \xleftarrow{\$} \mathbb{F}_p^n \times \mathbb{F}_p^n$ to P_0, and $(\Delta, \boldsymbol{w} = \Delta \cdot \boldsymbol{u} + \boldsymbol{v})$ for random MAC key $\Delta \xleftarrow{\$} \mathbb{F}_p$ to P_1.

Notation. We write $[x]$ to denote the set $\{1, \ldots, x\}$, and $[\pm n] = \{-n, \ldots, n\}$. For a set S, we write S^* to denote $S \setminus \{0\}$, and uniform sampling is denoted $x \xleftarrow{\$} S$. Assignment of the value X to a variable x is written $x \leftarrow X$.

2.1 PCFs for Vector-OLE

A VOLE-PCF over $\mathbb{F}_p$ enables two parties, P_0 and P_1, to on-the-fly generate a pseudorandom VOLE correlation of near-unbounded length from short correlated keys. Formally, a VOLE-PCF $\mathsf{PCF} = (\mathsf{KeyGen}, \mathsf{Eval})$ is a pair of algorithms with the following syntax:

$\mathsf{PCF}.\mathsf{KeyGen}(\Delta) \to (\mathsf{k}_0^{\mathsf{PCF}}, \mathsf{k}_1^{\mathsf{PCF}})$**.** On $\Delta \in \mathbb{F}_p$ key generation outputs $(\mathsf{k}_0^{\mathsf{PCF}}, \mathsf{k}_1^{\mathsf{PCF}})$, where $\mathsf{k}_1^{\mathsf{PCF}}$ contains Δ.
$\mathsf{PCF}.\mathsf{Eval}(b, \mathsf{k}_b^{\mathsf{PCF}}, x) \to y_b$**.** On a party index $b \in \{0, 1\}$, a key $\mathsf{k}_b^{\mathsf{PCF}}$, and an input $x \in \mathcal{X}$, evaluation outputs $y_0 = (u_x, v_x) \in \mathbb{F}_p^2$ or $y_1 = w_x \in \mathbb{F}_p$.

[5] A typical choice is $|p| \approx 2\lambda$.

A VOLE-PCF is *correct* if evaluations $((u_x, v_x), w_x)$ on any x in the PCF domain are such that $w_x = \Delta \cdot u_x + v_x$. A VOLE-PCF is *pseudorandom* if no adversary given Δ can distinguish evaluations on polynomially-many adaptively-chosen x from uniform VOLE-correlated samples. A VOLE-PCF is *secure* if $\mathsf{k}_0^{\mathsf{PCF}}$ leaks no information about Δ, and if no adversary given $\mathsf{k}_1^{\mathsf{PCF}}$ can distinguish, for adaptively chosen x, evaluations $\mathsf{PCF.Eval}(0, \mathsf{k}_0^{\mathsf{PCF}}, x)$ from random VOLE-correlated pairs $(u_x, \mathsf{PCF.Eval}(1, \mathsf{k}_1^{\mathsf{PCF}}, x) - \Delta \cdot u_x)$ where $u_x \xleftarrow{\$} \mathbb{F}_p$, modeling security against corrupted P_0 and P_1 respectively. Our formalizations differ from the standard for PCFs; our definition replaces the algorithm $\mathsf{PCF.KeyGen}$ with a distributed setup protocol that admits limited adversarial influence over the PCF keys, and requires that the aforementioned properties hold against this adversary.

2.2 Two-Party Signing from VOLE-PCF

In this subsection we overview the relevance of VOLE-PCFs to two-party signing, as originally independently observed in separate contexts, for which there is independent interest in a unified and self-contained presentation. In short, VOLE-PCFs provide a beautifully simple solution to the following two distinct nontrivial tasks that are fundamental to distributed signing.

1. Authenticating a pseudorandom nonce with the guarantee that the nonce was fixed at setup, thereby replacing the standard two-round commit-and-open procedure to generate unbiased shared nonces with a single round of interaction, and serving a role analogous to that of exponent-VRFs [BHLS25] in a designated-verifier setting.
2. Succint secure multiplication between a fixed value (a signing key) and a fresh random value (a nonce), in contrast with techniques employed in previous works from, e.g., additively homomorphic encryption [Lin17; CGGMP20] or Gilboa's OT-based OLE protocol [Gil99; DKLs18; DKLs24], which are much more computation- or communication-intensive.

We elaborate on both tasks with illustrative examples that distribute signing between parties P_A and P_B.[6] To simplify our protocol descriptions we consider the case where only one party, P_A, receives output but we note that our approaches generalize to the case in which both parties receive output.

Two-Party Schnorr and Pseudorandom Nonce Authentication. Fix a cyclic (additive) group $\mathbb{G}_{\mathsf{S}}$ of order q_{S} with generator G_{S}, and a hash function $\mathsf{H}_{\mathsf{S}} \colon \mathbb{G}_{\mathsf{S}}^2 \times \{0,1\}^* \to \mathbb{F}_{q_{\mathsf{S}}}$. The Schnorr signature scheme [Sch90] works as below.

Key pairs: a uniform secret key $\mathsf{sk} \xleftarrow{\$} \mathbb{F}_{q_{\mathsf{S}}}^*$, and public key $\mathsf{pk} \leftarrow \mathsf{sk} \cdot G_{\mathsf{S}}$.

Signing a message $m \in \{0,1\}^*$**:** sample a nonce $k \xleftarrow{\$} \mathbb{F}_{q_{\mathsf{S}}}$, set $R = k \cdot G_{\mathsf{S}}$, set $e \leftarrow \mathsf{H}_{\mathsf{S}}(\mathsf{pk}, R, m)$, and output a signature $\sigma = (R, e \cdot \mathsf{sk} + k)$.

Verifying a signature $\sigma = (R, s) \in \mathbb{G}_{\mathsf{S}} \times \mathbb{F}_{q_{\mathsf{S}}}$**:** given m and pk compute $e \leftarrow \mathsf{H}_{\mathsf{S}}(\mathsf{pk}, R, m)$ and output 1 if $e \cdot \mathsf{pk} + R = s \cdot G_{\mathsf{S}}$, otherwise output 0.

We begin with a naive (insecure) first attempt at two-party Schnorr. During setup the parties sample $\mathsf{sk}_A \xleftarrow{\$} \mathbb{F}_{q_{\mathsf{S}}}$ and $\mathsf{sk}_B \xleftarrow{\$} \mathbb{F}_{q_{\mathsf{S}}}$, respectively, and define $(\mathsf{pk}_A, \mathsf{pk}_B) \leftarrow (\mathsf{sk}_A \cdot G_{\mathsf{S}}, \mathsf{sk}_B \cdot G_{\mathsf{S}})$, $\mathsf{sk} = \mathsf{sk}_A + \mathsf{sk}_B$, and $\mathsf{pk} = \mathsf{pk}_A + \mathsf{pk}_B$.[7] Then, given a message m, distributed signing works as follows:

- Party P_A samples $k_A \xleftarrow{\$} \mathbb{F}_{q_{\mathsf{S}}}$ and sends $R_A \leftarrow k_A \cdot G_{\mathsf{S}}$ to P_B.
- Party P_B samples $k_B \xleftarrow{\$} \mathbb{F}_{q_{\mathsf{S}}}$, sets $R = R_A + k_B \cdot G_{\mathsf{S}}$, sets $e \leftarrow \mathsf{H}_{\mathsf{S}}(\mathsf{pk}, R, m)$, $s_B \leftarrow e \cdot \mathsf{sk}_B + k_B$, and sends (R, s_B) to P_A.
- Party P_A recomputes e, sets $s_A \leftarrow e \cdot \mathsf{sk}_A + k_A$, and outputs $\sigma = (R, s_A + s_B)$.

Insecurity stems from the ability of a malicious P_B, receiving R_A before choosing R_B, to bias the aggregate R. This bias can be eliminated via the introduction of a standard commit-and-open procedure at the expense of an additional preliminary round in which P_B commits to R_B. As was argued in several previous works [NRSW20; GKMN21; KOR23], other sources of insecurity, including potential catastrophic

[6] We denote the pair of signers as P_A and P_B to separate their roles from the roles in the underlying VOLE-PCF. We highlight that, depending on the protocol variant, P_A, for example, simultaneously plays P_0 in one VOLE-PCF, and P_1 in another.

[7] For security one must add a commit-and-open procedure. We omit these details for simplicity.

failures rooted in the hardness of generating high-quality randomness and maintaining state continiuty, are mitigated by modern instantiations such as EdDSA with determinism and statelessness, two highly desirable properties that are not satisfied by the two-party scheme above. Alternative approaches to two-round unbiased nonce generation have been proposed, but each has weaknesses. FROST [KG20], for example, admits limited nonce bias and therefore requires a strong interactive assumption to satisfy a game-based threshold unforeability notion that is weaker than simulation-based security, and is not deterministic. Two-round, deterministic, stateless solutions [GKMN21; KOR23; BHLS25] come at a much higher concrete communication and computational cost.

It was first observed in [KOR23] that VOLE-PCFs offer the following compelling solution to two-round unbiased nonce generation. Assume that keys $(\mathsf{k}_0^{\mathsf{PCF}}, \mathsf{k}_1^{\mathsf{PCF}}) \leftarrow \mathsf{PCF}.\mathsf{KeyGen}(\Delta)$ for a VOLE-PCF over $\mathbb{F}_{q_\mathsf{S}}$ are distributed by a trusted dealer at setup, where $\Delta \xleftarrow{\$} \mathbb{F}_{q_\mathsf{S}}$ and $\mathsf{k}_1^{\mathsf{PCF}}$ are known to P_A. This PCF can be used to produce and authenticate a pseudorandom nonce for P_B, thereby addressing the first task listed above. Fix a message m to be signed, then:

- P_A picks $k_A \xleftarrow{\$} \mathbb{F}_{q_\mathsf{S}}$ and sends $R_A \leftarrow k_A \cdot G_\mathsf{S}$.
- P_B runs $(k_B, v_0) \leftarrow \mathsf{PCF}.\mathsf{Eval}(0, \mathsf{k}_0^{\mathsf{PCF}}, m)$, sets $R_B \leftarrow k_B \cdot G_\mathsf{S}$ and $V_0 \leftarrow v_0 \cdot G_\mathsf{S}$, computes (e, s_B) as before, and sends (R_B, V_0, s_B) to P_A.
- P_A runs $w_0 \leftarrow \mathsf{PCF}.\mathsf{Eval}(1, \mathsf{k}_1^{\mathsf{PCF}}, m)$, aborts if $w_0 \cdot G_\mathsf{S} \neq \Delta \cdot R_B + V_0$ and otherwise, recomputes e, defines s_A as before, and outputs $\sigma = (R_A + R_B, s_A + s_B)$.

VOLE-PCF security against P_A implies that k_B is indistinguishable from a truly random element of $\mathbb{F}_{q_\mathsf{S}}$. By correctness, $w_0 = \Delta \cdot k_B + v_0$ and P_A's check passes. Security against P_B ensures that $\mathsf{k}_0^{\mathsf{PCF}}$ leaks no information about Δ, and therefore the probability that P_B can send $k_B^* \neq k_B$ and $v_0^* \neq v_0$ such that the check passes is negligible, as $w_0 = \Delta \cdot k_B^* + v_0^*$ implies that $(v_0 - v_0^*) \cdot (k_B^* - k_B)^{-1} = \Delta$.

This yields a 2-round protocol with very low communication: three elements of $\mathbb{G}_\mathsf{S}$ and one element of $\mathbb{F}_{q_\mathsf{S}}$. As described the protocol is not deterministic, but this is easy to fix with an additional VOLE-PCF instance in which the roles of P_A and P_B are reversed and letting P_A similarly compute and authenticate k_A deterministically at the cost of one more $\mathbb{G}_\mathsf{S}$-element in communication.

Two-Party ECDSA and Secure Key-Nonce Multiplication. We now turn to the second non-trivial task for which VOLE-PCFs provide a graceful solution: the secure multiplication of a long-term secret (typically, a share of the secret key) with an ephemeral secret (typically, a nonce). An approach to distributing ECDSA signatures, introduced in [Lin17], involves sharing the secret key sk additively but sharing the nonce k *multiplicatively*, and therefore we use ECDSA to demonstrate the utility of VOLE-PCFs in this task. Fix a cyclic (additive) group $\mathbb{G}_\mathsf{E}$ of order q_E with generator G_E, but further assume that $\mathbb{G}_\mathsf{E}$ is an elliptic curve equiped with a "projection to the first coordinate" mapping that maps an elliptic curve point $R \in \mathbb{G}_\mathsf{E}$ to an element $r := R.\mathsf{x} \bmod q_\mathsf{E} \in \mathbb{F}_{q_\mathsf{E}}$, with the property $R.\mathsf{x} = (-R).\mathsf{x}$. Fix a hash function $\mathsf{H}_\mathsf{E} \colon \{0,1\}^* \to \mathbb{F}_{q_\mathsf{E}}$. The ECDSA signature scheme [JMV01] works as follows:

Key pairs: a uniform secret key $\mathsf{sk} \xleftarrow{\$} \mathbb{F}_{q_\mathsf{E}}$, and public key $\mathsf{pk} \leftarrow \mathsf{sk} \cdot G_\mathsf{E}$.
Signing a message $m \in \{0,1\}^*$**:** sample a nonce $k \xleftarrow{\$} \mathbb{F}_{q_\mathsf{E}}$, set $R = k \cdot G_\mathsf{E}$, set $r = R.\mathsf{x} \bmod q_\mathsf{E}$, and output $\sigma = (r, k^{-1} \cdot (\mathsf{H}_\mathsf{E}(m) + r \cdot \mathsf{sk}))$.
Verifying a signature $\sigma = (r, s) \in \mathbb{F}_{q_\mathsf{E}}^2$**:** given m output 1 if $(s^{-1} \cdot (\mathsf{H}_\mathsf{E}(m) + r \cdot \mathsf{sk}) \cdot G_\mathsf{E}).\mathsf{x} \bmod q_\mathsf{E} = r$, and otherwise output 0.

Concretely, during setup P_A and P_B sample $\mathsf{sk}_A \xleftarrow{\$} \mathbb{F}_{q_\mathsf{E}}$ and $\mathsf{sk}_B \xleftarrow{\$} \mathbb{F}_{q_\mathsf{E}}$ respectively, and set $(\mathsf{pk}_A, \mathsf{pk}_B) \leftarrow (\mathsf{sk}_A \cdot G_\mathsf{E}, \mathsf{sk}_B \cdot G_\mathsf{E})$.[8] Define $\mathsf{sk} = \mathsf{sk}_A + \mathsf{sk}_B$ and $\mathsf{pk} = \mathsf{pk}_A + \mathsf{pk}_B$. Consider the incomplete distributed signing procedure below:

- P_A picks $k_A \xleftarrow{\$} \mathbb{F}_{q_\mathsf{E}}$ and sends $R_A \leftarrow k_A \cdot G_\mathsf{E}$ to P_B.
- P_B picks $k_B \xleftarrow{\$} \mathbb{F}_{q_\mathsf{E}}$ and computes $R = k_B \cdot R_A$.

In the above $k = k_A \cdot k_B$, and the nonce R is essentially computed as a Diffie-Hellman key. Our goal in the asymmetric setting is for P_A to obtain $s = k^{-1} \cdot (\mathsf{H}_\mathsf{E}(m) + r \cdot \mathsf{sk})$. As P_A knows k_A it suffices to obtain $k_B^{-1} \cdot (\mathsf{H}_\mathsf{E}(m) + r \cdot \mathsf{sk})$. Observe that as

$$k_B^{-1} \cdot (\mathsf{H}_\mathsf{E}(m) + r \cdot \mathsf{sk}) = \underbrace{k_B^{-1} \cdot (\mathsf{H}_\mathsf{E}(m) + r \cdot \mathsf{sk}_B)}_{v_0} + \underbrace{(k_B^{-1} \cdot r)}_{u_0} \cdot \mathsf{sk}_A,$$

[8] See previous footnote.

and v_0, u_0 can be locally computed by P_B, it suffices for P_A and P_B to securely obtain shares of $u_0 \cdot \mathsf{sk}_A$, where v_0 is an ephemeral secret known to P_B and sk_A is a long-term secret known to P_A, to finalize the signature. As first observed in [Adj+24], this is precisely what is achieved by a secure protocol for VOLE over $\mathbb{F}_{q_\mathsf{E}}$, which we instantiate with a VOLE-PCF. Under the assumption that the parties receive $(\mathsf{k}_0^{\mathsf{PCF}}, \mathsf{k}_1^{\mathsf{PCF}}) \leftarrow \mathsf{PCF.KeyGen}(\mathsf{sk}_A)$ for a VOLE-PCF over $\mathbb{F}_{q_\mathsf{E}}$ during a trusted setup, then P_A and P_B can finalize their signature as follows:

- P_A evaluates $w \leftarrow \mathsf{PCF.Eval}(1, \mathsf{k}_1^{\mathsf{PCF}}, m)$.
- P_B evaluates $(u, v) \leftarrow \mathsf{PCF.Eval}(0, \mathsf{k}_0^{\mathsf{PCF}}, m)$ such that $w = u \cdot \mathsf{sk}_A + v$ and, together with the second round message R, sends $v' = v_0 - v$ and $u' = u_0 - u$.
- P_A computes $s_A = v' + \mathsf{sk}_A \cdot u' + w = (v_0 - v) + \mathsf{sk}_A \cdot (u_0 - u) + w = v_0 + u_0 \cdot \mathsf{sk}_A$, and outputs $s = k_A^{-1} \cdot s_A$.

A similar approach was used in [FHKRS25] to distributively compute BBS+ signatures. A careful reader may have noted that the 2-round ECDSA, as described, neither prevents a malicious P_B from biasing the nonce, nor ensures deterministic signing. However, both issues can be fixed exactly as in the two-party Schnorr protocol described above with two additional VOLE-PCF instances used to deterministically generate and authenticate the nonces, requiring three VOLE-PCFs in total.

Two alternatives are available in the case that deterministic signing is not required. A single additional VOLE-PCF instance (two in total) can be used to authenticate P_B's nonce while P_A is permitted to pick their nonce freely but is required to prove knowledge of its discrete-log with a straight-line extractable proof, at the cost of the strong assumptions for efficient straight-line extractable ZK proofs. In [Adj+24] this is done by sending $k_A \cdot \mathsf{pk}_B$, which P_B can compare to $\mathsf{sk}_B \cdot R_A$, under the knowledge-of-exponent assumption. Parties can also use no additional VOLE-PCF instance (one in total) and use a straight-line extractable ZK proof to authenticate both nonces, letting P_B bias the nonce at the cost of reducing security to the *doubly-enhanced* existential unforgeability of ECDSA, a strengthening of EUF-CMA introduced in [Adj+24] and shown to hold (tightly) in the generic group model, instead of standard unforgeability (EUF-CMA).

2.3 Distributing Key Generation

The observation that VOLE-PCFs enable efficient two-round two-party signatures is not novel to our work, though we believe there is value in the unified picture presented herein, but there is a fundamental question remains unaddressed: *how are VOLE-PCF keys distributed?* In the previous section, while demonstrating the usefulness of VOLE-PCFs, we intentionally avoided this question and assumed that the signers P_A and P_B *somehow* receive VOLE-PCF keys at setup. This sweeps a core challenge under the rug. While there exist several efficient VOLE-PCFs over $\mathbb{F}_p$ [Boy+20a; Boy+22; CD23; BCMRS25], *none of these are known to admit an efficient distributed key generation (DKG) protocol.* Consequently, prior works that rely on PCGs/PCFs for distributed signatures [FHKRS25; Fen+25; ANOSS22; KOR23; Kon26] make design choices that sidestep this issue at a significant cost. All previous works fall into one of three categories.

Category I: PCF-Based Protocols that Assume a Trusted Setup. Works such as [FHKRS25, Appendix F] and [Fen+25] describe PCF-based threshold BBS+ and EdDSA, respectively, from a VOLE-PCF over $\mathbb{F}_p$ for which no efficient DKG is known. These works simply assume that a trusted dealer generates and distributes the PCF keys to the signers ahead of time.

Category II: PCG-Based Protocols. Pseudorandom correlation generators (PCGs) are to PCFs what pseudorandom generators are to pseudorandom functions; a PCG for VOLE lets two parties stretch short correlated seeds into a pseudorandom length-N VOLEinstance $((\boldsymbol{u}, \boldsymbol{v}), (\Delta, \boldsymbol{w}))$. Distributed signatures from VOLE-PCGs have been presented by [FHKRS25; ANOSS22] for BBS+ and ECDSA, respectively. There exist several fast VOLE-PCG candidates over $\mathbb{F}_p$ [BCGI18; Boy+19b; SGRR19; Boy+19a; CRR21; Boy+22; RRT23; KPRR25; Guo+26], each compiled from a distributed point function (DPF) into a PCG under a coding-theoretic assumption. In contrast with PCFs, where entries of the potentially superpolynomial-dimension VOLE correlation vectors can be accessed one at a time, a PCG is restricted to the simultaneous generation of an a-priori fixed amount of correlated pseudorandomness all at once. In the context of distributed signatures, this effectively means that (1) after setup the number of signing

sessions is correspondingly bounded, and (2) that the parties must generate and store VOLEs of length proportional to the maximum supported number of signing sessions. Furthermore, PCG-based approaches force statefullness; to avoid devastating nonce re-use attacks signers must maintain a session counter to track which VOLE entries were used thus far. Hence, although PCGs *do* admit efficient DKGs, computation and long-term storage grows with the upper bound on the maximum number of signing sessions, and PCG-based protocols are fundamentally incompatible with stateless signing.

Category III: Protocols from PCFs for the "Wrong" Correlation. Given the limitations on PCG-based constructions works such as [KOR23; Kon26] have instead settled for using VOLE-PCFs that admit an efficient distributed setup but generate correlations *over the wrong field*. There are two candidates.

1. The VOLE-PCF from [Roy22] generates VOLEs over a small (polynomial-sized) field $\mathbb{F}$, or alternatively restricts Δ to being sampled from a publicly-known polynomial-sized subset of the field, which can be generalized with the Chinese remainder theorem to VOLE correlations over smooth integer rings as recently observed in [Kon26].
2. The VOLE-PCF from [OSY21] generates VOLEs over $\mathbb{Z}_{\hat{N}}$, where $\hat{N}$ is an RSA modulus (a product of two large primes).

Neither of these works support generating VOLEs over $\mathbb{F}_p$, as required for our approach outlined above. The works of [KOR23; Kon26] cleverly leverage these VOLE-PCFs over the wrong structure, but with a price in either efficiency or security.

The work of [KOR23] uses the VOLE-PCF from [Roy22] over $\mathbb{F}_p$ to build Schnorr signatures, but under the aforementioned restriction on Δ, precluding the work from achieving malicious security. The authors settle on *covert* security; the adversary is permitted to cheat during setup, then arbitrarily bias the nonces if that cheating goes unnoticed, which can happen with inverse polynomial probability. The work of [KOR23] provides a second construction from the PCF over $\mathbb{Z}_{\hat{N}}$ from [OSY21] that achieves fully malicious security. Correlations are lifted back to a correlation over $\mathbb{F}_p$ using a mechanism that enforces a range check that significantly increases communication and computation as elements of $\mathbb{Z}_{\hat{N}}$ are typically an order of magnitude larger than elements of $\mathbb{F}_p$, and exponentiations over $\mathbb{Z}_{\hat{N}}$ are around two orders of magnitude more expensive ([OSY21] reports an estimated runtime of 188 ms per party, ignoring all setup costs). The work of [Kon26] instead combines several instances of the VOLE-PCF from [Roy22] into a VOLE-PCF over a large ring $\mathbb{Z}_m$, where m is a primordial modulus (a product of all primes smaller than a threshold) larger than p^2. This VOLE-PCF is used to securely compute the product $(k_B^{-1} \cdot r \bmod p) \cdot \mathsf{sk}_A$ over the integers which is locally reduced modulo p, opening the door to an "out of range" selective failure attack known to have devastating consequences when not properly addressed [MYG24]. To address this issue security is downgraded to the security model of Lindell [Lin17] that restricts parties to sequential signing, and requires parties to abort *the entire protocol* whenever a single failure is detected.

2.4 A Template for Efficient VOLE-PCFs

All known efficient VOLE-PCFs over $\mathbb{F}_p$ are based on variants of the learning parity with noise (LPN) assumption. To the best of our knowledge there are three constructions: a VOLE-PCF from Variable-Density LPN [Boy+20a; CD23], a VOLE-PCF from Expand-Accumulate LPN [Boy+22], and a more recent VOLE-PCF from sparse-LPN [BCMRS25]. Similarly to the known LPN-based PCG constructions these constructions follow a common template, at the heart of which are two ingredients: FSS and Dual LPN.

Function Secret Sharing (FSS). An FSS scheme for a class of functions $\mathcal{F} = \{f : [N] \to \mathbb{F}_p\}$ is a pair of algorithms $(\mathsf{Gen}, \mathsf{Eval})$ with the following syntax:

$\mathsf{FSS.Gen}(f)$: On a function $f \in \mathcal{F}$ generation outputs a key pair (k_0, k_1).
$\mathsf{FSS.Eval}(k_b, x)$: On a key k_b and a point $x \in [N]$, evaluation outputs y_b.

An FSS must be *correct*, in that y_0, y_1 are shares of $y = f(x)$; for every $f \in \mathcal{F}$, $(k_0, k_1) \leftarrow \mathsf{FSS.Gen}(f)$, for all $x \in [N]$ it is the case that $\mathsf{FSS.Eval}(k_0, x) + \mathsf{FSS.Eval}(k_1, x) = f(x)$. An FSS scheme must be *secure*; informally, each key k_b should not real any information about f. Finally, an FSS must be *efficient*; a key share should be much smaller than N,[9] typically $O(\log N)$.

[9] Otherwise, a trivial FSS can be obtained by simply sharing the truth-table of f.

Very efficient constructions of FSS schemes are known for simple function classes from PRGs, e.g., point functions [GI14] where $f(x) = 0$ everywhere except on a single point, and interval functions [BGI16] where $f(x) = 0$ if $x \notin [a, b]$ for some interval $[a, b]$ and otherwise $f(x) = v$ for some value v. An FSS for point functions is called a *distributed point function* (DPF), the truth table of a point function is called a *unit vector*, and the truth table of an interval function is called an *interval vector*.

By the linearity of additive sharing one can then generically build FSS for the sum of t point functions, called a distributed multi-point function (DMPF), or the sum of t interval functions by simply having $\mathsf{FSS.Gen}$ for the DMPF output t DPF keys and $\mathsf{FSS.Eval}$ for the DMPF output the sum of the outputs of the t DPFs. Let $\mathcal{P}_t$ denote the class of sums of t point functions, and $\mathcal{I}_t$ denote the class of sums of t interval functions. We omit the implicit domain size N from the index for simplicity. Note that $\mathcal{P}_t \subsetneq \mathcal{I}_t$.

Dual LPN (or Syndrome Decoding). The dual LPN (or syndrome decoding) assumption over $\mathbb{F}_p$ with codeword length n, rate r, matrix distribution $\mathcal{H}$ supported over $\mathbb{F}_p^{r \cdot n \times n}$, and noise distribution $\mathcal{E}$ supported over $\mathbb{F}_p^n$, states the following: sample $H \xleftarrow{\$} \mathcal{H}$, $\boldsymbol{e} \xleftarrow{\$} \mathcal{E}$. Then the pair $(H, H \cdot \boldsymbol{e})$ should be indistinguishable from the pair $(H, \boldsymbol{z})$ where $\boldsymbol{z} \xleftarrow{\$} \mathbb{F}_p^{r \cdot n}$.

Template VOLE-PCF. To build a VOLE-PCF over $\mathbb{F}_p$, all existing works [Boy+20a; Boy+22; CD23; BCMRS25] rely on a flavor of the dual LPN assumption that satisfies the following conditions:

- The codeword length n can be superpolynomial in the security parameter, and the rate r is either a constant or some inverse polynomial.
- The distribution $\mathcal{H}$ samples the rows of H independently from an efficiently samplable distribution denoted $\mathcal{R}$, such that each row of H is *sparse* in that it contains only polynomially-many nonzero entries.
- The noise vector $\boldsymbol{e} \xleftarrow{\$} \mathcal{E}$ can be sampled in time $\mathsf{poly}(t, \log n)$ for some fixed (polynomial) parameter t, and $\boldsymbol{e}$ is the truth table of some "FSS-friendly" function $f \in \mathcal{I}_t$, i.e., there is $f_{\boldsymbol{e}} \in \mathcal{I}_t$ such that $\boldsymbol{e}_i = f_{\boldsymbol{e}}(i)$ for every $i \in [N]$.

Let H denote a hash function (modeled as a random oracle) that maps any input x to a random sample from $\mathcal{R}$ (in practice, this is done by hashing x to a string fed as randomness to the sampler for $\mathcal{R}$). Then, under a dual LPN assumption satisfying the above conditions, the VOLE-PCF from FSS is as follows:

$\mathsf{PCF.KeyGen}(\Delta)$**:** sample $\boldsymbol{e} \xleftarrow{\$} \mathcal{E}$, run $(k_0, k_1) \leftarrow \mathsf{FSS.Gen}(f_{\Delta \cdot \boldsymbol{e}})$ (note that if $f \in \mathcal{I}_t$ then $\Delta \cdot f \in \mathcal{I}_t$). Define $\mathsf{k}_0^{\mathsf{PCF}} = (f_{\boldsymbol{e}}, k_0)$ and $\mathsf{k}_1^{\mathsf{PCF}} = (\Delta, k_1)$.

$\mathsf{PCF.Eval}(b, \mathsf{k}_b^{\mathsf{PCF}}, x)$: If $b = 0$ compute $u_x \leftarrow \mathsf{H}(x) \cdot \boldsymbol{e} = \sum_{i \,:\, \mathsf{H}(x)_i \neq 0} \mathsf{H}(x)_i \cdot f_{\boldsymbol{e}}(i)$ and $v_x \leftarrow -\sum_{i \,:\, \mathsf{H}(x)_i \neq 0} \mathsf{H}(x)_i \cdot \mathsf{FSS.Eval}(k_0, i)$ (recall that $\mathsf{H}(x) \in \mathbb{F}_p^n$ is sparse). If $b = 1$ compute $w_x = \sum_{i \,:\, \mathsf{H}(x)_i \neq 0} \mathsf{H}(x)_i \cdot \mathsf{FSS.Eval}(k_1, i)$.

It follows by inspection that $w_x = \Delta \cdot u_x + v_x$ for every input x, where $u_x = \mathsf{H}(x) \cdot \boldsymbol{e}$. If the parties make Q PCF evaluations on inputs $(x_1, \ldots, x_Q)$, the evaluation on each input x_i corresponds to lazily sampling the i-th row of a matrix $H \xleftarrow{\$} \mathcal{H}$ and computing the i-th entry of $H \cdot \boldsymbol{e}$. As n is superpolynomial and r is inverse polynomial, the polynomially-bounded parties necessarily make at most $Q \ll r \cdot n$ queries and security follows from the dual LPN variant.

All existing VOLE-PCF candidates follow the template above and differ only in their choices of $\mathcal{H}$, $\mathcal{E}$, and parameters (n, r, t). The VDLPN construction [Boy+20a; CD23] lets $\boldsymbol{e}$ be a concatenation of t variable-length unit vectors and H be a matrix with sparse rows of variable density. The EALPN construction [Boy+22] sets $\boldsymbol{e}$ to a concatenation of t interval vectors and H to a a random sparse matrix with fixed-weight rows. The sparse-LPN construction [BCMRS25] uses L noise vectors $(\boldsymbol{e}_1, \ldots, \boldsymbol{e}_L)$ for some small number of *levels* L, where each $\boldsymbol{e}_i$ is a concatenation of t (equal-length) unit vectors, and sets $(H_1, \ldots, H_L)$ to be products of sparse matrices (of increasing size and sparsity).[10]

[10] We note that in [BCMRS25] and in the full version of this work, the sparse-LPN PCF is described as a recursive construction from the *primal* LPN assumption, which states that samples of the form $(G, G \cdot \boldsymbol{s} + \boldsymbol{e})$ for a generator matrix G, a secret $\boldsymbol{s}$, and a noise $\boldsymbol{e}$, are indistinguishable from random. Nevertheless, it is not too hard to "unpack" the recursive construction and rewrite it in a dual form where H is a concatenation of products of sparse matrices, and $\boldsymbol{e}$ is a sparse noise of variable density, similar in spirit to the VDLPN construction from [Boy+20a]. We omit further discussion in this overview for the sake of keeping the descriptions high level and unified.

The Bottom Line. The main message of this section is that securely distributing a PCF key boils down to securely distributing $(k_0, k_1) \leftarrow \mathsf{FSS.Gen}(f_{\Delta \cdot e})$ for $f_{\Delta \cdot e}$ a sum of t point or interval functions, which, in turn, reduces to distributing FSS shares for those t point or interval functions.

2.5 On the Cost of Distributing VOLE-PCFs

In the previous section we demonstrated how building a DKG for any known VOLE-PCF over $\mathbb{F}_p$ reduces to distributing $\mathsf{FSS.Gen}$ for point or interval functions. We now focus on constructing a DKG for a DPF, but note that the interval function case is nearly identical.[11] A relaxation of DPFs, called *known-index* DPFs (KI-DPFs) in which one party learns the position at which the function is non-zero suffices. In the template above for VOLE-PCFs this translates to revealing the LPN noise to one of the parties, which is revealed anyway. KI-DPFs are typically built from puncturable pseudorandom functions (PPRFs) [BW13; KPTZ13; BGI14][12] using the celebrated Goldreich-Goldwasser-Micali (GGM [GGM84]) construction. We recall the construction due to [Boy+19a] below.

KI-DPFs from Length-Doubling PRGs. Fix a length-doubling pseudorandom generator $G\colon \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$, typically instantiated in practice from two calls to AES. The KI-DPF construction is best presented with reference to a picture, such as that in Figure 1. Consider labeling the full binary tree represented in Figure 1 inductively from a label k for the root node by defining $(v_0, v_1) = G(v)$ as the labels of the children of a node with label v.

Given a point function $f_{\alpha,\beta}\colon [N] \to \{0,1\}^*$ with $f_{\alpha,\beta}(\alpha) = \beta$, and assuming for simplicity that N is a power of two, the KI-DPF algorithm Gen deals P_1 a random string $k_1 \in \{0,1\}^\lambda$ to be used as the root label. On input $x \in [N]$, viewed as a $\log N$-bit string, the KI-DPF algorithm $\mathsf{Eval}(k_1, x)$ returns the label of the leaf node on the path from the root to a leaf defined by x, where 0 denotes left child and 1 denote right child (in Figure 1, the red node's label would be the output on input $x = 0111$). Append to k_1 the value $v_\alpha = \mathsf{Eval}(k_1, \alpha) + \beta$.

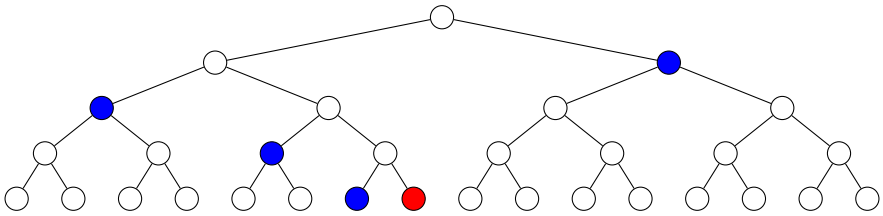

Fig. 1. A full GGM tree, $N = 16$. The root is labeled with the key k_1. The red leaf is at the position α (the nonzero in the point function), and the blue nodes denote the co-path to the red leaf. The blue nodes labels are given as part of the key k_0.

The key k_0 consists of the labels on the co-path from the root to the leaf at position $\alpha \in [N]$, denoted in blue on Figure 1. Observe that given k_0 party P_0 can recompute the leaves at all positions except α. Define $\mathsf{Eval}(k_0, x)$ to return $\mathsf{Eval}(k_1, x)$ for every $x \neq \alpha$, which can be computed from the blue labels, and $\mathsf{Eval}(k_0, \alpha) = v_\alpha$. It is the case that $\mathsf{Eval}(k_0, x) - \mathsf{Eval}(k_1, x) = f_{\alpha,\beta}(x)$ for every $x \in [N]$.

Distributing KI-DPF Keys with Generic MPC. A first option for a DKG for the GGM KI-DPF above is to make non-blackbox use of the underlying PRG in a 2-party maliciously secure protocol: P_1 picks k_1, P_0 picks α, and the parties iteratively run a secure protocol that takes as input shares of a value v_i from both parties and a bit α_i from P_0, computes $G(v_i) = (v^0_{i+1}, v^1_{i+1})$, and outputs $v^{1-\alpha_i}_{i+1}$ to P_0 and shares of $v_{i+1} = v^{\alpha_i}_{i+1}$ to the parties. A full-fledged maliciously secure protocol would require additional

[11] The known construction of FSS for intervals from [BGI16] closely follows the template of their DPF construction, and exhibits the same efficiency features.

[12] At a high level, a PPRF is a pseudorandom function F_K equipped with a *puncturing* algorithm which, given a PRF key K and a point z, outputs a *punctured key* K_z. Given a punctured key K_z one can evaluate the PPRF $\mathsf{F}_{K_z}(x) = \mathsf{F}_K(x)$ at all points except z, and $\mathsf{F}_K(z)$ is indistinguishable from random given K_z.

mechanisms to ensure consistency of the inputs used throughout the protocol, but we ignore this issue to simplify the discussion.

Unfortunately, this approach is prohibitively expensive. Assume that the parties use the VOLE-PCF from [Boy+22] with $t = 1120$ nonzero entries in the noise vector (parameter set LPN3 from [Guo+26, Table 2]; all other VOLE-PCFs require much higher t) and set $N = 5 \cdot 2^{35}/t > 2^{27}$ (thereby setting 2^{35} as an upper bound on the number of distributed signing sessions), that PRG G is instantiated with two calls to a fixed-key AES as is the case for virtually all works on PCG/PCFs, and that the parties use the state of the art maliciously-secure distributed AES protocol from [WRK17].[13] When amortizing over more than 1024 executions, their protocol computes one AES call in 114ms (over a WAN network) over one thread of an Amazon EC2 c4.8xlarge instance, using 2.6 MB of communication per party (5.2 MB total). Ignoring other costs and estimating one instance of the KI-DPF as 29×2 calls to a maliciously secure AES protocol (two calls per level of a depth-28 tree and an additional call to extend the 128-bit leaf to 256 bits), since $t = 1120$ we get a (loose) lower bound of 7405 s of computation and 338 GB of communication. Replacing AES with an MPC-friendly block cipher such as LowMC [ARSTZ15] would likely save an order of magnitude from both—[BCGI18] estimated a 23-fold improvement in computation in a similar setting—but at the cost of relying on a much less well-studied cipher.

Distributing KI-DPF Keys via Doerner-shelat. An alternative is a variant of the Doerner-shelat protocol [Ds17] as proposed in [Boy+19a] and improved for the malicious setting in [WYKW21]. At a high level, this variant requires $\log n$ oblivious transfers, one for each level of the tree, where the two sender inputs are the XOR of all the "left children nodes" and the XOR of all the "right children nodes" and the selection bits are $1 - \alpha_i$. Given the blue node labels P_0 can recompute all terms but one in each XOR, and deduce the next blue node from that. This makes blackbox use of the PRG and can be made maliciously secure with a simple sacrificing procedure that reveals a linear combination of a subset of the leaf nodes. Unfortunately, due to the need to compute XORs of all left or right children at each level of the tree the computational overhead of this approach scales with the total number of leaves N. Even with the fastest hardware instruction to evaluate one AES call in about 20 cycles [MSY21] for the VOLE-PCF from [Boy+22], targeting as before 2^{35} correlations and assuming a 2.4 GHz CUP, this translates to 4295 s of computation in the semi-honest setting.

2.6 Our Approach

At the heart of our work is the following simple, but powerful observation: existing VOLE-PCFs over $\mathbb{F}_p$ are *way too fast* for our application to distributed signatures. Take, for example, the recent VOLE-PCF from [BCMRS25] that generates 2^{35} correlations at a rate of 100k correlations/s, or one VOLE entry in 10 µs. Distributed signatures require between one and three VOLE entries in the approach sketched above; two-party Schnorr requires one or two VOLE-PCFs, and ECDSA requires three in the asymmetric case, or two assuming the doubly-enhanced unforgeability of ECDSA. This translates to an overhead of 10–30 µs in runtime. Costs in this range are completely dominated by the latency. For many distributed signature applications, often in the blockchain setting, participants can be arbitrarily far apart. As discussed in [Adj+24] a single roundtrip between two machines can take between 50–100 ms for distant machines in the US [DKLs19], and 100–400 ms for machines in far apart countries, e.g., 348 ms for India-Belgium [DKLs19]. In this regime, whether a VOLE-PCF evaluation takes 10 µs or 5 ms makes virtually *no difference* to end-to-end runtime.

Algebraic KI-DPFs. Leveraging this observation, we replace the AES-based KI-DPF used in VOLE-PCF constructions, which while blazing fast lacks an efficient DKG, with an algebraic, slower but *DKG-friendly*, KI-DPF under group-based assumptions. We take inspiration from two recent works [BCMPR24; CHHK25] that construct efficient group-based PPRFs for different purposes (placing PPRFs in NC^1 for [BCMPR24], and garbling arithmetic circuits for [CHHK25]). We start from these two constructions and introduce a new, slightly more involved, group-based PPRF construction that in a sense interpolates the two and strikes a better key-size/efficiency tradeoff for our application. We demonstrate that our new PPRF

[13] This uses authenticated garbling. Protocols with less communication and slightly less computation can be obtained using GMW/SPDZ-style MPC, but they will require many rounds. For example, even in the honest majority setting, the state of the art protocol of [Mor+25] requires between 16 and 30 rounds of communication. As the DKG for KI-DPF already requires 28 sequential AES computations, this will induce a prohibitive latency.

admits an efficient maliciously-secure distributed setup, leveraging Quicksilver-style VOLE-based zero-knowledge proofs [YSWW21]. We show how to turn our new generalized PPRF into a KI-DPF using a correlation-robust hash function. Finally, we describe an end-to-end maliciously-secure DKG for our KI-DPF. The final protocol requires a single non-blackbox distributed evaluation of the hash function (in contrast with the GGM-based distributed setup sketched above which requires 80 distributed AES evaluations), which can be instantiated with AES. To further improve efficiency we introduce a new "MPC-friendly" correlation-robust hash function that leverages chains of elliptic curves.

Choice of VOLE-PCF. We briefly elaborate on our choice of VOLE-PCF to make DKG-friendly. Among the three listed above[Boy+20a; Boy+22; BCMRS25], we selected the candidate of [BCMRS25] from sparse-LPN.

The candidate of [Boy+20a] from variable-density LPN is asymptotically superior, but is based on a less standard assumption. Furthermore, this candidate is much slower in the range of parameters of interest. Using the generalized (and optimized) parameter selection process from [Duc24, Section 7.3.3], counting only the time for AES calls (ignoring all other factors) and using the estimate of 20.8 cycles per AES block from [MSY21], on a 2.4GHz processor, the throughput for $2^{30}, 2^{35}$, and 2^{40} outputs respectively are 614, 412, and 291 VOLEs per second (single core). In the same range of parameters the VOLE-PCF from [BCMRS25] has an estimated throughput of $5 \cdot 10^5$, 10^5, and $2 \cdot 10^4$ VOLEs per second, respectively.

The candidate of [Boy+22], on the other hand, performs comparably to [BCMRS25] in this parameter range with much shorter keys (albeit under a more recent assumption). However, this VOLE-PCF requires FSS for intervals, whereas the candidate from sparse-LPN only requires a DPF. Building efficient DKG-friendly FSS for intervals appears harder than building DKG-friendly KI-DPF.[14] We leave this as an open problem.

2.7 First Contribution: Puncturable PRFs with Distributed Setup

PPRFs are a core building block for KI-DPFs; the AES-based KI-DPF described in Section 2.5 is exactly the seminal GGM PPRF construction from [BW13; KPTZ13; BGI14] with the addition of a term $\beta + \mathsf{F}_K(\alpha)$ to fill in the hole left by the punctured evaluation procedure. As a starting point for building a DKG-friendly KI-DPF, we explain how to build DKG-friendly PPRFs with domain $[N]$. Assume for convenience that $N = 2^n$ is a power of two. Fix a group $\mathbb{G}$ of order p with generator G, and let $\mathsf{H}\colon \mathbb{G} \to \mathbb{F}_p$ denote a hash function.

The BCMPR PPRF [BCMPR24]. The BCMPR PPRF has the following procedures:

$\mathsf{F.Setup}(1^n)$**:** Sample $\rho, r \xleftarrow{\$} \mathbb{F}_p$, set $H \leftarrow \rho \cdot G$ and $H_i \leftarrow r^i \cdot H$ for all $i \in [\pm n]$. Output $\mathsf{mpk} = (H_i)_{i \in [\pm n] \setminus \{-n\}}$, $\mathsf{msk} = (r, H)$.

$\mathsf{F.KeyGen}(\mathsf{mpk})$**:** Output $(\mathsf{sk}_1, \ldots, \mathsf{sk}_n) \xleftarrow{\$} \mathbb{F}_p^n$.

$\mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, x)$**:** On $(\mathsf{msk}, \mathsf{sk}) \in \mathbb{G}^{2n} \times \mathbb{F}_p^n$ and bitstring $(x_1, \ldots, x_n) \in [N]$, output $y = \mathsf{H}((\prod_{i=1}^n \mathsf{sk}_i^{(-1)^{x_i}}) \cdot H)$.

$\mathsf{F.Puncture}(\mathsf{msk}, \mathsf{sk}, z)$**:** On $(\mathsf{msk}, \mathsf{sk}) \in \mathbb{G}^{2n} \times \mathbb{F}_p^n$, and bitstring $(z_1, \ldots, z_n) \in [N]$, output $\mathsf{psk} = (\mathsf{sk}_i \cdot r^{(-1)^{z_i}})_{i \le n}$.

$\mathsf{F.PEval}(\mathsf{mpk}, \mathsf{psk}, x)$**:** If $x \neq z$ then output $y = \mathsf{H}((\prod_{i=1}^n \mathsf{psk}_i^{(-1)^{x_i}}) \cdot H_{i^*})$ for $i^* \leftarrow -\sum_{i=1}^n (-1)^{x_i + z_i}$.

Punctured evaluation $\mathsf{F.PEval}$ is correct for $x \neq z$ as $i^* \neq -n$ and $H_{i^*} \in \mathsf{mpk}$. For intermediate value $Y \in \mathbb{G}$ such that $y = \mathsf{H}(Y)$ it is the case that

$$
\begin{aligned}
Y &= \left(\prod_{i \in [n]} \mathsf{psk}_i^{(-1)^{x_i}}\right) \cdot H_{i^*} = \left(\prod_{i \in [n]} \mathsf{sk}_i^{(-1)^{x_i}} \cdot r^{(-1)^{(x_i + z_i)}}\right) \cdot \left(r^{i^*} \cdot H\right) \\
&= \left(r^{i^* + \sum_{i \in [n]} (-1)^{x_i + z_i}}\right) \cdot \left(\prod_{i \in [n]} \mathsf{sk}_i^{(-1)^{x_i}}\right) \cdot H = \prod_{i \in [n]} \mathsf{sk}_i^{(-1)^{x_i}} \cdot H.
\end{aligned}
$$

[14] There exists a generic construction of FSS for intervals from DPFs [BGI16] that is too inefficient for our purposes.

When $x = z$ the same equation shows that $\mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, z) = \mathsf{H}(u \cdot H_{-n})$, where $u = \prod_{i\in[n]} \mathsf{psk}_i^{(-1)^{x_i}}$ is a value known given (psk, z).

Security follows from the *power-DDH assumption* which essentially states that H_{-n} is pseudorandom given $(H_{-n+1}, \ldots, H_n)$.[15] In the multi-instance setting, with multiple secret keys for the same setup, security follows from the assumption that given arbitrary distinct $(\mathsf{psk}_j)_j$ and a random H_{-n}, the values $y_j = \mathsf{H}(u_j \cdot H_{-n})$ (where the u_j are terms computable from $(\mathsf{psk}_j)_j$ and the punctured points z_j) are pseudorandom, provided that H satisfies a suitable form of correlation robustness.

Observe that distributing setup is very easy. One party can generate and send $\mathsf{mpk} = (H_i)_i$, proving well-formedness using a simple VOLE-ZK proof (or alternatively a Σ-protocol), and sample sk. The second party picks z. To distribute psk the parties engage in n oblivious transfers (OTs) with sender input $(\mathsf{sk}_i \cdot r, \mathsf{sk}_i/r)$ and receiver selection bit z_i. Security against malicious adversaries further requires that all pairs of OT inputs have the same ratio r^2, which is achieved with a little extra work.

The CHHK PPRF [CHHK25]. The CHHK PPRF has the following procedures:

$\mathsf{F.Setup}(1^n)$**:** Sample $\rho, r \xleftarrow{\$} \mathbb{F}_p$, set $H \leftarrow \rho \cdot G$ and $H_i \leftarrow r^i \cdot H$ for all $i \in [\pm N]$. Output $\mathsf{mpk} = (H_i)_{i\in[\pm N]^*}$, $\mathsf{msk} = (r, H)$.
$\mathsf{F.KeyGen}(\mathsf{mpk})$**:** Output $\mathsf{sk} \xleftarrow{\$} \mathbb{F}_p$.
$\mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, x)$**:** Output $y = \mathsf{H}(\mathsf{sk} \cdot H_x)$.
$\mathsf{F.Puncture}(\mathsf{msk}, \mathsf{sk}, z)$**:** Output $\mathsf{psk} = \mathsf{sk} \cdot r^z$.
$\mathsf{F.PEval}(\mathsf{mpk}, \mathsf{psk}, x)$**:** If $x \neq z$ then output $y = \mathsf{H}(\mathsf{psk} \cdot H_{x-z})$.

Correctness follows by inspection. When $x \neq z$, $\mathsf{psk} \cdot H_{x-z} = (\mathsf{sk} \cdot r^z) \cdot (r^{x-z} \cdot H) = \mathsf{sk} \cdot H_x$. Security stems from the fact that, under the power-DDH assumption, $H = H_0$ is pseudorandom given mpk and $(H_i)_{i\neq 0}$. Hence, given psk, $\mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, z) = \mathsf{sk} \cdot H_z = \mathsf{psk} \cdot H$ also looks random. This suffices for security in a single instance setting, where a single pair $(\mathsf{sk}, \mathsf{psk})$ is computed from a given mpk. As above the correlation-robustness of H ensures security in a multi-instance setting, where the same mpk is reused to generate many pairs $(\mathsf{sk}_j, \mathsf{psk}_j)_j$. As for the PPRF of [BCMPR24], it is not hard to construct a DKG for this PPRF: let one party pick $(\mathsf{mpk}, \mathsf{sk})$, use $n = \log N$ OTs to distribute psk, and add a little work to proving well-formedness of mpk and ensure consistency.

A New DKG-friendly PPRF. The two PPRFs described above are comparably DKG-friendly, and have similar computational costs dominated by one exponentiation and a hash (plus a few multiplications for the BCMPR PPRF [BCMPR24]). However, the key size differs. The secret key sk and the punctured secret key psk of CHHK [CHHK25] are much shorter, at a single $\mathbb{F}_p$ element instead of k, and while the master public key mpk is exponentially larger at 2^{n+1} group elements instead of n it can be reused across arbitrarily many instances.[16]Recall that a VOLE-PCF key is essentially t PPRF keys, where t is the Hamming weight of the noise vector of the underlying dual LPN instance. For t PPRFs over $[2^n]$ the key size is $O(t \cdot n)$ with BCMPR and $O(t + 2^n)$ with CHHK; the best choice depends on the tradeoff between t and n. We introduce a new PPRF that, in our setting, can strike a better tradeoff between t and n than the above alternatives.

From a bird's-eye view, the BCMPR PPRF writes the input and constraint in binary, and enforces a check of the form $\mathsf{HD}(x, z) \neq 0$.[17] The CHHK PPRF, on the other hand, directly operates on elements of $[N]$ and enforces the check $x - z \neq 0$. We interpolate between these two extremes by building a new PPRF that represents the input over the k-ary basis, writing $x \in [N]$ as $(x_1, \ldots, x_m)$ with $x_i \in \{0, \ldots, k-1\}$ and $m = \log_k(N)$. Our new PPRF ensures that evaluation given a key constrained at $(z_1, \ldots, z_m)$ is possible only when $\sum_{i\in[m]}(x_i - z_i)^2 \neq 0$. That is, we enforce $\mathsf{ED}_k(x, z) \neq 0$, where ED_k refers to the Euclidean distance of inputs viewed as vectors over the k-ary basis.

Our new PPRF has the following procedures, where $m = \log_k(N)$:

$\mathsf{F.Setup}(1^n)$**:** Sample $\rho, r \xleftarrow{\$} \mathbb{F}_p$, let $H \leftarrow \rho \cdot G$, $H_i \leftarrow r^{-i} \cdot H$ for all $i \in [m(k-1)^2]$. Output $\mathsf{mpk} = (H_i)_{i\in[m(k-1)^2]}$, $\mathsf{msk} = (r, H)$.
$\mathsf{F.KeyGen}(\mathsf{mpk})$**:** Output $(\mathsf{sk}_0, \ldots, \mathsf{sk}_{m+1}) \xleftarrow{\$} \mathbb{F}_p^{m+2}$.

[15] Note that the usual formulation of the power-DDH assumption is slightly different but this variant is equivalent.
[16] We used $2n$ elements H_i in our simplified description. It is not too hard to observe that only n are needed since for bits b_i, $\sum_{i=1}^n (-1)^{b_i}$ always has the same parity as n.
[17] HD refers to the Hamming distance.

$\mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, x)$**:** For $\mathsf{sk}(x) = \mathsf{sk}_0^{\sum_{i\in[m]} x_i^2} \cdot \prod_{i\in[m]} \mathsf{sk}_i^{x_i} \cdot \mathsf{sk}_{m+1}$, output $y_x = \mathsf{H}(\mathsf{sk}(x) \cdot H)$.

$\mathsf{F.Puncture}(\mathsf{msk}, \mathsf{sk}, z)$**:** For $\mathsf{psk}_0 = r \cdot \mathsf{sk}_0$, $\mathsf{psk}_i = r^{-2z_i} \cdot \mathsf{sk}_i$ for $i \in [m]$, and $\mathsf{psk}_{m+1} = r^{\sum_{i=1}^m z_i^2} \cdot \mathsf{sk}_{m+1}$, output $\mathsf{psk} = (\mathsf{psk}_0, \ldots, \mathsf{psk}_{m+1})$.

$\mathsf{F.PEval}(\mathsf{mpk}, \mathsf{psk}, x)$**:** If $x \neq z$ set $i^* \leftarrow \mathsf{ED}_k(x, z) = \sum_{i\in[m]}(x_i - z_i)^2$, $\mathsf{psk}(x) = \mathsf{psk}_0^{\sum_{i\in[m]} x_i^2} \cdot \prod_{i\in[m]} \mathsf{psk}_i^{x_i} \cdot \mathsf{psk}_{m+1}$, output $y'_x = \mathsf{H}(\mathsf{psk}(x) \cdot H_{i^*})$.

Correctness uses fact that $\mathsf{ED}_k(x, z) = \sum_{i=1}^m (x_i^2 + z_i^2) - 2\langle x, z\rangle$ where $\langle\cdot\rangle$ denotes the inner product. Given $\mathsf{psk}$and input x,

$$\mathsf{psk}(x) = r^{\sum_i x_i^2 - 2\sum_i x_i z_i + \sum_i z_i^2} \cdot \mathsf{sk}(x) = r^{\mathsf{ED}_k(x,z)} \cdot \mathsf{sk}(x).$$

Hence when $i^* = \mathsf{ED}_k(x, z)$ we have $H_{i^*} = r^{-\mathsf{ED}_k(x,z)} \cdot H$, and therefore $y_x = y'_x$. As before, security reduces to the power-DDH assumption assuming H is correlation-robust.

This new PPRF offers a tunable tradeoff between the size of mpk, at $\log_k(N) \cdot (k-1)^2$ group elements, and the sizes of $\mathsf{sk}, \mathsf{psk}$ at $\log_k(N)$ elements of $\mathbb{F}_p$. In our instantiation based on [BCMRS25], we have $t = 1300$ to $t = 2700$ PPRF instances per layer with the same mpk, and values $N \in [2^{30}/t, 2^{40}/t]$. For such values, the new PPRF always enables a much better tradeoff than both BCMPR and CHHK.

2.8 Second Contribution: from PPRFs to FSS via DKG-friendliness of the Correlation-Robust Hash

From the above PPRF we immediately obtain a KI-DPF with reusable setup for a point function $f_{\alpha,\beta}$ such that $f(\alpha) = \beta$ via the standard generic transform that "plants" the missing share:

$\mathsf{DPF.Setup}(1^\lambda)$**:** Output $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{F.Setup}(1^\lambda)$.

$\mathsf{DPF.Gen}(\mathsf{mpk}, \mathsf{msk}, f_{\alpha,\beta})$**:** Set $\mathsf{sk} \xleftarrow{\$} \mathsf{F.KeyGen}(\mathsf{mpk})$, $y'_\alpha = \mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, \alpha) + \beta$, $\mathsf{psk} \xleftarrow{\$} \mathsf{F.Puncture}(\mathsf{msk}, \mathsf{sk}, \alpha)$. Output $k_0 = (\mathsf{msk}, \mathsf{sk})$, $k_1 = (\mathsf{mpk}, \mathsf{psk}, y'_\alpha, \alpha)$.

$\mathsf{DPF.Eval}(k_b, x)$**:** If $b = 0$ parse $(\mathsf{msk}, \mathsf{sk}) \leftarrow k_0$, and output $\mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, x)$. If $b = 1$ parse $(\mathsf{mpk}, \mathsf{psk}, y'_\alpha, \alpha) \leftarrow k_1$, if $x \neq \alpha$ output $\mathsf{F.PEval}(\mathsf{mpk}, \mathsf{psk}, x)$, and otherwise output y'_α.

Distributing Key Generation. The main challenge lies in extending the efficient DKG of the PPRF to the KI-DPF setting. Concretely, assuming that the underlying PPRF admits an efficient DKG and that the parties are given shares (β_0, β_1) of β, the parties must securely compute the following functionality:[18]

Input: P_0 inputs $k_0 = (\mathsf{msk}, \mathsf{sk})$, β_0, and P_1 inputs (α, β_1).
Output: The parties receive shares (s_0, s_1) of $\mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, \alpha)$.

Once the parties get (s_0, s_1) then P_0 simply sends $s_0 + \beta_0$ to P_1, from which P_1 reconstructs $y'_\alpha = \mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, \alpha) + \beta = (s_0 + \beta_0) + s_1 + \beta$. Opening up our DKG-friendly PPRF, the parties want to get shares of $\mathsf{H}(\mathsf{sk}(\alpha) \cdot H)$. Observe that P_1 can compute $\mathsf{psk}(\alpha)$ locally, and[19]

$$\mathsf{psk}(\alpha) = \mathsf{psk}_0^{\sum_{i=1}^m \alpha_i^2} \cdot \prod_{i=1}^m \mathsf{psk}_i^{\alpha_i} \cdot \mathsf{psk}_{m+1} = r^{\mathsf{ED}_k(\alpha,\alpha)} \cdot \mathsf{sk}(\alpha) = \mathsf{sk}(\alpha).$$

With this in mind, we tweak the PPRF setup and add $\rho = \mathsf{DLOG}_G(H)$ to msk. We denote a sharing of a value $v \in \mathbb{F}_p$ over $\mathbb{F}_p$ as $\langle\!\langle v \rangle\!\rangle_p$.[20] The DKG proceeds in four steps:

[18] In the VOLE-PCF construction, β is of the form $\Delta \cdot e$ where Δ is known to one party, and e is a nonzero entry of the noise vector known to the other. Shares of β can be obtained with one secure multiplication protocol.

[19] We note that our final construction, in the malicious setting, uses a different approach, where a masked version of $\mathsf{sk}(\alpha)$ is computed in parallel by the parties. The description in this section conveys the high-level intuition but is simplified to the honest-but-curious setting.

[20] The reader can think of it as additive sharing at this stage, but we will later use authenticated sharing for security against malicious adversaries.

1. P_0 and P_1 input $(\rho, \mathsf{psk}(\alpha))$ to a secure multiplication, and receive output $(\langle\!\langle \rho \cdot \mathsf{psk}(\alpha) \rangle\!\rangle_p^0, \langle\!\langle \rho \cdot \mathsf{psk}(\alpha) \rangle\!\rangle_p^1)$.
2. P_0 and P_1 compute shares of $\mathsf{psk}(\alpha) \cdot H$ *over the group* $\mathbb{G}$ as $F_0 = \langle\!\langle \rho \cdot \mathsf{psk}(\alpha) \rangle\!\rangle_p^0 \cdot G$ and $F_1 = \langle\!\langle \rho \cdot \mathsf{psk}(\alpha) \rangle\!\rangle_p^1 \cdot G$, respectively.
3. P_0 and P_1 run a secure protocol that, on input F_0, F_1, reconstructs $F = F_0 + F_1$ and outputs shares $\langle\!\langle F \rangle\!\rangle$ of F *over a field* $\mathbb{F}$, where $\mathbb{F}$ can be, for example, $\mathbb{F}_2$ in which case F is shared bitwise, or the field over which the curve $\mathbb{G}$ is defined $\mathbb{F}_q$ in which case F is a pair of $\mathbb{F}_q$ elements.
4. P_0 and P_1 run a secure protocol to compute shares of $\mathsf{H}(F)$ from $\langle\!\langle F \rangle\!\rangle$.

Step 1 requires one secure multiplication. Step 2 is local. Step 3 and 4, however, can be quite expensive; a naive implementation would require secure bit decomposition, and secure evaluation of a hash function on bitwise-shared inputs. These steps are the efficiency bottleneck of our DKG.

We summarize what our approach achieves. A DKG for the GGM KI-DPF outlined in Section 2.5 requires at least $2 \log N$ secure evaluations of a symmetric primitive (such as AES). Using our algebraic KI-DPF, the DKG cost is reduced to that of our PPRF: around $3 \log_k(N) \cdot (k-1)^2$ exponentiations to compute $(H_i)_i$ and prove well-formedness, around $\log N$ OTs on committed authenticated values with a some additional consistency proofs, and a single secure hash evaluation.

A DKG-friendly Correlation-Robust Hash over Chains of Curves. Let us denote $\mathbb{G}'$ the elliptic curve underlying our target signature scheme (e.g., secp256k1 for ECDSA or the version of Schnorr used in Bitcoin's Taproot update). Let p' denote its order. To further improve efficiency, assume that we have the following chain of curves: an elliptic curve $\mathbb{G}$ of order p defined over a field $\mathbb{F}_q$, and an elliptic curve $\overline{\mathbb{G}}$ of order q defined over a field $\mathbb{F}_{p'}$, where p' is the order of $\mathbb{G}'$. In general, such chains can be constructed using standard methods for constructing cryptographically-secure elliptic curves defined over a target field (a process which is well-understood, albeit nontrivial when one wants to get fast curve operations and optimal security). In the specific case of secp256k1, the curve underlying both ECDSA and Schnorr in leading blockchains, it turns out that such a chain exists with $\mathbb{G}' = \mathbb{G}$ and $p' = p$, as secp256k1, defined by the formula $y^2 = x^3 + 7 \bmod q$, is a Koblitz curve defined over a field $\mathbb{F}_q$ that forms a 2-cycle with the curve secq256k1 of order q defined by the same formula $y^2 = x^3 + 7 \bmod p$ over the field $\mathbb{F}_p$.

Given such a chain of curve, we introduce a new hash function $\mathsf{H} \colon \mathbb{G} \to \mathbb{F}_{p'}$. For the security of our PPRF and KI-DPF, we need H to satisfy the following form of correlation-robustness: given a secret point H, no polynomial-time adversary should be able to distinguish values of the form $\mathsf{H}(x \cdot H)$ (for distinct x's in $\mathbb{Z}_p$) from random (in other words, the function $F_H \colon x \mapsto \mathsf{H}(x \cdot H)$ should be a PRF). Now, fix a generator $\overline{G}$ of $\overline{\mathbb{G}}$. Given this target security notion, our new hash function is constructed as follows (recall that $.\mathsf{x}$ denotes projection to the first coordinate of an elliptic curve point): given $F \in \mathbb{G}$, return $((F.\mathsf{x}) \cdot \overline{G}).\mathsf{x}$. That is, H projects its input to its x-coordinate, parses the projection as a scalar over $\overline{\mathbb{G}}$, computes a scalar multiplication, and projects the result again. This type of "double exponentiation" bears some resemblance with the power-DDH-based correlation-robust hash function from [AHI11]. However, our construction targets a different correlation ([AHI11] targets additive correlations over $\mathbb{F}_p$, we target "scalar multiplication over $\mathbb{G}$" correlations) and our construction's exponentiations are all scalar multiplications over a group ([AHI11] uses a standard exponentiation over the exponent group) interleaved with projections. Nevertheless, the security of our construction relies "in spirit" on some form of the power-DDH assumption, where the powers are replaced by scalar multiplications over a curve followed by a projection. This assumption, while new, is simple and plausible; our assumption is a static, falsifiable variant of power-DDH. To support its security, we formally prove in Section 8 that it holds in the elliptic-curve generic group model (when modelling both $\mathbb{G}$ and as generic elliptic curves), a variant of the generic group model introduced in [GS22] which accounts for some of the idiosyncrasies of elliptic curves within the generic group model (such as the existence of a projection map).

Distributed Hashing. Now, Steps 3-4 of our DKG consist of two sequential secure evaluations of an elliptic-curve addition formula:

3. One secure evaluation of $\mathbb{G}$'s addition formula to reconstruct F as $F_0 + F_1$ and distribute additive shares of F over $\mathbb{F}_q$, where projecting to the x-coordinate amounts to keeping only the first share.

3'. Parties locally compute $F'_i \leftarrow \langle\!\langle F \rangle\!\rangle^i.\mathsf{x} \cdot G'$ from $\langle\!\langle F \rangle\!\rangle^i.\mathsf{x}$.

4. One secure addition over $\overline{\mathbb{G}}$ to reconstruct shares of $F' = F'_0 + F'_1$ over $\mathbb{F}_{p'}$.

A curve addition of points (x_1, y_1) and (x_2, y_2) proceeds by computing the slope $\lambda = (y_1 - y_2)/(x_1 - x_2)$ and setting $(x_3, y_3) = (\lambda^2 - x_1 - x_2, \lambda(x_1 - x_3) - y_1)$. A secure division can be done with two secure multiplications via a standard technique[21] when the denominator is nonzero, which in our case holds with overwhelming probability.[22] Hence, a full curve addition boils down to four secure multiplications, one opening, and a few local share additions.

Malicious Security. Thus far, we have ignored security against malicious adversaries. To enhance our protocol to the malicious setting we simply replace additive sharing with authenticated secret sharing, and use standard "SPDZ-style" checks [DPSZ12] to enforce honest behavior.[23] However, an issue appears in Steps 2 and 3' of our DKG in which the parties take shares of a scalar and locally compute a scalar multiplication, obtaining *unauthenticated* shares of a point on the curve. To obtain full malicious security we must add a mechanism to check that the (unauthenticated) point shares input to the secure curve addition protocols are consistent with the (authenticated) shares of the corresponding scalar, which cannot be done via a SPDZ-style check due to the difference in algebraic structure. We solve this issue by employing the optimized bucket-based cut-and-choose approach from Wolverine [WYKW21], which multiplies the cost of the curve addition protocol by $B + 1$ for bucketing parameter B that depends on the total number of hash evaluations and the target statistical security guarantee. For our parameters $B \in \{3, 4\}$.

2.9 A VOLE-PCF from DKG-friendly KI-DPFs

Putting everything together, we obtain a DKG-friendly VOLE-PCF by plugging our new DKG-friendly KI-DPF into the VOLE-PCF from [BCMRS25], taking advantage of the significant opportunity for optimization described below.

Per-Level Optimization. The BCMRS VOLE-PCF from [BCMRS25] is *leveled*. At each level it generates a noisy codeword as a sparse encoding of a pseudorandom message, then uses this codeword as the pseudorandom message in the next level. At each level the codeword is perturbated with a fixed-weight noise vector, so the construction must deal with noise vectors of fixed weight but varying size. Concretely, using the parameters from [BCMRS25], a noise vector has $t = 2700$ nonzero entries, and the length at level ℓ is n_ℓ, where $n_1 = 2^{15}$ and $n_{\ell+1} = 2^5 \cdot n_\ell$. Equivalently, for each level ℓ, the VOLE-PCF keys contain t KI-DPF keys over a domain of size n_ℓ/t.[24]

Recall that the choice of arity k of the basis for the KI-DPF depends on the tradeoff between the number of DPFs and the size of their domain. For lower levels the GGM-based KI-DPF from [Boy+19a] and DKG of Doerner-shelat can be faster, despite the linear-in-n_ℓ/t computation, due to its fully blackbox nature and reliance on exclusively symmetric primitives. Therefore, after selecting sparse-LPN parameters for the VOLE-PCF from [BCMRS25], we use a script to independently select the best possible choice of KI-DPF for each level among $\{\text{GGM, BCMPR, CHHK, } (k\mathsf{NR})_{k \le N}\}$[25], where $k\mathsf{NR}$ denotes our new KI-DPF with arity parameter k. The numbers in Table 1 were obtained with a 4-level instantiation of [BCMRS25] for a target number of 2^{35} VOLE correlations, using the GGM KI-DPF for levels 1 and 2, and $k\mathsf{NR}$ with $k = 16$ at level 3 and $k = 18$ at level 4.

LPN Variants. We turn to the exact variant of sparse-LPN underlying the security of our construction. A standard optimization, common to our work and all previous works on malicious DKG for PCG/PCFs, is to allow a party to execute a selective-failure attack when setting up the keys. Take as a non-notationally dense example the BCMPR PPRF. To distribute $\mathsf{psk}_i = \mathsf{sk}_i \cdot r^{(-1)^{z_i}}$ for $i \in [m]$, the DKG protocol instructs one party to sample sk_i, commit to $(\mathsf{sk}_i \cdot r, \mathsf{sk}_i \cdot r^{-1})$, prove properties of the commitments, and

[21] To compute a/b, securely compute shares of $a \cdot r$, $b \cdot r$ for a random r, open $b \cdot r$, and let the parties locally compute shares of $a/b = ar/(br)$

[22] One of the two points to add is always distributed uniformly random in the group.

[23] Our full DKG, including the DKG for the PPRF, additionally employs a VOLE-based zero-knowledge proof system such as Quicksilver [YSWW21] to enforce honest behavior. We omit discussions of the implementation details, as these techniques are standard.

[24] This is a slight simplification: We further optimize our parameters, by choosing different weights $t^{(\ell)}$ for each level ℓ. The larger LPN dimensions on higher levels allow for fewer noise coordinates and, therefore, fewer KI-DPF instances (with in turn larger domains).

[25] The best choices were always combinations of GGM and $k\mathsf{NR}$ with different k.

prepare an OT with inputs $(\mathsf{sk}_i \cdot r, \mathsf{sk}_i \cdot r^{-1})$. The other party uses z_i as selection bit and checks that the opening is correct. It is easy to see that the first party can guess z_i,place an incorrect opening in one of the messages, and confirm their guess. If an incorrect guess is made the malicious behavior is detected and the protocol aborts. Allowing a selective failure attack, which permits much more efficient DKGs, translates to assuming the security of a variant of LPN termed *LPN with static leakage* that models this kind of leakage. In LPN with static leakage the adversary places a guess about the secret noise, e.g., a subset of non-noisy positions, and learns if this guess is correct. If the guess is incorrect then the attack automatically fails. We use the same leakage as all prior work [Boy+19a; Boy+20c; WYKW21] and refer the reader to [Boy+20c] for discussion and a reduction to standard LPN.

In this work, we introduce another slight variant of LPN to correct a subtle mismatch between the guarantees provided by LPN and the security requirements of distributed signatures. Recall that the VOLE-PCF lets the parties generate shares of the entries of $H \cdot (\Delta \cdot \boldsymbol{e})$, where $\boldsymbol{e}$ is a noise vector and H a code matrix. In the security game of the distributed signature, the adversary plays in some number N of signing session, and can freely choose the messages m_i to be signed. In a *stateless* setting this message is mapped via a random oracle to a row $\boldsymbol{h}_i = \mathsf{H}(m_i)$ of the lazily sampled matrix H, and the parties locally construct shares of $\Delta \cdot \boldsymbol{h}_i \cdot \boldsymbol{e}$. One might expect to reduce security to a dual-LPN instance with N samples, but there is a mismatch. In dual-LPN, the N rows of H are picked independently and uniformly, whereas in the signature security game the adversary gets to pick the messages m_i adaptively. Specifically, the adversary can *rejection sample* the rows of H locally, trying many candidates $\boldsymbol{h}_i = \mathsf{H}(m_i)$ until they find a row with certain properties. A straightforward fix is to instead reduce to the security of LPN with Q samples where Q is a bound on the number of RO-queries of the adversary, but this is clearly wasteful as one must pick parameters assuming the adversary gets to see all Q samples of the form $\boldsymbol{h}_i \cdot \boldsymbol{e}$ when in reality the adversary sees only N samples.[26]

To address this mismatch, we introduce the *filtered* (sparse) LPN assumption that captures exactly the above situation. In Q-query N-sample filtered LPN the adversary gets to query up to Q random rows and receives LPN samples $\boldsymbol{h} \cdot \boldsymbol{e}$ for N of them that it adaptively selects (thus filtering the rows). A common framework to analyze the security of LPN variants is the *linear test framework* [Boy+20a; CRR21] that captures many known attacks. After formally introducing filtered LPN,we observe that the security analysis of sparse-LPN in the linear test framework from [BCMRS25] extends almost immediately to filtered sparse-LPN (provided in the full version), and allows the derivation of provable parameters in which the row-weight of H is increased slightly to account for filtering. This assumption may be of independent interest, as it captures the right security notion for all LPN-based VOLE-PCFs when they are used in a stateless regime, a well-motivated setting beyond the realm of distributed signatures [GKMN21; KOR23].

2.10 Two-Round Distributed Signing from the VOLE-PCF

We return to the design of a distributed signature from our DKG-friendly VOLE-PCF over $\mathbb{F}_p$, starting from the outline provided in 2.2. The basic scheme can be further improved in the random oracle model to a point where the communication overhead of using a VOLE-PCF to authenticate a nonce is *exactly zero*. To demonstrate this, we start at the deterministic variant of distributed Schnorr, where both the nonces are authenticated with independent VOLE-PCFs.

Setup. The parties run a DKG to compute a public key $\mathsf{pk} = \mathsf{sk} \cdot G_\mathsf{S}$, and additive shares $\mathsf{sk}_A, \mathsf{sk}_B$ of sk. The parties execute two maliciously-secure DKGs to compute $(\mathsf{k}_0^{\mathsf{PCF}}, \mathsf{k}_1^{\mathsf{PCF}}) \leftarrow \mathsf{PCF.KeyGen}(\Delta_A)$ and $(\ell_0^{\mathsf{PCF}}, \ell_1^{\mathsf{PCF}}) \leftarrow \mathsf{PCF.KeyGen}(\Delta_B)$ for VOLE-PCF instances over $\mathbb{F}_{q_\mathsf{S}}$ in which the parties take reversed roles, i.e., Δ_A and Δ_B are random elements of $\mathbb{F}_{q_\mathsf{S}}$ is known to P_A and P_B, respectively.

Signing a message m.

- P_A runs $(k_A, v_1) \leftarrow \mathsf{PCF.Eval}(0, \ell_0^{\mathsf{PCF}}, m)$, sets $R_A \leftarrow k_A \cdot G_\mathsf{S}$ and $V_1 \leftarrow v_1 \cdot G_\mathsf{S}$. Send (R_A, V_1).
- P_B runs $w_1 \leftarrow \mathsf{PCF.Eval}(1, \ell_1^{\mathsf{PCF}}, m)$, verifies $w_1 \cdot G_\mathsf{S} = \Delta_B \cdot R_A + V_1$ and aborts if the check fails, runs $(k_B, v_0) \leftarrow \mathsf{PCF.Eval}(0, \mathsf{k}_0^{\mathsf{PCF}}, m)$, sets $R_B \leftarrow k_B \cdot G_\mathsf{S}$ and $V_0 \leftarrow v_0 \cdot G_\mathsf{S}$, computes $R \leftarrow R_A + R_B$, $e \leftarrow \mathsf{H}_\mathsf{S}(\mathsf{pk}, R, m)$, and $s_B \leftarrow e \cdot \mathsf{sk}_B + k_B$. Finally, P_B sends (R_B, V_0, s_B) to P_A.

[26] Note that in the stateful setting, where the parties are allowed to maintain and update a persistent state, the issue disappears because the parties can feed the message counter i to H instead of m_i.

– P_A runs $w_0 \leftarrow \mathsf{PCF.Eval}(1, \mathsf{k}_1^{\mathsf{PCF}}, m)$ and verifies that $w_0 \cdot G_{\mathsf{S}} = \Delta_A \cdot R_B + V_0$, aborts if the check fails, recomputes e, sets $s_A = e \cdot \mathsf{sk}_A + k_A$, and outputs $\sigma = (R_A + R_B, s_A + s_B)$.

The total communication is 4 elements of $\mathbb{G}_{\mathsf{S}}$ and one element of $\mathbb{F}_{q_{\mathsf{S}}}$. The terms used to authenticate the nonces V_0, V_1 are highlighted in red. Explicitly communicating these terms, in fact, is not necessary because each party can locally compute the authentication term of the other. For example, from the verification equation $w_1 \cdot G_{\mathsf{S}} =_? \Delta_B \cdot R_A + V_1$, P_B can compute $\tilde{V}_1 = w_1 \cdot G_{\mathsf{S}} - \Delta_B \cdot R_A$ that is used by an honest P_A. The security of the VOLE-PCF guarantees that P_A can only compute the same $V_1 = \tilde{V}_1$ if R_A was computed correctly (up to a negligible error equal to the probability of guessing Δ_A). Therefore, for a hash function H' modeled as a random oracle, P_B can use $\mathsf{H}'(\tilde{V}_1)$ to *mask the next message*. In the ROM, P_B's message to P_A looks random and causes the signature to fail unless P_A can query H' on (knows) $V_1 = \tilde{V}_1$.

Observe that a similar approach can be used to remove V_0 by having P_A locally compute $\tilde{V}_0 = w_0 \cdot G_{\mathsf{S}} - \Delta_A \cdot R_B$ and unmask the message received from P_B with $\mathsf{H}'(\tilde{V}_0)$ before computing σ. An honest P_B must preemptively compute $\mathsf{H}'(V_0)$ for the masking; if P_B behaved dishonestly then the probability of querying $\tilde{V}_0$ is negligible, and the masking process will result in a uniformly random string thereby causing the signature to abort with overwhelming probability. Changes are in red below:

Even Better Signing.

– P_A runs $(k_A, v_1) \leftarrow \mathsf{PCF.Eval}(0, \ell_0^{\mathsf{PCF}}, m)$, sets $V_1 \leftarrow v_1 \cdot G_{\mathsf{S}}$, and sends $R_A \leftarrow k_A \cdot G_{\mathsf{S}}$ to P_B.
– P_B runs $w_1 \leftarrow \mathsf{PCF.Eval}(1, \ell_1^{\mathsf{PCF}}, m)$, computes $\tilde{V}_1 = w_1 \cdot G_{\mathsf{S}} - \Delta_B \cdot R_A$, runs $(k_B, v_0) \leftarrow \mathsf{PCF.Eval}(0, \mathsf{k}_0^{\mathsf{PCF}}, m)$, sets $V_0 \leftarrow v_0 \cdot G_{\mathsf{S}}$, $R_B \leftarrow k_B \cdot G_{\mathsf{S}}$, and $R \leftarrow R_A + R_B$, computes $e \leftarrow \mathsf{H}_{\mathsf{S}}(\mathsf{pk}, R, m)$, $s'_B \leftarrow e \cdot \mathsf{sk}_B + k_B + \mathsf{H}(\tilde{V}_1) + \mathsf{H}(V_0)$, and sends (R_B, s'_B) to P_A.
– P_A runs $w_0 \leftarrow \mathsf{PCF.Eval}(1, \mathsf{k}_1^{\mathsf{PCF}}, m)$ and compute $\tilde{V}_0 = w_0 \cdot G_{\mathsf{S}} - \Delta_A \cdot R_B$. Recomputes e, sets $s_B = s'_B - \mathsf{H}(V_1) - \mathsf{H}(\tilde{V}_0)$, $s_A = e \cdot \mathsf{sk}_1 + k_A$, and outputs $\sigma = (R_A + R_B, s_A + s_B)$.

With these changes, we get a two-round two-party protocol for distributed, stateless, deterministic Schnorr signatures using only two group elements and one $\mathbb{F}_{q_{\mathsf{S}}}$ elements—a total of *96 bytes*, only 50% more than the size of a Schnorr signature—of communication. If randomized stateful signatures suffice the first VOLE-PCF call can be removed by letting P_A pick R_A at random without security loss, provided that P_B obtains a different nonce R_B in every signing session, ensured by letting the parties maintain a counter ctr and evaluating $\mathsf{PCF.Eval}$ on ctr instead of m, to mitigate nonce-reuse attacks.

Further Improvements for ECDSA. As discussed in Section 2.2, when applying the same template for ECDSA an additional VOLE-PCF instance is used to compute shares of the product $u \cdot \mathsf{sk}_A$ for u known to P_B. Instead of using another VOLE-PCF instance to authenticate R_B it is possible to use the VOLE-PCF instance with MAC key $\mathsf{sk}_A = \Delta_A$. The public key $\mathsf{pk}_A = \mathsf{sk}_A \cdot G_{\mathsf{E}}$ allows an attack on authentication, so this variant requires P_B to prove knowledge of (k_B, v_0) such that $R_B = k_B \cdot G$ and $V_0 = v_0 \cdot G$ to guarantee that an ECDSA forgery can be computed from the outputs of a malicious P_B sending $(R_B^*, V_0^*) \neq (R_B, V_0)$ that P_A would accept. To further compress we leverage an observation from [Adj+24]: under the knowledge of exponent assumption (KEA), $\pi_B := k_B \cdot (\mathsf{sk}_A \cdot G_{\mathsf{E}})$ and $\pi_0 := v_0 \cdot (\mathsf{sk}_A \cdot G_{\mathsf{E}})$ functions as a deterministic proof of knowledge that can be verified by P_A with sk_A. We can apply the same trick as above by masking the message of P_B with $\mathsf{H}(V_0, \tilde{V}_1, \pi_0, \pi_B)$. This yields a protocol with identical communication and computation as above, paying the KEA assumption for the ability to use only two executions of the VOLE-PCF DKG and reducing setup runtime by 22 seconds on one core of a laptop (according to our estimates).

3 Formalized Preliminaries

3.1 Notation

We denote by λ and ρ the computational and statistical security parameters, respectively. We denote by $\mathsf{negl}(\lambda)$ a negligible function in λ and by $\mathsf{poly}(\lambda)$ a polynomial function in λ. PPT refers to probabilistic polynomial-time. For integers $a \leq b$ we write $[a, b] := \{a, \ldots, b\}$, $[a, b) := \{a, \ldots, b-1\}$, and $[b] := [1, b]$. We denote a vector $\boldsymbol{v}$ in bold, and the hamming weight of a vector is $\omega(\boldsymbol{v})$.

In this work, we work over Abelian cyclic groups with additive notation. We denote by $\mathcal{G}$ a PPT algorithm that on input a security parameter λ outputs a tuple $(\mathbb{G}, G, p)$, where $\mathbb{G}$ is a cyclic group of prime order p generated by G.

Cycle of Elliptic Curve Groups. Let $\mathbb{G}$ and $\overline{\mathbb{G}}$ be a cycle of prime-order elliptic curve groups such that $\mathbb{G} = \langle G \rangle$ with order p is defined over the base field $\mathbb{F}_q$, and $\overline{\mathbb{G}} = \langle \overline{G} \rangle$ with order q is defined over the base field $\mathbb{F}_p$.[27] For an elliptic curve point $P = (x, y) \in \mathbb{G} \setminus \{\mathcal{O}\}$ (or $\in \overline{\mathbb{G}}$), we write $(\cdot).\mathsf{x} \colon \mathbb{G} \to \mathbb{F}_q$ to project onto the x coordinate such that $P.\mathsf{x} = x$.

EC-GGM. The Generic Group Model (GGM) [Sho97] is a heuristic framework for security analyses against adversaries that are only granted black-box access to a cryptographic group. Elements of a generic group are represented by encodings, i.e., the image of the group element under a bijective map from the group onto itself that destroys the underlying group structure. An adversary performs group operations over these encoded elements, such as scalar multiplication and addition, via an oracle. The Elliptic Curve Generic Group Model (EC-GGM) [GS22] is an adaptation of the GGM that accounts for the additional algebraic structure of elliptic curves, namely by explicitly enforcing symmetry of the encoding.

Definition 1 (Elliptic Curve Generic Group Model (EC-GGM)). Let $\mathbb{G}$ be an elliptic curve group over the base field $\mathbb{F}_q$ generated by a point $G \in \mathbb{G} \setminus \{\mathcal{O}\}$ of prime order p. Each point $P \in \mathbb{G}$ has a unique discrete logarithm $x \in \mathbb{F}_p$ such that $P = x \cdot G$, and in the EC-GGM this value is represented by its image under a random bijective map:

$$\pi \colon \mathbb{F}_p \to \mathbb{G} \text{ such that } \pi(0) = \mathcal{O}, \pi(1) = G, \text{ and } \pi(-x) = -\pi(x) \text{ for all } x \in \mathbb{F}_p.$$

In the EC-GGM an adversary exclusively interacts with the generic group via the following oracles.

Mapping: $\mathsf{map}(x) \to \pi(x)$**.** On input $x \in \mathbb{F}_p$, outputs $\pi(x)$.
Addition: $\mathsf{add}(a, P, b, Q)$**.** On inputs $a, b \in \mathbb{F}_p$ and $P, Q \in \mathbb{G}$, outputs $\pi\left(a \cdot \pi^{-1}(P) + b \cdot \pi^{-1}(Q)\right)$.

3.2 Signature Schemes

We recall the definition of existentially unforgeable digital signatures in Definition 2 [GMR88]. We present the ECDSA [Nat19] and Schnorr [Sch91] signature schemes in Figure 2.

Definition 2 (Existentially Unforgeable Digital Signature Scheme). A digital signature scheme $\Sigma = (\mathsf{KG}, \mathsf{Sign}, \mathsf{Ver})$ for the message space $\mathcal{M}$ is a tuple of algorithms with the following syntax:

$\Sigma.\mathsf{KG}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$**.** On input a security parameter 1^λ, key generation outputs a public key pk and a signing key sk.
$\Sigma.\mathsf{Sign}(\mathsf{sk}, m) \to \sigma$**.** On input a secret key sk and a message m, signing outputs a signature σ.
$\Sigma.\mathsf{Ver}(\mathsf{pk}, m, \sigma) \to \{0, 1\}$**.** On input a public key pk, message $m \in \mathcal{M}$ and signature σ, verification outputs 1 for accept and 0 for reject.

Perfect Corretness. For all $\lambda \in \mathbb{N}$, all $(\mathsf{pk}, \mathsf{sk}) \leftarrow \Sigma.\mathsf{KG}(1^\lambda)$ and all messages $m \in \mathcal{M}$:

$$\Pr\left[\Sigma.\mathsf{Ver}\left(\mathsf{pk}, m, \sigma\right) = 1 | \sigma \leftarrow \Sigma.\mathsf{KG}(\mathsf{sk}, m)\right] = 1.$$

Existential Unforgeability. For all PPT adversaries and for all $\lambda \in \mathbb{N}$:

$$\Pr\left[\mathsf{Exp}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A}}(1^\lambda) = 1\right] = \mathsf{negl}(\lambda).$$

[27] We can instantiate such a cycle for example by using $\mathbb{G} = \texttt{secp256k1}$ and $\overline{\mathbb{G}} = \texttt{secq256k1}$.

Fig. 2: ECDSA, Schnorr and Existential Unforgeability

ECDSA Signature Scheme
$\mathsf{ECDSA} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ in group $\mathcal{G}_\mathsf{E} = (\mathbb{G}_\mathsf{E}, G_\mathsf{E}, q_\mathsf{E})$
with function $\mathsf{H}_\mathsf{E} : \{0,1\}^* \to \mathbb{F}_{q_\mathsf{E}}$.

$\mathsf{KeyGen}(1^\lambda, \mathcal{G})$:

$\mathsf{sk} \in_R \mathbb{F}^*_{q_\mathsf{E}}$
$\mathsf{pk} := \mathsf{sk}_\mathsf{E} \cdot G_\mathsf{E}$
Output $(\mathsf{pk}, \mathsf{sk})$

$\mathsf{Sign}(\mathsf{sk}, m \in \{0,1\}^*)$:

$z := \mathsf{H}_\mathsf{E}(m) \in \mathbb{F}_{q_\mathsf{E}}$
$k \xleftarrow{\$} \mathbb{F}^*_{q_\mathsf{E}}$
$r := (k \cdot G_\mathsf{E}).\mathsf{x}$
$s := k^{-1} \cdot (z + r \cdot \mathsf{sk})$
Output (r, s)

$\mathsf{Verify}(\mathsf{pk}, m, \sigma)$:

$(r, s) := \sigma$
$z := \mathsf{H}_\mathsf{E}(m)$
$u_1 := z \cdot s^{-1}$
$u_2 := r \cdot s^{-1}$
$(x, y) := u_1 \cdot G_\mathsf{E} + u_2 \cdot \mathsf{pk}_\mathsf{E}$
If $r = x \bmod p$**:**
  Return 1.
Else: Return 0.

Schnorr Signature Scheme
$\mathsf{Schnorr} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ in group $\mathcal{G}_\mathsf{S} = (\mathbb{G}_\mathsf{S}, G_\mathsf{S}, q_\mathsf{S})$
with function $\mathsf{H}_\mathsf{S} : \{0,1\}^* \to \mathbb{F}_{q_\mathsf{S}}$.

$\mathsf{KeyGen}(1^\lambda, \mathcal{G})$:

$\mathsf{sk} \in_R \mathbb{F}^*_{q_\mathsf{S}}$
$\mathsf{pk} := \mathsf{sk} \cdot G_\mathsf{S}$
Output $(\mathsf{pk}, \mathsf{sk})$

$\mathsf{Sign}(\mathsf{sk}, m \in \{0,1\}^*)$:

$k \xleftarrow{\$} \mathbb{F}_{q_\mathsf{S}}$
$R := k \cdot G_\mathsf{S}$
$e := \mathsf{H}_\mathsf{S}(\mathsf{pk}, R, m)$
$s := k + e \cdot \mathsf{sk}$
Output (R, s)

$\mathsf{Verify}(\mathsf{pk}, m, \sigma_\mathsf{S})$:

$(R, s) := \sigma_\mathsf{S}$
$e := \mathsf{H}_\mathsf{S}(\mathsf{pk}, R, m)$
If $s \cdot G_\mathsf{S} = R + e \cdot \mathsf{pk}$**:**
  Return 1.
Else: Return 0.

Existential Unforgeability of a signature scheme $\Sigma = (\mathsf{KG}, \mathsf{Sign}, \mathsf{Ver})$,
EUF-CMA Game with oracle $\mathcal{O}^{\mathsf{sig}}(\cdot)$:

$\mathsf{Exp}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A}}(1^\lambda)$:

$Q := \emptyset$
$(\mathsf{pk}, \mathsf{sk}) \leftarrow \Sigma.\mathsf{KG}(1^\lambda)$
$(\sigma, m^*) \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{sig}}(\cdot)}(1^\lambda, \mathcal{G}, \mathsf{pk})$
If $m^* \in Q$ **or** $1 \neq \Sigma.\mathsf{Ver}(\mathsf{pk}, m^*, \sigma^*)$**:**
  Return 0.
Else: Return 1.

$\mathcal{O}^{\mathsf{sig}}(m \in \{0,1\}^*)$:

$Q := Q \cup \{m\}$
$\sigma \leftarrow \Sigma.\mathsf{Sign}(\mathsf{sk}, m)$
Return σ

3.3 Pseudorandom Generators, Functions and Correlation Robust Functions

We recall the definition of pseudorandom generators.

Definition 3 (Pseudorandom Generators (PRGs)). Let $\lambda \in \mathbb{N}$. A pseudorandom generator (PRG) is a polynomial-time function $\mathsf{PRG} : \{0,1\}^\lambda \to \{0,1\}^{\ell(\lambda)}$ such that

- $\ell(\lambda) > \lambda$.
- for all $s \xleftarrow{\$} \{0,1\}^\lambda$ it holds that $\mathsf{PRG}(s) \overset{c}{\approx} u$, where $u \xleftarrow{\$} \{0,1\}^{\ell(\lambda)}$.

Next, we recall the definition for strong [GGM84] and weak [NR95] pseudorandom functions in Definition 4. These are defined with respect to Figure 3.

Definition 4 (Pseudorandom Functions). Let $\mathcal{K} = (K_\lambda)_\lambda$, $\mathcal{Y} = (Y_\lambda)_\lambda$, $\mathcal{Z} = (z_\lambda)_\lambda$ be ensembles of finite sets indexed by $\lambda \in \mathbb{N}$.

A family of polynomial-time computable functions $\mathcal{F} = (F_\lambda)_\lambda$ where $F_\lambda : K_\lambda \times Z_\lambda \to Y_\lambda$ is a pseudorandom function (PRF) if for all $\lambda \in \mathbb{N}$ and all PPT adversaries $\mathcal{A}$:

$$\left|\Pr\left[\mathsf{Exp}^{\mathsf{PRF}}_{\mathcal{A},0}\left(1^\lambda, F_\lambda\right) = 1\right] - \Pr\left[\mathsf{Exp}^{\mathsf{PRF}}_{\mathcal{A},1}\left(1^\lambda, F_\lambda\right) = 1\right]\right| = \mathsf{negl}(\lambda).$$

A family of polynomial-time computable functions $\mathcal{F}$ is a weak pseudorandom function (wPRF) if for all $\lambda \in \mathbb{N}$ and all PPT adversaries $\mathcal{A}$:

$$\left|\Pr\left[\mathsf{Exp}^{\mathsf{wPRF}}_{\mathcal{A},0}\left(1^\lambda, F_\lambda\right) = 1\right] - \Pr\left[\mathsf{Exp}^{\mathsf{wPRF}}_{\mathcal{A},1}\left(1^\lambda, F_\lambda\right) = 1\right]\right| = \mathsf{negl}(\lambda).$$

We recall the definition of correlation robust hash functions in Definition 5. Informally, a hash function H is correlation robust (CR) [IKNP03] (resp. strong CR) for correlation C if samples $(x, \mathsf{H}(C(s,x)))$ are pseudorandom for uniform x (resp. adversarially-chosen x), where s is a uniform secret.

Definition 5 (Correlation Robust Hash Functions). Let $\mathcal{J} = (J_\lambda)_\lambda$, $\mathcal{S} = (S_\lambda)_\lambda$, $\mathcal{X} = (X_\lambda)_\lambda$, $\mathcal{Y} = (Y_\lambda)_\lambda$, $\mathcal{Z} = (Z_\lambda)_\lambda$ be ensembles of finite sets indexed by security parameter $\lambda \in \mathbb{N}$. We define correlation robustness with respect to the games in Figure 3.

A family of polynomial-time computable hash functions $\mathcal{H} = (\mathsf{H}_\lambda)_\lambda$ where $\mathsf{H}_\lambda : J_\lambda \times X_\lambda \to Y_\lambda$ is correlation robust (CR) for the correlations $\mathcal{C} = (C_\lambda)_\lambda$ with $C_\lambda : S_\lambda \times Z_\lambda \to X_\lambda$ if for all PPT adversaries $\mathcal{A}$:

$$\left|\Pr\left[\mathsf{Exp}^{\mathsf{CRH}}_{\mathcal{A},0}(1^\lambda, H_\lambda, C_\lambda) = 1\right] - \Pr\left[\mathsf{Exp}^{\mathsf{CRH}}_{\mathcal{A},1}(1^\lambda, H_\lambda, C_\lambda) = 1\right]\right| = \mathsf{negl}(\lambda).$$

The family $\mathcal{H}$ is weak correlation robust (wCR) for $\mathcal{C}$ if for all $\lambda \in \mathbb{N}$ and all PPT $\mathcal{A}$:

$$\left|\Pr\left[\mathsf{Exp}^{\mathsf{wCRH}}_{\mathcal{A},0}(1^\lambda, H_\lambda, C_\lambda) = 1\right] - \Pr\left[\mathsf{Exp}^{\mathsf{wCRH}}_{\mathcal{A},1}(1^\lambda, H_\lambda, C_\lambda) = 1\right]\right| = \mathsf{negl}(\lambda).$$

Figure 3 clarifies the correspondence between PRF Definition 4 and CRH Definition 5 observed by [GOR11]; given a strong (resp. weak) CR hash $\mathsf{H}_\lambda : J_\lambda \times X_\lambda \to Y_\lambda$ for a correlation C_λ the composed function $F_\lambda := \mathsf{H}_\lambda \circ C_\lambda : J_\lambda \times S_\lambda \times Z_\lambda \to Y_\lambda$ is a strong (resp. weak) PRF $F_\lambda : (J_\lambda \times S_\lambda) \times Z_\lambda \to Y_\lambda$ with key space $K_\lambda = J_\lambda \times S_\lambda$ and input space Z_λ. A strong (resp. weak) PRF $F_\lambda : (K_\lambda \times S_\lambda) \times Z_\lambda \to Y_\lambda$ with input space Z_λ that is the composition of two functions $\mathsf{H}_\lambda : J_\lambda \times X_\lambda \to Y_\lambda$ and $C_\lambda : S_\lambda \times Z_\lambda \to X_\lambda$ implies that H_λ is strong (resp. weak) CR for the correlation C_λ.

We consider the correlation $C(S, z) := z \cdot S$ where $S \in \mathbb{G}$ and $z \in \mathbb{F}_p$ for a cryptographic group $\mathbb{G}$ of order p. This correlation is elsewhere referred to as "exponential" in $\mathbb{G}$ [BCMPR24; CHHK25] but is herein referred to as "multiplicative" in $\mathbb{G}$ as our groups are written additively. The weak and strong variants of correlation robustness for multiplicative correlations over the group $\mathbb{G}$, denoted $\mathbb{G}$-wCR and $\mathbb{G}$-CR, are presented in Definition 6.

Fig. 3: Pseudorandom and Correlation Robust Hash Functions

Below are the games for Pseudorandom Functions (PRF, Definition 4) and Correlation Robust Hash Functions (CRH, Definition 5).

These games are defined with respect to the following, indexed by security parameter $\lambda \in \mathbb{N}$:

- Ensembles of finite sets:
  $\mathcal{J} = (J_\lambda)_\lambda$, $\mathcal{K} = (K_\lambda)_\lambda$, $\mathcal{S} = (S_\lambda)_\lambda$, $\mathcal{X} = (X_\lambda)_\lambda$, $\mathcal{Y} = (Y_\lambda)_\lambda$, $\mathcal{Z} = (z_\lambda)_\lambda$.
- $\mathcal{F} = (F_\lambda)_\lambda$: a family of polynomial-time computable functions, where
  $F_\lambda : K_\lambda \times Z_\lambda \to Y_\lambda$.
- $\mathcal{H} = (\mathsf{H}_\lambda)_\lambda$: a family of polynomial-time computable functions, where
  $\mathsf{H}_\lambda : J_\lambda \times X_\lambda \to Y_\lambda$.
- $\mathcal{C} = (C_\lambda)_\lambda$: a collection of correlations, where $C_\lambda : S_\lambda \times Z_\lambda \to X_\lambda$.
- Φ_λ: the set of all functions $f : Z_\lambda \to Y_\lambda$.

Pseudorandomness

sPRF Game with oracle $\mathcal{O}_b^{\mathsf{PRF}}(\cdot)$ is **Strong Pseudorandomness**,

wPRF Game with oracle $\mathcal{O}_b^{\mathsf{wPRF}}(\cdot)$ is **Weak Pseudorandomness**.

$\mathsf{Exp}^{\mathsf{PRF},\,\mathsf{wPRF}}_{\mathcal{A},b}(1^\lambda, F_\lambda)$:

$\mathcal{Q} := \emptyset$
$k \xleftarrow{\$} K_\lambda$
$f_0 := F_\lambda(k,)$
$f_1 \xleftarrow{\$} \Phi_\lambda$
$b' \leftarrow \mathcal{A}^{\mathcal{O}_b^{\mathsf{PRF}}(\cdot),\,\mathcal{O}_b^{\mathsf{wPRF}}(\cdot)}(1^\lambda)$
Output b'

$\mathcal{O}_b^{\mathsf{PRF}}(z \in Z_\lambda)$:

$Q := Q \cup \{z\}$
$y := f_b(z)$
Return y

$\mathcal{O}_b^{\mathsf{wPRF}}(\cdot)$:

$z \xleftarrow{\$} Z_\lambda$
$Q := Q \cup \{z\}$
$y := f_b(z)$
Return y

Correlation Robustness

CR Game with oracle $\mathcal{O}_b^{\mathsf{CRH}}(\cdot)$ is **Strong Correlation Robustness**

wCR Game with oracle $\mathcal{O}_b^{\mathsf{wCRH}}(\cdot)$ is **Weak Correlation Robustness**.

$\mathsf{Exp}^{\mathsf{CRH},\,\mathsf{wCRH}}_{\mathcal{A},b}(1^\lambda, H_\lambda, C_\lambda)$:

$\mathcal{Q} := \emptyset$
$j \xleftarrow{\$} J_\lambda$, $s \xleftarrow{\$} S_\lambda$
$h_0 := \mathsf{H}_\lambda(j, C_\lambda(s, \cdot))$
$h_1 \xleftarrow{\$} \Phi_\lambda$
$b' \leftarrow \mathcal{A}^{\mathcal{O}_b^{\mathsf{CRH}}(\cdot),\,\mathcal{O}_b^{\mathsf{wCRH}}(\cdot)}(1^\lambda)$
Output b'

$\mathcal{O}_b^{\mathsf{CRH}}(z \in Z_\lambda)$:

$Q := Q \cup \{z\}$
$y := h_b(z)$
Return y

$\mathcal{O}_b^{\mathsf{wCRH}}(\cdot)$:

$z \xleftarrow{\$} Z_\lambda$
$Q := Q \cup \{z\}$
$y := h_b(z)$
Return y

Definition 6 (Correlation Robustness for Multiplicative Correlations over $\mathbb{G}$). Let $\mathcal{Y} = (Y_\lambda)_\lambda$ be an ensemble of finite sets indexed by security parameter $\lambda \in \mathbb{N}$, let $(\mathbb{G}, G, p) \xleftarrow{\$} \mathcal{G}(1^\lambda)$ be an additive cryptographic group of order $p = 2^{\omega(\lambda)}$, and let $C_\mathbb{G}(S, x) := x \cdot S$ denote scalar multiplication in $\mathbb{G}$. A hash function family $\mathcal{H} = (\mathsf{H}_\lambda)_\lambda$ where $\mathsf{H}_\lambda : \mathbb{G} \to Y_\lambda$ is strongly correlation robust for scalar multiplication ($\mathbb{G}$-sCR) if for all $\lambda \in \mathbb{N}$ and for all PPT adversaries $\mathcal{A}$:

$$\left|\Pr\left[\mathsf{Exp}^{\mathsf{CRH}}_{\mathcal{A},0}(1^\lambda, \mathsf{H}_\lambda, C_\mathbb{G}) = 1\right] - \Pr\left[\mathsf{Exp}^{\mathsf{CRH}}_{\mathcal{A},1}(1^\lambda, \mathsf{H}_\lambda, C_\mathbb{G}) = 1\right]\right| = \mathsf{negl}(\lambda).$$

The family $\mathcal{H}$ is weakly correlation robust for scalar multiplication ($\mathbb{G}$-wCR) if for all PPT adversaries $\mathcal{A}$:

$$\left|\Pr\left[\mathsf{Exp}^{\mathsf{wCRH}}_{\mathcal{A},0}(1^\lambda, \mathsf{H}_\lambda, C_\mathbb{G}) = 1\right] - \Pr\left[\mathsf{Exp}^{\mathsf{wCRH}}_{\mathcal{A},1}(1^\lambda, \mathsf{H}_\lambda, C_\mathbb{G}) = 1\right]\right| = \mathsf{negl}(\lambda).$$

3.4 Non-Interactive Zero Knowledge Proofs of Knowledge

A standard Σ-protocol [CDS94] for a relation $\mathcal{R}$ is a three-move public-coin interactive protocol between a probabilistic prover $\mathcal{P}(\phi, w)$ and a deterministic verifier $\mathcal{V}(\phi)$ that satisfies completeness, special honest-verifier zero knowledge, and special soundness. Essentially, perfect completeness requires that for all $(\phi, w) \in \mathcal{R}$ the verifier always accepts an interaction with the honest prover. Special honest-verifier zero knowledge requires the existence of an efficient simulator $\mathcal{S}(\phi)$ that, without the witness w, outputs transcripts that are computationally indistinguishable from a real execution between $\mathcal{P}, \mathcal{V}$. Special soundness (k-special soundness) requires the existence of an efficient extractor that can compute a witness from any two (resp. k) accepting transcripts.

A standard Σ-protocol with transcripts (A, e, z) can be compiled into a non-interactive zero knowledge proof of knowledge (NIZKPoK) via the Fiat-Shamir transform [FS87], which replaces the computation of the challenge e sent by $\mathcal{V}$ with $e \leftarrow \mathsf{H}(\phi, A)$, where H is a random oracle. The resulting NIZKPoK can be proven to satisfy completeness, zero knowledge, and knowledge soundness. Completeness, again, is the property that verification always accepts proofs of true statements computed with a witness. Zero knowledge, again, requires the existence of an efficient simulator that can output accepting proofs without a witness as input. Knowledge soundness similarly requires the existence of an extractor that can output a witness for a statement given non-black-box access to an adversary $\mathcal{A}$ that outputs accepting proofs for that statement.

We are interested in NIZKPoKs for Eq 1 in $\mathcal{G}_\mathsf{E} = (\mathbb{G}_\mathsf{E}, G_\mathsf{E}, q_\mathsf{E})$:

$$\mathcal{R}^{\mathsf{DL}}_{\mathbb{G}} := \{((G, \boldsymbol{H}), \boldsymbol{x}) \mid \mathbf{x} \cdot G = \mathbf{H};\ (G, \boldsymbol{H}) \in \mathbb{G}^*\} \tag{1}$$

When $\mathbf{H}$ is a single group element this relation can be proven via Schnorr:

$\mathsf{Prv}^{\mathsf{H}}_{\mathsf{DL}}(\phi = (G, H), x) \to \pi$**.** Sample $a \leftarrow \mathbb{Z}^*_p$, compute $A := a \cdot G$, $e \leftarrow \mathsf{H}(\phi, A)$, $z := a + e \cdot x$. Output $\pi := (A, e, z)$.

$\mathsf{Ver}^{\mathsf{H}}_{\mathsf{DL}}(\phi = (G, H), \pi = (A, e, z))$**.** Output 1 if $e = \mathsf{H}(\phi, A)$ and $z \cdot G = A + e \cdot H$. Otherwise, output 0.

For $\mathbf{H}$ longer than a single group element, for example for $\mathbf{H} = (H_1, H_2)$ as in our ECDSA signing protocol $\Pi_{\mathtt{ECDSA}}$, the relation can be proven by the above in parallel for each element of $\mathbf{H}$. This can then be compressed via the batching technique of Gennaro et al. [GLSY04].

$\mathsf{Prv}^{\mathsf{H}}_{\mathsf{DL}}(\phi = (G, (H_1, \ldots, H_b)), (x_1, \ldots, x_b)) \to \pi$**.** Sample $a \leftarrow \mathbb{Z}^*_p$, compute $A := a \cdot G$, $e \leftarrow \mathsf{H}(\phi, A)$, $z := a + \cdot \sum_{i \in [b]} x_i \cdot e^i$. Output $\pi := (A, e, z)$.

$\mathsf{Ver}^{\mathsf{H}}_{\mathsf{DL}}(\phi = (G, (H_1, \ldots, H_b)), \pi = (A, e, z))$**.** Output 1 if $e = \mathsf{H}(\phi, A)$ and $z \cdot G = A + \sum_{i \in [b]} (e^i) \cdot H_i$. Otherwise, output 0.

This batching technique reduces the size of proofs by a multiplicative factor b, increasing special soundness of the underlying Σ-protocol from 2 to $b+1$; the knowledge extractor must rewind $\mathcal{A}$ to output $b+1$ transcripts in order to recover a witness. In our case, as our statements are of length 2, this results in a proof size of one group element and two scalar field elements[28] at the cost of our reduction rewinding $\mathcal{A}$ *twice* to recover three accepting transcripts.

[28] Note that we include the challenge in the proof; given access to the input-output table for H this allows our knowledge extractor to recover the statement ϕ efficiently from the pair $(A, e = \mathsf{H}(\phi, A))$.

In our signing protocol $\Pi_{\mathtt{ECDSA}}$ we denote by $(\mathsf{Prv}^{\mathsf{H}}_{\mathsf{DL}}, \mathsf{Ver}^{\mathsf{H}}_{\mathsf{DL}})$ a knowledge sound NIZKPoK for $\mathcal{R}^{\mathsf{DL}}_{\mathbb{G}_{\mathsf{E}}}$, where $\mathsf{H} : \{0,1\}^* \to \mathbb{F}_{q_{\mathsf{E}}}$ is a programmable random oracle. The simulator and extractor are briefly described below:

$\mathcal{S}^{\mathsf{H}}(\phi = (G, \mathbf{H})) \to \pi$**.** Sample $e, z \in \mathbb{F}_{q_{\mathsf{E}}}$ such that the preimage of e under H is undefined. Define $A := z \cdot G_{\mathsf{E}} - \sum_{i \in [b]} (e^i) H_i$. Program $\mathsf{H}(\phi, A) := e$. Output $\pi := (A, e, z)$.

$\mathcal{E}^{\mathsf{H}, \mathcal{A}}(1^\lambda) \to (\phi, w)$**.** The knowledge extractor rewinds $\mathcal{A}$ to retrieve $b+1$ accepting transcripts (A, e_1, z_1), $\ldots, (A, e_{b+1}, z_{b+1})$ and recovers all elements of the witness via Lagrange interpolation.

4 LPN Variants

4.1 Learning Parity with Noise

We consider the so-called primal variant of learning parity with noise (LPN) that we parametrize by the distributions of the linear code and of the noise vector: Let $\mathsf{CodeGen}$ be a probabilistic polynomial time code generation algorithm such that $\mathsf{CodeGen}(m, n, \mathbb{F})$ outputs a matrix $\mathbf{A} \in \mathbb{F}^{m \times n}$, and let $\mathsf{Noise}(t, m, \mathbb{F})$ be an efficiently sampleable noise distribution which outputs t-sparse vectors in $\mathbb{F}^m$. We denote by $\mathcal{V}_k^n(\mathbb{F})$ the set of vectors in $\mathbb{F}^n$ that have exactly k non-zero entries. For a vector $\boldsymbol{v} \in \mathcal{V}_k^n(\mathbb{F})$, we parse $\boldsymbol{v} = (i, v_i)_{i \in [k]}$ which means that on entry i, the non-zero value of $\boldsymbol{v}$ is v_i. We denote by $\mathcal{M}_k^{m \times n}(\mathbb{F})$ the set of $m \times n$ matrices such that each row is in $\mathcal{V}_k^n(\mathbb{F})$.

Definition 7 (Regular Noise Distribution). Let $t, m \in \mathbb{N}$ and such that $t \mid m$. Define $\mathsf{Reg}_t(m, \mathbb{F})$ to be the uniform distribution of vectors $\mathbf{e} = (\mathbf{e}_1, \ldots, \mathbf{e}_t) \in \mathbb{F}^m$ conditioned on each $\mathbf{e}_i$ having a single non-zero entry. We assume that each $\mathbf{e}_i \in \mathbb{F}^{m/t}$ is represented by $(\alpha, \beta) \in [m/t] \times \mathbb{F}$.

Definition 8 (Sparse-Product Noise Distribution). Let $t, m, \beta \in \mathbb{N}$ such that $t \mid m$. Define $D^{\mathsf{sp}}_{\mathsf{Noise},\beta}(t, m, \mathbb{F})$ to be the distribution over $\mathbb{F}^{m \times m} \times \mathbb{F}^m$ outputting matrices $\mathbf{B} := (\boldsymbol{b}_1, \ldots, \boldsymbol{b}_m)^T$ by sampling m row vectors $\boldsymbol{b}_i \xleftarrow{\$} \mathcal{V}_\beta^m(\mathbb{F})$, and a random noise vector $\mathbf{e} \leftarrow \mathsf{Noise}(t, m, \mathbb{F})$ and outputting $(\mathbf{B}, \mathbf{B} \cdot \mathbf{e})$.

Definition 9 (Sparse Random Code). Let $k, n, m \in \mathbb{N}$ such that $3 \leq k \leq n$. Define $\mathsf{SparseCodeGen}(k, m, n, \mathbb{F})$ to output a random matrix $\mathbf{A} \in \mathbb{F}^{m \times n}$ such that each row of $\mathbf{A}$ has only k non-zero elements.

Definition 10 (Regular Random Code). Let $k, n, m \in \mathbb{N}$ such that $3 \leq k \leq n$ and $k \mid n$. Define $\mathsf{RegCodeGen}(k, m, n, \mathbb{F})$ to output a random matrix $\mathbf{A} \in \mathbb{F}^{m \times n}$, such that for every $j \in [m]$ row $\boldsymbol{a}_j = [\boldsymbol{a}_{j,1}, \boldsymbol{a}_{j,2}, \ldots, \boldsymbol{a}_{j,k}]$ is the concatenation of k vectors $\boldsymbol{a}_{j,i} \in \mathbb{F}^{n/k}$ with Hamming weight $\omega(\boldsymbol{a}_{j,i}) = 1$. Define $\mathsf{RegCodeGen}(k, n, \mathbb{F})$ to output a random k-regular vector $\boldsymbol{a}_j \in \mathbb{F}^n$ as above.

Definition 11 (Learning Parity with Noise (LPN) Assumption). Let λ be a security parameter, secret size $n = n(\lambda)$, output size $m = m(\lambda)$, noise sparsity $t = t(\lambda)$. Let $\mathbb{F}$ be a finite field, $\mathsf{CodeGen}$ a code generation algorithm, and Noise a noise distribution. We say that the Learning Parity with Noise (LPN) problem is hard relative to $\mathsf{CodeGen}$ if

$$(\mathbf{A}, \mathbf{A} \cdot \mathbf{s} + \mathbf{e}) \approx_c (\mathbf{A}, \mathbf{y}),$$

where $\mathbf{A} \leftarrow \mathsf{CodeGen}(m, n, \mathbb{F})$, $\mathbf{s} \xleftarrow{\$} \mathbb{F}^n$, $\mathbf{e} \leftarrow \mathsf{Noise}(t, m, \mathbb{F})$, and $\mathbf{y} \xleftarrow{\$} \mathbb{F}^m$. We denote by $\mathsf{LPN}(n, m, \mathsf{Noise}, \mathbb{F})$ the assumption with these parameters.

The following specific variants of the LPN assumption are relevant to our work:

- *Sparse LPN with Regular Noise*, denoted by $\mathsf{SparseLPN}(k, \cdot, \cdot, \mathsf{Reg}_t, \cdot)$, is the LPN assumption instantiated with $\mathsf{SparseCodeGen}(k, \cdot, \cdot, \cdot)$ and Reg_t.
- *Sparse LPN with Sparse-Product Noise* is the assumption that the following two tuples are indistinguishable

  $$(\mathbf{A}, \mathbf{A} \cdot \mathbf{s} + \mathbf{B} \cdot \mathbf{e}, \mathbf{B}) \approx_c (\mathbf{A}, \mathbf{y}, \mathbf{B}),$$

  for $(\mathbf{B}, \mathbf{B} \cdot \mathbf{e}) \xleftarrow{\$} D^{\mathsf{sp}}_{\mathsf{Noise},\beta}$, and $\mathbf{A}, \mathbf{s}$ and $\mathbf{y}$ are defined as before.
- ⋆ *Filtered Sparse LPN with Sparse-Product Noise* allows $\mathcal{A}$ to rejection sample the rows of the code. The assumption is denoted $\mathsf{FilteredSparseLPN}(k, n, m, D^{\mathsf{sp}}_{\mathsf{Reg}_t,\beta}, \mathbb{F})$ when instantiated with $\mathsf{RegCodeGen}(k, \cdot, \cdot, \cdot)$, and $D^{\mathsf{sp}}_{\mathsf{Reg}_t,\beta}$. FLPN appears in Definition 12.
- *Sparse LPN with Regular Noise and Static Leakage* in Definition 13 allows $\mathcal{A}$ to perform a selective-failure attack on the error vector $\mathbf{e}$ prior to seeing LPN samples. The assumption is denoted $\mathsf{SLPNSL}(k, \cdot, \cdot, \mathsf{Reg}_t, \cdot)$ when instantiated with $\mathsf{RegCodeGen}(k, \cdot, \cdot, \cdot)$ and Reg_t.

Security. A common way to analyze the security of a variant of LPN is in the linear test framework as follows. For a code $\mathbf{A} \leftarrow \mathsf{RegCodeGen}(k, m, n, \mathbb{F})$, the goal of an unbounded adversary $\mathcal{A}$ on input $\mathbf{A}$ is to output an attack vector $\boldsymbol{v} \in \mathbb{F}^m$ that induces bias over the distribution $\mathcal{D}_{\mathbf{A}}$ below.

$$\mathcal{D}_{\mathbf{A}} := \left\{ \mathbf{A} \cdot \mathbf{s} + \mathbf{e} \middle| \mathbf{s} \xleftarrow{\$} \mathbb{F}^n, \mathbf{e} \xleftarrow{\$} \mathsf{Reg}_t(m, \mathbb{F}) \right\}$$

Inducing bias over the distribution means that the probability that $\boldsymbol{v}^\top \cdot \boldsymbol{x} = 0$ for $\boldsymbol{x} \leftarrow \mathcal{D}_{\mathbf{A}}$ is noticably different from $\frac{1}{2}$. The possible bias over $\mathcal{D}_{\mathbf{A}}$ decomposes into the possible bias over $\mathbf{A}$ and $\mathbf{e}$; the relevant values to the analysis are below.

- The dual distance of the code matrix $\mathbf{A}$. If $\mathbf{A} \leftarrow \mathsf{RegCodeGen}(\cdots)$ output by the generation algorithm have high dual distance $D := \delta(\mathbf{A})$ with high probability then the possibility of $\mathcal{A}$ to find a vector $\boldsymbol{v} \in \mathbb{F}^m$ with hamming weight $\omega(\boldsymbol{v}) < D$ such that $\boldsymbol{v}^\top \cdot \mathbf{A} = 0$ is low.
- The sparsity of the error vector $\mathbf{e}$. For $\mathbf{e} \leftarrow \mathsf{Reg}_t(m, \mathbb{F})$ the probability that $\boldsymbol{v}^\top \cdot \mathbf{e} = 0$ decreases as the hamming weight $\omega(\boldsymbol{v})$ increases.

The introduction of *filtering* and *static leakage* provide the adversary with two separate advantages. Filtering allows $\mathcal{A}$ to rejection sample the rows of $\mathbf{A}$, first generating a large matrix $\mathbf{A}^{(\mathcal{R})} \in \mathbb{F}^{Q_{\mathsf{FLPN}} \times n}$ and then selecting any submatrix $\mathbf{A} \in \mathbb{F}^{m \times n}$ as the code. Filtering therefore impacts the dual distance estimates of the code $\mathbf{A}$. Static leakage, on the other hand, allows $\mathcal{A}$ to mount a selective failure attack and learn non-zero entries of $\mathbf{e}$ with some probability. If $\mathcal{A}$ learns $I_0 \subset [m]$ such that $\mathbf{e}[i] = 0$ for all $i \in I_0$ then $\mathcal{A}$ can focus its attack against $\mathbf{A}$ on the relevant rows. We expand upon Filtered LPN in Section 4.2, and SLPN with static leakage in Section 4.3.

4.2 Filtered LPN

We present the definition for Filtered Sparse Learning Parity with Sparse-Product Noise in Definition 12, which was discussed in our Technical Overview (see **LPN variants**).

Definition 12 (Filtered Sparse Learning Parity with Sparse-Product Noise). Let $\lambda \in \mathbb{N}$ be a security parameter, n be the secret length, k be the code regularity such that $k|n$, m be the output length, $t = t(\lambda)$ be the noise regularity such that $t|m$, $\beta = \beta(\lambda)$ be the product matrix sparsity such that $\beta|m$, and $\mathbb{F}$ be a finite field. Consider the game $\mathsf{Exp}^{\mathsf{FLPN}}_{\mathcal{A},b}$ in Figure 4, defined relative to algorithm $\mathsf{RegCodeGen}$, and distribution $D^{\mathsf{sp}}_{\mathsf{Reg}_t,\beta}$. The FLPN assumption holds if for all PPT adversaries $\mathcal{A}$ making $Q_{\mathsf{FLPN}} = Q_{\mathsf{FLPN}}(\lambda)$ queries to oracle $\mathcal{O}^{\mathsf{FLPN}}_b$:

$$\left| \Pr\left[\mathsf{Exp}^{\mathsf{FLPN}}_{\mathcal{A},0}\left(1^\lambda, k, n, m, D^{\mathsf{sp}}_{\mathsf{Reg}_t,\beta}, \mathbb{F}\right) = 1 \right] - \Pr\left[\mathsf{Exp}^{\mathsf{FLPN}}_{\mathcal{A},1}\left(1^\lambda, k, n, m, D^{\mathsf{sp}}_{\mathsf{Reg}_t,\beta}, \mathbb{F}\right) = 1 \right] \right| = \mathsf{negl}(\lambda).$$

If the FLPN assumption holds for the parameters then we say that the FLPN problem is hard relative to the parameter set, denoted by $\mathsf{FilteredSparseLPN}(k, n, m, D^{\mathsf{sp}}_{\mathsf{Reg}_t,\beta}, \mathbb{F})$.

Observe that while the game $\mathsf{Exp}^{\mathsf{FLPN}}_{\mathcal{A},b}$ in Figure 4 is presented in terms of a sparse-product noise distribution (Definition 8), a variant for (non-product) sparse noise Noise can be inferred by eliminating the vector $\boldsymbol{b}_i$, i.e., on a query $i \in [m]$ the oracle $\mathcal{O}^{\mathsf{FLPN}}(i) \mapsto \boldsymbol{a}_i \cdot \boldsymbol{s} + e_i$.

Parameter Adjustment for High Dual Distance. As discussed, the advantage of an adversary $\mathcal{A}$ in FLPN, in contrast with standard LPN, is that $\mathcal{A}$ can sample Q_{FLPN} many independent rows $\{(i, \boldsymbol{a}_i)\}_{i \in [Q_{\mathsf{FLPN}}]}$, thereby allowing $\mathcal{A}$ to rejection sample the code matrix to select whichever subset of the N_{FLPN} rows are most convenient in the attack of its choosing.

Consider the matrix $\mathbf{A}^{(\mathcal{R})}$ consisting of the rows $\mathbf{A}^{(\mathcal{R})}[i][:] := \boldsymbol{a}_i$, where $(i, \boldsymbol{a}_i, \cdot) \in \mathcal{R}$ is an element of the set of queries and responses maintained by oracle $\mathcal{O}^{\mathsf{mtx}}$ in $\mathsf{Exp}^{\mathsf{FLPN}}_{\mathcal{A},b}$:[29]

$$\mathcal{R} := \left\{ (i, \boldsymbol{a}_i, \boldsymbol{b}_i) \middle| \begin{array}{r} \boldsymbol{a}_i \leftarrow \mathsf{RegCodeGen}(k, n, \mathbb{F}) \\ \boldsymbol{b}_i \leftarrow D^{\mathsf{sp}}_{\mathsf{Reg}_t,\beta} \end{array} \right\}. \tag{2}$$

[29] Note that we do not restrict $\mathcal{A}$ to queries $i \in [Q_{\mathsf{FLPN}}]$, but the resulting security analysis is easier to express when we index the queries as elements of $i \in [Q_{\mathsf{FLPN}}]$.

To argue security of FLPN it is sufficient to show that with high probability $\mathbf{A}^{(\mathcal{R})}$ has high dual distance $D = \delta\left(\mathbf{A}^{(\mathcal{R})}\right)$, meaning that any vector $\boldsymbol{v}$ in the left kernel of $\mathbf{A}^{(\mathcal{R})}$ must have hamming weight $\omega(\boldsymbol{v}) \geq \delta\left(\mathbf{A}^{(\mathcal{R})}\right)$ [ADINZ17; ABCM25].

An analysis of the dual distance of k-regular sparse codes output by $\mathsf{RegCodeGen}(k, m, n, \mathbb{F})$ can be found in [BCMRS25]. Therein, for fixed statistical security parameter ρ the authors present a recursive formula to compute a tight lower bound D on the dual distance $\delta(\mathbf{A})$ of the matrices output by $\mathsf{RegCodeGen}(k, n, m, \mathbb{F}_2)$ that holds with high probability.

$$\Pr\left[\delta(\mathbf{A}) \leq D \mid \mathbf{A} \leftarrow \mathsf{RegCodeGen}(k, n, m, \mathbb{F}_2)\right] \leq 2^{-\rho}.$$

We recall the recursive formula in Eq 3.

$$D := \arg\max_{d>0} \left(\sum_{w=1}^{\lfloor (d-1)/2 \rfloor} \binom{Q_{\mathsf{FLPN}}}{2w} \cdot \left(p_{w\cdot k,0}\right)^k \leq 2^{-\rho} \right) \tag{3}$$

Towards intuition, the summed turm recursively computes the probability that the xor of $2w$ random k-regular vectors is $u = 0$ as below, for $N = \frac{n}{k}$ and $c = w \cdot k$:

$$p_{c,u} := \frac{p_{c-1,u}}{N} + \frac{1}{N^2} \cdot \begin{pmatrix} (N-2u+2)\cdot(N-2\cdot u+1)\cdot p_{c-1,u-1} \\ +4u\,(N-2u)\cdot p_{c-1,u} \\ +(2u+2)\cdot(2u+1)\cdot p_{c-1,u+1} \end{pmatrix},$$

$$\text{where} \quad p_{1,0} = \frac{1}{N} \qquad p_{c,c} = \prod_{i\in[2c-1]} \frac{N-i}{N}, \qquad p_{c,c'} = 0 \text{ for } c' > c.$$

Most importantly, the only dependence of the bound D on the number of rows m is the binomial coefficient. As filtering is a local computation performed by a PPT adversary $\mathcal{A}$ we set the number of rows to be maximal $Q_{\mathsf{FLPN}} = 2^\lambda$ in Eq 3, and use the formula to inform our choice of parameters for FLPN given fixed security parameters λ and ρ. We adjust the sparsity k such that the term $\left(p_{w\cdot k,0}\right)^k$ compensates for the growth of the binomial coefficient attributable to the increase $m \to Q_{\mathsf{FLPN}} = 2^\lambda$.

The resulting recommended parameters are reported in Section 13, Table 5. We report values of k that result in a non-trivial lower bound $D > 0$ for $\rho \in \{40, 64, 80\}$ and $m \in \{2^{30}, 2^{35}, 2^{40}\}$. This corresponds to the parameters for sparse LPN *without filtering* for fixed $\rho \in \{40, 64\}$ in [BCMRS25], reporting values of (n, k) such that $D > 0$.

4.3 Sparse Learning Parity with Regular Noise and Static Leakage

We recall Sparse Learning Parity with Regular Noise (SLPN) in Definition 13.

Definition 13 (Sparse Learning Parity with Regular Noise (SLPN) Assumption with Static Leakage). Let $\lambda \in \mathbb{N}$ be a security parameter, n be the secret length, k be the code regularity such that $k \mid n$, m be the output length, t be the noise regularity such that $t \mid m$, and $\mathbb{F}$ be a finite field. Consider the Sparse Learning Parity with Noise game $\mathsf{Exp}^{\mathsf{LPN}}_{\mathcal{A},b}$ in Figure 4, defined relative to algorithm $\mathsf{RegCodeGen}$ and distribution Reg. The SLPN assumption holds if for all PPT adversaries $\mathcal{A}$:

$$\left| \begin{array}{l} \Pr\left[\mathsf{Exp}^{\mathsf{SLPN}}_{\mathcal{A},0}(1^\lambda, k, n, m, \mathsf{Reg}_t, \mathbb{F}) = 1\right] - \\ \Pr\left[\mathsf{Exp}^{\mathsf{SLPN}}_{\mathcal{A},1}(1^\lambda, k, n, m, \mathsf{Reg}_t, \mathbb{F}) = 1\right] \end{array} \right| = \mathsf{negl}(\lambda).$$

If the SLPN assumption holds for the parameters then we say that the SLPN problem is hard relative to this parameter set. If the SLPN assumption holds given that $\mathcal{A}$ has access to the static leakage oracle $\mathcal{O}^{\mathsf{lk}}$ then we say that the SLPN with Static Leakage problem is hard relative to the parameter set. When the assumption holds when instantiated with $\mathsf{RegCodeGen}(k, \cdot, \cdot, \cdot)$ and Reg_t with static leakage, this is denoted by $\mathsf{SLPNSL}(k, \cdot, \cdot, \mathsf{Reg}_t, \cdot)$.

SLPN with static leakage allows $\mathcal{A}$ to mount a selective failure attack prior to obtaining any LPN samples. The selective failure attack can be considered a pre-game to LPN, and has been analyzed in prior work [Boy+19a; WYKW21; BBMS22]. A variant of LPN, in this case SLPN, with static leakage allows $\mathcal{A}$ to query the oracle $\mathcal{O}^{\mathsf{lk}}$ (Figure 4) prior to seeing any samples. In the selective failure attack $\mathcal{A}$ submits a tuple of subsets of indices $((j, I_j))$, where for each $j \in [t]$ the subset $I_j \subseteq [(j-1) \cdot \frac{m}{t} + 1, j \cdot \frac{m}{t}]$ is essentially the adversary's guess at some non-zero indices in each error block $\mathbf{e}_j$.

In the case that $\mathcal{A}$ guesses incorrectly, meaning that $i_j \notin I_j$ for $i_j \in [(j-1) \cdot \frac{m}{t} + 1, j \cdot \frac{m}{t}]$ such that $\mathbf{e}[i_j] \neq 0$ for any $j \in [t]$, the attack fails and the LPN game aborts. Otherwise, $\mathcal{A}$ can proceed having learned $I_0 := \left\{ i \in [m] \setminus \bigcup_{j \in [t]} I_j \right\}$ such that $\mathbf{e}[i] = 0$ for all $i \in I_0$.

As an example, and to expand upon Section 4.2 in a manner relevant to our security analysis, in the *filtered LPN with static leakage* game $\mathcal{A}$ first engages with the oracle $\mathcal{O}^{\mathsf{lk}}$ for the selective failure attack, then proceeds to interact with $\mathcal{O}^{\mathsf{FLPN}}$. We note that as the error vector is independent of $\mathbf{A}$, $\mathcal{A}$ is granted access to $\mathcal{O}^{\mathsf{mtx}}$ at the start of this game.

Security of LPN with static leakage can be reduced to LPN without static leakage, or analyzed in the linear test framework to determine the possibility that an unbounded adversary can find an attack vector $\boldsymbol{v}$ that biases samples from $\mathcal{D}_{\mathbf{A}}$ towards 0 as discussed in Section 4. When taking into account the zero coordinates that $\mathcal{A}$ learned about $\mathbf{e}$ conditioned on the selective failure attack not failing, the loose reduction in [Boy+20c] implies a dramatic increase to the LPN parameters but does not reflect an actual security loss.

Since the introduction of LPN with static leakage it has been conjectured that *LPN with static leakage is nearly as secure* as LPN without static leakage [Boy+20c]. At a high level, this conjecture can be supported by the observation that the adversary's selective failure attack via $\mathcal{O}^{\mathsf{lk}}$ happens before a query to $\mathcal{O}^{\mathsf{LLPN}}$ and therefore the leakage is bounded to a single bit of information on the secret—$\mathcal{A}$ simply learns that the guess is or is not correct [AS22].

Fig. 4: LPN Games and the Static Leakage Oracle

Below are the security games for Filtered LPN (FLPN, Definition 12) and LPN with Regular Noise. Both are presented with the options to include static leakage by including the highlighted text. The games are defined with respect to:

- $\mathsf{RegCodeGen}(k, m, n, \mathbb{F})$ is an algorithm with k-regular output in $\mathbb{F}^{m \times n}$
- $\mathsf{RegCodeGen}(k, n, \mathbb{F})$ is an algorithm with k-regular output in $\mathbb{F}^n$,
- $D^{\mathsf{sp}}_{\mathsf{Reg}_t,\beta}$ is a distribution over $\mathbb{F}^{m \times m} \times \mathbb{F}^m$ consisting of:
  - $\mathcal{V}^m_\beta(\mathbb{F})$, a distribution over $\mathbb{F}^m$ for β-sparse product vectors, and
  - $\mathsf{Reg}_t(m, \mathbb{F})$, a distribution over $\mathbb{F}^m$ for t-regular error vectors.

Filtered Sparse Learning Parity with SP-Noise and Static Leakage

FSLPN Game with oracles $\mathcal{O}^{\mathsf{mtx}}(\cdot), \mathcal{O}^{\mathsf{FLPN}}_b(\cdot)$ and $\mathcal{O}^{\mathsf{lk}}(\cdot)$:

$\underline{\mathsf{Exp}^{\mathsf{FLPN}}_{\mathcal{A},b}\left(1^\lambda, k, n, m, D^{\mathsf{sp}}_{\mathsf{Reg}_t,\beta}, \mathbb{F}\right)}$:

$\mathsf{allow\text{-}leak} := \top$
$\mathcal{R}, \mathcal{L} := \emptyset$
$\mathbf{s} \xleftarrow{\$} \mathbb{F}^n$
$\mathbf{e} \leftarrow \mathsf{Reg}_t(m, \mathbb{F})$
$b' \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{mtx}}, \mathcal{O}^{\mathsf{FLPN}}_b, \mathcal{O}^{\mathsf{lk}}}(1^\lambda)$
Output b'

$\underline{\mathcal{O}^{\mathsf{mtx}}(i)}$:

If $(i, \boldsymbol{a}, \boldsymbol{b}) \in \mathcal{R}$**:**
  Return $(i, \boldsymbol{a}, \boldsymbol{b})$
$\boldsymbol{a} \leftarrow \mathsf{RegCodeGen}(k, n, \mathbb{F})$
$\boldsymbol{b} \leftarrow \mathcal{V}^m_\beta(\mathbb{F})$
$\mathcal{R} := \mathcal{R} \cup \{(i, \boldsymbol{a}, \boldsymbol{b})\}$
Return $(\boldsymbol{a}, \boldsymbol{b})$

$\underline{\mathcal{O}^{\mathsf{FLPN}}_b(i)}$:

$\mathsf{allow\text{-}leak} := \bot$
If $(i, \boldsymbol{a}, \boldsymbol{b}) \notin \mathcal{R}$**:**
  Return $\bot$
If $(i, u) \in \mathcal{L}$**:**
  Return u
$u_0 \xleftarrow{\$} \mathbb{F}$
$u_1 := \boldsymbol{a}^\top \cdot \mathbf{s} + \boldsymbol{b}^\top \cdot \mathbf{e}$
$\mathcal{L} := \mathcal{L} \cup \{(i, u_b)\}$
Return u_b

Sparse Learning Parity with Regular Noise and Static Leakage

LPN Game with oracles $\mathcal{O}^{\mathsf{lpn}}_b(\cdot)$ and $\mathcal{O}^{\mathsf{lk}}(\cdot)$:

$\underline{\mathsf{Exp}^{\mathsf{SLPN}}_{\mathcal{A},b}\left(1^\lambda, k, n, m, \mathsf{Reg}_t, \mathbb{F}\right)}$:

$\mathsf{allow\text{-}leak} := \top$
$\mathcal{L} := \emptyset$
$\mathbf{A} \leftarrow \mathsf{RegCodeGen}(k, m, n, \mathbb{F})$
$\mathbf{s} \xleftarrow{\$} \mathbb{F}^n$
$\mathbf{e} \leftarrow \mathsf{Reg}_t(m, \mathbb{F})$
$(\mathbf{e}_1, \ldots, \mathbf{e}_t) := \mathbf{e}$
Output $b' \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{lpn}}_b, \mathcal{O}^{\mathsf{lk}}}\left(1^\lambda\right)$

$\underline{\mathcal{O}^{\mathsf{lpn}}_b(i \in [0, m))}$:

$\mathsf{allow\text{-}leak} := \bot$
If $(i, \boldsymbol{a}_i, u) \in \mathcal{L}$**:**
  Return $(i, \boldsymbol{a}_i, u)$
$u_0 \xleftarrow{\$} \mathbb{F}$
$u_1 := \boldsymbol{a}_i^\top \cdot \mathbf{s} + e_i$
$\mathcal{L} := \mathcal{L} \cup \{(i, \boldsymbol{a}_i, u_b)\}$
Return $(\boldsymbol{a}_i, u_b)$

$\underline{\mathcal{O}^{\mathsf{lk}}\left(((j, I_j))_{j \in [t]}\right)}$:

If $\mathsf{allow\text{-}leak} = \bot$**:**
  Return $\bot$.
For $j \in [t]$**:**
  For i_j**:** $\mathbf{e}_j[i_j] \neq 0$.
  If $i_j \notin I_j$**: abort.**
$\mathsf{allow\text{-}leak} := \bot$.

5 Puncturable PRFs

In this section we present a new definition of PPRFs with reusable setup and give instantiations.

5.1 Definition

In the following, we present a definition of puncturable pseudorandom functions (PPRFs). Our definition differs from traditional definitions (e.g., those introduced in [BW13; BGI14; KPTZ13]) in that it incorporates additional details tailored to the distributed setting that will be discussed in Section 10. In particular, we consider a reusable setup, where a pair of $(\mathsf{msk}, \mathsf{mpk})$ can be used to generate multiple secret-key and punctured-key pairs. This property is inspired by the recent work of [CHHK25]. In Section 5.2, we show instantiations for our definition.

Definition 14 (Puncturable Pseudorandom Functions with Reusable Setup). Let $\lambda \in \mathbb{N}$. Let $\ell = \ell(\lambda)$, and $N_i = N_i(\lambda) \in \mathbb{N}$ for $i \in [\ell]$. Let $\{\mathcal{Y}_i\}_{i\in[\ell]}$ be finite sets. Define $\mathcal{X} := \bigcup_{i\in[\ell]}[0, N_i)$, and $\mathcal{Y} := \bigcup_{i\in[\ell]} \mathcal{Y}_i$. A puncturable pseudorandom function (PPRF) with reusable setup defined by $\mathsf{interface} = \{\lambda, \{N_i\}_{i\in[\ell]}, \{\mathcal{Y}_i\}_{i\in[\ell]}\}$ is a tuple of polynomial time algorithms as follows:

$\mathsf{F.Setup}(1^\lambda) \to \mathsf{pp}$: The deterministic setup algorithm takes as input a security parameter $\lambda \in \mathbb{N}$ and outputs public parameters pp. We assume that all the following algorithms have access to pp.

$\mathsf{F.GenMSK}(\mathsf{pp}) \to \mathsf{msk}$: The probabilistic master secret-key generation algorithm samples a master secret key msk.

$\mathsf{F.ComputeMPK}(\mathsf{msk}) \to \mathsf{mpk}$: The deterministic master public-key generation algorithm computes the master public key mpk from the master secret key msk.

$\mathsf{F.KeyGen}(N, \mathsf{msk}) \to \mathsf{sk}$: The probabilistic key generation algorithm takes as input a number $N \in \mathbb{N}$ and a master secret key msk, and outputs a secret key sk or $\perp$.We assume that sk contains N.

$\mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, x) \to y$: The deterministic evaluation algorithm takes as input a master secret key msk, a secret key sk, and an input $x \in \mathcal{X}$, and outputs a value $y \in \mathcal{Y}$ or $\perp$.

$\mathsf{F.Puncture}(\mathsf{msk}, \mathsf{sk}, z) \to \mathsf{psk}$: The deterministic puncturing algorithm takes as input a master secret key msk, a secret key sk (containing a domain length $N \in \mathbb{N}$), and an input $z \in \mathcal{X}$, and outpts a punctured secret key psk or $\perp$. We assume that psk contains N.

$\mathsf{F.PEval}(\mathsf{mpk}, \mathsf{psk}, x) \to y$: The punctured evaluation algorithm takes as input a master public key mpk, a punctured secret key psk and an input $x \in \mathcal{X}$. It outputs a value $y \in \mathcal{Y}$ or $\perp$.

We require a PPRF F with reusable setup defined by any $\mathsf{interface} = \{\lambda, \{N_i\}_{i\in[\ell]}, \{\mathcal{Y}_i\}_{i\in[\ell]}\}$ to satisfy the following properties:

- **Domain Validation:** There exists a deterministic public algorithm
  - $\mathsf{CheckDomain}(N) \to \top/\perp$

  that on input $N \in \mathbb{N}$, outputs $\top$ *iff* there exists $i \in [\ell]$ such that $N \le N_i$.
  All algorithms of F that (explicitly or implicitly) take $N \in \mathbb{N}$ as input, can run $\mathsf{CheckDomain}(N)$ as a subroutine and output $\perp$ if this algorithm returns $\perp$.

- **Key Validation:** There exist deterministic
  - $\mathsf{CheckMSK}(\mathsf{msk}) \to \top/\perp$
  - $\mathsf{CheckSK}(\mathsf{msk}, N\mathsf{sk}) \to \top/\perp$

  such that

$$\Pr\left[\mathsf{F.CheckMSK}(\mathsf{msk}) = \top\right] = 1,$$

  for all $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$ and $\mathsf{msk} \in \mathrm{supp}(\mathsf{GenMSK}(\mathsf{pp}))$. And

$$\Pr\left[\mathsf{F.CheckSK}(\mathsf{msk}, N, \mathsf{sk}) = \top\right] = 1,$$

  for all $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$, and all $(\mathsf{msk}, \mathsf{sk} \neq \perp)$ pairs satisfying $\mathsf{sk} \in \mathrm{supp}(\mathsf{KeyGen}(N, \mathsf{msk}))$.

– **Perfect Correctness:** For any $N \in \mathbb{N}$ where $\mathsf{CheckDomain}(N) = \top$, and all $z \in \mathcal{X}$

$$\Pr\left[\begin{array}{l}\mathsf{F.PEval}(\mathsf{mpk}, \mathsf{psk}, x) \\ \qquad = \mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, x)\end{array} : \begin{array}{l}\mathsf{pp} \leftarrow \mathsf{F.Setup}(1^\lambda) \\ \mathsf{msk} \xleftarrow{\$} \mathsf{F.GenMSK}(\mathsf{pp}) \\ \mathsf{mpk} \leftarrow \mathsf{F.ComputeMPK}(\mathsf{msk}) \\ \mathsf{sk} \xleftarrow{\$} \mathsf{F.KeyGen}(N, \mathsf{msk}) \\ \mathsf{psk} \leftarrow \mathsf{F.Puncture}(\mathsf{msk}, \mathsf{sk}, z)\end{array}\right] = 1,$$

for all $x \in \mathcal{X}$ satisfying $x \neq z$. We also require the above to hold for all msk and sk satisfying $\mathsf{CheckMSK}(\mathsf{msk}) = \top$ and $\mathsf{CheckSK}(\mathsf{msk}, N, \mathsf{sk}) = \top$.

– **Pseudorandomness:** For any probabilistic polynomial-time adversary $\mathcal{A}$,

$$\Pr[\mathsf{Exp}^{\mathsf{pprf\text{-}pr}}_{\mathcal{A}}(1^\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

where $\mathsf{Exp}^{\mathsf{pprf\text{-}pr}}_{\mathcal{A}}(1^\lambda)$ is defined as follows:

- **Setup:** The challenger first samples a random bit $b \xleftarrow{\$} \{0,1\}$. It then computes $\mathsf{pp} \leftarrow \mathsf{F.Setup}(1^\lambda)$, and $\mathsf{msk} \xleftarrow{\$} \mathsf{F.GenMSK}(\mathsf{pp})$, and $\mathsf{mpk} \leftarrow \mathsf{F.ComputeMPK}(\mathsf{msk})$. It also initiates an array $SK = \emptyset$ and a counter $\mathsf{cnt} = 0$. It then sends $(\mathsf{pp}, \mathsf{mpk})$ to the adversary. This phase is done only once at the beginning of the game.
- **Challenge Evaluation Queries:** $\mathcal{A}$ can send polynomially many challenges as follows:
  1. The adversary $\mathcal{A}$ sends an arbitrary triple (i, N, x) to the challenger, where $i \in \mathbb{N}$ or $i = \mathsf{null}$ is an index, $N \in \mathbb{N}$ is a domain length, and $x \in \mathcal{X}$ is an input.
  2. The challenger returns $\perp$ (without aborting) and waits for a new challenge query, if $i \neq \mathsf{null} \wedge SK[i] = \mathsf{null}$ or $\perp \leftarrow \mathsf{F.CheckDomain}(N)$.
  3. If $i = \mathsf{null}$, the challenger samples $\mathsf{sk} \xleftarrow{\$} \mathsf{F.KeyGen}(N, \mathsf{msk})$, updates $\mathsf{cnt} \leftarrow \mathsf{cnt} + 1$, and sets $SK[\mathsf{cnt}] \leftarrow \mathsf{sk}$. Otherwise, it sets $\mathsf{sk} \leftarrow SK[i]$.
  4. If $b = 0$, it computes $y \leftarrow \mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, x)$, and if $b = 1$, it samples a random value $y \xleftarrow{\$} \mathcal{Y}$. It finally returns y to $\mathcal{A}$.
- **Guess:** Finally $\mathcal{A}$ outputs a bit b'.

The experiment outputs 1 if $b' = b$, and 0 otherwise.

– **Selective Security:** For any probabilistic polynomial-time adversary $\mathcal{A}$,

$$\Pr[\mathsf{Exp}^{\mathsf{pprf\text{-}sec}}_{\mathcal{A}}(1^\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

where $\mathsf{Exp}^{\mathsf{pprf\text{-}sec}}_{\mathcal{A}}(1^\lambda)$ is defined as follows:

- **Setup:** The challenger first samples a random bit $b \xleftarrow{\$} \{0,1\}$. It then computes $\mathsf{pp} \leftarrow \mathsf{F.Setup}(1^\lambda)$, and $\mathsf{msk} \xleftarrow{\$} \mathsf{F.GenMSK}(\mathsf{pp})$, and $\mathsf{mpk} \leftarrow \mathsf{F.ComputeMPK}(\mathsf{msk})$. It then sends $(\mathsf{pp}, \mathsf{mpk})$ to the adversary. This phase is done only once at the beginning of the game.
- **Challenge Puncturing Queries:** $\mathcal{A}$ can send polynomially many challenges as follows:
  1. The adversary $\mathcal{A}$ sends a challenge pair (N, z) to the challenger, where $N \in \mathbb{N}$ is a domain length, and $z \in \mathcal{X}$ is a puncturing point.
  2. The challenger returns $\perp$ (without aborting) and waits for a new challenge query, if $\perp \leftarrow \mathsf{F.CheckDomain}(N)$. Otherwise, it continues as follows.
  3. The challenger samples $\mathsf{sk} \xleftarrow{\$} \mathsf{F.KeyGen}(N, \mathsf{msk})$ and computes $\mathsf{psk} \leftarrow \mathsf{F.Puncture}(\mathsf{msk}, \mathsf{sk}, z)$. If $b = 0$, it computes $y \leftarrow \mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, z)$, and if $b = 1$, it samples a random value $y \xleftarrow{\$} \mathcal{Y}$. It finally returns (psk, y) to $\mathcal{A}$.
- **Guess:** Finally $\mathcal{A}$ outputs a bit b'.

The experiment outputs 1 if $b' = b$, and 0 otherwise.

5.2 PPRF Constructions

In this section, we re-introduce the following three PPRF constructions in terms of our new definition of PPRFs with reusable setup.

GGM PPRF The classical PPRF construction based on the GGM construction for PRFs [BW13; BGI14; KPTZ13; SW14]. The PRF key is a seed of size λ and a length-doubling PRG is used to recursively expand it in form of a binary tree. To evaluate the PRF at some index x, we start at the root and at each level go left or right depending on the binary decomposition of x.
A key punctured at some point z is the co-path to the leaf z. Hence, a punctured key is of size $\lambda \cdot \log N$ when N is the domain size. We recall this construction in Figure 5.

CHHK25 This PPRF is introduced by [CHHK25]. It works over a cyclic group $\mathbb{G}$ of prime order p and essentially works as follows: a secret key is simply composed of a random element $a \xleftarrow{\$} \mathbb{Z}_p^*$, and the evaluation of the PRF on input x is computed as $\mathsf{H}(a \cdot r^x \cdot H)$, where H is a correlation-robust hash function, $r \xleftarrow{\$} \mathbb{Z}_p^*$, and H is a random group element. A punctured key that can evaluate the output of the PRF on all inputs except for z consists of $(\alpha = a \cdot r^z, (r^s \cdot H)_{s \neq 0}, z)$. This key would allow one to compute the function on any input $x \neq z$ by computing $\mathsf{H}(\alpha \cdot H_{s_x})$, where $s_x = x - z$.
As observed by [CHHK25], one could consider the elements $(r^s \cdot H)_{s \neq 0}$ as a *reusable* master public key and treat (r, H) as the corresponding master secret key. In this way, a new PPRF is instantiated each time by sampling independent random secret keys $a \xleftarrow{\$} \mathbb{Z}_p^*$. We recall this construction in Figure 6.

k-ary Naor-Reingold (k-NR) The work of [BCMPR24] introduced a PPRF in NC^1 from a modification of the Naor-Reingold PRF construction. Basically, their PPRF was a special case of a more general *constrained PRF* for inner-product membership (IPM) predicates. [30] An IPM predicate is defined by a vector z and a set S and is satisfied on any input vectors x *iff* $\langle x, z \rangle \in S$. The authors of [BCMPR24] observe that when the inputs are encoded in the binary basis $\{+1, -1\}$, then the inner-product membership predicate contains puncturing. More precisely, for such vectors it holds that $\langle x, z \rangle = n$ *iff* $x = z$. Therefore, defining $S := \{-n, -n+2, \ldots, n-2\}$ (excludes $\{n\}$) and generating a constrained key for the IPM predicate defined by (z, S), allows one to compute the PRF on all inputs except for z. Here, we introduce a more general case: instead of decomposing the input into bits, we use a more general k-ary decomposition for $k \in \mathbb{N}$ and encode an input $x \in [0, N)$ as $\boldsymbol{x} = \mathsf{enc}_{\mathsf{in}}(x) \in [0, k)^n$ where $n := \lceil \log_k(N) \rceil$ such that
such that

$$x = \sum_{i=1}^{n} k^{i-1} \cdot x_i.$$

To phrase our puncturing constraint as an inner-product membership constraint, we use different encodings for the PPRF inputs x and the punctured point z:

$$\mathsf{enc}_{\mathsf{in}}^1(x) = (\|\mathsf{enc}_{\mathsf{in}}(x)\|^2, 1, \mathsf{enc}_{\mathsf{in}}(x)),$$
$$\mathsf{enc}_{\mathsf{in}}^2(z) = (1, \|\mathsf{enc}_{\mathsf{in}}(z)\|^2, -2\mathsf{enc}_{\mathsf{in}}(z)).$$

Observe that

$$\begin{aligned}\langle \mathsf{enc}_{\mathsf{in}}^1(x), \mathsf{enc}_{\mathsf{in}}^2(z) \rangle &= \|\mathsf{enc}_{\mathsf{in}}(x)\|^2 + \|\mathsf{enc}_{\mathsf{in}}(z)\|^2 - 2 \cdot \langle \mathsf{enc}_{\mathsf{in}}(x), \mathsf{enc}_{\mathsf{in}}(z) \rangle \\ &= \|\mathsf{enc}_{\mathsf{in}}(x) - \mathsf{enc}_{\mathsf{in}}(z)\|^2\end{aligned}$$

and note that $\|\mathsf{enc}_{\mathsf{in}}(x) - \mathsf{enc}_{\mathsf{in}}(z)\|^2 = 0 \iff x = z$. Now we define our constraint set as

$$S := \{\|\mathsf{enc}_{\mathsf{in}}(x) - \mathsf{enc}_{\mathsf{in}}(z)\|^2 \mid x, z \in [0, N), x \neq z\} \subseteq [1, n \cdot (k-1)^2].$$

On the reusable setup. Our adaptation of these constructions differ from their original schemes in the following minor point: we extend the KeyGen algorithm to support multiple domain lengths. On input a domain length N, the algorithm outputs $\mathsf{sk} = (N, \overline{\mathsf{sk}})$ where $\overline{\mathsf{sk}}$ is a freshly sampled secret key of the original scheme. Essentially, this instantiates a collection of *independent* PPRFs. Since each secret key $\overline{\mathsf{sk}}$ is independent, all properties of the PRF defined by $\overline{\mathsf{sk}}$ on the corresponding domain $[0, N)$ is independent on the PRF defined by $\overline{\mathsf{sk}'}$ on another domain $[0, N')$. Therefore, the proof of all the following theorems proceed in the same way as the proof of the correctness, pseudorandomness and security of the original constructions by applying a standard hybrid argument over all generated independent secret keys.

[30] A constrained PRF is a PRF that allows generation of a *constrained key* that can compute the function on only a specific subset of inputs. The constraint subset is usually described as the set of inputs that satisfy a specific predicate.

Fig. 5: GGM PPRF(GGM-PPRF)

The construction is defined by $\mathsf{interface} = \{\lambda, \{N_i\}_{i\in[\ell]}, \mathbb{F}\}$, where

- $\lambda \in \mathbb{N}$ is a security parameter.
- $N_i = N_i(\lambda) \in \mathbb{N}$ denotes a domain length for $i \in [\ell]$.
- $\mathbb{F} = \mathbb{F}(\lambda)$ is a field that is the range of the PPRF. For all $a \in \mathbb{F}$, it should hold that $|a| = \lambda$, where $|\cdot|$ denotes the bit size.

Given $\mathsf{interface}$, let spec contain the following for each $N \in \{N_i\}_{i\in[\ell]}$:

- $\mathcal{X} := \bigcup_{i\in[\ell]}[0, N_i)$.
- $h := \lceil \log(N) \rceil$.
- $\mathsf{bin} : [0, N) \mapsto \{0,1\}^h$ is the binary composition function.
- Let $\mathsf{PRG} : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$, and $\mathsf{PRG}' : \{0,1\}^\lambda \to \mathbb{F}^2$ are pseudorandom generators as per Definition 3.
- For any input x, let $\mathsf{PRG}_0(x)$ and $\mathsf{PRG}_1(x)$ denote the two half of $\mathsf{PRG}(x)$, i.e., $\mathsf{PRG}(x) = \mathsf{PRG}_0(x)||\mathsf{PRG}_1(x)$. Similarly, $\mathsf{PRG}'_0(x)$ and $\mathsf{PRG}'_1(x)$ denote the two half of $\mathsf{PRG}'(x)$, i.e., $\mathsf{PRG}'(x) = (\mathsf{PRG}'_0(x), \mathsf{PRG}'_1(x))$.

$\underline{\mathsf{F.KeyGen}(1^\lambda)}$:

- Sample $s \xleftarrow{\$} \{0,1\}^\lambda$.
- Output $\mathsf{sk} := s$.

$\underline{\mathsf{F.Puncture}(\mathsf{sk}, z \in [0, N))}$:

- Parse $\mathsf{sk} = s$.
- Compute $z_1||\cdots||z_h \leftarrow \mathsf{bin}(z)$.
  // $z_i \in \{0,1\}$.
- For $i \in [h]$, define $\overline{z}_i := 1 \oplus z_i$.
- Set $K_1 := \mathsf{PRG}_{\overline{z}_1}(s)$.
- For $i \in [h-1]$:
  - $K_i := \mathsf{PRG}_{\overline{z}_i}(\cdots \mathsf{PRG}_{z_1}(s) \cdots)$
- Set $K_h :=$ $\mathsf{PRG}'_{\overline{z}_h}(\cdots \mathsf{PRG}_{z_1}(s) \cdots)$.
- Output $\mathsf{psk} := ((K_1, \ldots, K_h), z)$.

$\underline{\mathsf{F.Eval}(\mathsf{sk}, x \in [0, N))}$:

- Parse $\mathsf{sk} = s$.
- Compute $x_1||\cdots||x_h \leftarrow \mathsf{bin}(x)$.
  // $x_i \in \{0,1\}$.
- Output $y :=$ $\mathsf{PRG}'_{x_h}(\cdots \mathsf{PRG}_{x_1}(s) \cdots)$

$\underline{\mathsf{F.PEval}(\mathsf{psk}, x \in [0, N))}$:

- Parse $\mathsf{psk} = ((K_1, \ldots, K_h), z)$.
- Compute $z_1||\cdots||z_h \leftarrow \mathsf{bin}(z)$.
  // $z_i \in \{0,1\}$.
- For $i \in [h]$, define $\overline{z}_i := 1 \oplus z_i$.
- Find the smallest $i \in [h]$ s.t. $x_1||\cdots||x_{i-1}||x_i = z_1||\cdots||z_{i-1}||\overline{z}_i$. If there is no such index, output $\perp$.
- Output $y :=$ $\mathsf{PRG}'_{x_h}(\cdots \mathsf{PRG}_{x_{i+1}}(K_i) \cdots)$

Theorem 1 (GGM PPRF, [BW13; BGI14; KPTZ13; SW14]). *The construction presented in Figure 5, defined by interface* $\{\lambda, \{N_i\}_{i\in[\ell]}, \mathbb{F}\}$ *is a selectively-secure puncturable pseudorandom function for* $\mathsf{spec} = \{\mathsf{PRG}, \mathsf{PRG}', \ldots\}$*, if* $\mathsf{PRG} : \{0,1\}^n \to \{0,1\}^{2n}$*, and* $\mathsf{PRG}' : \{0,1\}^n \to \mathbb{F}^2$ *are pseudorandom generators (as per Definition 3).*

We note that our definition of PPRFs (Definition 14) requires the existence of a reusable setup. Although the GGM PPRF (as well as our adaptation of it) does not require a setup phase or public parameters (i.e., $\mathsf{msk}, \mathsf{mpk}$), it still fits this definition by treating the $\mathsf{Setup}, \mathsf{GenMSK}, \mathsf{ComputeMPK}$ as trivial functions that output null.

In the following theorems, we recall the power-DDH assumption of Definition 15 that ensures the security of the CHHK25 and our new k-NR constructions of Figures 6 and 7.

Definition 15. Let $B = B(\cdot)$ be a polynomial. The B-power-DDH assumption holds with respect to $\mathcal{G}$ if for all sufficiently large $\lambda \in \mathbb{N}$, the following distributions are computationally indistinguishable:

$$\mathcal{D}_0 := \left\{ (H_i)_{i\in[\pm B(\lambda)]} : r \xleftarrow{\$} \mathbb{Z}_p^*, H \xleftarrow{\$} \mathbb{G}, (H_i)_{i\in[\pm B(\lambda)]} \leftarrow (r^i \cdot H)_{i\in[\pm B(\lambda)]} \right\}$$

$$\mathcal{D}_1 := \left\{ (H_i)_{i\in[\pm B(\lambda)]} : r \xleftarrow{\$} \mathbb{Z}_p^*, H, H_0 \xleftarrow{\$} \mathbb{G}, (H_i)_{i\in[\pm B(\lambda)]^*} \leftarrow (r^i \cdot H)_{i\in[\pm B(\lambda)]^*} \right\},$$

where $(\mathbb{G}, G, p) \xleftarrow{\$} \mathcal{G}(1^\lambda)$.

Theorem 2 (CHHK25, [CHHK25]). *The construction presented in Figure 6, defined by interface* $\{\lambda, \{N_i\}_{i\in[\ell]}, \mathbb{F}\}$ *is a selectively-secure puncturable pseudorandom function for* $\mathsf{spec} = \{B, \mathcal{G}, \mathsf{H}, \ldots\}$*, if* $2B$*-power-DDH is hard relative to* $\mathcal{G}$ *(Definition 15) and* H *is a correlation-robust hash function for multiplicative correlations over* $\mathbb{G}$ *(as per Definition 6).*

Theorem 3 (k-ary Naor-Reingold (k-NR)). *The construction presented in Figure 6, defined by interface* $\{\lambda, \{N_i\}_{i\in[\ell]}, \mathbb{F}\}$ *is a selectively-secure PPRF with reusable setup for* $\mathsf{spec} = \{k, n, \mathcal{G}, \mathsf{H}, \ldots\}$*, if* B*-power-DDH is hard relative to* $\mathcal{G}$ *(Definition 15), where* $B = \lfloor n \cdot (k-1)^2 \rfloor$ *and* H *is a correlation-robust hash function for multiplicative correlations over* $\mathbb{G}$ *(as per Definition 6).*

Proof. We briefly go over the proof of correctness. The security follows from the same argument as in [BCMPR24].

To observe the correctness, let $z \in [0, N)$ and consider $(\mathsf{msk}, \mathsf{mpk})$ as well as $\mathsf{sk} \xleftarrow{\$} \mathsf{F.KeyGen}(\mathsf{msk})$ and $\mathsf{pk} \leftarrow \mathsf{F.Puncture}(\mathsf{msk}, \mathsf{sk}, z)$. Write $\boldsymbol{z} = (\|\mathsf{enc}_{\mathsf{in}}(z)\|^2, -2\mathsf{enc}_{\mathsf{in}}(z))$. For an input $x \in [0, N) \setminus \{z\}$, let $\boldsymbol{x} = \mathsf{enc}_{\mathsf{in}}^{1\prime}(x)$. The output of the PRF is computed as $\mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, x) = \mathsf{H}(t \cdot H)$ where

$$t = a_0^{\|\mathsf{enc}_{\mathsf{in}}(x)\|^2} \cdot \prod_{i=1}^{n'} a_i^{x_i}.$$

On the other hand, we have:

$$\mathsf{F.PEval}(\mathsf{mpk}, \mathsf{pk}, x) = \mathsf{H}(t' \cdot H_s) = \mathsf{H}(t' \cdot r^s \cdot H),$$

where $s := \langle \mathsf{enc}_{\mathsf{in}}^1(x), \mathsf{enc}_{\mathsf{in}}^2(z) \rangle$ and

Fig. 6: CHHK25 PPRF (CHHK25-PPRF)

The construction is defined by $\mathsf{interface} = \{\lambda, \{N_i\}_{i\in[\ell]}, \mathbb{F}\}$, where
- $\lambda \in \mathbb{N}$ is a security parameter.
- $N_i = N_i(\lambda) \in \mathbb{N}$ denotes a domain length for $i \in [\ell]$.
- $\mathbb{F} = \mathbb{F}(\lambda)$ is a field that is the range of the PPRF.

Given $\mathsf{interface}$, let spec contain the following for each $N \in \{N_i\}_{i\in[\ell]}$:
- $\mathcal{X} := \bigcup_{i\in[\ell]}[0, N_i)$.
- $B := \lfloor N/2 \rfloor$.
- $S := [\pm 2B] \setminus \{0\}$.
- $\mathsf{enc}_{\mathsf{in}} \colon [0, N) \to [\pm B]$ is an efficiently computable injection (e.g., $\mathsf{enc}_{\mathsf{in}}(x) = x - B$).
- $\mathcal{G}(1^\lambda) \to (\mathbb{G}, G, p)$ is an efficient probabilistic algorithm that on input a security parameter λ, outputs the description of an additive cyclic group $\mathbb{G}$ with generator G and of order p.
- $\mathsf{H} \colon \mathbb{G}_\lambda \to \mathbb{F}_\lambda$ is a family of correlation-robust hash functions for multiplicative correlations as per Definition 6.

F.Setup(1^λ):

- Run $(\mathbb{G}, G, p) \xleftarrow{\$} \mathcal{G}(1^\lambda)$.
- Output $\mathsf{pp} := (\mathbb{G}, G, p)$.

F.GenMSK(pp):

- Parse $\mathsf{pp} = (\mathbb{G}, G, p)$.
- Sample $\rho, r \xleftarrow{\$} \mathbb{Z}_p^*$. Let $H := \rho \cdot G$.
- Output $\mathsf{msk} := (r, H)$.

F.ComputeMPK(msk):

- Parse $\mathsf{msk} = (r, H)$.
- For $s \in S$, compute $H_s := r^s \cdot H$.
- Output $\mathsf{mpk} := (H_s)_{s\in S}$.

F.KeyGen(N, msk):

- Output $\perp$ if $\mathsf{CheckDomain}(N) = \perp$.
- Sample $a \xleftarrow{\$} \mathbb{Z}_p^*$.
- Output $\mathsf{sk} := (N, a)$.

F.Puncture(msk, sk, z):

- Parse $\mathsf{msk} = (r, H)$, and $\mathsf{sk} = (N, a)$.
- Set $\bar{z} := \mathsf{enc}_{\mathsf{in}}(z)$.
- Output $\mathsf{psk} = (N, \alpha := a \cdot r^{\bar{z}}, \bar{z})$.

F.Eval(msk, sk, x):

- Parse $\mathsf{msk} = (r, H)$, and $\mathsf{sk} = (N, a)$.
- Set $\bar{x} := \mathsf{enc}_{\mathsf{in}}(x)$.
- Output $y = \mathsf{H}(a \cdot H_{\bar{x}})$.

F.PEval(mpk, psk, x):

- Parse $\mathsf{mpk} = (H_s)_{s\in S}$, and $\mathsf{psk} = (N, \alpha, \bar{z})$.
- Set $\bar{x} := \mathsf{enc}_{\mathsf{in}}(x)$.
- Let $s := \bar{x} - \bar{z} \in [\pm 2B]$.
- If $s = 0$, output $\perp$ and abort.
- Otherwise, output $y = \mathsf{H}(\alpha \cdot H_s)$.

F.CheckDomain(N):

- Output $\top$ *iff* $\exists i \in [\ell] : N \le N_i$.

F.CheckMSK(msk):

- Output $\top$ *iff* $\mathsf{msk} \in \mathbb{Z}_p^* \times \mathbb{G}$.

F.CheckSK(msk, N, sk):

- Parse $\mathsf{sk} = (\mathsf{sk}_0, \mathsf{sk}_1)$.
- Output $\top$ *iff* $\mathsf{sk}_0 = N \wedge \mathsf{sk}_1 \in \mathbb{Z}_p^*$.

Fig. 7: Naor-Reingold PPRF with k-ary Decomposition (k-NR-PPRF)

The construction is defined by $\mathsf{interface} = \{\lambda, \{N_i\}_{i\in[\ell]}, \mathbb{F}\}$, where
- $\lambda \in \mathbb{N}$ is a security parameter.
- $N_i = N_i(\lambda) \in \mathbb{N}$ denotes a domain length for $i \in [\ell]$.
- $\mathbb{F} = \mathbb{F}(\lambda)$ is a field that is the range of the PPRF.

Given $\mathsf{interface}$, let spec contain the following for each $N \in \{N_i\}_{i\in[\ell]}$:
- $\mathcal{X} := \bigcup_{i\in[\ell]}[0, N_i)$.
- $k = k(\lambda)$ is a parameter that determines the base of logarithm for the encoding of the inputs.
- $n := \lceil \log_k(N) \rceil$.
- $n' := n + 1$.
- $S := [1, n \cdot (k-1)^2]$.
- $\mathsf{enc}_{\mathsf{in}} \colon [0, N) \to [0, k)^n$, $x \mapsto \tilde{\boldsymbol{x}}$ such that $x = \sum_{i=1}^{n} k^{i-1} \cdot \tilde{x}_i$.
- $\mathsf{enc}^{1'}_{\mathsf{in}} \colon [0, N) \to \{1\} \times [0, k)^n$, $x \mapsto (1, \mathsf{enc}_{\mathsf{in}}(x))$
- $\mathsf{enc}^{1}_{\mathsf{in}} \colon [0, N) \to [0, (k-1)^2] \times \{1\} \times [0, k)^n$, $x \mapsto (\|\mathsf{enc}_{\mathsf{in}}(x)\|^2, \mathsf{enc}^{1'}_{\mathsf{in}}(x))$
- $\mathsf{enc}^{2'}_{\mathsf{in}} \colon [0, N) \to [0, (k-1)^2] \times [-2k+2, 0]^n$, $z \mapsto (\|\mathsf{enc}_{\mathsf{in}}(z)\|^2, -2\mathsf{enc}_{\mathsf{in}}(z))$
- $\mathsf{enc}^{2}_{\mathsf{in}} \colon [0, N) \to \{1\} \times [0, (k-1)^2] \times [-2k+2, 0]^n$, $z \mapsto (1, \mathsf{enc}^{2'}_{\mathsf{in}}(z))$
  NB: $\|\mathsf{enc}_{\mathsf{in}}(x) - \mathsf{enc}_{\mathsf{in}}(z)\|^2 = \langle \mathsf{enc}^{1}_{\mathsf{in}}(x), \mathsf{enc}^{2}_{\mathsf{in}}(z) \rangle$
- $\mathcal{G}(1^\lambda) \to (\mathbb{G}, G, p)$ is an efficient probabilistic algorithm that on input a security parameter λ, outputs the description of an additive cyclic group $\mathbb{G}$ with generator G and of order p.
- $\mathsf{H} : \mathbb{G}_\lambda \to \mathbb{F}_\lambda$ is a family of correlation-robust hash functions for multiplicative correlations as per Definition 6.

F.Setup(1^λ):

- Run $(\mathbb{G}, G, p) \xleftarrow{\$} \mathcal{G}(1^\lambda)$.
- Output $\mathsf{pp} := (\mathbb{G}, G, p)$.

F.GenMSK(pp):

- Parse $\mathsf{pp} = (\mathbb{G}, G, p)$.
- Sample $\rho, r \xleftarrow{\$} \mathbb{Z}_p^*$.
- Let $H := \rho \cdot G$.
- Sample $a_0 \xleftarrow{\$} \mathbb{Z}_p$.
- Output $\mathsf{msk} := (r, H, a_0)$.

F.ComputeMPK(msk):

- Parse $\mathsf{msk} = (r, H, a_0)$.
- Let $\alpha_0 := a_0 \cdot r^{-1}$
- For $s \in S$, compute $H_s := r^s \cdot H$.
- Output $\mathsf{mpk} := ((H_s)_{s\in S}, \alpha_0)$.

F.KeyGen(N, msk):

- Output $\perp$ if $\mathsf{CheckDomain}(N) = \perp$.
- Sample $\mathbf{a} = (a_1, \ldots, a_{n'}) \xleftarrow{\$} \mathbb{Z}_p^{n'}$.
- Output $\mathsf{sk} := (N, \mathbf{a})$.

F.Puncture($\mathsf{msk}, \mathsf{sk}, z$):

- Parse $\mathsf{msk} = (r, H, a_0)$, and $\mathsf{sk} = (N, \mathbf{a})$.
- Set $\mathbf{z} := \mathsf{enc}^{2'}_{\mathsf{in}}(z)$.
- For $i \in [n']$, set $\alpha_i := a_i \cdot r^{-z_i}$.
- $\mathsf{psk} := (N, \boldsymbol{\alpha} = (\alpha_i)_{i\in[n']}, \mathbf{z})$.
- Output psk.

F.Eval($\mathsf{msk}, \mathsf{sk}, x$):

- Parse $\mathsf{msk} = (r, H, a_0)$, and $\mathsf{sk} = (N, \mathbf{a})$.
- Set $\mathbf{x} := \mathsf{enc}^{1'}_{\mathsf{in}}(x)$.
- $t := a_0^{\|\mathsf{enc}_{\mathsf{in}}(x)\|^2} \cdot \prod_{i=1}^{n'} a_i^{x_i}$.
- Output $y := \mathsf{H}(t \cdot H)$.

F.PEval($\mathsf{mpk}, \mathsf{psk}, x$):

- Parse $\mathsf{mpk} = ((H_s)_{s\in S}, \alpha_0)$, and $\mathsf{psk} = (N, \boldsymbol{\alpha}, \mathbf{z})$.
- $\mathbf{x} := \mathsf{enc}^{1'}_{\mathsf{in}}(x)$.
- $s := \|\mathsf{enc}_{\mathsf{in}}(x)\|^2 + \langle \mathbf{z}, \mathbf{x} \rangle$.
- If $s = 0$, output $\perp$.
- $t' := \alpha_0^{\|\mathsf{enc}_{\mathsf{in}}(x)\|^2} \cdot \prod_{i=1}^{n'} \alpha_i^{x_i}$
- Output $y := \mathsf{H}(t' \cdot H_s)$.

F.CheckDomain(N):

- Output $\top$ *iff* $\exists i \in [\ell] : N \le N_i$.

F.CheckMSK(msk):

- Output $\top$ *iff* $\mathsf{msk} \in \mathbb{Z}_p^* \times \mathbb{G} \times \mathbb{Z}_p$.

F.CheckSK($\mathsf{msk}, N, \mathsf{sk}$):

- Parse $\mathsf{sk} = (\mathsf{sk}_0, \mathsf{sk}_1)$.
- Output $\top$ *iff* $\mathsf{sk}_0 = N \wedge \mathsf{sk}_1 \in \mathbb{Z}_p^{n'}$.

$$
\begin{aligned}
t' &= \alpha_0^{\|\mathsf{enc}_{\mathsf{in}}(x)\|^2} \cdot \prod_{i=1}^{n'} \alpha_i^{x_i} \\
&= (a_0 \cdot r^{-1})^{\|\mathsf{enc}_{\mathsf{in}}(x)\|^2} \cdot \prod_{i=1}^{n'} (a_i \cdot r^{-z_i})^{x_i} \\
&= a_0^{\|\mathsf{enc}_{\mathsf{in}}(x)\|^2} \cdot \prod_{i=1}^{n'} a_i^{x_i} \cdot r^{-\|\mathsf{enc}_{\mathsf{in}}(x)\|^2 + \sum_{i=1}^{n'} -z_i \cdot x_i} \\
&= a_0^{\|\mathsf{enc}_{\mathsf{in}}(x)\|^2} \cdot \prod_{i=1}^{n'} a_i^{x_i} \cdot r^{-s}.
\end{aligned}
$$

Since r^s and r^{-s} cancel out, we have $t' \cdot H_s = t \cdot H$ and therefore the same result as from $\mathsf{F.Eval}$. □

5.3 A Compound PPRF

Here, we introduce a *compound* PPRF that is a mix of the GGM PPRF (Figure 5), the CHHK25 PPRF (Figure 6) and the k-NR PPRF (Figure 7). Essentially, our main observation is that the structured elements of the CHHK25 and k-NR are similar and can be shared with each other. In turn, we require the security of the power-DDH assumption for a larger parameter. In particular, the public parameters of our construction contain a set $S^{\max}$ which will be used to compute mpk as $(H_s = r^s \cdot H)_{s \in S^{\max}}$. Parsing $S^{\max} = [a, b]$, we require the B-power-DDH (Definition 15) to hold for $B = \max(a, b)$. Additionally, as shown in Figure 8, we introduce a selection function SEL that indicates which PPRF should be used as a subroutine at any point. This function selects the suitable PPRF with its associated spec (among the three constructions of Figures 5,6, and 7), that is used to properly call their underlying algorithms.

We present Theorem 4 to state the security and correctness of our compound PPRF.

Fig. 8: Compound PPRF (Compound-PPRF)

The construction is defined by $\mathsf{interface} = \{\lambda, \{\mathcal{X}_i\}_{i \in [\ell]}, \mathbb{F}\}$, where
- $\lambda \in \mathbb{N}$ is a security parameter.
- $\mathcal{X}_i = [0, N_i)$ denotes a domain, where $N_i = N_i(\lambda)$, for $i \in [\ell]$.
- $\mathbb{F} = \mathbb{F}(\lambda)$ is a field that is the range of the PPRF.

Given $\mathsf{interface}$, let spec contain the following:
- $\mathcal{X} := \cup_{i \in [\ell]} \mathbb{N} \times \mathcal{X}_i$ denotes the overall domain of the PPRF.
- $\mathsf{F}_1, \mathsf{F}_2, \mathsf{F}_3$ are PPRFs presented in Figures 5, 6, and 7, respectively, defined by $\mathsf{interface} = \{\lambda, \{\mathcal{X}_i\}_{i \in [\ell]}, \mathbb{F}\}$.
- SEL is an efficient deterministic function that on an input $N \in \{N_i\}_{i \in [\ell]}$, outputs an index $\mathsf{ind} \in \{1, 2, 3\}$ together with a set of specifications $\mathsf{spec}_{\mathsf{ind}}$ for $\mathsf{F}_{\mathsf{ind}}$.
- $\mathcal{G}(1^\lambda) \to (\mathbb{G}, G, p)$ is an efficient probabilistic algorithm that on input a security parameter λ, outputs the description of an additive cyclic group $\mathbb{G}$ with generator G and of order p.
- $\mathsf{H}\colon \mathbb{G}_\lambda \to \mathbb{F}_\lambda$ is a family of correlation-robust hash functions for multiplicative correlations as per Definition 6.

F.Setup(1^λ):

- Initiate $S^{\max} := \emptyset$, $N_2^{\max} := 0$, and $k_3^{\max} := 0$.
- For $i \in [\ell]$:
  - $(\mathsf{ind}, \mathsf{spec}_{\mathsf{ind}}) \leftarrow \mathsf{SEL}(N_i)$.
  - If $\mathsf{ind} = 1$, then skip.
  - Else if $\mathsf{ind} = 2$:
    - Read S, N from $\mathsf{spec}_{\mathsf{ind}}$.
    - $S^{\max} \leftarrow S^{\max} \cup S$.
    - $N_2^{\max} \leftarrow \max(N_2^{\max}, N)$.
  - Else if $\mathsf{ind} = 3$:
    - Read k and S from $\mathsf{spec}_{\mathsf{ind}}$
    - $S^{\max} \leftarrow S^{\max} \cup S$.
    - $k_3^{\max} \leftarrow \max(k_3^{\max}, k)$.
- Run $(\mathbb{G}, G, p) \xleftarrow{\$} \mathcal{G}(1^\lambda)$.
- Output $\mathsf{pp} := (\mathbb{G}, G, p, (S^{\max}, N_2^{\max}, k_3^{\max}))$.

F.GenMSK(pp):

- Sample $a_0, \rho, r \xleftarrow{\$} \mathbb{Z}_p^*$.
- Let $H := \rho \cdot G$.
- Output $\mathsf{msk} := (r, H, a_0)$.

F.ComputeMPK(msk):

- Parse $\mathsf{msk} = (r, H, a_0)$.
- Let $\alpha_0 := a_0 \cdot r^{-1}$.
- For $s \in S^{\max}$, $H_s := r^s \cdot H$.
- Output $\mathsf{mpk} := ((H_s)_{s \in S^{\max}}, \alpha_0)$.

F.KeyGen(N', msk):

- Output $\perp$ if $\mathsf{CheckDomain}(N') = \perp$ or $\mathsf{CheckMSK}(\mathsf{msk}) = \perp$.
- Find the smallest i s.t. $N' \le N_i$.
- Run $(\mathsf{ind}, \mathsf{spec}_{\mathsf{ind}}) \leftarrow \mathsf{SEL}(N_i)$.
- $\mathsf{sk} \leftarrow \mathsf{F}_{\mathsf{ind}}.\mathsf{KeyGen}(N', \mathsf{msk})$.
- Output sk.

F.Puncture(msk, sk, z):

- Read N' from sk.
- Output $\perp$ if $\mathsf{CheckDomain}(N') = \perp$ or $\mathsf{CheckSK}(\mathsf{msk}, N', \mathsf{sk}) = \perp$.
- Find the smallest i s.t. $N' \le N_i$.
- Run $(\mathsf{ind}, \mathsf{spec}_{\mathsf{ind}}) \leftarrow \mathsf{SEL}(N_i)$.
- Parse $\mathsf{msk} = (r, H, a_0)$.
- $\mathsf{psk} \leftarrow F_{\mathsf{ind}}.\mathsf{Puncture}(\mathsf{msk}, \mathsf{sk}, z)$.
- Output psk.

F.Eval(msk, sk, x):

- Read N' from sk.
- Output $\perp$ if $\mathsf{CheckDomain}(N') = \perp$ or $\mathsf{CheckSK}(\mathsf{msk}, N', \mathsf{sk}) = \perp$.
- Find the smallest i s.t. $N' \le N_i$.
- Run $(\mathsf{ind}, \mathsf{spec}_{\mathsf{ind}}) \leftarrow \mathsf{SEL}(N_i)$.
- Parse $\mathsf{msk} = (r, H, a_0)$.
- Output $y \leftarrow F_{\mathsf{ind}}.\mathsf{Eval}(\mathsf{msk}, \mathsf{sk}, x)$.

F.PEval(mpk, psk, x):

- Read N' from psk.
- Output $\perp$ if $\mathsf{CheckDomain}(N') = \perp$.
- Find the smallest i s.t. $N' \le N_i$.
- Run $(\mathsf{ind}, \mathsf{spec}_{\mathsf{ind}}) \leftarrow \mathsf{SEL}(N_i)$.
- Parse $\mathsf{mpk} = (\mathbb{G}, p, (H_s)_{s \in S}, \alpha_0)$.
- Output $y \leftarrow F_{\mathsf{ind}}.\mathsf{PEval}(\mathsf{mpk}, \mathsf{psk}, x)$.

F.CheckDomain(N'):

- Output $\top$ *iff* $\exists i \in [\ell] : N' \le N_i$.

F.CheckMSK(msk):

- Output $\top$ *iff* $\mathsf{msk} \in \mathbb{Z}_p^* \times \mathbb{G} \times \mathbb{Z}_p^*$.

F.CheckSK(msk, N', sk):

- Find the smallest i s.t. $N' \le N_i$.
- Run $(\mathsf{ind}, \mathsf{spec}_{\mathsf{ind}}) \leftarrow \mathsf{SEL}(N_i)$.
- Output $\top$ *iff* $\mathsf{F}_{\mathsf{ind}}.\mathsf{CheckSK}(\mathsf{msk}, N', \mathsf{sk}) = \top$.

Theorem 4. *The construction presented in Figure 8, defined by interface* $\{\lambda, \{N_i\}_{i \in [\ell]}, \mathbb{F}\}$ *is a selectively-secure puncturable pseudorandom function for* $\mathsf{spec} = \{\mathsf{F}_1, \mathsf{F}_2, \mathsf{F}_3, \mathsf{SEL}, \ldots\}$*, if for all* $N \in \{N_i\}_{i \in [\ell]}$*, the function* $\mathsf{F}_{\mathsf{ind}}$ *is a selectively-secure PPRF with reusable setup for* $\mathsf{spec}_{\mathsf{ind}}$*, where* $(\mathsf{ind}, \mathsf{spec}_{\mathsf{ind}}) \leftarrow \mathsf{SEL}(N)$.

6 Distributed Point Functions from Puncturable PRFs

In this section, we introduce a new definition of distributed point functions and present a transformation from our PPRF to a DMPF construction.

6.1 Definition

Our definition of distributed point function relies on the standard definition of point functions, which we recall in Definition 16.

Definition 16 (Point Functions). Let $\mathbb{G}$ be an additive group. Let $(\alpha, \beta) \in \mathbb{N} \times \mathbb{G}$. A point function $f_{(\alpha,\beta)} \colon \mathbb{N} \to \mathbb{G}$ is defined as follows:

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

for all $x \in \mathbb{N}$.

In the following, we introduce a definition of distributed point functions. Our definition differs from traditional DPF definitions (e.g. [Boy+20b; BCMRS25]) in additional properties tailored to distributed setting of our PCF, discussed in Section 11. In particular, we consider a reusable setup, similar to the one from our PPRF definition of Section 5, where a public parameter pp can be used to generate multiple key shares (k_0, k_1). Additionally, the setup and the key generation algorithms can be under the influence of a corrupted party. Specifically, for $\sigma \in \{0, 1, \perp\}$, a corrupted party P_σ can influence the computations of both the public parameters and the key values, where P_0 and P_1 denote corruptible parties 0 or 1, and $P_\perp$ denotes *no* corruption in the process.
As indicated in Definition 17, we use superscripts $\mathcal{A}, \sigma$ to denote the influence of the corruptible party $\mathcal{A} = P_\sigma$ on algorithms $\mathsf{DPF.Setup}$ and $\mathsf{DPF.Gen}$ via interaction. Observe that the superscript $\sigma \in \{0, 1, \perp\}$ of Setup and Gen algorithms specifies the corrupted party, while the input $\sigma \in \{0, 1\}$ of the evaluation algorithm Eval specifies the value's share of party P_σ.

Definition 17 (Distributed Point Functions (DPFs) with Reusable Setup under Adversarial Influence). Let $\lambda \in \mathbb{N}$. Let $\ell = \ell(\lambda)$, and $N_i = N_i(\lambda) \in \mathbb{N}$ for $i \in [\ell]$. Let $\mathbb{G} = \mathbb{G}(\lambda)$ be an additive group. Define $\mathcal{X} := \bigcup_{i \in [\ell]} [0, N_i)$. A distributed point function (DPF) with reusable setup for the class of point functions $\mathcal{F} = \{f_{(\alpha,\beta)} : \mathbb{N} \to \mathbb{G}\}$ is defined by $\mathsf{interface} = \{\lambda, \{N_i\}_{i \in [\ell]}, \mathbb{G}\}$ is a tuple of polynomial time algorithms as follows:

$\mathsf{DPF.Setup}^{\mathcal{A},\sigma}(1^\lambda) \to (\mathsf{st}_0, \mathsf{st}_1)$: The probabilistic setup algorithm takes as input a security parameter $\lambda \in \mathbb{N}$, and interacts with a PPT adversary $\mathcal{A}$ with respect to the value $\sigma \in \{0, 1, \perp\}$, and outputs two states st_0 and st_1.

$\mathsf{DPF.Gen}^{\mathcal{A},\sigma}(\mathsf{st}_0, \mathsf{st}_1, N, (\alpha, \beta)) \to (k_0, k_1)$: The probabilistic key generation algorithm takes as input $\mathsf{st}_0, \mathsf{st}_1$, a domain length $N = N(\lambda)$, a pair $(\alpha, \beta) \in \mathbb{N} \times \mathbb{G}$, and interacts with a PPT adversary $\mathcal{A}$ with respect to the value $\sigma \in \{0, 1, \perp\}$, to output a key pair (k_0, k_1) or returns $\perp$. We assume that each returned key contains N.

$\mathsf{DPF.Eval}(\sigma, k_\sigma, x) \to y_\sigma$: The deterministic evaluation algorithm is a deterministic polynomial time algorithm that takes as input $\sigma \in \{0, 1\}$, a key k_σ, and an input $x \in \mathcal{X}$, and outputs $y_\sigma \in \mathbb{G}$ or $\perp$.

We require a DPF DPF with reusable setup defined by any $\mathsf{interface} = \{\lambda, \{N_i\}_{i \in [\ell]}, \mathbb{G}\}$ to satisfy the following properties:

- **Domain Validation:** There exists a deterministic public algorithm
  - $\mathsf{CheckDomain}(N) \to \top/\perp$

  that on input $N \in \mathbb{N}$, outputs $\top$ *iff* there exists $i \in [\ell]$ such that $N \leq N_i$.
  All algorithms of DPF that (explicitly or implicitly) take $N \in \mathbb{N}$ as input, can run $\mathsf{CheckDomain}(N)$ as a subroutine and output $\perp$ if this algorithm returns $\perp$.

– **Perfect Correctness:** For any PPT adversary $\mathcal{A}$ and $\sigma \in \{0,1,\perp\}$, any point function $f_{(\alpha,\beta)} \in \mathcal{F}$ (as per Definition 16), all $N \in \mathbb{N}$ satisfying $\mathsf{CheckDomain}(N) = \top$, and all $x \in \mathcal{X}$,

$$\Pr\left[y_0 - y_1 = f_{(\alpha,\beta)}(x) : \begin{array}{l}(\mathsf{st}_0,\mathsf{st}_1) \xleftarrow{\$} \mathsf{DPF.Setup}^{\mathcal{A},\sigma}(1^\lambda)\\ (k_0,k_1) \xleftarrow{\$} \mathsf{DPF.Gen}^{\mathcal{A},\sigma}(\mathsf{st}_0,\mathsf{st}_1,N,(\alpha,\beta))\\ y_0 \leftarrow \mathsf{DPF.Eval}(0,k_0,x)\\ y_1 \leftarrow \mathsf{DPF.Eval}(1,k_1,x)\end{array}\right] = 1\ .$$

– **Pseudorandom Outputs:** For any PPT adversary $\mathcal{A}$ and $\sigma \in \{0,1\}$, any $Q = \mathsf{poly}(\lambda)$, any polynomial sequence of polynomial-size point functions $f_1, f_2, \ldots \in \mathcal{F}$, and set of inputs $(x_i)_{i\in[Q]}$, where $x_i \in \mathcal{X}$,

$$\left\{(y_i)_{i\in[Q]} : \begin{array}{l}(\mathsf{st}_0,\mathsf{st}_1) \xleftarrow{\$} \mathsf{DPF.Setup}^{\mathcal{A},\perp}(1^\lambda)\\ f \in f_1, f_2, \ldots\\ N \in \{N_i\}_{i\in[\ell]}\\ (k_0,k_1) \xleftarrow{\$} \mathsf{DPF.Gen}^{\mathcal{A},\perp}(\mathsf{st}_0,\mathsf{st}_1,N,f=(\alpha,\beta))\\ y_i \leftarrow \mathsf{DPF.Eval}(\sigma,k_\sigma,x_i)\end{array}\right\}$$
$$\overset{c}{\approx} \left\{(y_i)_{i\in[Q]} :\ y_i \xleftarrow{\$} \mathbb{G}\right\},$$

where $\mathsf{DPF.CheckDomain}(N) = \top$.

– **Security:** There exist PPT simulators $\mathsf{SetupSim}$ and GenSim such that for any non-uniform PPT algorithm $\mathcal{A}$ and any $\sigma \in \{0,1\}$,

$$\Pr[\mathsf{Exp}^{\mathsf{dpf\text{-}sec},\sigma}_{\mathcal{A}}(1^\lambda) = 1] \le \mathsf{negl}(\lambda),$$

where $\mathsf{Exp}^{\mathsf{dpf\text{-}sec},\sigma}_{\mathcal{A}}(1^\lambda)$ is defined as follows:

- **Setup:** The challenger samples $b \xleftarrow{\$} \{0,1\}$.
  - If $b = 0$, the challenger samples $(\mathsf{st}_0,\mathsf{st}_1) \xleftarrow{\$} \mathsf{DPF.Setup}^{\mathcal{A},\sigma}(1^\lambda)$ by interacting with $\mathcal{A}$.
  - If $b = 1$, the challenger samples $(\mathsf{st}_\sigma,\mathsf{td}) \leftarrow \mathsf{SetupSim}^{\mathcal{A},\sigma}(1^\lambda)$ by interacting with $\mathcal{A}$.
- **Key Queries:** $\mathcal{A}$ can query polynomially-many $(N, f_{(\alpha,\beta)})$, where $f_{(\alpha,\beta)} \in \mathcal{F}$ is an arbitrary polynomial-sized point function and $N \in \mathbb{N}$. The challenger responds as follows:
  - If $b = 0$, the challenger samples $k_\sigma \leftarrow \mathsf{DPF.Gen}^{\mathcal{A},\sigma}(\mathsf{st}_0,\mathsf{st}_1,N,(\alpha,\beta))$ by interacting with $\mathcal{A}$.
  - If $b = 1$, the challenger samples $k_\sigma \leftarrow \mathsf{GenSim}^{\mathcal{A},\sigma}(\mathsf{td},\mathsf{Leak}_\sigma(f_{(\alpha,\beta)}))$ by interacting with $\mathcal{A}$. The leakage function $\mathsf{Leak}_\sigma(f_{(\alpha,\beta)})$ leaks by default the domain and range of $f_{(\alpha,\beta)}$, i.e., $(N,\mathbb{G})$.

  The challenger then returns k_σ to $\mathcal{A}$.
- **Guess:** $\mathcal{A}$ returns $b' \in \{0,1\}$.

The experiment outputs 1 if $b' = b$, and 0 otherwise.

Definition 18 (Known-Index DPFs with Reusable Setup).
A *known-index* distributed point function with reusable setup (KI-DPF) is a DPF as defined in Definition 17 where the index α of the point function $f_{(\alpha,\beta)}$ is included in k_0. In the security property of such constructions, the leakage function $\mathsf{Leak}_0(f_{(\alpha,\beta)})$, additionally outputs the indices α.

6.2 DPFs from PPRFs

In the following, given a PPRF with reusable setup (as in Definition 14), we present a construction of KI-DPF under adversarial influence in Figure 9.

Theorem 5. *Let* $\mathsf{interface}_{\mathsf{DPF}} = \{\lambda, \{N_i\}_{i\in[\ell]}, \mathbb{G}\}$. *The construction presented in Figure 9 defined by* $\mathsf{interface}_{\mathsf{DPF}}$ *is a known-index DPF for the class of point functions* $\mathcal{F} = \{f_{(\alpha,\beta)} : \mathbb{N} \to \mathbb{G}\}$, *as per Definition 18.*

Proof. In the following, we prove the properties of Definition 18 for the construction of Figure 9.

Fig. 9: Transforming PPRFs to KI-DPFs

The construction is defined by $\mathsf{interface}_{\mathsf{DPF}} = \{\lambda, \{N_i\}_{i\in[\ell]}, \mathbb{G}\}$, where $\mathbb{G}$ is an additive group.

- Let F be a PPRF with reusable setup as per Definition 14 defined by $\{\lambda, \{N_i\}_{i\in[\ell]}, \mathbb{G}\}$ (read from $\mathsf{interface}_{\mathsf{DPF}}$).
- Let $\mathcal{A}$ be a PPT algorithm and $\sigma \in \{0, 1, \perp\}$.

$\mathsf{DPF.Setup}^{\mathcal{A},\sigma}(1^\lambda)$:

- Run $\mathsf{pp} \leftarrow \mathsf{F.Setup}(1^\lambda)$.
- If $\sigma = \perp$ or $\sigma = 0$:
  - $\mathsf{msk} \xleftarrow{\$} \mathsf{F.GenMSK}(\mathsf{pp})$.
- If $\sigma = 1$:
  - $\mathsf{msk} \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp})$.
  - If $\mathsf{F.CheckMSK}(\mathsf{msk}) = \perp$, return $\perp$.
- Compute $\mathsf{mpk} \leftarrow \mathsf{F.ComputeMPK}(\mathsf{msk})$.
- Output $\mathsf{st}_0 := (\mathsf{pp}, \mathsf{msk})$ and $\mathsf{st}_1 := (\mathsf{pp}, \mathsf{mpk})$.

$\mathsf{DPF.Eval}(\sigma, k_\sigma, x)$:

- If $\sigma = 0$:
  - Parse $k_0 = (N, \alpha, \mathsf{mpk}, \mathsf{psk}, y^*)$.
  - If $x = \alpha$, output $y_0 := y^*$.
  - Else, output $y_0 := \mathsf{F.PEval}(\mathsf{mpk}, \mathsf{psk}, x)$.
- If $\sigma = 1$:
  - Parse $k_1 = (N, \mathsf{msk}, \mathsf{sk})$.
  - Output $y_1 := \mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, x)$.

$\mathsf{DPF.Gen}^{\mathcal{A},\sigma}(\mathsf{st}_0, \mathsf{st}_1, N, (\alpha, \beta))$:

- Parse $\mathsf{st}_0 = (\mathsf{pp}, \mathsf{msk})$ and $\mathsf{st}_1 = (\mathsf{pp}, \mathsf{mpk})$.
- If $\sigma = \perp$ or $\sigma = 0$:
  - $\mathsf{sk} \xleftarrow{\$} \mathsf{F.KeyGen}(N, \mathsf{msk})$.
- If $\sigma = 1$:
  - $\mathsf{sk} \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{msk})$.
  - If $\mathsf{F.CheckSK}(\mathsf{msk}, N, \mathsf{sk}) = \perp$, return $\perp$.
- $\mathsf{psk} \leftarrow \mathsf{F.Puncture}(\mathsf{msk}, \mathsf{sk}, \alpha)$.
- $y^* \leftarrow \mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, \alpha) + \beta$.
- $k_0 := (N, \alpha, \mathsf{mpk}, \mathsf{psk}, y^*)$.
- $k_1 := (N, \mathsf{msk}, \mathsf{sk})$.
- Output (k_0, k_1).

– **Correctness.** Let $\lambda \in \mathbb{N}$, and $N \in \{N_i\}_{i\in[\ell]}$. Let $\mathcal{A}$ be a PPT algorithm and $\sigma \in \{0, 1, \perp\}$. Let $(\alpha, \beta) \in \mathbb{N} \times \mathbb{G}$. We show that and all $x \in [0, N)$,

$$\Pr\left[y_0 - y_1 = f_{(\alpha,\beta)}(x) : \begin{array}{l} (\mathsf{st}_0, \mathsf{st}_1) \xleftarrow{\$} \mathsf{DPF.Setup}^{\mathcal{A},\sigma}(1^\lambda) \\ (k_0, k_1) \xleftarrow{\$} \mathsf{DPF.Gen}^{\mathcal{A},\sigma}(\mathsf{st}_0, \mathsf{st}_1, N, (\alpha, \beta)) \\ y_0 \leftarrow \mathsf{DPF.Eval}(0, k_0, x) \\ y_1 \leftarrow \mathsf{DPF.Eval}(1, k_1, x) \end{array} \right] = 1 \ .$$

In the construction, if Setup and Gen succeed without aborting, for any input $x \in [0, N)$, the values y_0 and y_1 are computed as follows:

$$y_0 = \begin{cases} \mathsf{F.PEval}(\mathsf{mpk}, \mathsf{psk}, x) & : x \neq \alpha \\ y^* & : x = \alpha \end{cases} , \quad y_1 = \mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, x) \ ,$$

where $\alpha \in [0, N)$, and either (I) $\mathsf{pp} \leftarrow \mathsf{F.Setup}(1^\lambda)$, $\mathsf{msk} \xleftarrow{\$} \mathsf{F.GenMSK}(\mathsf{pp})$ and $\mathsf{sk} \xleftarrow{\$} \mathsf{F.KeyGen}(\mathsf{msk})$, if $\sigma \in \{\perp, 0\}$, or (II) $\mathsf{pp} \leftarrow \mathsf{F.Setup}(1^\lambda)$, and $\mathsf{msk} \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp})$ such that $\mathsf{F.CheckMSK}(\mathsf{msk}) = \top$, and $\mathsf{sk} \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{msk})$ such that $\mathsf{F.CheckSK}(\mathsf{msk}, N, \mathsf{sk}) = \top$, if $\sigma = 1$. Therefore, for any $x \in [0, N)$, it holds that:

$$y_0 - y_1 = \begin{cases} \mathsf{F.PEval}(\mathsf{mpk}, \mathsf{psk}, x) - \mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, x) & : x \neq \alpha \\ y^* - \mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, x) & : x = \alpha \end{cases} .$$

Due to the perfect correctness property of the underlying PPRF, $\mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, x) = \mathsf{F.PEval}(\mathsf{mpk}, \mathsf{psk}, x)$ for any $x \in [0, N)$ such that $x \neq \alpha$, and based on construction, $y^* = \mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, \alpha) + \beta$. Therefore,

$$y_0 - y_1 = \begin{cases} 0, & \text{if } x \neq \alpha \\ \beta, & \text{if } x = \alpha \end{cases} = f_{(\alpha,\beta)}(x),$$

which concludes the proof of correctness.

– **Security.** To prove security, we need to construct simulators $\mathsf{SetupSim}$ and GenSim such that

$$\Pr[\mathsf{Exp}_{\mathcal{A}}^{\mathsf{dpf\text{-}sec},\sigma}(1^\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

as per Definition 18.
Let $\mathsf{SetupSim}^{\mathcal{A},\sigma}(1^\lambda)$ to be as follows:

- Computes $\mathsf{pp} \leftarrow \mathsf{F.Setup}(1^\lambda)$.
- If $\sigma \in \{0, \perp\}$, it computes $\mathsf{msk} \xleftarrow{\$} \mathsf{F.GenMSK}(\mathsf{pp})$, and if $\sigma = 1$, it obtains $\mathsf{msk} \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp})$ by interacting with adversary. (The simulator returns $\perp$ if $\mathsf{F.CheckMSK}(\mathsf{msk}) = \perp$)
- Computes $\mathsf{mpk} \leftarrow \mathsf{F.ComputeMPK}(\mathsf{msk})$.
- Sets $\mathsf{st}_0 = (\mathsf{pp}, \mathsf{msk})$, $\mathsf{st}_1 = (\mathsf{pp}, \mathsf{mpk})$, and $\mathsf{td} = \mathsf{msk}$.

Assuming $(\alpha, N, \mathbb{G}) \leftarrow \mathsf{Leak}_0(f_{(\alpha,\beta)})$ and $(N, \mathbb{G}) \leftarrow \mathsf{Leak}_1(f_{(\alpha,\beta)})$,

- If $\sigma \in \{0, \perp\}$, $\mathsf{GenSim}^{\mathcal{A},\sigma}(\mathsf{td}, \alpha, N, \mathbb{G})$ proceeds as follows:
  1. Computes $\mathsf{mpk} \leftarrow \mathsf{F.ComputeMPK}(\mathsf{msk})$.
  2. Computes $\mathsf{sk} \xleftarrow{\$} \mathsf{F.KeyGen}(N, \mathsf{msk})$.
  3. Computes $\mathsf{psk} \leftarrow \mathsf{F.Puncture}(\mathsf{msk}, \mathsf{sk}, \alpha)$.
  4. Samples $y^* \xleftarrow{\$} \mathbb{G}$.
  5. Outputs $k_0 = (N, \alpha, \mathsf{mpk}, \mathsf{psk}, y^*)$.
- If $\sigma = 1$, $\mathsf{GenSim}^{\mathcal{A},1}(\mathsf{td}, N, \mathbb{G})$ proceeds as follows:
  1. Interacting with the adversary $\mathcal{A}$, it receives $\mathsf{sk} \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{msk})$. If $\mathsf{F.CheckSK}(\mathsf{msk}, N, \mathsf{sk}) = \perp$, it returns $\perp$.
  2. It then outputs $k_1 = (N, \mathsf{msk}, \mathsf{sk})$.

It is easy to see that the distribution of k_1 output by the simulators above is the same as the distribution of k_1 output by the original setup and key generation algorithms, as they are generated through interacting with the adversary and $\mathsf{F.CheckMSK}(\mathsf{msk}) = \top = \mathsf{F.CheckSK}(\mathsf{msk}, N, \mathsf{sk})$. Therefore, the security for $\sigma = 1$ follows immediately. Next, we prove the security for $\sigma = 0$ or $\sigma = \perp$ as follows:
Assume there exists a non-uniform PPT adversary $\mathcal{A}$ that wins the experiment $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{dpf\text{-}sec},\sigma}(1^\lambda)$ with non-negligible advantage $\epsilon = \epsilon(\lambda)$. we construct a polynomial-time adversary $\mathcal{B}$ that can break the selective security of the underlying PPRF F.

1. $\mathcal{B}$ receives $(\mathsf{pp}, \mathsf{mpk})$ from its challenger.
2. Upon receiving any query $(N, f_{(\alpha,\beta)})$ from $\mathcal{A}$, $\mathcal{B}$ sends the challenge (N, α) to its challenger and receives (psk, y). It then computes and sends $(N, \alpha, \mathsf{mpk}, \mathsf{psk}, y + \beta)$ to the adversary $\mathcal{A}$.
3. Eventually, $\mathcal{B}$ obtains the bit b' from $\mathcal{A}$ and sends it to its challenger.

If y is computed as the evaluation of the challenge α, then the tuple of step 3 has the same distributions as the original key k_0 outputed by key generation algorithm, and if y is sampled randomly, then the tuple has the same distribution as the key k_0 outputed by GenSim. Therefore, if $\mathcal{A}$ can distinguish the two hybrids with non-negligible advantage ε, then the above adversary $\mathcal{B}$ can break the selective security of the underlying PPRF with advantage ε. This concludes the security proof of the construction.

– **Pseudorandom Outputs.** In order to prove pseduorandomness, we need to show that for $\sigma \in \{0, 1\}$, any polynomial sequence of polynomial-size point functions $f_1, f_2, \ldots \in \mathcal{F}$, and set of inputs $(x_i)_{i\in[Q]}$,

$$\left\{ (y_i)_{i\in[Q]} : \begin{array}{l} (\mathsf{st}_0, \mathsf{st}_1) \xleftarrow{\$} \mathsf{DPF.Setup}^{\mathcal{A},\perp}(1^\lambda) \\ f \in f_1, f_2, \ldots \\ N \in \{N_i\}_{i\in[\ell]} \\ (k_0, k_1) \xleftarrow{\$} \mathsf{DPF.Gen}^{\mathcal{A},\perp}(\mathsf{st}_0, \mathsf{st}_1, N, f = (\alpha, \beta)) \\ y_i \leftarrow \mathsf{DPF.Eval}(\sigma, k_\sigma, x_i) \end{array} \right\}$$
$$\overset{c}{\approx} \left\{ (y_i)_{i\in[Q]} : \; y_i \xleftarrow{\$} \mathbb{G} \right\},$$

We start from the left-hand-side distribution and proceed as follows. For any $x \in \mathcal{X}$, if $\sigma = 1$, consider the following hybrids:

Hyb_0: This is the original DPF evaluation of the construction: $y_1 \leftarrow \mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, x)$.
Hyb_1: This is similar to the original algorithm except that $y_1 \xleftarrow{\$} \mathbb{G}$.

Claim. The two hybrids Hyb_0 and Hyb_1 are computationally indistinguishable.

Proof. Based on the pseudorandomness property of the underlying PPRF scheme, the evaluation $\mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, x)$ for any input x is indistinguishable from a uniformly sampled value y. This is simply because any distinguisher that can distinguish between the two, can be used to construct an adversary against the pseudorandomness property of the underlying PPRF scheme as follows. The adversary simply queries the PPRF evaluation algorithm for $\leq Q$ inputs, and forwards the responses to the distinguisher. It then outputs the bit b' outputted by the distinguisher. It is easy to see that if $b = 0$, the distribution of the responses is similar to Hyb_0 and if $b = 1$, this distribution is similar to Hyb_1, and therefore, the adversary has a similar advantage to the distinguisher.
This implies that the value y_1 in hybrid Hyb_0 is going to be indistinguishable from a uniformly sampled value, which concludes the proof.

□

If $\sigma = 0$, consider the following hybrids:
Hyb_0: This is similar to the original DPF evaluation:
1. If $x \neq \alpha$, $y_0 \leftarrow \mathsf{F.PEval}(\mathsf{mpk}, \mathsf{psk}, x)$.
2. If $x = \alpha$, $y_0 = y^*$.

Hyb_1: This is similar to the previous hybrid except that the punctured value is replaced by PPRF evaluation.
1. If $x \neq \alpha$, $y_0 \leftarrow \mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, x)$.
2. If $x = \alpha$, $y_0 = y^*$.

Claim. The two hybrids Hyb_0 and Hyb_1 are indistinguishable.

Proof. Based on the correctness of the underlying PPRF, $\mathsf{F.PEval}(\mathsf{mpk}, \mathsf{psk}, x) = \mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, x)$ if $x \neq \alpha$. This immediately implies the indistinguishability of the two hybrids. □

Hyb_2: In this hybrid, the value of y_0 is sampled uniformaly at random:
1. If $x \neq \alpha$, $y_0 \xleftarrow{\$} \mathbb{G}$.
2. If $x = \alpha$, $y_0 \xleftarrow{\$} \mathbb{G}$.

Claim. The two hybrids Hyb_1 and Hyb_2 are computationally indistinguishable.

Proof. As the pseudorandomness property of the PPRF states, the value of $\mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, x)$ is indistinguishable from a uniformly sampled value. The proof is exactly similar to the previous argument. Viewing $y^* = \mathsf{F.Eval}(\mathsf{msk}, \mathsf{sk}, \alpha) + \beta$ from the original evaluation, the distribution of the value y_0 in hybrid Hyb_0 is indistinguishable from the uniform distribution, and this concludes the proof. □

And this concludes the proof.

□

Corollary 6. *Applying the transformation of Figure 9 on the compound PPRF construction of Figure 8 with interface* $\{\lambda, \{N_i\}_{i\in[\ell]}, \mathbb{G}\}$*, results in a compound known-index distributed point function with* $\mathsf{interface}_{\mathsf{DPF}} = \{\lambda, \{N_i\}_{i\in[\ell]}, \mathbb{G}\}$*. We call the resulting DPF,* Compound-DPF*.*

7 Pseudorandom Correlation Functions for VOLE

7.1 Definition

Here we recall the definition of pseudorandom correlation functions specified to vector oblivious linear evaluation (VOLE) correlations (Definition 19). We call these PCFs for VOLE (Definition 20).

The following defines VOLE correlations.

Definition 19 (VOLE Correlation). Let $n, r \in \mathbb{N}$, and $p \in \mathbb{P}$ be prime. A subfield vector oblivious linear evaluation (VOLE) correlation of dimension n over the finite field $\mathbb{F}_{p^r}$ is a tuple of vectors $((\boldsymbol{u}, \boldsymbol{v}), (\Delta, \boldsymbol{w})) \in (\mathbb{F}_p^n \times \mathbb{F}_{p^r}^n) \times (\mathbb{F}_{p^r} \times \mathbb{F}_{p^r}^n)$ such that $\boldsymbol{w} = \boldsymbol{u} \cdot \Delta + \boldsymbol{v}$. We denote by $((\boldsymbol{u}, \boldsymbol{v}), (\Delta, \boldsymbol{w})) := [\![\boldsymbol{u}]\!] \xleftarrow{\$} \mathsf{VOLE}^n$ the sampling of a random VOLE correlation of dimension n. When $n = 1$ the superscript is omitted. If sampling takes an input then that input is fixed, for example $((u, v), (\Delta, w)) := [\![u]\!] \xleftarrow{\$} \mathsf{VOLE}(\Delta)$ fixes $\Delta \in \mathbb{F}_{p^r}$.

Notation. When the parameters and Δ are clear from context we use notational shorthand for brevity. Angled brackets $\langle \boldsymbol{x} \rangle$ denote a subtractive sharing of $\Delta \cdot \boldsymbol{x}$, i.e., $\langle \boldsymbol{x} \rangle = (\langle \boldsymbol{x} \rangle_0, \langle \boldsymbol{x} \rangle_1)$ such that $\langle \boldsymbol{x} \rangle_1 - \langle \boldsymbol{x} \rangle_0 = \Delta \cdot \boldsymbol{x}$. Double brackets $[\![\boldsymbol{x}]\!]$ denote a full VOLE correlation, i.e., $[\![\boldsymbol{x}]\!] = \big((\boldsymbol{x}, \langle \boldsymbol{x} \rangle_0), (\Delta, \langle \boldsymbol{x} \rangle_1)\big)$. With subscript $\sigma \in \{0, 1\}$ the double brackets indicate $[\![x]\!]_0 = (\boldsymbol{x}, \langle \boldsymbol{x} \rangle_0)$ and $[\![x]\!]_1 = (\Delta, \langle \boldsymbol{x} \rangle_1)$. Linear operations over VOLE-authenticated data is denoted as $[\![\boldsymbol{z}]\!] = \alpha \cdot [\![\boldsymbol{x}]\!] + [\![\boldsymbol{y}]\!] + \beta$, which is shorthand for the following computation:

$$[\![\boldsymbol{z}]\!]_0 = \alpha \cdot [\![\boldsymbol{x}]\!]_0 + [\![\boldsymbol{y}]\!]_0 + \beta = \big(\alpha \cdot \boldsymbol{x} + \boldsymbol{y} + \beta, \alpha \cdot \langle \boldsymbol{x} \rangle_0 + \langle \boldsymbol{y} \rangle_0\big) \qquad \text{and}$$
$$[\![\boldsymbol{z}]\!]_1 = \alpha \cdot [\![\boldsymbol{x}]\!]_1 + [\![\boldsymbol{y}]\!]_1 + \beta = \big(\Delta, \alpha \cdot \langle \boldsymbol{x} \rangle_1 + \langle \boldsymbol{y} \rangle_1 + \Delta \cdot \beta\big).$$

Next, we introduce a definition for PCFs *under adversarial influence*. Our definition differs from the traditional definitions of PCF (e.g., [Boy+20a]) in that the key generation algorithm can be run in interaction with an adversary. This would let us cover scenarios where one of the parties involved in the PCF is corrupted and can somehow influence the key generation, especially when a distributed protocol simulates it. More precisely, $\mathsf{PCF}.\mathsf{Gen}^{\mathcal{A},\sigma}$ for $\sigma \in \{\bot, 0, 1\}$ is interpreted as follows: $\sigma = \bot$ denotes that none of the parties are corrupted, therefore the adversary does not have any influence, in which case the definition collapses to traditional PCF definitions without the adversarial influence. And $\sigma = b \in \{0, 1\}$ denotes that Party b is corrupted.

Definition 20 (PCF for VOLE under Adversarial Influence). Let $\lambda \in \mathbb{N}$ be the security parameter, let $r, p \in \mathbb{N}$ with p prime, and let $\mathcal{X} = \mathcal{X}(\lambda)$ denote a domain. A PCF for VOLE defined by $\mathsf{interface} = \{\lambda, \mathcal{X}, \mathbb{F}_{p^r}\}$ is a tuple with the following syntax:

$\mathsf{PCF}.\mathsf{Gen}^{\mathcal{A},\sigma}(1^\lambda, \Delta) \to (\mathsf{k}_0^{\mathsf{PCF}}, \mathsf{k}_1^{\mathsf{PCF}})$**.** For adversary $\mathcal{A}$ and $\sigma \in \{\bot, 0, 1\}$, key generation is an interactive process that on input $\Delta \in \mathbb{F}_{p^r}$ outputs a pair $(\mathsf{k}_0^{\mathsf{PCF}}, \mathsf{k}_1^{\mathsf{PCF}})$ or $\bot$.

$\mathsf{PCF}.\mathsf{Eval}(\sigma, \mathsf{k}_\sigma^{\mathsf{PCF}}, x) \to [\![y]\!]_\sigma$**.** Evaluation is a deterministic PT algorithm that on input $\sigma \in \{0, 1\}$, $\mathsf{k}_\sigma^{\mathsf{PCF}}$ and $x \in \mathcal{X}$, outputs $[\![y]\!]_\sigma$ where $[\![y]\!]_0 \in \mathbb{F}_p \times \mathbb{F}_{p^r}$ and $[\![y]\!]_1 \in \mathbb{F}_{p^r} \times \mathbb{F}_{p^r}$.

For any $\mathsf{interface} = \{\lambda, \mathcal{X}, \mathbb{F}_{p^r}\}$, we say that $\mathsf{PCF} = (\mathsf{Gen}^{\mathcal{A},\sigma}, \mathsf{Eval})$ defined by $\mathsf{interface}$ is a *secure PCF for VOLE under adversarial influence* if the properties of perfect correctness, strong VOLE-correlated pseudorandomness, and strong security, as defined with respect to the games in Figure 10, hold.

Perfect Correctness. For all PPT adversaries $\mathcal{A}$, all $\sigma \in \{\bot, 0, 1\}$, all $\mathsf{pp}_{\mathsf{PCF}}$, all $x \in \mathcal{X}$, and all $\Delta \in \mathbb{F}_{p^r}$:

$$\Pr\left[[\![y]\!] \in \mathsf{VOLE}(\Delta) \;\middle|\; \begin{array}{c} (\mathsf{k}_0^{\mathsf{PCF}}, \mathsf{k}_1^{\mathsf{PCF}}) \leftarrow \mathsf{Gen}^{\mathcal{A},\sigma}(1^\lambda, \Delta) \\ [\![y]\!] := \big(\mathsf{Eval}(0, \mathsf{k}_0^{\mathsf{PCF}}, x), \mathsf{Eval}(1, \mathsf{k}_1^{\mathsf{PCF}}, x)\big) \end{array} \right] = 1.$$

Strong VOLE-Correlated Pseudorandomness. For all non-uniform PPT adversaries $\mathcal{A}$ making at most $Q = Q(\lambda)$ queries to oracle $\mathcal{O}_b^{\mathsf{pr}}$ there exists an $N \in \mathbb{N}$ such that for all $\lambda > N$, for all $\mathsf{pp}_{\mathsf{PCF}}$, and for all $\Delta \in \mathbb{F}_{p^r}$:

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

Strong Security. For all non-uniform PPT adversaries $\mathcal{A}$ making at most $Q = Q(\lambda)$ queries to oracle $\mathcal{O}_b^{\mathsf{sec}}$ there exists an $N \in \mathbb{N}$ such that for all $\lambda > N$, for all $\mathsf{pp}_{\mathsf{PCF}}$, and for all $\Delta, \Delta' \in \mathbb{F}_{p^r}$:

$$\left| \Pr\left[\mathsf{Exp}_{\mathcal{A},0}^{\mathsf{sec}}(1^\lambda, \Delta, \Delta') = 1\right] - \Pr\left[\mathsf{Exp}_{\mathcal{A},1}^{\mathsf{sec}}(1^\lambda, \Delta, \Delta') = 1\right] \right| = \mathsf{negl}(\lambda).$$

Fig. 10: PCF Strong Correlated Pseudorandomness and Security Games

Let $\left(\mathsf{Gen}^{\mathcal{A},\sigma}, \mathsf{Eval}\right)$ be a PCF for VOLE defined by $\mathsf{interface} = \{\lambda, \mathcal{X}, \mathbb{F}_{p^r}\}$.

Strong Correlated Pseudorandomness game with oracle $\mathcal{O}_b^{\mathsf{pr}}(\cdot)$:

$\underline{\mathsf{Exp}^{\mathsf{pr}}_{\mathcal{A},b}(1^\lambda, \Delta)}$:

$\mathcal{Q} := \emptyset$
$(\mathsf{k}_0^{\mathsf{PCF}}, \mathsf{k}_1^{\mathsf{PCF}}) \leftarrow \mathsf{Gen}^{\mathcal{A},\perp}(1^\lambda, \Delta)$
$b' \leftarrow \mathcal{A}^{\mathcal{O}_b^{\mathsf{pr}}(\cdot)}(1^\lambda)$
Output b'

$\underline{\mathcal{O}_b^{\mathsf{pr}}(x \in \mathcal{X})}$:

If $(x, [\![y]\!]) \in \mathcal{Q}$**: Return** $[\![y]\!]$
If $b = 0$**:** $[\![y]\!] \xleftarrow{\$} \mathsf{VOLE}(\Delta)$
Else $(b = 1)$**:**
  $[\![y]\!]_0 := \mathsf{Eval}(0, \mathsf{k}_0^{\mathsf{PCF}}, x)$
  $[\![y]\!]_1 := \mathsf{Eval}(1, \mathsf{k}_1^{\mathsf{PCF}}, x)$
$\mathcal{Q} := \mathcal{Q} \cup \{(x, [\![y]\!])\}$
Return $[\![y]\!]$

Strong PCF Security game with oracle $\mathcal{O}_{b,\sigma}^{\mathsf{sec}}(\cdot)$:
If $\sigma = 1$ *oracle outputs* $\mathsf{VOLE}(\Delta^*, \cdot)$ *for* $\Delta^* \leftarrow \mathcal{A}$.
If $\sigma = 0$ *oracle outputs* $\mathsf{VOLE}(\Delta, \cdot)$ *and* $\mathcal{A}$ *gets* $\mathsf{k}_0^{\mathsf{PCF}}$ *from* $\mathsf{Gen}(1^\lambda, \Delta)$ *or* $\mathsf{Gen}(1^\lambda, \Delta')$.

$\underline{\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},b}(1^\lambda, \Delta, \Delta')}$:

$\mathcal{Q} := \emptyset$.
$\sigma \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}_{\mathsf{PCF}})$
If $\sigma = 1$**:**
  $\Delta^* \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}_{\mathsf{PCF}}, 1)$.
Else:
  $\Delta^* := \Delta$ **if** $b = 1$ **else** Δ'.
$(\mathsf{k}_0^{\mathsf{PCF}}, \mathsf{k}_1^{\mathsf{PCF}}) \leftarrow \mathsf{Gen}^{\mathcal{A},\sigma}(1^\lambda, \Delta^*)$
$b' \leftarrow \mathcal{A}^{\mathcal{O}_{b,\sigma}^{\mathsf{sec}}(\cdot)}(\sigma, \mathsf{k}_\sigma^{\mathsf{PCF}})$
Output b'

$\underline{\mathcal{O}_{b,0}^{\mathsf{sec}}(x \in \mathcal{X})}$:

If $(x, [\![y]\!]_1) \in \mathcal{Q}$**:**
  Return $[\![y]\!]_1$
$[\![y]\!]_0 \leftarrow \mathsf{Eval}(0, \mathsf{k}_0^{\mathsf{PCF}}, x)$
If $b = 0$**:**
  $[\![y]\!]_1 \leftarrow \mathsf{VOLE}(\Delta, [\![y]\!]_0)$
Else $(b = 1)$**:**
  $[\![y]\!]_1 \leftarrow \mathsf{Eval}(1, \mathsf{k}_1^{\mathsf{PCF}}, x)$
$\mathcal{Q} := \mathcal{Q} \cup \{(x, [\![y]\!]_1)\}$
Return $[\![y]\!]_1$

$\underline{\mathcal{O}_{b,1}^{\mathsf{sec}}(x \in \mathcal{X})}$:

If $(x, [\![y]\!]_0) \in \mathcal{Q}$**:**
  Return $[\![y]\!]_0$
$[\![y]\!]_1 \leftarrow \mathsf{Eval}(1, \mathsf{k}_1^{\mathsf{PCF}}, x)$
If $b = 0$**:**
  $[\![y]\!]_0 \leftarrow \mathsf{VOLE}(\Delta^*, [\![y]\!]_1)$
Else $(b = 1)$**:**
  $[\![y]\!]_0 \leftarrow \mathsf{Eval}(0, \mathsf{k}_0^{\mathsf{PCF}}, x)$
$\mathcal{Q} := \mathcal{Q} \cup \{(x, [\![y]\!]_0)\}$
Return $[\![y]\!]_0$

7.2 The Recursive LPN-Based PCF for VOLE of [BCMRS25]

We now re-state the PCF for VOLE construction of [BCMRS25], that is based on the sparse learning noise with parity (sparse LPN), in terms of our new definition of PCF for VOLE under adversarial influence. We make the following slight modifications to the original construction:

– The key generation algorithm is executed in interaction with an adversary that can corrupt one of the parties.

Theorem 7. *The protocol of Figure 11, defined by interface* $\{\lambda, \mathcal{X}, \mathbb{F}_{p^r}\}$ *is a secure pseudorandom function for VOLE under adversarial influence (as per Definition 20) for* $\mathsf{spec} = \{L, t^{(1)}, \ldots, t^{(L)}, m^{(0)}, \ldots, m^{(L)}, k^{(1)}, \ldots, k^{(L)}, k^B, \mathsf{DPF}, \mathsf{H}_A^{(1)}, \ldots, \mathsf{H}_A^{(L)}, \mathsf{H}_B, \ldots\}$ *under the following assumptions:*

1. $\mathsf{H}_A^{(\ell)}$ *for all* $\ell \in [L]$ *and* H_B *are random oracles,*
2. $\mathsf{SparseLPN}(k^{(\ell)}, m^{(\ell-1)}, m^{(\ell)}, \mathsf{Reg}_{t^{(\ell)}}, \mathbb{F}_{p^r})$ *with static leakage is hard for* $\ell \in [L-1]$ *,*
3. $\mathsf{FilteredSparseLPN}(k^{(L)}, m^{(L-1)}, Q_{\mathcal{O}}, D^{\mathsf{sp}}_{\mathsf{Reg}_{t^{(L)}},\beta}, \mathbb{F}_{p^r})$ *with static leakage is hard where* $Q_{\mathcal{O}}$ *is a bound on the number of queries to either oracles* $\mathcal{O}_1^{\mathsf{pr}}$ *or* $\mathcal{O}_1^{\mathsf{sec}}$ *(see Figure 10), and*
4. DPF *is a secure DPF with reusable setup for the class of point functions* $\mathbb{N} \to \mathbb{F}_{p^r}$ *with pseudorandom outputs in* $\mathbb{F}_{p^r}$*.*

Proof. The proof of the theorem proceeds along the same line as the proof of Theorem 1 of [BCMRS25]. In the following, we outline the differences in the protocol and in the proof of the theorem.

Adversarial Influence. The main difference between the PCF of Figure 11 and the PCF of [BCMRS25] is the key generation algorithm which can be influenced by a corrupted party.In particular, for $b \in \{0, 1\}$, a corrupted party P_b can influence $\mathsf{PCF.Gen}^{\mathcal{A},b}$, where $\mathcal{A} = P_b$ as follows:

– $\mathcal{A}$ chooses its share of the seed VOLE $[\![\mathbf{s}^{(b)}]\!]$.
// This influence does not affect any of the properties.
– If $b = 0$, then $\mathcal{A} = P_0$ chooses the error vectors $\mathbf{e}^{(\ell)}$ for $\ell \in [L]$.
// This influence does not affect any of the properties.
– $\mathcal{A}$ influences $\mathsf{DPF.Setup}$ and $\mathsf{DPF.Gen}$.
// Taken into account by the properties of the DPF.
– If $b = 1$, the $\mathcal{A} = P_1$ performs a selective failure attack by trying to guess the non-zero indices in each error block.
// Taken into account by the static leakage of the sparse LPN.

DPF with reusable setup. Contrary to [BCMRS25] that uses L distributed *multi-point* function (DMPF) to distribute $\Delta \cdot \mathbf{e}^{(\ell)}$ for $\ell \in [L]$, we use a DPF with reusable setup (as per Definition 17). By leveraging the $t^{(\ell)}$-regularity of each error vector $\mathbf{e}^{(\ell)}$, we run the key generation algorithm of the DPF $t^{(\ell)}$ times for each level $\ell \in [L]$, to distribute each of the $t^{(\ell)}$ blocks of $\mathbf{e}^{(\ell)}$. At each level $\ell \in [L]$, we use an encoding $\mathsf{idx}^{(\ell)}$ that assigns every input of the domain of that level $[m^{(\ell)}]$ to one of the blocks and an index within that block.

The evaluation algorithm is essentially the same algorithm as that of [BCMRS25], where the DMPFs are replaced by a DPF with reusable setup.

Correctness. The proof of correctness is the same as that of [BCMRS25] by resorting to the correctness of DPF.

Pseudorandomness. The proof of [BCMRS25] used the hardness of SparseLPN (with level-dependent parameters) and the pseudorandomness of the DMPF outputs to prove that the outputs of the recursive evaluation function at each level is indistinguishable from random VOLE-correlated pairs of the same dimensions.

Here also, to prove the pseudorandomness at each level $\ell \in [L-1]$, we resort to the hardness of the $\mathsf{SparseLPN}(k^{(\ell)}, m^{(\ell-1)}, m^{(\ell)}, \mathsf{Reg}_{t^{(\ell)}}, \mathbb{F}_{p^r})$ and the pseudorandomness of the outputs of the DPF with reusable setup. For the last level, we assume the hardness of $\mathsf{FilteredSparseLPN}(k^{(L)}, m^{(L-1)}, Q_{\mathcal{O}}, D^{\mathsf{sp}}_{\mathsf{Reg}_{t^{(L)}},\beta}, \mathbb{F}_{p^r})$,

in order to take into account that an adversary, having access to $\mathsf{H}_A^{(L)}$ and H_B can query these oracles more times than it sees the output of the evaluation algorithm.

Note that none of the adversarial influence in the key generation listed above affects the pseudorandomness. This is because pseudorandomness in only required against an outsider adversary that does not corrupt any of the parties, i.e., $\mathsf{PCF}.\mathsf{Gen}^{\mathcal{A},b}$ is run with $b = \perp$.

Security. The security of the construction against party 0 (i.e., when $b = 0$) is straightforward and follows from the correctness of the scheme. The security against party 1 is more involved but follows exactly the same structure as that of [BCMRS25]. Essentially for each level $\ell \in [L]$, (1) the hardness of $\mathsf{SparseLPN}(k^{(\ell)}, m^{(\ell-1)}, m^{(\ell)}, \mathsf{Reg}_{t^{(\ell)}}, \mathbb{F}_{p^r})$ with static leakage and (2) the security of DPF with adversarial influence ensure the security against party 1. The static leakage of the assumption accounts for the selective failure attack that party 1 can perform in the key generation. For the last level, we assume the hardness of $\mathsf{FilteredSparseLPN}(k^{(L)}, m^{(L-1)}, Q_{\mathcal{O}}, D^{\mathsf{sp}}_{\mathsf{Reg}_{t^{(L)}},\beta}, \mathbb{F}_{p^r})$, with static leakage in order to take into account the selective failure attack as well as the unlimited access of the adversary to the random oracles.

Moreover, note that the two first items of the list of adversarial influence in the key generation, i.e., the party choosing its share of the seed VOLE or the error vector, do not affect the security. More precisely, for any $b \in \{0, 1\}$, we want the security to hold *against* party b that $\mathcal{A}$ corrupts. Therefore, the adversary choosing its own input does not inject any insecurity. □

Fig. 11: PCF for VOLE of [BCMRS25] under Adversarial Influence (Recursive-LPN-PCF)

The construction is defined by $\mathsf{interface} = \{\lambda, \mathcal{X}, \mathbb{F}_{p^r}\}$, where
- $\lambda \in \mathbb{N}$ is a security parameter.
- $\mathcal{X} = \mathcal{X}(\lambda)$ is the domain of the PCF.
- $\mathbb{F}_{p^r} = \mathbb{F}_{p^r}(\lambda)$ is a finite field that is the range of the PCF.

Given $\mathsf{interface}$, let spec contain the following:
- Let $L, t^{(1)}, \ldots, t^{(L)}, m^{(0)}, \ldots, m^{(L)}, k^{(1)}, \ldots, k^{(L)}, k^B \in \mathbb{N}$, such that $t^{(\ell)} < m^{(\ell)}$, $m^{(0)} < \cdots < m^{(L)}$, $k^{(\ell)} \leq m^{(\ell-1)}$ for $\ell \in [L]$, $k^B \leq m^{(L)}$.
- Let $\mathsf{DPF} = (\mathsf{DPF}.\mathsf{Setup}, \mathsf{DPF}.\mathsf{Gen}, \mathsf{DPF}.\mathsf{Eval})$ be a distributed point function with reusable setup for the class of point functions $\mathbb{N} \to \mathbb{F}_{p^r}$ specified by $\{\lambda, \{m^{(\ell)}/t^{(\ell)}\}_{\ell\in[L]}, \mathbb{F}_{p^r}\}$.
- Let $\mathsf{H}_A^{(1)}, \ldots, \mathsf{H}_A^{(L)}$ and H_B be random oracles such that

$$\mathsf{H}_A^{(\ell)} \colon [m^{(\ell)}] \to \mathcal{V}_{k^{(\ell)}}^{m^{(\ell-1)}}(\mathbb{F}_p) \text{ for } \ell \in [L-1], \qquad \mathsf{H}_A^{(L)} \colon \mathcal{X} \to \mathcal{V}_{k^{(L)}}^{m^{(L-1)}}(\mathbb{F}_p) \qquad \mathsf{H}_B \colon \mathcal{X} \to \mathcal{V}_{k^B}^{m^{(L)}}(\mathbb{F}_p).$$

- Let $\mathsf{idx}^{(\ell)} \colon [m^{(\ell)}] \to [t^{(\ell)}] \times [m^{(\ell)}/t^{(\ell)}]$ defined by $j \mapsto (\lceil j/t^{(\ell)} \rceil, j \bmod t^{(\ell)})$.

$\mathsf{PCF}.\mathsf{Gen}^{\mathcal{A},b}(1^\lambda, \Delta \in \mathbb{F}_{p^r})$**:**

1. If $b = \perp$, $[\![\mathbf{s}^{(0)}]\!] \leftarrow \mathsf{VOLE}_\Delta^{m^{(0)}}$. If $b = 0$, receive $[\![\mathbf{s}^{(0)}]\!]_0 \leftarrow \mathcal{A}$. If $b \in \{\perp, 0\}$, compute $[\![\mathbf{s}^{(0)}]\!]_1 \leftarrow \mathsf{VOLE}_\Delta^{m^{(0)}}.\mathsf{RSample}(1^\lambda, 0, [\![\mathbf{s}^{(0)}]\!]_0)$. If $b = 1$, receive $\langle \mathbf{s}^{(0)} \rangle_1 \leftarrow \mathcal{A}$, set $[\![\mathbf{s}^{(0)}]\!]_1 := (\Delta, \langle \mathbf{s}^{(0)} \rangle_1)$, and compute $[\![\mathbf{s}^{(0)}]\!]_0 \leftarrow \mathsf{VOLE}_\Delta^{m^{(0)}}.\mathsf{RSample}(1^\lambda, 1, [\![\mathbf{s}^{(0)}]\!]_1)$.
2. Run $(\mathsf{st}_0^{\mathsf{DPF}}, \mathsf{st}_1^{\mathsf{DPF}}) \leftarrow \mathsf{DPF}.\mathsf{Setup}^{\mathcal{A},b}(1^\lambda)$, and send $\mathsf{st}_b^{\mathsf{DPF}}$ to $\mathcal{A}$.
3. For $\ell \in [L]$:
   (a) If $b = 0$, receive $\mathbf{e}^{(\ell)} = \big((\alpha_i^{(\ell)}, \beta_i^{(\ell)})\big)_{i\in[t]} \leftarrow \mathcal{A}$. Otherwise sample $\mathbf{e}^{(\ell)} = \big((\alpha_i^{(\ell)}, \beta_i^{(\ell)})\big)_{i\in[t]} \leftarrow \mathsf{Reg}_t(m^{(\ell)}, \mathbb{F}_p)$.
   (b) If $b = 1$, receive $I \subseteq [m^{(\ell)}/t^{(\ell)}]^t$ from $\mathcal{A}$, and return $\perp$ if $\boldsymbol{\alpha}^{(\boldsymbol{\ell})} \notin I$.
   (c) For $i \in [t^{(\ell)}]$ run

$$k \leftarrow \mathsf{DPF}.\mathsf{Gen}^{\mathcal{A},b}\Big((\mathsf{st}_0^{\mathsf{DPF}}, \mathsf{st}_1^{\mathsf{DPF}}), \frac{m^{(\ell)}}{t^{(\ell)}}, (\alpha_i^{(\ell)} - 1, \Delta \cdot \beta_i^{(\ell)})\Big).$$

   If $k = \perp$, return $\perp$; Otherwise parse $(\mathsf{k}_{0,(\ell,i)}^{\mathsf{DPF}}, \mathsf{k}_{1,(\ell,i)}^{\mathsf{DPF}}) := k$.

(d) If $b \neq \perp$, return $\mathsf{k}^{\mathsf{DMPF}}_{b,(\ell,i)}$ to $\mathcal{A}$.

4. Output $\mathsf{k}^{\mathsf{PCF}}_0 = \Big(\llbracket \mathbf{s}^{(0)} \rrbracket_0, \mathsf{st}^{\mathsf{DPF}}_0, \big(\mathbf{e}^{(\ell)}, (\mathsf{k}^{\mathsf{DMPF}}_{0,(\ell,i)})_{i\in[t^{(\ell)}]}\big)_{\ell\in[L]}\Big)$,
$\mathsf{k}^{\mathsf{PCF}}_1 = \Big(\llbracket \mathbf{s}^{(0)} \rrbracket_1, \mathsf{st}^{\mathsf{DPF}}_1, \big((\mathsf{k}^{\mathsf{DMPF}}_{1,(\ell)})_{i\in[t^{(\ell)}]}\big)_{\ell\in[L]}\Big)$.

$\mathsf{PCF}.\mathsf{Eval}(\sigma, \mathsf{k}^{\mathsf{PCF}}_\sigma, x \in \mathcal{X})$**:**

1. Compute $\mathbf{A}^{(L)}_x \leftarrow \mathsf{H}^{(L)}_A(x)$ and $\mathbf{B}_x \leftarrow \mathsf{H}_B(x)$.
2. For $(j,\cdot) \in \mathbf{A}^{(L)}_x$ compute $\llbracket s^{(L-1)}_j \rrbracket_\sigma \leftarrow \mathsf{EvalRec}(\sigma, \mathsf{k}^{\mathsf{PCF}}_\sigma, L-1, j)$.
3. For $(j,\cdot) \in \mathbf{B}_x$ compute $\langle e^{(L)}_j \rangle_\sigma \leftarrow \mathsf{DPF}.\mathsf{Eval}(\sigma, \mathsf{st}^{\mathsf{DPF}}_\sigma, \mathsf{k}^{\mathsf{DPF}}_{\sigma,(L,j_1)}, j_2)$ with $(j_1, j_2) \leftarrow \mathsf{idx}^{(L)}(j)$.[a]
4. Compute and return

$$\llbracket s^{(L)}_x \rrbracket_\sigma = \sum_{(j,a_j)\in\mathbf{A}^{(L)}_x} a_j \cdot \llbracket s^{(L-1)}_j \rrbracket_\sigma + \sum_{(j,b_j)\in\mathbf{B}_x} b_j \cdot \llbracket e^{(L)}_j \rrbracket_\sigma \ .$$

$\mathsf{EvalRec}(\sigma, \mathsf{k}^{\mathsf{PCF}}_\sigma, \ell \in [0, L-1], i \in [m^{(\ell)}])$**:**

1. If $\ell = 0$, return $\llbracket s^{(0)}_i \rrbracket_\sigma$.
2. Otherwise compute $\mathbf{A}^{(\ell)}_i \leftarrow \mathsf{H}^{(\ell)}_A(i)$.
3. For $(j,\cdot) \in \mathbf{A}^{(\ell)}_i$ compute $\llbracket s^{(\ell-1)}_j \rrbracket_\sigma \leftarrow \mathsf{EvalRec}(\sigma, \mathsf{k}^{\mathsf{PCF}}_\sigma, \ell-1, j)$.
4. Compute $\langle e^{(\ell)}_i \rangle_\sigma \leftarrow \mathsf{DPF}.\mathsf{Eval}(\sigma, \mathsf{st}^{\mathsf{DPF}}_\sigma, \mathsf{k}^{\mathsf{DPF}}_{\sigma,(L,i_1)}, i_2)$ with $(i_1, i_2) \leftarrow \mathsf{idx}^{(\ell)}(i)$.
5. Compute and return

$$\llbracket s^{(\ell)}_i \rrbracket_\sigma = \sum_{(j,a_j)\in\mathbf{A}^{(\ell)}_i} a_j \cdot \llbracket s^{(\ell-1)}_j \rrbracket_\sigma + \llbracket e^{(\ell)}_i \rrbracket_\sigma .$$

[a] Recall that $\llbracket e^{(\ell)}_j \rrbracket_0 = (e^{(\ell)}_j, \langle e^{(\ell)}_j \rangle_0)$ and $\llbracket e^{(\ell)}_j \rrbracket_1 = (\Delta, \langle e^{(\ell)}_j \rangle_1)$ for $\ell \in [L]$.

8 An Arithmetization-Friendly Correlation Robust Hash Function

The power-DDH-based PPRFs that we consider require a mapping from group elements into the field over which our VOLE correlation is defined. To reuse the setup across different PPRFs, this map should be a correlation robust hash function [IKNP03] (CRH, Definition 5, 6). As we evaluate the hash function over secret shared data as part of our distributed setup, we require a construction that can be efficiently evaluated in such a setting, as discussed in our Technical Overview (see **A DKG-friendly correlation-robust hash over chains of curves**).

We consider the correlation $C(S, z) := z \cdot S$ where $S \in \mathbb{G}$ and $z \in \mathbb{F}_p$ for a cryptographic group $\mathbb{G}$ of order p. This correlation is elsewhere referred to as "exponential" in $\mathbb{G}$ [BCMPR24; CHHK25] but is herein referred to as "multiplicative" in $\mathbb{G}$ as our groups are written additively. The weak and strong variants of correlation robustness for multiplicative correlations over the group $\mathbb{G}$, denoted $\mathbb{G}$-wCR and $\mathbb{G}$-CR, are presented in Definition 6.

Our goal is a $\mathbb{G}$-CR Hash function, which we prove by way of constructing a candidate PRF. Referencing the relationship between sCR and PRF (resp. wCR and wPRF) in Section 3.3, a function $\mathsf{H}_\lambda : \mathbb{G} \to Y_\lambda$ is $\mathbb{G}$-CR if the composed function $\mathsf{F}_H(\cdot) : \mathbb{F}_p \to Y_\lambda$ that maps $z \mapsto \mathsf{H}(z \cdot H)$ is a PRF with key $H \in \mathbb{G}$. In this section we present a function $\mathsf{F}_H(\cdot)$ in Definition 21 that we prove is a PRF with Theorem 9, which implies the $\mathbb{G}$-sCR hash function H in Lemma 8. Our new $\mathbb{G}$-CR hash function admits efficient evaluation over secret shared input, as demonstrated by the protocol $\Pi_{\mathsf{DPF.Hash}}$ in Figure 19.

Definition 21 (Cyclic Multiplication). Let $(\mathbb{G}, \overline{\mathbb{G}})$ be a 2-cycle of elliptic curve groups such that the group $\mathbb{G} = \langle G \rangle$ has order p over the field $\mathbb{F}_q$, and the group $\overline{\mathbb{G}} = \langle \overline{G} \rangle$ has order q over the field $\mathbb{F}_p$. For $H \in \mathbb{G}$ and $\rho \in \mathbb{F}_p$ we define $\mathsf{F}_H(\cdot)$ and $\mathsf{F}_\rho(\cdot)$, respectively, as below.

$$\mathsf{F}_H \colon \mathbb{F}_p \to \mathbb{F}_p \qquad z \mapsto ((z \cdot H).\mathsf{x} \cdot \overline{G}).\mathsf{x}$$
$$\mathsf{F}_\rho \colon \mathbb{F}_p \to \mathbb{F}_p \qquad z \mapsto ((z \cdot \rho \cdot G).\mathsf{x} \cdot \overline{G}).\mathsf{x}$$

Lemma 8 (A $\mathbb{G}$-sCR Hash). *Let $(\mathbb{G}, \overline{\mathbb{G}})$ be a 2-cycle of elliptic curve groups such that the group $\mathbb{G} = \langle G \rangle$ has order p over the field $\mathbb{F}_q$, and the group $\overline{\mathbb{G}} = \langle \overline{G} \rangle$ has order q over the field $\mathbb{F}_p$. The function $\mathsf{H}(\cdot)$, defined below, is a $\mathbb{G}$-sCR hash function.*

$$\mathsf{H} \colon \mathbb{G} \to \mathbb{F}_p \qquad P \mapsto \left((P.\mathsf{x}) \cdot \overline{G}\right).\mathsf{x}$$

Theorem 9 (Cyclic Multiplication is a PRF in the EC-GGM). *The function $\mathsf{F}_H : \mathbb{F}_p \to \mathbb{F}_p$ in Definition 21 is a pseudorandom function in the EC-GGM for generic $\mathbb{G}$ and $\overline{\mathbb{G}}$.*

Proof. We define a sequence of games Game-n which runs an **Init** process that uniformly samples a random bit $b \xleftarrow{\$} \{0,1\}$ then executes the adversary $\mathcal{A}$ with access to a series of oracles. The adversary's task is to distinguish $b = 0$ from $b = 1$; $\mathcal{A}$ wins Game-n if it outputs $b' = b$, and $\mathcal{G}_n$ is a random variable that indicates $\mathcal{A}$ wins Game-n. Certain games raise flags when some set of conditions are met, and we use $\mathcal{F}_n^a$ as random variables that indicate that flag F-a was raised in Game-n.

Game-0: The first game is the actual PRF security game in the EC-GGM; the adversary $\mathcal{A}$ has access to oracles $(\mathsf{map}, \mathsf{add})$ for $\mathbb{G}$ and $(\overline{\mathsf{map}}, \overline{\mathsf{add}})$ for $\overline{\mathbb{G}}$, along with the PRF oracle $\mathcal{O}_b^{\mathsf{PRF}}$. The PRF oracle $\mathcal{O}_0^{\mathsf{PRF}}$ outputs the real PRF output (computed with the EC-GGM encodings), and $\mathcal{O}_1^{\mathsf{PRF}}$ outputs a random output in $\mathbb{F}_p$. We want to show that the adversary's advantage is negligible:

$$\Pr[\mathcal{G}_0] \leq \frac{1}{2} + \mathsf{negl}(\lambda). \tag{4}$$

We begin by assuming towards contradiction that there exists a PPT $\mathcal{A}$ and non-negligible function $\varepsilon(\lambda)$ such that:

$$\Pr[\mathcal{G}_0] = \frac{1}{2} + \varepsilon(\lambda). \tag{5}$$

Game-0 (real/ideal games)

Init

1. Sample $b \xleftarrow{\$} \{0,1\}$

2. $\rho \xleftarrow{\$} \mathbb{F}_p$ (PRF key)
3. $F\colon \mathbb{F}_p \to \mathbb{F}_p$ random map
4. $\pi\colon \mathbb{F}_p \to \mathbb{G}$ random map satisfying EC-GGM constraints (bijective, $\pi(0) = \mathcal{O}$, $\pi(-i) = -\pi(i)$)
5. $\overline{\pi}\colon \mathbb{F}_q \to \overline{\mathbb{G}}$ random map satisfying EC-GGM constraints (bijective, $\overline{\pi}(0) = \mathcal{O}$, $\overline{\pi}(-i) = -\overline{\pi}(i)$)

$\mathsf{map}(i)$
1. return $\pi(i)$

$\overline{\mathsf{map}}(i)$
1. return $\overline{\pi}(i)$

$\mathsf{add}(c_1, P_1, c_2, P_2)$
1. return $\pi(c_1 \cdot \pi^{-1}(P_1) + c_2 \cdot \pi^{-1}(P_2))$

$\overline{\mathsf{add}}(c_1, P_1, c_2, P_2)$
1. return $\overline{\pi}(c_1 \cdot \overline{\pi}^{-1}(P_1) + c_2 \cdot \overline{\pi}^{-1}(P_2))$

$\mathcal{O}_b^{\mathsf{PRF}}(z)$
1. if $b = 0$ (real)
   (a) $T_1 = \mathsf{map}(z \cdot \rho) \in \mathbb{G}$
   (b) $t_1 = T_1.\mathsf{x} \in \mathbb{F}_q$
   (c) $T_2 = \overline{\mathsf{map}}(t_1) \in \overline{\mathbb{G}}$
   (d) $y = T_2.\mathsf{x} \in \mathbb{F}_p$
   (e) return (z, y)
2. else if $b = 1$ (ideal)
   (a) $y := F(z)$
   (b) return (z, y)

Game-1: The next game behaves identically to Game-0 except that the EC-GGM encoding maps π, $\overline{\pi}$, and when $b = 1$ the random function F are sampled *lazily*, meaning that the outputs are defined upon query as opposed to at **Init**. The game is presented below, and the changes from Game-0 are highlighted.

Game-1 (lazy GGM and RO)

Init
1. Sample $b \xleftarrow{\$} \{0, 1\}$.
2. $\rho \xleftarrow{\$} \mathbb{F}_p$ (PRF key)
3. $F\colon \mathbb{F}_p \to \mathbb{F}_p$ empty map
4. $\pi\colon \mathbb{F}_p \to \mathbb{G}$ empty map except $\pi(0) = \mathcal{O}$, $\pi(1) = G$, $\pi(-1) = -G$
5. $\overline{\pi}\colon \mathbb{F}_q \to \overline{\mathbb{G}}$ empty map except $\overline{\pi}(0) = \mathcal{O}$, $\overline{\pi}(1) = \overline{G}$, $\overline{\pi}(-1) = -\overline{G}$

$\mathsf{map}(i)$
1. if $i \notin \mathrm{Dom}(\pi)$:
   (a) sample $P \in_R \mathbb{G} \setminus \mathrm{Im}(\pi)$
   (b) set $\pi(i) := P$, $\pi(-i) := -P$
2. return $\pi(i)$

$\overline{\mathsf{map}}(i)$
1. if $i \notin \mathrm{Dom}(\overline{\pi})$:
   (a) sample $P \in_R \overline{\mathbb{G}} \setminus \mathrm{Im}(\overline{\pi})$
   (b) set $\overline{\pi}(i) := P$, $\overline{\pi}(-i) := -P$
2. return $\overline{\pi}(i)$

$\mathsf{add}(c_1, P_1, c_2, P_2)$
1. for $j = 1, 2$, if $P_j \notin \mathrm{Im}(\pi)$:
   (a) sample $i_j \xleftarrow{\$} \mathbb{F}_p \setminus \mathrm{Dom}(\pi)$
   (b) set $\pi(i_j) := P_j$, $\pi(-i_j) := -P_j$
2. return $\mathsf{map}(c_1 \cdot \pi^{-1}(P_1) + c_2 \cdot \pi^{-1}(P_2))$

$\overline{\mathsf{add}}(c_1, P_1, c_2, P_2)$
1. for $j = 1, 2$, if $P_j \notin \mathrm{Im}(\overline{\pi})$:
   (a) sample $i_j \xleftarrow{\$} \mathbb{F}_q \setminus \mathrm{Dom}(\overline{\pi})$
   (b) set $\overline{\pi}(i_j) := P_j$, $\overline{\pi}(-i_j) := -P_j$
2. return $\overline{\mathsf{map}}(c_1 \cdot \overline{\pi}^{-1}(P_1) + c_2 \cdot \overline{\pi}^{-1}(P_2))$

$\mathcal{O}_b^{\mathsf{PRF}}(z)$

1. if $z \in \mathrm{Dom}(F)$, return $(z, y := F(z))$
2. if $b = 0$ (real)
   (a) $T_1 = \mathsf{map}(z \cdot \rho) \in \mathbb{G}$
   (b) $t_1 = T_1.\mathsf{x} \in \mathbb{F}_q$
   (c) $T_2 = \overline{\mathsf{map}}(t_1) \in \overline{\mathbb{G}}$
   (d) $y = T_2.\mathsf{x} \in \mathbb{F}_p$
   (e) set $F(z) := y$ and return (z, y)
3. else if $b = 1$ (ideal)
   (a) sample $y \xleftarrow{\$} \mathbb{F}_p$
   (b) set $F(z) := y$ and return (z, y)

As stated, Game-1 behaves identically to Game-0 except that the maps π, $\overline{\pi}$, and the outputs of $\mathcal{O}_1^{\mathsf{PRF}}$ are sampled lazily. The output of $\mathcal{O}_0^{\mathsf{PRF}}$ is unchanged, as the PRF output is determined by the key ρ. The outputs of the oracles in Game-1 are identically distributed to those in Game-0, and therefore the advantage of $\mathcal{A}$ is identical:

$$\Pr[\mathcal{G}_0] = \Pr[\mathcal{G}_1]. \tag{6}$$

Game-2: The next game behaves as Game-1 except that the outputs of $\mathsf{map}, \overline{\mathsf{map}}$ are sampled with replacement. Flag F-1 is raised in the events that a group element P already in the image $\mathrm{Im}(\pi)$ or $\mathrm{Im}(\overline{\pi})$, respectively, is sampled *again* by either of the lazy EC-GGM oracles $\mathsf{map}, \overline{\mathsf{map}}$. This event represents a collision; the flag is raised if a lazily sampled encoding is not injective. The oracles are presented below, where the changes from Game-1 are highlighted.

Game-2 (random sampling of encodings)

$\mathsf{map}(i)$
1. if $i \notin \mathrm{Dom}(\pi)$:
   (a) sample $P \in_R \mathbb{G}$, and raise F-1 if $P \in \mathrm{Im}(\pi)$
   (b) set $\pi(i) := P$, $\pi(-i) := -P$
2. return $\pi(i)$

$\overline{\mathsf{map}}(i)$
1. if $i \notin \mathrm{Dom}(\overline{\pi})$:
   (a) sample $P \in_R \overline{\mathbb{G}}$, and raise F-1 if $P \in \mathrm{Im}(\overline{\pi})$
   (b) set $\overline{\pi}(i) := P$, $\overline{\pi}(-i) := -P$
2. return $\overline{\pi}(i)$

As stated, Game-2 behaves identically to Game-1 except that the outputs of map and $\overline{\mathsf{map}}$ are sampled with replacement. If in Game-2 the encodings are lazily sampled as injective then flag F-1 is not raised and the distribution of the outputs of the oracles are identical to those in Game-1; conditioned on the flag not being raised, Game-2 is perfectly indistinguishable from Game-1.

$$\Pr[\mathcal{G}_1] = \Pr\left[\mathcal{G}_2 \middle| \neg\mathcal{F}_2^1\right] \tag{7}$$

From the law of total probability, we can express the success probability in Game-2 as the sum:

$$\begin{aligned}
\Pr[\mathcal{G}_2] &= \Pr[\mathcal{G}_2 \mid \mathcal{F}_2^1] \cdot \Pr[\mathcal{F}_2^1] + \Pr[\mathcal{G}_2 \mid \neg\mathcal{F}_2^1] \cdot (1 - \Pr[\mathcal{F}_2^1]) \\
&\geq \Pr[\mathcal{G}_2 \mid \neg\mathcal{F}_2^1] \cdot (1 - \Pr[\mathcal{F}_2^1]) \\
&= \Pr[\mathcal{G}_1] \cdot (1 - \Pr[\mathcal{F}_2^1]).
\end{aligned} \tag{8}$$

We now bound $\Pr\left[\mathcal{F}_2^1\right]$, the probability that flag F-1 is raised in Game-2. Let $q_{\mathcal{O}}$ denote the number of queries to an oracle $\mathcal{O}$. A query to map (resp. $\overline{\mathsf{map}}$) can add one new point $P = \pi(i)$ and its inverse $-P = \pi(-i)$ to the map. A query to add (resp. $\overline{\mathsf{add}}$) indirectly queries map (resp. $\overline{\mathsf{map}}$) and therefore can similarly add one new point $P = \pi(c_1 \cdot \pi^{-1}(P_1) + c_2 \cdot \pi^{-1}(P_2))$ and its inverse to the map. A query to

$\mathcal{O}_0^{\mathsf{PRF}}$ indirectly queries map (resp. $\overline{\mathsf{map}}$) and therefore can similarly add one new point $P = \pi(z \cdot \rho)$ and its inverse to the map. In total, the number of random points sampled by map and $\overline{\mathsf{map}}$ are at most:

$$\begin{aligned} \mathsf{rnd}_{\mathsf{map}} &\leq q_{\mathsf{map}} + q_{\mathsf{add}} + (1-b) \cdot q_{\mathcal{O}_b^{\mathsf{PRF}}}, \\ \mathsf{rnd}_{\overline{\mathsf{map}}} &\leq q_{\overline{\mathsf{map}}} + q_{\overline{\mathsf{add}}} + (1-b) \cdot q_{\mathcal{O}_b^{\mathsf{PRF}}}. \end{aligned} \tag{9}$$

The size of the domain and image of the maps $\pi, \overline{\pi}$ are bounded by the number of oracle queries:

$$\begin{aligned} |\pi| &\leq 2 \cdot \left(q_{\mathsf{map}} + 3 \cdot q_{\mathsf{add}} + (1-b) q_{\mathcal{O}_b^{\mathsf{PRF}}} + 1 \right) + 1 \\ |\overline{\pi}| &\leq 2 \cdot \left(q_{\overline{\mathsf{map}}} + 3 \cdot q_{\overline{\mathsf{add}}} + (1-b) q_{\mathcal{O}_b^{\mathsf{PRF}}} + 1 \right) + 1 \end{aligned} \tag{10}$$

corresponding to the:

- 3 starting points $\mathcal{O}$, and $G, -G$,
- $2\left(q_{\mathsf{map}} + q_{\mathsf{add}} + (1-b) \cdot q_{\mathcal{O}_b^{\mathsf{PRF}}}\right)$ points $Q, -Q$ sampled by map, and
- $4q_{\mathsf{add}}$ adversarially-input points $P_1, -P_1, P_2, -P_2$.

The probability that flag F-1 is raised in Game-2, computed in Eq 11, is then negligible when $p, q = \Omega(2^{2\lambda})$.

$$\begin{aligned} \Pr[\mathcal{F}_2^1] \leq & \frac{\mathsf{rnd}_{\mathsf{map}} \cdot |\pi|}{p} + \frac{\mathsf{rnd}_{\overline{\mathsf{map}}} \cdot |\overline{\pi}|}{q} \\ \overset{(9,10)}{\leq} & \frac{(q_{\mathsf{map}} + q_{\mathsf{add}} + (1-b) \cdot q_{\mathcal{O}_b^{\mathsf{PRF}}}) \cdot (2q_{\mathsf{map}} + 6q_{\mathsf{add}} + (1-b) \cdot 2q_{\mathcal{O}_b^{\mathsf{PRF}}} + 3)}{p} \\ & + \frac{(q_{\overline{\mathsf{map}}} + q_{\overline{\mathsf{add}}} + (1-b) \cdot q_{\mathcal{O}_b^{\mathsf{PRF}}}) \cdot (2q_{\overline{\mathsf{map}}} + 6q_{\overline{\mathsf{add}}} + (1-b) \cdot 2q_{\mathcal{O}_b^{\mathsf{PRF}}} + 3)}{q} \end{aligned} \tag{11}$$

Game-3: The next game behaves exactly as Game-2 except that the preimage of the inputs to $\mathsf{add}, \overline{\mathsf{add}}$ are sampled with replacement. Flag F-2 is raised in the event that a scalar i_j already in the domain $\mathrm{Dom}(\pi)$ or $\mathrm{Dom}(\overline{\pi})$ is sampled *again* by either of the lazy EC-GGM oracles $\mathsf{add}, \overline{\mathsf{add}}$. This event represents a collision; the flag is raised if a lazily sampled encoding is not a well defined function. The oracles are presented below, where the changes from Game-2 are highlighted.

Game-3 (random sampling of preimages)

$\mathsf{add}(c_1, P_1, c_2, P_2)$

1. for $j = 1, 2$, if $P_j \notin \mathrm{Im}(\pi)$:
   (a) sample $i_j \xleftarrow{\$} \mathbb{F}_p$, and raise F-2 if $i_j \in \mathrm{Dom}(\pi)$
   (b) set $\pi(i_j) := P_j$, $\pi(-i_j) := -P_j$ (overwrites possible previous definitions)
2. return $\mathsf{map}(c_1 \cdot \pi^{-1}(P_1) + c_2 \cdot \pi^{-1}(P_2))$

$\overline{\mathsf{add}}(c_1, P_1, c_2, P_2)$

1. for $j = 1, 2$, if $P_j \notin \mathrm{Im}(\overline{\pi})$:
   (a) sample $i_j \xleftarrow{\$} \mathbb{F}_q$, and raise F-2 if $i_j \in \mathrm{Dom}(\overline{\pi})$
   (b) set $\overline{\pi}(i_j) := P_j$, $\overline{\pi}(-i_j) := -P_j$ (overwrites possible previous definitions)
2. return $\overline{\mathsf{map}}(c_1 \cdot \overline{\pi}^{-1}(P_1) + c_2 \cdot \overline{\pi}^{-1}(P_2))$

As stated, Game-3 is identical to Game-2 except that the preimages of π and $\overline{\pi}$ are sampled with replacement. If a duplicate preimage is not sampled then the outputs of the oracles are identically distributed to those in Game-2; conditioned on the flag F-2 not being raised, Game-3 is perfectly indistinguishable from Game-2.

$$\Pr[\mathcal{G}_2] = \Pr[\mathcal{G}_3 \mid \neg \mathcal{F}_3^2]. \tag{12}$$

As oracles map and $\overline{\mathsf{map}}$ are are the same in Game-2 and Game-3, but the size of the maps in Game-3 are less than in Game-2, we have that F-1 occurs with probability:

$$\Pr[\mathcal{F}_3^1] \leq \Pr[\mathcal{F}_2^1] \tag{13}$$

Similarly to the previous game hop, from the law of total probability we can express the success probability in Game-3 as a sum and conclude:

$$\Pr[\mathcal{G}_3] \geq \Pr[\mathcal{G}_2] \cdot (1 - \Pr[\mathcal{F}_3^2]). \tag{14}$$

We now bound $\Pr[\mathcal{F}_3^2]$, the probability that F-2 is raised in Game-3. A query to add (resp. $\overline{\mathsf{add}}$) can add two random points P_1, P_2 to the map π (resp. $\overline{\pi}$), and their inverses $-P_1, -P_2$. In total, the number of pairs added by add and $\overline{\mathsf{add}}$ are:

$$\mathsf{rnd}_{\mathsf{add}} \leq 2 \cdot q_{\mathsf{add}} \quad \text{and} \quad \mathsf{rnd}_{\overline{\mathsf{add}}} \leq 2 \cdot q_{\overline{\mathsf{add}}}. \tag{15}$$

The random preimages of these points can collide with any of the $|\pi|$ elements of $\mathrm{Dom}(\pi)$ (resp. $|\overline{\pi}|$ of $\mathrm{Dom}(\overline{\pi})$). The probability that F-2 is raised in Game-3 in Eq 16 is negligible when $p, q = \Omega(2^{2\lambda})$.

$$\begin{aligned}
\Pr[\mathcal{F}_3^2] \leq & \frac{\mathsf{rnd}_{\mathsf{add}} \cdot |\pi|}{p} + \frac{\mathsf{rnd}_{\overline{\mathsf{add}}} \cdot |\overline{\pi}|}{q} \\
\overset{(10,15)}{\leq} & \frac{2 \cdot q_{\mathsf{add}} \cdot (2q_{\mathsf{map}} + 6q_{\mathsf{add}} + (1-b) \cdot 2q_{\mathcal{O}_b^{\mathsf{PRF}}} + 3)}{p} \\
& + \frac{2 \cdot q_{\overline{\mathsf{add}}} \cdot (2q_{\overline{\mathsf{map}}} + 6q_{\overline{\mathsf{add}}} + (1-b) \cdot 2q_{\mathcal{O}_b^{\mathsf{PRF}}} + 3)}{q}
\end{aligned} \tag{16}$$

Game-4: Instead of sampling the secret values, as in Game-3, this game symbolically represents any secret values as formal variables throughout the execution. The symbols are:

- X_ρ: the PRF key introduced at **Init**,
- $(Y_z, \overline{Y}_z)$: symbolic pairs of variables introduced by $\mathcal{O}_0^{\mathsf{PRF}}$, where $\overline{Y}_z$ represents the projection of Y_z to the x-coordinate; for an evaluation of the PRF the pair $(z \cdot X_\rho, Y_z)$ is added to π, and the pair $(\overline{Y}_z, T_2)$ is added to $\overline{\pi}$.

In Game-4, the encodings π and $\overline{\pi}$ are defined as a map from linear polynomials in the formal variables above to the respective groups $\mathbb{G}$ and $\overline{\mathbb{G}}$. In this game after the adversary produces output the new **Collapse** is called to detect any error in simulation. Simulation is considered erroneous (to have failed) and the flag F-3 is raised if a random assignment of the formal variables above does not result in bijective maps π and $\overline{\pi}$. The oracles are presented below, where the changes from Game-3 are highlighted. The behavior of oracles map and $\overline{\mathsf{map}}$ are unchanged when queried on input a formal variable.

Game-4 (symbolization)

Init

1. Sample $b \xleftarrow{\$} \{0,1\}$.
2. $X_\rho :=$ fresh formal variable (symbolic PRF key)
3. $\pi \colon \mathbb{F}_p[X_\rho]^{\leq 1} \to \mathbb{G} \cup \{Y_z, \dots\}$ empty map except $\pi(0) = \mathcal{O}$, $\pi(1) = G$, $\pi(-1) = -G$
4. $\overline{\pi} \colon \mathbb{F}_q[\overline{Y}_z, \dots]^{\leq 1} \to \overline{\mathbb{G}}$ empty map except $\overline{\pi}(0) = \mathcal{O}$, $\overline{\pi}(1) = \overline{G}$, $\overline{\pi}(-1) = -\overline{G}$

$\mathcal{O}_b^{\mathsf{PRF}}(z)$

1. if $z \in \mathrm{Dom}(F)$, return $(z, y := F(z))$
2. if $b = 0$ (real)
   (a) $Y_z :=$ fresh formal variable
   (b) $\overline{Y}_z :=$ fresh formal variable
   (c) set $\pi(z \cdot X_\rho) := Y_z$ and $\pi(-z \cdot X_\rho) := -Y_z$
   (d) $T_2 = \overline{\mathsf{map}}(\overline{Y}_z) \in \overline{\mathbb{G}}$
   (e) $y = T_2.\mathsf{x} \in \mathbb{F}_p$
   (f) set $F(z) := y$ and return (z, y)
3. else if $b = 1$ (ideal)
   (a) sample $y \xleftarrow{\$} \mathbb{F}_p$
   (b) set $F(z) := y$ and return (z, y)

Collapse At this point for $j = (1-b) \cdot q_{\mathcal{O}_b^{\mathsf{PRF}}}$ the domain and image of the maps are defined as below:

$$\pi : \mathbb{F}_p[X_\rho]^{\leq 1} \to \mathbb{G} \cup \{Y_{z_1}, \ldots, Y_{z_j}\}$$

$$\overline{\pi} : \mathbb{F}_q[\overline{Y}_{z_1}, \ldots, \overline{Y}_{z_j}]^{\leq 1} \to \overline{\mathbb{G}}$$

1. Sample $\rho \xleftarrow{\$} \mathbb{F}_p$ for X_ρ, and for all $f(X_\rho) = z_i \cdot X_\rho \in \mathrm{Dom}(\pi)$:
   (a) raise F-3 if the evaluation $z_i \cdot \rho \in \mathrm{Dom}(\pi)$
   (b) replace $f(X_\rho)$ with $f(\rho)$ in $\mathrm{Dom}(\pi)$.
2. For $i \in [j]$:
   (a) sample $T_{z_i} \xleftarrow{\$} \mathbb{G}$ for Y_{z_i}
   (b) raise F-3 if $Y_{z_i} \in \mathrm{Im}(\pi)$
   (c) set $\overline{T}_{z_i} := T_{z_i}.\mathsf{x}$ for $\overline{Y}_{z_i}$
   (d) raise F-3 if $\overline{T}_{z_i} \in \mathrm{Dom}(\overline{\pi})$
   (e) for all $\overline{f}(\overline{Y}_{z_i}, \ldots, \overline{Y}_{z_j}) \in \mathrm{Dom}(\overline{\pi})$:
      i. raise F-3 if the projection $\overline{f}(\overline{T}_{z_i}, Y_{z_{i+1}} \ldots, \overline{Y}_{z_j}) \in \mathrm{Dom}(\overline{\pi})$
      ii. substitute $\overline{f}(\overline{T}_{z_i}, \overline{Y}_{z_{i+1}} \ldots, \overline{Y}_{z_j})$ for $\overline{f}(\overline{Y}_{z_i}, \ldots, \overline{Y}_{z_j})$ in $\mathrm{Dom}(\overline{\pi})$.

Oracle $\mathcal{O}_0^{\mathsf{PRF}}$ samples the output $T_2 \in \overline{\mathbb{G}}$ randomly and therefore $y := T_2.\mathsf{x} = F(z)$ is random for each new input z, just as with oracle $\mathcal{O}_1^{\mathsf{PRF}}$; in Game-4 the output of $\mathcal{O}_0^{\mathsf{PRF}}$ and $\mathcal{O}_1^{\mathsf{PRF}}$ are identically distributed conditioned on no flags being raised.

$$\Pr\left[\mathcal{G}_4 \middle| \neg(\mathcal{F}_4^1 \vee \mathcal{F}_4^2 \vee \mathcal{F}_4^3)\right] = \frac{1}{2}. \tag{17}$$

Eq 10 relating the sizes of the maps to the number of oracle queries holds in both Game-3 and Game-4. In Game-3 the maps π and $\overline{\pi}$ are bijective and uniformly random if neither F-1 nor F-2 is raised. In Game-4 the maps π and $\overline{\pi}$ are bijective and uniformly random if neither F-1 nor F-3 is raised. Conditioned on no flags being raised, the distribution of the output of oracles $\mathsf{map}, \mathsf{add}, \overline{\pi}, \overline{\mathsf{add}}$ is identical.

Unconditionally, the distribution of the output of $\mathcal{O}_1^{\mathsf{PRF}}$ is uniform in both Game-3 and Game-4. Now, consider oracle $\mathcal{O}_0^{\mathsf{PRF}}$. In Game-3 on input z oracle map is called on $(z \cdot \rho)$ which outputs $T_1 \in \mathbb{G}$, and F-1 is raised when $z \cdot \rho \notin \mathrm{Dom}(\pi)$ but the output $T_1 = \pi(i)$ for some $i \neq z \cdot \rho$. In Game-4 on input z the map $\pi(z \cdot X_\rho) \mapsto Y_z$ is defined as $z \cdot X_\rho \mapsto Y_z$ for formal variables X_ρ, Y_z which are substituted by $\rho \xleftarrow{\$} \mathbb{F}_p$ and $T_z \xleftarrow{\$} \mathbb{G}$ at Collapse, and F-3 is raised if there is some z_i such that the output $T_{z_i}.\mathsf{x} \in \mathrm{Dom}(\overline{\pi})$. Conditioned on no flags being raised, $\mathcal{O}_0^{\mathsf{PRF}}$ simply computes the random bijective encodings π and $\overline{\pi}$ in both Game-3 and Game-4.

In both games a flag is raised if the lazily defined maps $\pi, \overline{\pi}$ are not bijective. If no flags are raised then the maps are bijective and the games are perfectly indistinguishable; conditioned on none of flag F-1 nor F-2 nor F-3 being raised, Game-4 is perfectly indistinguishable from Game-3.

$$\frac{1}{2} \overset{(17)}{=} \Pr[\mathcal{G}_4 \mid \neg(\mathcal{F}_4^1 \vee \mathcal{F}_4^2 \vee \mathcal{F}_4^3)] = \Pr[\mathcal{G}_3 \mid \neg(\mathcal{F}_3^1 \vee \mathcal{F}_3^2)] \tag{18}$$

We bound the probability that any flag is raised in Game-4, starting with F-1. As oracles map and $\overline{\mathsf{map}}$ are the same in Game-3 and Game-4, but in Game-4 the former is not called by $\mathcal{O}_0^{\mathsf{PRF}}$ and the number of group elements in the images of the maps are potentially fewer, the probability of F-1 being raised in Game-4 is:

$$\Pr[\mathcal{F}_4^1] \leq \Pr[\mathcal{F}_3^1]. \tag{19}$$

Similarly, as oracles add and $\overline{\mathsf{add}}$ are the same in Game-3 and Game-4 but in Game-4 the number of scalars in the domain of the maps are potentially fewer the probability of F-2 being raised in Game-4 is:

$$\Pr[\mathcal{F}_4^2] \leq \Pr[\mathcal{F}_3^2]. \tag{20}$$

First, note that F-3 is never raised if $b = 1$.

$$\Pr[\mathcal{F}_4^3] = \Pr[\mathcal{F}_4^3 \mid b = 0] \cdot \Pr[b = 0] = \left(\frac{1}{2}\right) \cdot \Pr[\mathcal{F}_4^3 \mid b = 0] \tag{21}$$

We bound the probability that F-3 is raised when $b = 0$ in Eq 24. The number of formal variables introduced prior to collapse are in Eq 22. When collapse is called these variables are substituted with uniformly random values, and F-3 is raised if these values are redundant.

$$\begin{aligned} \mathsf{rndcol}_X &= 1 \text{ for } \rho \\ \mathsf{rndcol}_T &= q_{\mathcal{O}_0^{\mathsf{PRF}}} \text{ many } T_{z_i} \\ \mathsf{rndcol}_{\overline{T}} &= q_{\mathcal{O}_0^{\mathsf{PRF}}} \text{ many } \overline{T}_{z_i} = T_{z_i}.\mathsf{x} \end{aligned} \tag{22}$$

The flag F-3 is also raised if ρ is sampled such that $\rho = a \cdot z_i^{-1}$ for some $z_i \in \mathrm{Dom}(F)$ and $a \in \mathrm{Dom}(\pi)$, of which there are $q_{\mathcal{O}_0^{\mathsf{PRF}}}$ and $|\pi|$, respectively. Finally, at any $i \in [q_{\mathcal{O}_0^{\mathsf{PRF}}}]$ iteration of the loop over variables Y_{z_i}, the flag F-3 is raised if there is a polynomial in $\mathrm{Dom}(\overline{\pi})$ such that its projection $\overline{Y}_{z_i} \mapsto \overline{T}_{z_i}$ is in $\mathrm{Dom}(\overline{\pi})$. If there are two such linear equations $p_a, p_b \in \mathrm{Dom}(\overline{\pi})$ where $p_a := c_a\overline{Y}_{z_i} + F_a + f_a$ and $p_b := c_b\overline{Y}_{z_i} + F_b + f_b$ with free terms f_a, f_b respectively, such that their projections are equal, then:

$$c_a\overline{T}_{z_i} + F_a + f_a = c_b\overline{T}_{z_i} + F_b + f_b \Longrightarrow \overline{T}_{z_i} = \frac{f_b - f_a}{c_a - c_b} \text{ and } F_a = F_b, \tag{23}$$

where $c_a = c_b$ implies that $f_a = f_b$ and therefore $p_a = p_b$. We determine the probability that $\overline{T}_{z_i}$ is sampled such that there are such c_a, c_b, f_a, f_b. It is clear that each of the scalars c_a, c_b was input by $\mathcal{A}$ via a call to $q_{\overline{\mathsf{add}}}$. Each of the scalars f was added to the polynomial p, so there similarly must have been an adversarial input c, P to $q_{\overline{\mathsf{add}}}$ such that $f = c \cdot \overline{\pi}^{-1}(P)$. Therefore, each of c_a, c_b, f_a, f_b can take one of at most $2 \cdot q_{\overline{\mathsf{add}}}$ adversarially-input values, but $c_a \neq c_b$ and $f_a \neq f_b$. There are $\binom{2 \cdot q_{\overline{\mathsf{add}}}}{2}$ pairs (c_1, c_2) and (f_1, f_2), so this value squared—which is $q_{\overline{\mathsf{add}}}^2 \cdot (2 \cdot q_{\overline{\mathsf{add}}} - 1)^2$—options for the tuple of scalars (c_1, c_2, f_1, f_2).

$$\begin{aligned} \Pr[\mathcal{F}_4^3 \mid b = 0] &\leq \underbrace{\frac{(\mathsf{rndcol}_X + \mathsf{rndcol}_T) \cdot |\pi|}{p}}_{\rho \in \mathrm{Dom}(\pi),\, T_{z_i} \in \mathrm{Im}(\pi)} + \underbrace{\frac{\mathsf{rndcol}_{\overline{T}} \cdot |\overline{\pi}|}{q}}_{\overline{T}_{z_i} \in \mathrm{Dom}(\overline{\pi})} \\ &\quad + \underbrace{\frac{q_{\mathcal{O}_0^{\mathsf{PRF}}} \cdot |\pi|}{p}}_{(a, z_i) \,:\, \rho = a \cdot z_i^{-1}} + \underbrace{\frac{q_{\mathcal{O}_0^{\mathsf{PRF}}} \cdot \left(q_{\overline{\mathsf{add}}}^2 \cdot (2 \cdot q_{\overline{\mathsf{add}}} - 1)^2\right)}{q}}_{(c_a, c_b, f_a, f_b) \,:\, \overline{T}_{z_i} = \frac{f_a - f_b}{c_b - c_a}} \\ &= \left(\frac{(2 \cdot q_{\mathcal{O}_0^{\mathsf{PRF}}} + 1) \cdot |\pi|}{p}\right) \\ &\quad + \left(\frac{q_{\mathcal{O}_0^{\mathsf{PRF}}} \cdot |\overline{\pi}| + q_{\mathcal{O}_0^{\mathsf{PRF}}} \cdot \left(q_{\overline{\mathsf{add}}}^2 \cdot (2 \cdot q_{\overline{\mathsf{add}}} - 1)^2\right)}{q}\right) \\ &= \left(\frac{(2 \cdot q_{\mathcal{O}_0^{\mathsf{PRF}}} + 1) \cdot |\pi|}{p}\right) \\ &\quad + \left(\frac{q_{\mathcal{O}_0^{\mathsf{PRF}}} \cdot \left(q_{\overline{\mathsf{add}}}^2 \cdot (2 \cdot q_{\overline{\mathsf{add}}} - 1)^2 + |\overline{\pi}|\right)}{q}\right) \\ &\leq p^{-1} \cdot \left(2 \cdot q_{\mathcal{O}_0^{\mathsf{PRF}}} + 1\right) \cdot \left(2 \cdot q_{\mathsf{map}} + 6 \cdot q_{\mathsf{add}} + 2 \cdot q_{\mathcal{O}_0^{\mathsf{PRF}}} + 3\right) \\ &\quad + q^{-1} \cdot q_{\mathcal{O}_0^{\mathsf{PRF}}} \cdot \begin{pmatrix} q_{\overline{\mathsf{add}}}^2 \cdot (2 \cdot q_{\overline{\mathsf{add}}} - 1)^2 \\ +2 \cdot q_{\overline{\mathsf{map}}} + 6 \cdot q_{\overline{\mathsf{add}}} + 2 \cdot q_{\mathcal{O}_0^{\mathsf{PRF}}} + 3 \end{pmatrix}. \end{aligned} \tag{24}$$

We write the probability that $\mathcal{A}$ wins Game-4 as:

$$\begin{aligned}
\frac{1}{2} &\geq \Pr[\mathcal{G}_4 \mid \neg(\mathcal{F}_4^1 \vee \mathcal{F}_4^2 \vee \mathcal{F}_4^3)] \cdot \Pr[\neg(\mathcal{F}_4^1 \vee \mathcal{F}_4^2 \vee \mathcal{F}_4^3)] \\
&\overset{(18)}{=} \Pr[\mathcal{G}_3 \mid \neg\mathcal{F}_3^1 \wedge \neg\mathcal{F}_3^2] \cdot \Pr[\neg(\mathcal{F}_4^1 \vee \mathcal{F}_4^2 \vee \mathcal{F}_4^3)] \\
&= \left(\frac{\Pr[\mathcal{G}_3] - \Pr[\mathcal{G}_3 \mid \mathcal{F}_3^1 \vee \mathcal{F}_3^2] \cdot \Pr[\mathcal{F}_3^1 \vee \mathcal{F}_3^2]}{\Pr[\neg\mathcal{F}_3^1 \wedge \neg\mathcal{F}_3^2]}\right) \cdot \Pr[\neg(\mathcal{F}_4^1 \vee \mathcal{F}_4^2 \vee \mathcal{F}_4^3)] \\
&\geq \big(\Pr[\mathcal{G}_3] - \Pr[\mathcal{F}_3^1 \vee \mathcal{F}_3^2]\big) \cdot \Pr[\neg(\mathcal{F}_4^1 \vee \mathcal{F}_4^2 \vee \mathcal{F}_4^3)].
\end{aligned} \tag{25}$$

Combining all of the games we reach a contradiction:

$$\begin{aligned}
\frac{1}{2} &\overset{(25)}{\geq} \big(\Pr[\mathcal{G}_3] - \Pr[\mathcal{F}_3^1 \vee \mathcal{F}_3^2]\big) \cdot \Pr[\neg(\mathcal{F}_4^1 \vee \mathcal{F}_4^2 \vee \mathcal{F}_4^3)] \\
&\overset{(14)}{\geq} \big(\Pr[\mathcal{G}_2] \cdot \Pr[\neg\mathcal{F}_3^2] - \Pr[\mathcal{F}_3^1 \vee \mathcal{F}_3^2]\big) \cdot \Pr[\neg(\mathcal{F}_4^1 \vee \mathcal{F}_4^2 \vee \mathcal{F}_4^3)] \\
&\overset{(8)}{\geq} \big(\Pr[\mathcal{G}_1] \cdot \Pr[\neg\mathcal{F}_2^1] \cdot \Pr[\neg\mathcal{F}_3^2] - \Pr[\mathcal{F}_3^1 \vee \mathcal{F}_3^2]\big) \cdot \Pr[\neg(\mathcal{F}_4^1 \vee \mathcal{F}_4^2 \vee \mathcal{F}_4^3)] \\
&\overset{(6)}{=} \big(\Pr[\mathcal{G}_0] \cdot \Pr[\neg\mathcal{F}_2^1] \cdot \Pr[\neg\mathcal{F}_3^2] - \Pr[\mathcal{F}_3^1 \vee \mathcal{F}_3^2]\big) \cdot \Pr[\neg(\mathcal{F}_4^1 \vee \mathcal{F}_4^2 \vee \mathcal{F}_4^3)] \\
&\overset{(5)}{=} \left(\frac{1}{2} + \underbrace{\varepsilon(\lambda)}_{\text{non-negl by (5)}}\right) \cdot \left(\underbrace{\Pr[\neg\mathcal{F}_2^1] \cdot \Pr[\neg\mathcal{F}_3^2]}_{1-\mathsf{negl}(\lambda) \text{ by } (11,\,16)} - \underbrace{\Pr[\mathcal{F}_3^1 \vee \mathcal{F}_3^2]}_{\mathsf{negl}(\lambda) \text{ by } (13,\,16)}\right) \\
&\quad \cdot \underbrace{\Pr[\neg(\mathcal{F}_4^1 \vee \mathcal{F}_4^2 \vee \mathcal{F}_4^3)]}_{1-\mathsf{negl}(\lambda) \text{ by } (19,20,24)} \\
&> \frac{1}{2}
\end{aligned}$$

Therefore, the assumed adversary $\mathcal{A}$ winning Game-0 with non-negligible advantage $\varepsilon(\lambda)$ cannot exist. $\square$

9 Auxiliary Functionalities for Distributed Setup

We describe our distributed setup protocols in terms of ideal functionalities $\mathcal{F}_{\mathsf{VOLE}}, \mathcal{F}_{\mathsf{ZK}}, \mathcal{F}^{p,q}_{\mathsf{MPC}}$. These are used by the distributed key generation protocol for the Naor-Reingold-based DPFs with reusable setup in Section 10.

9.1 VOLE

We use a standard subfield VOLE functionality $\mathcal{F}_{\mathsf{VOLE}}$ (Figure 12) to generate the seed of our PCF. The functionality is similar to previous works such as Wolverine [WYKW21], but we allow for P_1 to freely choose Δ.

Fig. 12: $\mathcal{F}^{p,r}_{\mathsf{VOLE}}$: Subfield VOLE over $\mathbb{F}_{p^r}$

Ideal functionality for two parties P_0 and P_1 parametrized by a finite field $\mathbb{F}_{p^r}$.

Init On (Init) from P_0 and (Init, $\Delta \in \mathbb{F}_{p^r}$) from P_1, store Δ.
Sample On (Sample, n) from both parties:

1. If no party is corrupt, sample $[\![\boldsymbol{x}]\!] \leftarrow \mathsf{VOLE}^n(\Delta)$.
2. If P_0 is corrupt, receive $[\![\boldsymbol{x}]\!]_0$ with $\boldsymbol{x} \in \mathbb{F}_p^n$ from $\mathcal{S}$. and compute $[\![\boldsymbol{x}]\!]_1 \leftarrow \mathsf{VOLE}^n(\Delta).\mathsf{RSample}(1^\lambda, 0, [\![\boldsymbol{x}]\!]_0)$.
3. If P_1 is corrupt, receive $\langle \boldsymbol{x} \rangle_1$ of length n from $\mathcal{S}$. and compute $[\![\boldsymbol{x}]\!]_0 \leftarrow \mathsf{VOLE}^n(\Delta).\mathsf{RSample}(1^\lambda, 1, [\![\boldsymbol{x}]\!]_1 := (\Delta, \langle \boldsymbol{x} \rangle_1))$.
4. Send $[\![\boldsymbol{x}]\!]_b$ to P_b for $b \in \{0, 1\}$.

9.2 Zero-Knowledge with Oblivious Opening

We model commit-and-prove-style zero-knowledge proofs as an arithmetic-black-box ideal functionality $\mathcal{F}_{\mathsf{ZK}}$ (Figure 13 for prover $\mathcal{P}$ and verifier $\mathcal{V}$, which can be realized for example using VOLE-based zero-knowledge proofs [WYKW21; BMRS21; YSWW21]. In fact our $\mathcal{F}_{\mathsf{ZK}}$ is based on $\mathcal{F}^R_{\mathsf{ComZK}}$ from [BBMRS21] who instantiated these protocols for rings. Compared to $\mathcal{F}^R_{\mathsf{ComZK}}$, we replace Affine and $\mathsf{CheckMult}$ with a more abstract Eval, and additionally add $\mathsf{CheckNonZero}$ for convenience. Beyond the standard methods, we add two methods OTOpen and $\mathsf{CheckDLog}$ with simple realizations as described below.

Other instantiations (e.g., based on Pedersen commitments and Σ protocols) are also possible, and our protocols do for the most part not need the straight-line extractability guaranteed by $\mathcal{F}_{\mathsf{ZK}}$.

Notation. To simplify the description of our protocols that use $\mathcal{F}_{\mathsf{ZK}}$, we use the following shorthand: We represent committed values $x \in \mathbb{F}_p$ by $[\![x]\!]_p$ and write $[\![x]\!]_p \leftarrow \mathcal{F}^p_{\mathsf{ZK}}.\mathsf{Input}(x)$ to denote that $\mathcal{P}$ commits to a value x; $y \leftarrow \mathcal{F}^p_{\mathsf{ZK}}.\mathsf{Open}([\![y]\!]_p)$ denotes that y is revealed to $\mathcal{V}$. Instead of explicit calls to Eval we write for example $[\![z]\!]_p \leftarrow a \cdot [\![x]\!]_p \cdot [\![y]\!]_p + [\![w]\!]_p + b$ to denote that $\mathcal{P}$ is now committed to $z := a \cdot x \cdot y + w + b$.

Oblivious Opening. We add the method OTOpen where the prover obliviously opens one out of n commitments chosen by the verifier. The realization is straight-forward by using $\binom{n}{1}$-Oblivious Transfer where the messages consist of the corresponding commitment opening information. This allows for a selected-failure attack where the prover sends invalid values in some of the messages, which is accounted for in $\mathcal{F}_{\mathsf{ZK}}$.

CheckDL. We add a second method $\mathsf{CheckDLog}$ that takes triples consisting of a commitment to a value x_i and two group elements $G_i, H_i \in \mathbb{G}$ and checks if $x_i \cdot G_i = H_i$. The check can be realized with VOLE-ZK as follows: Let $q_i = \Delta \cdot x_i + v_i$ be the VOLE equation authenticating x_i such that $\mathcal{P}$ holds x_i and v_i and $\mathcal{V}$ holds Δ and q_i. $\mathcal{P}$ sends $V_i := v_i \cdot G_i$ to $\mathcal{V}$, who verifies that $q_i \cdot G_i - \Delta \cdot H_i = V_i$. To check

Fig. 13: $\mathcal{F}_{\mathsf{ZK}}^{p}(\mathsf{sid})$: Commit&Prove Zero-Knowledge over $\mathbb{F}_p$

Ideal functionality for two parties $\mathcal{P}$ and $\mathcal{V}$ parametrized by a finite field $\mathbb{F}_p$ and a group $\mathbb{G} = \langle G \rangle$ both of prime order order p.

Init Initialize an empty dictionary $\mathsf{D} = \emptyset$.
Input On $(\mathsf{Input}, \mathsf{id}, x)$ from $\mathcal{P}$ and $(\mathsf{Input}, \mathsf{id})$ from $\mathcal{V}$ such that id is a fresh identifier, store $\mathsf{D}[\mathsf{id}] = x$. // $[\![x]\!]_p \leftarrow \mathcal{F}_{\mathsf{ZK}}^{p}.\mathsf{Input}(x)$
Random On $(\mathsf{Random}, \mathsf{id})$ from $\mathcal{P}$ and $\mathcal{V}$ such that id is a fresh identifier, sample $x \xleftarrow{\$} \mathbb{F}$ and store $\mathsf{D}[\mathsf{id}] = x$.
// $[\![x]\!]_p \leftarrow \mathcal{F}_{\mathsf{ZK}}^{p}.\mathsf{Random}()$
Open On $(\mathsf{Open}, \mathsf{id})$ from $\mathcal{P}$ and $\mathcal{V}$ such that id is an existing identifier, look up $x := \mathsf{D}[\mathsf{id}]$ and send x to $\mathcal{V}$.
// $x \leftarrow \mathcal{F}_{\mathsf{ZK}}^{p}.\mathsf{Open}([\![x]\!]_p)$
CheckZero On $(\mathsf{CheckZero}, \mathsf{id})$ from $\mathcal{P}$ and $\mathcal{V}$ such that id is an existing identifier, look up $x := \mathsf{D}[\mathsf{id}]$ and if $x = 0$, send $(\mathsf{success})$ to $\mathcal{V}$; otherwise abort.
// $\bot/\top \leftarrow \mathcal{F}_{\mathsf{ZK}}^{p}.\mathsf{CheckZero}([\![x]\!]_p)$
CheckNonZero On $(\mathsf{CheckNonZero}, \mathsf{id})$ from $\mathcal{P}$ and $\mathcal{V}$ such that id is an existing identifier, look up $x := \mathsf{D}[\mathsf{id}]$ and if $x \neq 0$, send $(\mathsf{success})$ to $\mathcal{V}$; otherwise abort.
// $\bot/\top \leftarrow \mathcal{F}_{\mathsf{ZK}}^{p}.\mathsf{CheckNonZero}([\![x]\!]_p)$
Eval On $(\mathsf{Eval}, \mathcal{C}, \mathsf{id}_1, \ldots, \mathsf{id}_n, \mathsf{id}'_1, \ldots, \mathsf{id}'_m)$ from $\mathcal{P}$ and $\mathcal{V}$ such that $\mathcal{C}\colon \mathbb{F}^n \to \mathbb{F}^m$ is an arithmetic circuit over $\mathbb{F}$ $(\mathsf{id}_1, \ldots, \mathsf{id}_n)$ are existing and $(\mathsf{id}'_1, \ldots, \mathsf{id}'_m)$ are fresh identifiers, look up $x_i := \mathsf{D}[\mathsf{id}_i]$ for $i \in [n]$, compute $(y_1, \ldots, y_m) \leftarrow \mathcal{C}(x_1, \ldots, x_n)$, and store $\mathsf{D}[\mathsf{id}'_j] := y_j$ for $j \in [m]$.
// $[\![z]\!]_p \leftarrow [\![x]\!]_p \cdot [\![y]\!]_p + [\![w]\!]_p \cdot 42$
OTOpen On $(\mathsf{OTOpen}, (\mathsf{id}_{i,1}, \ldots, \mathsf{id}_{i,m}, y_i)_{i \in [0,n)})$ from $\mathcal{P}$ and $(\mathsf{OTOpen}, (\mathsf{id}_{i,1}, \ldots, \mathsf{id}_{i,m})_{i \in [0,n)}, z \in [0,n))$ from $\mathcal{V}$ such that the $\mathsf{id}_{i,j}$ are existing identifiers and $y_i \in \{0,1\}^*$: If $\mathcal{P}$ is corrupted, receive $I \subseteq [0,n)$ from $\mathcal{S}$, and send $\bot$ to $\mathcal{V}$ if $z \notin I$. Otherwise, look up $x_j := \mathsf{D}[\mathsf{id}_{z,j}]$ for $j \in [m]$ and send $(x_1, \ldots, x_m, y_z)$ to $\mathcal{V}$.
// $x_i, y_i \leftarrow \mathcal{F}_{\mathsf{ZK}}^{p}.\mathsf{OTOpen}(([\![x_j]\!]_p, y_j)_j, i)$
CheckDL On $(\mathsf{CheckDL}, (\mathsf{id}_i, G_i \in \mathbb{G}, H_i \in \mathbb{G})_{i \in [n]})$ from $\mathcal{P}$ and $\mathcal{V}$ such that $(\mathsf{id}_1, \ldots, \mathsf{id}_n)$ are existing identifiers, look up $x_i := \mathsf{D}[\mathsf{id}_i]$ for $i \in [n]$, and return $\top$ iff $x_i \cdot G_i = H_i$, and $\bot$ otherwise.
// $\bot/\top \leftarrow \mathcal{F}_{\mathsf{ZK}}^{p}.\mathsf{CheckDLog}([\![x]\!]_p, G, H)$

a larger batch, $\mathcal{P}$ can send a hash $h := \mathsf{H}((v_i \cdot G_i)_{i\in[n]})$, and $\mathcal{V}$ compares $h \stackrel{?}{=} \mathsf{H}((q_i \cdot G_i - \Delta \cdot H_i)_{i\in[n]})$. Successfully cheating implies guessing Δ or breaking the collision resistance of the hash function (cf. batch opening in Wolverine [WYKW21]).

9.3 MPC

We need to perform secure computation over two different fields $\mathbb{F}_p$ and $\mathbb{F}_q$. We model this with a single reactive arithmetic black box (ABB) functionality $\mathcal{F}_{\mathsf{MPC}}^{p,q}$ (Figure 14), which for each field allows parties to Input values to the computation, compute arithmetic circuits on values (EvalCircuit), and receive Output. $\mathcal{F}_{\mathsf{MPC}}^{p,q}$ can be realized by using two instances of the standard reactive MPC functionalities from, e.g., SPDZ [DPSZ12; Dam+13].

We use a single functionality for both fields because we also need an additional check that two values over $\mathbb{F}_p$ and $\mathbb{F}_q$ are consistent: Recall that we make use of a cycle of elliptic curve groups $(\mathbb{G}, \overline{\mathbb{G}})$. We represent elements $H \in \mathbb{G}$ coordinate-wise over the base field $\mathbb{F}_q$ and define CheckEC as follows: Given $x \in \mathbb{F}_p$ and $H \in \mathbb{G} \subset \mathbb{F}_q^2$, it verifies that $H = x \cdot G$, i.e., that x is the discrete logarithm of H w.r.t. G. We define the check analogously for $\overline{\mathbb{G}}$, and let it verify values in large batches for a more efficient implementation which we describe below.

Fig. 14: MPC over $\mathbb{F}_p$ and $\mathbb{F}_q$

Let $\mathbb{G}, \overline{\mathbb{G}}$ be an elliptic curve cycle such that $\mathbb{G}$ has base field $\mathbb{F}_q$ and scalar field $\mathbb{F}_p$. In the following let $\mathbb{F}_M \in \{\mathbb{F}_p, \mathbb{F}_q\}$.

Init Initialize two empty dictionaries $\mathsf{D}_p = \mathsf{D}_q = \emptyset$.
Input On (Input, P_i, $\mathbb{F}_M$, id, x) from P_i and (Input, P_i, $\mathbb{F}_M$, id) from all other parties such that id is a fresh identifier for $\mathbb{F}_M$, store $\mathsf{D}_M[\mathsf{id}] = x$.
Output(All) On (Output, P_i, $\mathbb{F}_M$, id) from all parties such that id is an existing identifier for $\mathbb{F}_M$, look up $x := \mathsf{D}_M[\mathsf{id}]$ and send x to P_i. On (OutputAll, $\mathbb{F}_M$, id) x is sent to all parties.
OutputShared On (OutputShared, $\mathbb{F}_M$, id) from all parties such that id is an existing identifier for $\mathbb{F}_M$, look up $x := \mathsf{D}_M[\mathsf{id}]$ and send random x_i to each P_i such that $x = \sum_i x_i$. (Corrupted P_i can choose their x_i.)
CheckZero On (CheckZero, $\mathbb{F}_M$, id) from all parties such that id is an existing identifier for $\mathbb{F}_M$, look up $x := \mathsf{D}_M[\mathsf{id}]$ and if $x = 0$, send (success) to all parties; otherwise send (failure).
EvalCircuit On (EvalCircuit, $\mathbb{F}_M$, $\mathcal{C}$, $\mathsf{id}_1, \ldots, \mathsf{id}_n$, $\mathsf{id}'_1, \ldots, \mathsf{id}'_m$) from all parties such that $\mathcal{C}\colon \mathbb{F}_M^n \to \mathbb{F}_M^m$ is an arithmetic circuit over $\mathbb{F}_M$ ($\mathsf{id}_1, \ldots, \mathsf{id}_n$) are existing and ($\mathsf{id}'_1, \ldots, \mathsf{id}'_m$) are fresh identifiers for $\mathbb{F}_M$: look up $x_i := \mathsf{D}_M[\mathsf{id}_i]$ for $i \in [n]$, compute $(y_1, \ldots, y_m) \leftarrow \mathcal{C}(x_1, \ldots, x_n)$, and store $\mathsf{D}_M[\mathsf{id}'_j] := y_j$ for $j \in [m]$.
CheckEC On (CheckEC, $\mathbb{G}$, $(\mathsf{id}_i^a, \mathsf{id}_i^x, \mathsf{id}_i^y)_{i\in[n]}$) such that the id_i^a are existing identifiers for $\mathbb{F}_p$ and the $\mathsf{id}_i^x, \mathsf{id}_i^y$ are existing identifiers for $\mathbb{F}_q$: look up $a_i := \mathsf{D}_p[\mathsf{id}_i^a]$, $x_i := \mathsf{D}_q[\mathsf{id}_i^x]$, and $y_i := \mathsf{D}_q[\mathsf{id}_i^y]$, and interpret $A_i := (x_i, y_i)$ for $i \in [n]$. If $A_i \notin \mathbb{G}$ for some $i \in [n]$, send all $(a_i, x_i, y_i)_{i\in[n]}$ to $\mathcal{S}$ and output whatever $\mathcal{S}$ responds. Otherwise: If $A_i = a_i \cdot G$ for all $i \in [n]$, send (success) to all parties; otherwise send (failure).
The input (CheckEC, $\mathcal{C}$, $\overline{\mathbb{G}}$, ...) is handled analogously with the groups and fields swapped.

Notation. Similar to $\mathcal{F}_{\mathsf{ZK}}$ we also use a shorthand: Values $x \in \mathbb{F}_p$ and $y \in \mathbb{F}_q$ input to $\mathcal{F}_{\mathsf{MPC}}^{p,q}$ are denoted by $\langle\!\langle x \rangle\!\rangle_p$ and $\langle\!\langle y \rangle\!\rangle_q$, respectively.

Point Addition in MPC. Given $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ such that $P, Q \neq \mathcal{O}$ and $x_1 \neq x_2$, the sum $P + Q = (x_3, y_3)$ can be computed by:

$$\begin{aligned} \lambda &= \frac{y_2 - y_1}{x_2 - x_1}, \\ x_3 &= \lambda^2 - x_1 - x_2, \\ y_3 &= \lambda \cdot (x_1 - x_3) - y_1. \end{aligned}$$

Let $\mathcal{C}_{\mathsf{ADD}}\colon \mathbb{F}^2 \times \mathbb{F}^2 \to \mathbb{F}^2$ be the corresponding arithmetic circuit consisting of 1 division, 1 squaring, and 1 multiplication. Evaluation on invalid inputs such that $x_1 = x_2$ result in division by zero, which can cause an abort during secure evaluation.

Realizing CheckEC. Let Π_{MPC} be a protocol in the $\mathcal{F}_{\mathsf{MPC}}^{p,q'}$-hybrid model that supports the same methods as $\mathcal{F}_{\mathsf{MPC}}^{p,q}$: The method CheckEC is defined in Figure 15 and all other methods are instantiated by by calling the corresponding method of $\mathcal{F}_{\mathsf{MPC}}^{p,q'}$.

The CheckEC is based on the combinatorial multiplication check from Wolverine [WYKW21]: To verify consistency of n tuples (a_i, A_i), we create $N := B \cdot n + C$ additional tuples (r_j, R_j). We randomly open C of these and verify that $r_j \cdot G = R_j$. Then, each of the original tuples (a_i, A_i) is checked with B randomly selected tuples (r_j, R_j) by verifying that $A_i + (r_j - a_i) \cdot G = R_j$ holds. The full protocol for CheckEC is given in Figure 15 and parameter choices for B, C are given in Table 2.

To verify n tuples, we compute $2nB + C$ point addition inside of MPC, which results in $8nB + 4C$ secure multiplications, so ca. $8B$ multiplications per tuple to check.

Fig. 15: CheckEC Protocol

The protocol is described in the $\mathcal{F}_{\mathsf{MPC}}^{p,q'}$-hybrid model, where $\mathcal{F}_{\mathsf{MPC}}^{p,q'}$ the same as $\mathcal{F}_{\mathsf{MPC}}^{p,q}$ without CheckEC.
Let $n, B, C \in \mathbb{N}$ and $N := B \cdot n + C$ such that $\binom{n}{B} \geq 2^\rho$. Let $\mathcal{C}_{\mathsf{ADD}}\colon \mathbb{F}_q^2 \cdot \mathbb{F}_q^2 \to \mathbb{F}_q^2$ be the elliptic curve addition circuit.
On input $(\mathsf{CheckEC}, \mathbb{G}, (\langle\!\langle a_j \rangle\!\rangle_p, \langle\!\langle A_j \rangle\!\rangle_q)_{j\in[n]})$:

1. Each P_i for $i \in \{0,1\}$ and $j \in [N]$:
   (a) Sample $s_{i,j} \in_R \mathbb{F}_p$ and compute $S_{i,j} := s_{i,j} \cdot G$.
   (b) Input the values: $\langle\!\langle s_{i,j} \rangle\!\rangle_p \leftarrow \mathcal{F}_{\mathsf{MPC}}^{p,q'}.\mathsf{Input}(\mathsf{P}_i, \mathbb{F}_p, s_{i,j})$ and $\langle\!\langle S_{i,j} \rangle\!\rangle_q \leftarrow \mathcal{F}_{\mathsf{MPC}}^{p,q'}.\mathsf{Input}(\mathsf{P}_i, \mathbb{F}_q, S_{i,j})$
2. For $j \in [N]$, compute
   (a) $\langle\!\langle s_j \rangle\!\rangle_p \leftarrow \langle\!\langle s_{0,j} \rangle\!\rangle_p + \langle\!\langle s_{1,j} \rangle\!\rangle_p$
   (b) $\langle\!\langle S_j \rangle\!\rangle_q \leftarrow \mathcal{F}_{\mathsf{MPC}}^{p,q'}.\mathsf{EvalCircuit}(\mathcal{C}_{\mathsf{ADD}}, \langle\!\langle S_{0,j} \rangle\!\rangle_q, \langle\!\langle S_{1,j} \rangle\!\rangle_q)$
3. Jointly sample a random permutation $\pi\colon [N] \to [N]$. Let $(\langle\!\langle r_j \rangle\!\rangle_p, \langle\!\langle R_j \rangle\!\rangle_q)_{j\in[N]}$ be the permuted tuples $(\langle\!\langle s_{\pi(j)} \rangle\!\rangle_p, \langle\!\langle S_{\pi(j)} \rangle\!\rangle_q)_{j\in[N]}$.
4. Verify the last C tuples, for $j \in [Bn+1, Bn+C]$ do in parallel:
   (a) $r_j \leftarrow \mathcal{F}_{\mathsf{MPC}}^{p,q'}.\mathsf{OutputAll}(\mathbb{F}_p, \langle\!\langle r_j \rangle\!\rangle_p)$, $R_j \leftarrow \mathcal{F}_{\mathsf{MPC}}^{p,q'}.\mathsf{OutputAll}(\mathbb{F}_q, \langle\!\langle R_j \rangle\!\rangle_q)$.
   (b) Output failure if $r_j \cdot G \neq R_j$.
5. For each input $i \in [n]$ and each $j \in [B(i-1)+1, Bi]$ do in parallel:
   (a) $t_j \leftarrow \mathcal{F}_{\mathsf{MPC}}^{p,q'}.\mathsf{OutputAll}(\mathbb{F}_p, \langle\!\langle r_j \rangle\!\rangle_p - \langle\!\langle a_i \rangle\!\rangle_p)$
   (b) $T_j := t_j \cdot G$
   (c) $\langle\!\langle R'_j \rangle\!\rangle_q \leftarrow \mathcal{F}_{\mathsf{MPC}}^{p,q'}.\mathsf{EvalCircuit}(\mathbb{F}_q, \mathcal{C}_{\mathsf{ADD}}, \langle\!\langle A_i \rangle\!\rangle_q, \langle\!\langle T_j \rangle\!\rangle_q)$
6. Check that $R_j = R'_j$ for all $j \in [Bn]$ by running

$$\mathcal{F}_{\mathsf{MPC}}^{p,q'}.\mathsf{CheckZero}\left(\{\langle\!\langle R'_j.\mathsf{x} \rangle\!\rangle_q - \langle\!\langle R_j.\mathsf{x} \rangle\!\rangle_q, \langle\!\langle R'_j.\mathsf{y} \rangle\!\rangle_q - \langle\!\langle R_j.\mathsf{y} \rangle\!\rangle_q\}_{j\in[Bn]}\right)$$

   and output success if it passes; otherwise output failure.

Theorem 10. *The protocol Π_{MPC} securely realizes $\mathcal{F}_{\mathsf{MPC}}^{p,q}$ in the $\mathcal{F}_{\mathsf{MPC}}^{p,q'}$-hybrid model for 2 parties with statistical security.*

Proof. Since all other methods are just wrappers around $\mathcal{F}_{\mathsf{MPC}}^{p,q'}$, we focus on the simulation of CheckEC here.

Computing Point Additions. Note that every time the protocol computes the sum of two points P, Q with $\mathcal{C}_{\mathsf{ADD}}$ one of the inputs is chosen uniformly at random and independently of the other input. Thus, we will have $P \neq \pm Q$ except with negligible probability such that we can assume $\mathcal{C}_{\mathsf{ADD}}$ always correctly implements point addition and abort due to division by zero never happens.

Table 2. Parameter choices for CheckEC to obtain $\rho = 40$ bits of statistical security.

n	$B = C$
$\geq$741 455	2
$\geq$ 6251	3
$\geq$ 566	4
$\geq$ 133	5
$\geq$ 51	6
$\geq$ 25	7
$\geq$ 15	8

n	$B = C$
$\geq$10	9
$\geq$ 7	10
$\geq$ 6	11
$\geq$ 4	12
$\geq$ 3	14
$\geq$ 2	16
$\geq$ 1	22

No corruptions. We distinguish two cases: First, if the inputs are well-formed, i.e., $A_i \in \mathbb{G}$ for all $i \in [n]$, then we only need to argue correctness of the protocol: If both parties are honest, then we have $S_{i,j} = s_{i,j} \cdot G$ for all i, j and therefore also $R_j = r_j \cdot G$ for all j and the C opened tuples (r_j, R_j) verify. Let $A_i = (a_i + \delta_i) \cdot G$ for $i \in [n]$. Then the protocol outputs success iff

$$A_i + T_j = R_j \iff (a_i + \delta_i) \cdot G + (r_j - a_i) \cdot G = r_j \cdot G \iff \delta_i + r_j = r_j \iff \delta_i = 0.$$

Hence it outputs success exactly when $\mathcal{F}_{\mathsf{MPC}}^{p,q}$ would output success.

Second, if there is an $A_i \notin \mathbb{G}$, $\mathcal{F}_{\mathsf{MPC}}^{p,q}$ breaks down and sends all values $a_i, A_i = (x_i, y_i)$ to $\mathcal{S}$. To ensure that $\mathcal{F}_{\mathsf{MPC}}^{p,q}$ outputs the same as the protocol, $\mathcal{S}$ simulates and execution of the CheckEC protocol with two honest parties using the obtained $(a_i, A_i)_i$ as inputs, which results in an output success or failure. This value is sent to $\mathcal{F}_{\mathsf{MPC}}^{p,q}$ which outputs it.

Corrupt P_b. Again, suppose that all inputs are well formed, i.e., $A_i \in \mathbb{G}$ for all $i \in [n]$. Then $\mathcal{S}$ receives success or failure from $\mathcal{F}_{\mathsf{MPC}}^{p,q}$.

$\mathcal{S}$ extracts P_b's values $s_{b,j} \in \mathbb{F}_p$ and $S_{b,j} \in \mathbb{F}_q^2$ in Step 1, and simulates the protocol up to Step 4 using honestly generated values for P_{1-b}'s values $s_{1-b,j}$ and $S_{1-b,j} = s_{1-b,j} \cdot G$. If one of the opened (r_j, R_j), $j \in [Bn + 1, Bn + C]$ is invalid in Step 4, then $\mathcal{S}$ sends failure to $\mathcal{F}_{\mathsf{MPC}}^{p,q}$ and simulates P_{1-b} stopping.

Otherwise, continue the simulation and in Step 5a, output uniformly random $t_j \xleftarrow{\$} \mathbb{F}_p$. Simulate the final CheckZero in Step 6: If $\mathcal{F}_{\mathsf{MPC}}^{p,q}$ had sent failure earlier or if one of the tuples (r_j, R_j), $j \in [Bn]$, is invalid, simulate a failing check and send failure to $\mathcal{F}_{\mathsf{MPC}}^{p,q}$. Otherwise (all tuples are valid and $\mathcal{F}_{\mathsf{MPC}}^{p,q}$ had signaled success), simulate a succeeding check and send success to $\mathcal{F}_{\mathsf{MPC}}^{p,q}$.

Up until the last step the simulation is perfectly indistinguishable from the real protocol. Now we need to argue that the simulated final check matches the outcome of the protocol except with negligible probability. We have four cases:

- All $(a_i, A_i)_i$ good and all $(r_j, R_j)_j$ good: Both protocol and simulation accept.
- All $(a_i, A_i)_i$ good, some $(r_j, R_j)_j$ bad: Both protocol and simulation reject.
- Some $(a_i, A_i)_i$ bad, all $(r_j, R_j)_j$ good: Both protocol and simulation reject.
- Some $(a_i, A_i)_i$ bad, some $(r_j, R_j)_j$ bad: Following the analysis of the corresponding "balls-and-bins" game in Wolverine [WYKW21], an adversary can make the protocol accept with probability at most $\binom{nB+C}{B}^{-1} \leq 2^{-\rho}$, whereas the simulation always rejects.

Thus, the whole simulation is statistically indistinguishable to the real protocol (in the $\mathcal{F}_{\mathsf{MPC}}^{p,q'}$-hybrid model).

Now suppose that one of the $A_i \notin \mathbb{G}$. Then as in the case for no corruptions above, $\mathcal{S}$ receives all values $a_i, A_i = (x_i, y_i)$ from $\mathcal{F}_{\mathsf{MPC}}^{p,q}$, simulates the real protocol with P_b, and signals $\mathcal{F}_{\mathsf{MPC}}^{p,q}$ to output success or failure depending on the result of the simulated protocol. This perfectly simulates the real protocol. □

10 Distributed Key Generation for Distributed Point Functions

10.1 The Ideal Functionality $\mathcal{F}_{\mathsf{DPF}}$

We define a generic ideal functionality $\mathcal{F}_{\mathsf{DPF}}$ for parties P_0 and P_1 that is parameterized by any DPF with reusable setup DPF according to Definition 17. It supports two methods where $\mathcal{F}_{\mathsf{DPF}}$ executes the algorithms DPF.Setup and DPF.KeyGen, respectively, while allowing the simulator $\mathcal{S}$ to provide influence. During Setup P_1 provides a value $\Delta \in \mathbb{F}_{p^r}$ and both parties obtain their part of the DPF setup state $(\mathsf{st}_0^{\mathsf{DPF}}, \mathsf{st}_1^{\mathsf{DPF}})$. To be able to force a value of Δ consistent with other functionalities, it is authenticated using an information-theoretic MAC, which $\mathcal{F}_{\mathsf{DPF}}$ verifies. For Keygen with common input N, t, and t pairs α_i, β_i as private input of P_0, $\mathcal{F}_{\mathsf{DPF}}$ generates t DPF key pairs $(\mathsf{k}_{0,(i)}^{\mathsf{DPF}}, \mathsf{k}_{1,(i)}^{\mathsf{DPF}})$ sharing the point functions that map $\alpha_i \mapsto \Delta \cdot \beta_i$.

Fig. 16: $\mathcal{F}_{\mathsf{DPF}}$ – DPF Setup+Keygen Functionality

Let interface $= \{\lambda, \{N_i\}_{i\in[\ell]}, \mathbb{F}_{p^r}\}$ be the interface of a be a distributed point function scheme for the class of point functions $\{f\colon \mathbb{N} \to \mathbb{F}_{p^r}\}$ as in Definition 17.

Setup On (Setup, $\Gamma_\Delta \in \mathbb{F}_{p^r}$, $K_\Delta \in \mathbb{F}_{p^r}$) from P_0 and (Setup, $\Delta \in \mathbb{F}_{p^r}$, $M_\Delta \in \mathbb{F}_{p^r}$) from P_1:

1. If $M_\Delta \neq \Gamma_\Delta \cdot \Delta + K_\Delta$, abort; otherwise store Δ.
2. Run $(\mathsf{st}_0^{\mathsf{DPF}}, \mathsf{st}_1^{\mathsf{DPF}}) \leftarrow \mathsf{DPF.Setup}^{\mathcal{S},b}(1^\lambda)$.
3. Send $\mathsf{st}_b^{\mathsf{DPF}}$ to P_b for $b \in \{0,1\}$.

This method can only be called once.

Keygen On (Keygen, $N \in \mathbb{N}$, $t \in \mathbb{N}$, $(\boldsymbol{z}, \boldsymbol{e}) \in (\mathbb{N} \times \mathbb{F}_{p^r})^t$) from P_0 and (Keygen, $N \in \mathbb{N}$, $t \in \mathbb{N}$) from P_1:

1. If $b = 1$, receive $I \subseteq [N]^t$ from $\mathcal{S}$. If $\boldsymbol{z} \notin I$, abort.
2. For $i \in [t]$:
   (a) Run $k \leftarrow \mathsf{DPF.Gen}^{\mathcal{S},b}((\mathsf{st}_0^{\mathsf{DPF}}, \mathsf{st}_1^{\mathsf{DPF}}), N, (z_i - 1, \Delta \cdot e_i))$.
   (b) If $k = \perp$ abort; otherwise parse $k = (\mathsf{k}_{0,(i)}^{\mathsf{DPF}}, \mathsf{k}_{1,(i)}^{\mathsf{DPF}})$.
3. If $b \in \{0,1\}$, send $(\mathsf{k}_{b,(i)}^{\mathsf{DPF}})_{i\in[t]}$ to $\mathcal{S}$, and await continue or abort from $\mathcal{S}$. If abort was received, abort.
4. Send $(\mathsf{k}_{b,(i)}^{\mathsf{DPF}})_{i\in[t]}$ to P_b for $b \in \{0,1\}$.

This method can only be called after Setup.

10.2 The Protocol Π_{DPF}

Our protocol Π_{DPF} realizes $\mathcal{F}_{\mathsf{DPF}}$ where DPF is instantiated with the PPRF-to-DPF transformation (Figure 9) applied to the Compound-PPRF in the $(\mathcal{F}_{\mathsf{ZK}}, \mathcal{F}_{\mathsf{MPC}}^{p,q})$-hybrid model. Recall that the compound PPRF was obtained by mixing the GGM-PPRF, CHHK25-PPRF and k-NR-PPRF. In particular, the construction is specified by a selection function SEL that on input a domain N, outputs one of these three PPRFs together with their specification.

While we decided to model it as a single UC protocol[31], it is still very modular and consists of the following subprotocols:

- $\Pi_{\mathsf{DPF.Macros}}$ (Figure 17) contains helper macros to ensure that consistent values are input into $\mathcal{F}_{\mathsf{ZK}}$ and $\mathcal{F}_{\mathsf{MPC}}^{p,q}$.
- $\Pi_{\mathsf{DPF.Setup}}$ (Figure 18) realizes the Setup method of $\mathcal{F}_{\mathsf{DPF}}$.
- $\Pi_{\mathsf{DPF.Hash}}$ (Figure 19), securely evaluates the arithmetization-friendly hash functions from Section 8.
- $\Pi_{\mathsf{DPF.CHHK25}}$ (Figure 20) realizes Gen for the DPF when the selection function of the Compound-PPRF selects CHHK25-PPRF.
- $\Pi_{\mathsf{DPF.kNR}}$ (Figure 21) realizes Gen for the DPF when the selection function of the Compound-PPRF selects k-NR-PPRF.

We do not introduce a protocol to realize Gen of the DPF from GGM-PPRF, and refer to Wolverine [WYKW21] instead.

[31] While it is possible to split $\mathcal{F}_{\mathsf{DPF}}$ into separate functionalities for the shared setup and the different DPF instantiations, it would require extra care and additional checks to ensure that consistent setup states are input in all functionalities.

Fig. 17: $\Pi_{\mathsf{DPF.Macros}}$ – Macros used in DPF Setup+Keygen Protocol

Macro `AuthCommit`$(\mathsf{P}_b; (x_i)_{i\in[n]}, (M_{x_i})_{i\in[n]}; \Gamma, (K_{x_i})_{i\in[n]})$

1. $[\![x_1]\!]^b_p, \ldots, [\![x_n]\!]^b_p \leftarrow \mathcal{F}^{p,b}_{\mathsf{ZK}}.\mathsf{Input}(x_1, \ldots, x_n)$.
2. P_{1-b} sends $\chi \xleftarrow{\$} \mathbb{F}_p$.
3. $[\![M_{\hat{x}}]\!]^b_p \leftarrow \mathcal{F}^{p,b}_{\mathsf{ZK}}.\mathsf{Input}(\sum_{i\in[n]} \chi^i \cdot M_{x_i})$.
4. P_{1-b} sends Γ and $(K_{x_i})_{i\in[n]}$.
5. Compute $[\![\hat{x}]\!]^b_p \leftarrow \sum_{i\in[n]} \chi^i \cdot [\![x_i]\!]^b_p$ and $K_{\hat{x}} := \sum_{i\in[n]} \chi^i \cdot K_{x_i}$.
6. If $\mathcal{F}^{p,b}_{\mathsf{ZK}}.\mathsf{CheckZero}\big(\Gamma \cdot [\![\hat{x}]\!]^b_p + K_{\hat{x}} - [\![M_{\hat{x}}]\!]^b_p\big) = \bot$, abort.
7. Return $[\![x_1]\!]^b_p, \ldots, [\![x_n]\!]^b_p$.

Macro `ComInput`$(p, \mathsf{P}_b; ([\![x_i]\!]^b_p)_{i\in[n]})$

1. $\langle\!\langle x_1 \rangle\!\rangle_p, \ldots, \langle\!\langle x_n \rangle\!\rangle_p \leftarrow \mathcal{F}^{p,q}_{\mathsf{MPC}}.\mathsf{Input}(p, \mathsf{P}_b, x_1, \ldots, x_n)$.
2. $[\![\sigma]\!]^b_p \leftarrow \mathcal{F}^{p,b}_{\mathsf{ZK}}.\mathsf{Random}()$, $\langle\!\langle \sigma \rangle\!\rangle_p \leftarrow \mathcal{F}^{p,q}_{\mathsf{MPC}}.\mathsf{Input}(p, \mathsf{P}_b, \sigma)$.
3. P_{1-b} sends $\chi \xleftarrow{\$} \mathbb{F}_p$.
4. $\mu = \mathcal{F}^{p,b}_{\mathsf{ZK}}.\mathsf{Open}([\![\sigma]\!]^b_p + \sum_{i=1}^n \chi^i \cdot [\![x_i]\!]^b_p)$.
5. If $\mathcal{F}^{p,q}_{\mathsf{MPC}}.\mathsf{CheckZero}(\langle\!\langle \sigma \rangle\!\rangle_p + \sum_{i=1}^n \chi^i \cdot \langle\!\langle x_i \rangle\!\rangle_p - \mu) = \bot$, abort.
6. Return $\langle\!\langle x_1 \rangle\!\rangle_p, \ldots, \langle\!\langle x_n \rangle\!\rangle_p$.

Fig. 18: $\Pi_{\mathsf{DPF.Setup}}$ – DPF Setup+Keygen Protocol

We read the public parameters $\mathsf{pp} = (\mathbb{G}, G, p, (S^{\max}, N_2^{\max}, k_3^{\max}))$ of the underlying Compound-PPRF F, in the $(\mathcal{F}^{p,q}_{\mathsf{MPC}}, \mathcal{F}^{p,0}_{\mathsf{ZK}}, \mathcal{F}^{p,1}_{\mathsf{ZK}}, \mathcal{F}^{q,0}_{\mathsf{ZK}}, \mathcal{F}^{q,1}_{\mathsf{ZK}})$-hybrid model.

Setup On $(\mathsf{Setup}, \Gamma_\Delta \in \mathbb{F}_p, K_\Delta \in \mathbb{F}_p)$ from P_0 and $(\mathsf{Setup}, \Delta \in \mathbb{F}_p, M_\Delta \in \mathbb{F}_p)$ from P_1:

1. Input Δ:
   (a) $\langle\!\langle \Delta \rangle\!\rangle_p, \langle\!\langle M_\Delta \rangle\!\rangle_p \leftarrow \mathcal{F}^{p,q}_{\mathsf{MPC}}.\mathsf{Input}(p, \mathsf{P}_1, \Delta, M_\Delta)$.
   (b) P_0 sends Γ_Δ, K_Δ to P_1.
   (c) If $\mathcal{F}^{p,q}_{\mathsf{MPC}}.\mathsf{CheckZero}(\Gamma_\Delta \cdot \langle\!\langle \Delta \rangle\!\rangle_p + K_\Delta - M_\Delta) = \bot$, abort.
2. Generate msk:
   (a) $[\![\rho]\!]^1_p, [\![r]\!]^1_p, [\![a_0]\!]^1_p \leftarrow \mathcal{F}^{p,1}_{\mathsf{ZK}}.\mathsf{Random}()$, and P_1 sets $\mathsf{msk} =: (\rho, r, a_0)$.
   (b) If $\mathcal{F}^{p,1}_{\mathsf{ZK}}.\mathsf{CheckNonZero}([\![\rho]\!]^1_p, [\![r]\!]^1_p, [\![a_0]\!]^1_p) = \bot$, abort.
3. P_1 computes $\mathsf{mpk} := ((H_s)_{s\in S}, \alpha_0) \leftarrow \mathsf{F.ComputeMPK}(\mathsf{msk})$. // $H_s = r^s \cdot \rho \cdot G$
4. P_1 sends mpk to P_0.
5. Prove consistency of mpk:
   (a) $\mathcal{F}^{p,1}_{\mathsf{ZK}}.\mathsf{CheckZero}([\![a_0]\!]^1_p - \alpha_0 \cdot [\![r]\!]^1_p)$ // $\alpha_0 = a_0 \cdot r^{-1}$
   (b) Compute $[\![\rho \cdot r]\!]^1_p \leftarrow [\![\rho]\!]^1_p \cdot [\![r]\!]^1_p$ and $[\![r^2]\!]^1_p \leftarrow [\![r]\!]^1_p \cdot [\![r]\!]^1_p$
   (c) $\mathcal{F}^{p,1}_{\mathsf{ZK}}.\mathsf{CheckDLog}(([\![\rho \cdot r]\!]^1_p, G, H_1))$ // $r \cdot \rho \cdot G = H_1$
   (d) $\mathcal{F}^{p,1}_{\mathsf{ZK}}.\mathsf{CheckDLog}\big(\{([\![r]\!]^1_p, H_i, H_{i+1})\}_{i\in[S_a,-1)\cup[1,S_b)}\big)$ // $r \cdot H_i = H_{i+1}$
   (e) If $S_a < 0$, run $\mathcal{F}^{p,1}_{\mathsf{ZK}}.\mathsf{CheckDLog}(([\![r^2]\!]^1_p, H_{-1}, H_1))$. // $r^2 \cdot H_{-1} = H_1$
6. Run $\langle\!\langle \rho \rangle\!\rangle_p \leftarrow \mathtt{ComInput}(\mathsf{P}_1, [\![\rho]\!]^1_p)$.
7. Pre-compute commitments:
   (a) Compute $[\![r^{2^{i-1}}]\!]^1_p$ for $i \in [\lceil \log_2(N_2^{\max}) \rceil]$.
   (b) Compute the following values:
      i. $[\![r^{2j}]\!]^1_p \leftarrow [\![r^2]\!]^1_p \cdot [\![r^{2j-2}]\!]^1_p$ for $j \in [2, k_3^{\max})$
      ii. $[\![r^{-j^2}]\!]^1_p \leftarrow \mathcal{F}^{p,1}_{\mathsf{ZK}}.\mathsf{Input}(r^{-j^2})$ for $j \in [1, k_3^{\max})$
      iii. $[\![a_0^{j^2}]\!]^1_p \leftarrow \mathcal{F}^{p,1}_{\mathsf{ZK}}.\mathsf{Input}(a_0^{j^2})$ for $j \in [2, k_3^{\max})$
      iv. $\mathcal{F}^{p,1}_{\mathsf{ZK}}.\mathsf{CheckDLog}\big(\{([\![r^{-j^2}]\!]^1_p, H_{j^2+1}, H_1)\}_{j\in[1,k_3^{\max})}\big)$ [a] // $r^{-j^2} \cdot H_{j^2+1} = H_1$
      v. $\mathcal{F}^{p,1}_{\mathsf{ZK}}.\mathsf{CheckZero}(\{[\![a_0^{j^2}]\!]^1_p \cdot [\![r^{-j^2}]\!]^1_p - \alpha_0^{j^2}\}_{j\in[2,k_3^{\max})})$ // $a_0^{j^2} = \alpha_0^{j^2} \cdot r^{j^2}$
      vi. $[\![\rho \cdot a_0^{-1}]\!]^1_p \leftarrow [\![\rho]\!]^1_p \cdot ([\![a_0]\!]^1_p)^{-1}$
8. P_0 outputs $\mathsf{st}_0^{\mathsf{DPF}} := \mathsf{mpk}$ and P_1 outputs $\mathsf{st}_1^{\mathsf{DPF}} := \mathsf{msk}$.

Keygen On $(\mathsf{Keygen}, N \in \mathbb{N}, (\boldsymbol{\alpha}, \boldsymbol{\beta}) \in (\mathbb{N} \times \mathbb{F}_p)^t)$ from P_0 and $(\mathsf{Keygen}, N \in \mathbb{N})$ from P_1:

1. Depending on the output of $\mathsf{SEL}(N)$, run one of the subprotocols $\Pi_{\mathsf{DPF.CHHK25}}$ (Figure 20), $\Pi_{\mathsf{DPF.kNR}}$ (Figure 21), or Wolverine.

[a] Assumes $(k_3^{\max} - 1)^2 + 1 \in S^{\max}$, which is the case if corresponding $\lceil \log_{k_3^{\max}} N_2^{\max} \rceil > 1$.

Fig. 19: $\Pi_{\mathsf{DPF.Hash}}$ – Subprotocol for Hash Evaluation

Given $\langle\!\langle \gamma_\ell \rangle\!\rangle_p$ compute $\langle\!\langle q_\ell \rangle\!\rangle_p$ such that $q_\ell = \mathsf{H}(\gamma_\ell \cdot G)$ in parallel for $\ell \in [t]$:

1. $\gamma_{\ell,0}, \gamma_{\ell,1} \leftarrow \mathcal{F}^{p,q}_{\mathsf{MPC}}.\mathsf{OutputShared}(\langle\!\langle \gamma_\ell \rangle\!\rangle_p)$ such that P_b receives $\gamma_{\ell,b}$ for $b \in \{0,1\}$. Abort if $\gamma_{\ell,b} = 0$. // $\gamma_{\ell,0} + \gamma_{\ell,1} = \gamma_\ell$
2. $[\![Q_{\ell,b}]\!]^b_q \leftarrow \mathcal{F}^{q,b}_{\mathsf{ZK}}.\mathsf{Input}(\gamma_{\ell,b} \cdot G)$ for $b \in \{0,1\}$.
3. $\mathcal{F}^{q,b}_{\mathsf{ZK}}.\mathsf{CheckZero}(\mathcal{F}^{q,b}_{\mathsf{ZK}}.\mathsf{Eval}(\mathcal{C}_{\mathsf{EC}}, [\![Q_{\ell,b}]\!]^b_q))$ for $b \in \{0,1\}$. // *verify* $Q_{\ell,b} \in \mathbb{G}$
4. $(\langle\!\langle Q_{\ell,b} \rangle\!\rangle_q)_{\ell \in [t]} \leftarrow \mathtt{ComInput}(q, \mathsf{P}_b, ([\![Q_{\ell,b}]\!]^b_q)_{\ell \in [t]})$ for $b \in \{0,1\}$.
5. $\langle\!\langle Q_\ell \rangle\!\rangle_q \leftarrow \mathcal{F}^{p,q}_{\mathsf{MPC}}.\mathsf{Eval}(q, \mathcal{C}_{\mathsf{ADD}}, \langle\!\langle Q_{\ell,0} \rangle\!\rangle_q, \langle\!\langle Q_{\ell,1} \rangle\!\rangle_q)$. // $Q_\ell = \gamma_\ell \cdot G$
6. $\mathcal{F}^{p,q}_{\mathsf{MPC}}.\mathsf{CheckEC}(\{(\langle\!\langle \gamma_\ell \rangle\!\rangle_p, \langle\!\langle Q_\ell \rangle\!\rangle_q)\}_{\ell \in [t]})$. // *verify* $Q_\ell = \gamma_\ell \cdot G$
7. Compute $\langle\!\langle \gamma'_\ell \rangle\!\rangle_q \leftarrow \langle\!\langle Q_\ell \rangle\!\rangle_q.\mathsf{x}$, run $\gamma'_{\ell,0}, \gamma'_{\ell,1} \leftarrow \mathcal{F}^{p,q}_{\mathsf{MPC}}.\mathsf{OutputShared}(\langle\!\langle \gamma'_\ell \rangle\!\rangle_q)$ s.t. P_b receives $\gamma'_{\ell,b}$ for $b \in \{0,1\}$. Abort if $\gamma'_{\ell,b} = 0$. // $\gamma'_{\ell,0} + \gamma'_{\ell,1} = Q_\ell.\mathsf{x}$
8. $[\![Q'_{\ell,b}]\!]^b_p \leftarrow \mathcal{F}^{p,b}_{\mathsf{ZK}}.\mathsf{Input}(\gamma'_{\ell,b} \cdot \overline{G})$ for $b \in \{0,1\}$.
9. $\mathcal{F}^{p,b}_{\mathsf{ZK}}.\mathsf{CheckZero}(\mathcal{F}^{p,b}_{\mathsf{ZK}}.\mathsf{Eval}(\mathcal{C}_{\mathsf{EC}}, [\![Q'_{\ell,b}]\!]^b_p))$ for $b \in \{0,1\}$. // *verify* $Q'_{\ell,b} \in \overline{\mathbb{G}}$
10. $\langle\!\langle Q'_{\ell,b} \rangle\!\rangle_p \leftarrow \mathtt{ComInput}(p, \mathsf{P}_b, [\![Q'_{\ell,b}]\!]^b_p)$ for $b \in \{0,1\}$.
11. $\langle\!\langle Q'_\ell \rangle\!\rangle_p \leftarrow \mathcal{F}^{p,q}_{\mathsf{MPC}}.\mathsf{Eval}(p, \mathcal{C}_{\mathsf{ADD}}, \langle\!\langle Q'_{\ell,0} \rangle\!\rangle_p, \langle\!\langle Q'_{\ell,1} \rangle\!\rangle_p)$.
12. $\mathcal{F}^{p,q}_{\mathsf{MPC}}.\mathsf{CheckEC}(\{(\langle\!\langle \gamma'_\ell \rangle\!\rangle_q, \langle\!\langle Q'_\ell \rangle\!\rangle_p)\}_{\ell \in [t]})$. // *verify* $Q'_\ell = \gamma'_\ell \cdot \overline{G}$
13. $\langle\!\langle q_\ell \rangle\!\rangle_p \leftarrow \langle\!\langle Q'_\ell \rangle\!\rangle_p.\mathsf{x}$. // $q_\ell = \mathsf{H}(Q_\ell)$

Fig. 20: $\Pi_{\mathsf{DPF.CHHK25}}$ – Subprotocol for CHHK25-PPRF Keygen

Let $\mathsf{interface} = \{\lambda, \{N_i\}_{i \in [\ell]}, \mathbb{F}\}$ be the interface of CHHK25-PPRF with spec for $N \in \{N_i\}_{i \in [\ell]}$ as in Figure 20.

1. P_0 samples $e_\ell \xleftarrow{\$} \mathbb{F}_p$ and $z_\ell \in [0, N)$, and $\langle\!\langle e_\ell \rangle\!\rangle_p \leftarrow \mathcal{F}^{p,q}_{\mathsf{MPC}}.\mathsf{Input}(p, \mathsf{P}_0, e_\ell)$ for $\ell \in [t]$. Let $\boldsymbol{z}_\ell \in \{0,1\}^n$ denote the bit decomposition of z_ℓ such that $\sum_{i=1}^n 2^{i-1} \cdot z_{\ell,i} = z_\ell$. // *defines point function* $z_\ell \mapsto \Delta \cdot e_\ell$
2. Key Generation, for $\ell \in [t]$ do in parallel:
   (a) $[\![a_{\ell,i}]\!]^1_p \leftarrow \mathcal{F}^{p,1}_{\mathsf{ZK}}.\mathsf{Random}()$ for $i \in [n]$.
   (b) P_1 computes $\mathsf{sk}_\ell := a_\ell := a_0^B \cdot \prod_{i=1}^n a_{\ell,i}$ and aborts if $a_\ell = 0$.
3. Oblivious puncturing, for $\ell \in [t]$ do:
   (a) $[\![\mu_\ell]\!]^1_p \leftarrow \mathcal{F}^{p,1}_{\mathsf{ZK}}.\mathsf{Random}()$.
   (b) Set $[\![\alpha^{(0)}_{\ell,1}]\!]^1_p \leftarrow [\![\mu_\ell]\!]^1_p \cdot [\![a_{\ell,1}]\!]^1_p$ and $[\![\alpha^{(1)}_{\ell,1}]\!]^1_p \leftarrow [\![\mu_\ell]\!]^1_p \cdot [\![a_{\ell,1}]\!]^1_p \cdot [\![r]\!]^1_p$.
   (c) Set $[\![\alpha^{(0)}_{\ell,i}]\!]^1_p = [\![a_{\ell,i}]\!]^1_p$ and $[\![\alpha^{(1)}_{\ell,i}]\!]^1_p \leftarrow [\![a_{\ell,i}]\!]^1_p \cdot [\![r^{2^{i-1}}]\!]^1_p$ for $i \in [2, n]$.
   (d) P_1 samples MAC keys $\Gamma_\alpha, K_{\alpha_{\ell,i}} \xleftarrow{\$} \mathbb{F}_p$ for $i \in [n]$, and computes MAC tags $M^{(b)}_{\alpha_{\ell,i}} := \Gamma_\alpha \cdot \alpha^{(b)}_{\ell,i} + K_{\alpha_{\ell,i}}$ for $i \in [n]$ and $b \in \{0,1\}$.
   (e) Run $\mathcal{F}^{p,1}_{\mathsf{ZK}}.\mathsf{OTOpen}\Big(([\![\alpha^{(0)}_{\ell,i}]\!]^1_p, M^{(0)}_{\alpha_{\ell,i}}), ([\![\alpha^{(1)}_{\ell,i}]\!]^1_p, M^{(1)}_{\alpha_{\ell,i}}); z_{\ell,i}\Big)$ for $i \in [n]$ such that P_0 receives either $\perp$ or $\left(\alpha^{(z_{\ell,i})}_{\ell,i}, M^{(z_{\ell,i})}_{\alpha_{\ell,i}}\right)_{i \in [n]}$. If $\perp$ was received, P_0 samples $\alpha_{\ell,i}, M_{\alpha_{\ell,i}} \xleftarrow{\$} \mathbb{F}_p$; otherwise it sets $\alpha_{\ell,i} = \alpha^{(z_{\ell,i})}_{\ell,i}$ and $M_{\alpha_{\ell,i}} = M^{(z_{\ell,i})}_{\alpha_{\ell,i}}$ for $i \in [n]$, or $\perp$. // $\alpha_{\ell,i} = a_{\ell,i} \cdot r^{z_{\ell,i}}$
4. Prepare PPRF evaluation at punctured point $\mathsf{F}(z_\ell) = \mathsf{H}(\gamma_\ell \cdot G)$:
   (a) $([\![\alpha_{\ell,i}]\!]^0_p)_{\ell \in [t], i \in [n]} \leftarrow \mathtt{AuthCommit}\big(\mathsf{P}_0; (\alpha_{\ell,i})_{\ell \in [t], i \in [n]}, (M_{\alpha_{\ell,i}})_{\ell \in [t], i \in [n]}; \Gamma_\alpha, (K_{\alpha_{\ell,i}})_{\ell \in [t], i \in [n]}\big)$
   (b) Compute $[\![\beta_{\ell,0}]\!]^0_p \leftarrow \alpha_0^B \cdot \prod_{i=1}^n [\![\alpha_{\ell,i}]\!]^0_p$ for $\ell \in [t]$. P_0 aborts if $\beta_{\ell,0} = 0$. // $\beta_{\ell,0} = \mu_\ell \cdot a_\ell \cdot r^{z_\ell - B}$
   (c) Run $(\langle\!\langle \beta_{\ell,0} \rangle\!\rangle_p)_{\ell \in [t]} \leftarrow \mathtt{ComInput}(p, \mathsf{P}_0; ([\![\beta_{\ell,0}]\!]^0_p)_{\ell \in [t]})$.
   (d) Compute $[\![\beta_{\ell,1}]\!]^1_p \leftarrow [\![\rho]\!]^1_p \cdot ([\![\mu_\ell]\!]^1_p)^{-1}$ for $\ell \in [t]$.
   (e) Run $(\langle\!\langle \beta_{\ell,1} \rangle\!\rangle_p)_{\ell \in [t]} \leftarrow \mathtt{ComInput}(p, \mathsf{P}_1; ([\![\beta_{\ell,1}]\!]^1_p)_{\ell \in [t]})$.
   (f) Compute $\langle\!\langle \gamma_\ell \rangle\!\rangle_p \leftarrow \langle\!\langle \beta_{\ell,0} \rangle\!\rangle_p \cdot \langle\!\langle \beta_{\ell,1} \rangle\!\rangle_p$ for $\ell \in [t]$. // $\gamma_\ell = \rho \cdot \alpha_\ell$
5. Compute $\langle\!\langle q_\ell \rangle\!\rangle_p \leftarrow \mathsf{H}(\langle\!\langle \gamma_\ell \rangle\!\rangle_p \cdot G)$ for $\ell \in [t]$ using $\Pi_{\mathsf{DPF.Hash}}$ (Figure 19).
6. Reveal punctured keys, for $\ell \in [t]$ do in parallel:
   (a) P_0 receives $\mu_\ell \leftarrow \mathcal{F}^{p,1}_{\mathsf{ZK}}.\mathsf{Open}([\![\mu_\ell]\!]^1_p)$.
   (b) P_0 computes $\alpha_\ell := \beta_{\ell,0} \cdot \mu_\ell^{-1}$ and sets $\mathsf{psk}_\ell := (\alpha_\ell, z_\ell)$.
7. Compute correction value v_ℓ, for $\ell \in [t]$ do in parallel:
   (a) Compute $\langle\!\langle v_\ell \rangle\!\rangle_p \leftarrow \langle\!\langle q_\ell \rangle\!\rangle_p - \langle\!\langle \Delta \rangle\!\rangle_p \cdot \langle\!\langle e_\ell \rangle\!\rangle_p$.
   (b) P_0 receives $v_\ell \leftarrow \mathcal{F}^{p,q}_{\mathsf{MPC}}.\mathsf{Output}(p, \mathsf{P}_0, \langle\!\langle v_\ell \rangle\!\rangle_p)$.
8. P_0 outputs $\left(\mathsf{k}^{\mathsf{DPF}}_{0,(\ell)} := (\mathsf{psk}_\ell, v_\ell)\right)_{\ell \in [t]}$ and P_1 outputs $\left(\mathsf{k}^{\mathsf{DPF}}_{1,(\ell)} := \mathsf{sk}_\ell\right)_{\ell \in [t]}$.

Fig. 21: $\Pi_{\mathsf{DPF.kNR}}$ – Subprotocol for k-NR-PPRF Keygen

Let $\mathsf{interface} = \{\lambda, \{N_i\}_{i\in[\ell]}, \mathbb{F}\}$ be the interface of k-NR-PPRF with spec for $N \in \{N_i\}_{i\in[\ell]}$ as in Figure 7.

1. P_0 samples $e_\ell \xleftarrow{\$} \mathbb{F}_p$ and $z_\ell \in [0, N)$, and $\langle\!\langle e_\ell \rangle\!\rangle_p \leftarrow \mathcal{F}_{\mathsf{MPC}}^{p,q}.\mathsf{Input}(p, \mathsf{P}_0, e_\ell)$ for $\ell \in [t]$. Let $\boldsymbol{z}_\ell \in [0,k)^n$ denote the k-ary decomposition of z_ℓ such that $\sum_{i=1}^n k^{i-1} \cdot z_{\ell,i} = z_\ell$. // *defines point function* $z_\ell \mapsto \Delta \cdot e_\ell$
2. Key Generation, for $\ell \in [t]$ do in parallel:
   (a) $[\![a_{\ell,1,i}]\!]_p^1, [\![a_{\ell,1+i}]\!]_p^1 \leftarrow \mathcal{F}_{\mathsf{ZK}}^{p,1}.\mathsf{Random}()$ for $i \in [n]$ and compute $[\![a_{\ell,1}]\!]_p^1 = \prod_{i=1}^n [\![a_{\ell,1,i}]\!]_p^1$.
   (b) P_1 sets $\mathsf{sk}_\ell := \boldsymbol{a}_\ell = (a_{\ell,1}, \ldots, a_{\ell,n'})$.
3. Oblivious puncturing, for $\ell \in [t]$ do:
   (a) Prepare commitments for punctured key:
      i. Sample masks $[\![\mu_{\ell,i}]\!]_p^1 \leftarrow \mathcal{F}_{\mathsf{ZK}}^{p,1}.\mathsf{Random}()$ for $i \in [n']$.
      ii. $[\![\alpha_{\ell,1,1}^{(j)}]\!]_p^1 \leftarrow [\![\mu_{\ell,1}]\!]_p^1 \cdot [\![a_{\ell,1,1}]\!]_p^1 \cdot [\![r^{-j^2}]\!]_p^1$ for $j \in [0,k)$
      iii. $[\![\alpha_{\ell,1,i}^{(j)}]\!]_p^1 \leftarrow [\![a_{\ell,1,j}]\!]_p^1 \cdot [\![r^{-j^2}]\!]_p^1$ for $j \in [0,k)$ and $i \in [2,n]$
      iv. $[\![\alpha_{\ell,1+i}^{(j)}]\!]_p^1 \leftarrow [\![\mu_{\ell,1+i}]\!]_p^1 \cdot [\![a_{\ell,1+i}]\!]_p^1 \cdot [\![r^{2j}]\!]_p^1$ for $j \in [0,k)$ and $i \in [n]$
   (b) Prepare commitments for evaluation at punctured point:
      i. Sample masks $[\![\nu_{\ell,i}]\!]_p^1 \leftarrow \mathcal{F}_{\mathsf{ZK}}^{p,1}.\mathsf{Random}()$ for $i \in [n]$ and compute $[\![\nu_\ell]\!]_p^1 \leftarrow \prod_{i=1}^n [\![\nu_{\ell,i}]\!]_p^1$.
      ii. $[\![\eta_{\ell,0,i}^{(j)}]\!]_p^1 \leftarrow [\![\nu_{\ell,i}]\!]_p^1 \cdot [\![a_0^{j^2}]\!]_p^1 \cdot [\![a_{\ell,1+i}^{j}]\!]_p^1$ for $j \in [0,k)$ and $i \in [n]$.
      iii. P_1 samples MAC keys $\Gamma_\eta, K_{\eta_{\ell,0,i}} \xleftarrow{\$} \mathbb{F}_p$ for $i \in [n]$, and computes MAC tags $M_{\eta_{\ell,0,i}}^{(j)} := \Gamma_\eta \cdot \eta_{\ell,0,i}^{(j)} + K_{\eta_{\ell,0,i}}$ for $i \in [n]$ and $j \in [0,k)$.
   (c) Run $\mathcal{F}_{\mathsf{ZK}}^{p,1}.\mathsf{OTOpen}\Big(\big([\![\alpha_{\ell,1,i}^{(j)}]\!]_p^1, [\![\alpha_{\ell,1+i}^{(j)}]\!]_p^1, [\![\eta_{\ell,0,i}^{(j)}]\!]_p^1, M_{\eta_{\ell,0,i}}^{(j)}\big)_{j\in[0,k)}, z_{\ell,i}\Big)$ for $i \in [n]$ such that P_0 receives either $\perp$ or $\big(\alpha_{\ell,1,i}^{(z_{\ell,i})}, \alpha_{\ell,1+i}^{(z_{\ell,i})}, \eta_{\ell,0,i}^{(z_{\ell,i})}, M_{\eta_{\ell,0,i}}^{(z_{\ell,i})},\big)_{i\in[n]}$. If $\perp$ was received, P_0 samples $\eta_{\ell,0,i}, M_{\eta_{\ell,0,i}}, \xleftarrow{\$} \mathbb{F}_p$; otherwise it sets $\alpha_{\ell,1,i} = \alpha_{\ell,1,i}^{(z_{\ell,i})}$, $\beta_{\ell,1+i,0} := \alpha_{\ell,1+i}^{(z_{\ell,i})}$, $\eta_{\ell,0,i} := \eta_{\ell,0,i}^{(z_{\ell,i})}$, $M_{\eta_{\ell,0,i}} := M_{\eta_{\ell,0,i}}^{(z_{\ell,i})}$.
   (d) P_0 compute $\beta_{\ell,1,0} := \prod_{i=1}^n \alpha_{\ell,1,i}$, and aborts if $\beta_{\ell,1,0} = 0$. // $\boldsymbol{\beta}_{\ell,0} = \boldsymbol{\alpha}_\ell \odot \boldsymbol{\mu}_\ell$
4. Prepare PPRF evaluation at punctured point $\mathsf{F}(z_\ell) = \mathsf{H}(\gamma_\ell \cdot G)$:
   (a) $([\![\eta_{\ell,0,i}]\!]_p^0)_{\ell\in[t],i\in[n]} \leftarrow \mathtt{AuthCommit}\big(\mathsf{P}_0; (\eta_{\ell,0,i})_{\ell\in[t],i\in[n]}, (M_{\eta_{\ell,0,i}})_{\ell\in[t],i\in[n]}; \Gamma_\eta, (K_{\eta_{\ell,0,i}})_{\ell\in[t],i\in[n]}\big)$
   (b) Compute $[\![\eta_{\ell,0}]\!]_p^0 \leftarrow \prod_{i=1}^n [\![\eta_{\ell,0,i}]\!]_p^0$ for $\ell \in [t]$.
   (c) Run $(\langle\!\langle \eta_{\ell,0} \rangle\!\rangle_p)_{\ell\in[t]} \leftarrow \mathtt{ComInput}(p, \mathsf{P}_0; ([\![\eta_{\ell,0}]\!]_p^0)_{\ell\in[t]})$.
   (d) Compute $[\![\eta_{\ell,1}]\!]_p^1 \leftarrow [\![\rho]\!]_p^1 \cdot ([\![a_0]\!]_p^1)^{-1} \cdot ([\![\nu_\ell]\!]_p^1)^{-1}$ for $\ell \in [t]$.
   (e) Run $(\langle\!\langle \eta_{\ell,1} \rangle\!\rangle_p)_{\ell\in[t]} \leftarrow \mathtt{ComInput}(p, \mathsf{P}_1; ([\![\eta_{\ell,1}]\!]_p^1)_{\ell\in[t]})$.
   (f) Compute $\langle\!\langle \gamma_\ell \rangle\!\rangle_p \leftarrow \langle\!\langle \eta_{\ell,0} \rangle\!\rangle_p \cdot \langle\!\langle \eta_{\ell,1} \rangle\!\rangle_p$ for $\ell \in [t]$.
5. Compute $\langle\!\langle q_\ell \rangle\!\rangle_p \leftarrow \mathsf{H}(\langle\!\langle \gamma_\ell \rangle\!\rangle_p \cdot G)$ for $\ell \in [t]$ using $\Pi_{\mathsf{DPF.Hash}}$ (Figure 19).
6. Reveal punctured keys, for $\ell \in [t]$ do in parallel:
   (a) P_0 receives $(\mu_{\ell,i})_{i\in[n']} \leftarrow \mathcal{F}_{\mathsf{ZK}}^{p,1}.\mathsf{Open}(([\![\mu_{\ell,i}]\!]_p^1)_{i\in[n']})$.
   (b) P_0 computes $\alpha_{\ell,i} := \beta_{\ell,i,0} \cdot \mu_{\ell,i}^{-1}$ for $i \in [n']$ and sets $\mathsf{psk}_\ell := (\boldsymbol{\alpha}_\ell, z_\ell)$.
7. Compute correction value v_ℓ: same as Step 7 of the CHHK25-PPRF keygen protocol in Figure 20.
8. P_0 outputs $\big(\mathsf{k}_{0,(\ell)}^{\mathsf{DPF}} := (\mathsf{psk}_\ell, v_\ell)\big)_{\ell\in[t]}$ and P_1 outputs $\big(\mathsf{k}_{1,(\ell)}^{\mathsf{DPF}} := \mathsf{sk}_\ell\big)_{\ell\in[t]}$.

Setup. After inputting Δ into $\mathcal{F}_{\mathsf{MPC}}^{p,q}$ and verifying the MAC, P_1 commits to a freshly generated master secret key msk and sends the corresponding master public key mpk to P_0. Then it proves in zero-knowledge that mpk is correct w.r.t. the committed msk. Finally, P_1 commits to certain values that are needed multiple times during key generation.

Macros. In the description of our protocol we use two helper macros `AuthCommit` and `ComInput` that are defined in Figure 17. `AuthCommit` verifies that a party commits to the right values by verifying information-theoretic MACs `ComInput` ensures that a party inputs the same values it is already committed to via $\mathcal{F}_{\mathsf{ZK}}$ into $\mathcal{F}_{\mathsf{MPC}}^{p,q}$.

Hash. Given sharings of $\gamma_\ell \in \mathbb{F}_p$ for $\ell \in [t]$, the subprotocol compute $\mathsf{H}(\gamma_\ell \cdot G) = ((\gamma_\ell \cdot G).\mathsf{x} \cdot \overline{G}).\mathsf{x}$: First the parties obtain additive shares of γ_ℓ and locally multiply their share with G to obtain (unauthenticated) additive shares of $\gamma_\ell \cdot G$ over the group $\mathbb{G}$. These are again input into $\mathcal{F}_{\mathsf{MPC}}^{p,q}$ as tuples over $\mathbb{F}_q$ and the point addition using an arithmetic circuit. $\mathcal{F}_{\mathsf{MPC}}^{p,q}.\mathsf{CheckEC}$ is used to enforce consistency of the result w.r.t. the original γ_ℓ, and these steps are repeated for the second scalar multiplication.

Keygen. Both $\Pi_{\mathsf{DPF.CHHK25}}$ and $\Pi_{\mathsf{DPF.kNR}}$ proceed in a similar fashion: P_1 commits to a new secret key sk and prepares commitments to a set of related values. These are used in an oblivious puncturing protocol where P_0 inputs its bit-decomposed (or k-ary) decomposed indices z_ℓ into OTOpen. It receives the values of some of these commitments, which are used to obtain a masked version of the punctured key psk punctured at point z_ℓ. To compute the correction value, the parties evaluate the PPRF at the punctured point: They prepare multiplicative shares of γ_ℓ, compute the product with $\mathcal{F}_{\mathsf{MPC}}^{p,q}$, and run the hashing protocol $\Pi_{\mathsf{DPF.Hash}}$. The output $q_\ell = \mathsf{H}(\gamma_\ell \cdot G)$ is then used compute the correction value $v_\ell = q_\ell - \Delta \cdot e_\ell$, which is together with psk output to P_1.

In the remainder of this section we prove:

Theorem 11. *The protocol Π_{DPF} (Figures 17, 18, 19, 20, 21) realizes the functionality $\mathcal{F}_{\mathsf{DPF}}$ (Figure 16) for $\mathbb{F}_p$ when* DPF = Compound-DPF.

10.3 Macro Properties

We first prove the soundness of the consistency checks used in the macros `AuthCommit` and `ComInput` given in $\Pi_{\mathsf{DPF.Macros}}$ (Figure 17).

Lemma 12 (Soundness of `AuthCommit`). *Let $x_1, \dots, x_n \in \mathbb{F}_p$. For $\Gamma, K_{x_1}, \dots, K_{x_n} \xleftarrow{\$} \mathbb{F}_p$, $K_{x_i} = \Gamma \cdot x_i + K_{x_i}$ for $i \in [n]$, and any $x'_1, \dots, x'_n, M'_{x_1}, \dots, M'_{x_n} \in \mathbb{F}_p$ if $[\![x'_1]\!]_p^b, \dots, [\![x'_n]\!]_p^b \leftarrow \mathtt{AuthCommit}(\mathsf{P}_b; *, *; \Gamma, (K_{x_i})_{i\in[n]})$ does not abort, then $x'_i = x_i$ for all $i \in [n]$ except with probability $(n+1)/p$.*

Proof. Let $x'_i = x_i + \delta_i$ for $i \in [n]$ be the values P_b commits to in Step 1, and let $M'_{\hat{x}}$ be the value P_b commits to in Step 3. Define $\hat{x} := \sum_{i=1}^n \chi^i \cdot x_i$, $K_{\hat{x}} := \sum_{i=1}^n \chi^i \cdot K_{x_i}$, $\hat{\delta} := \sum_{i=1}^n \chi^i \cdot \delta_i$, and $\hat{x}' := \sum_{i=1}^n \chi^i \cdot x'_i = \hat{x} + \hat{\delta}$. Suppose at least one of $\delta_i \neq 0$. Then we have $\hat{\delta} = 0$ with probability at most n/p, since $\chi \xleftarrow{\$} \mathbb{F}_p$ independently of δ. Suppose this does not happen, and observe that `AuthCommit` succeeds iff

$$\Gamma \cdot \hat{x}' + K_{\hat{x}} = M'_{\hat{x}} \iff \Gamma \cdot \hat{\delta} + \Gamma \cdot \hat{x} + K_{\hat{x}} = M'_{\hat{x}}$$

$$\iff \Gamma \cdot \hat{\delta} + M_{\hat{x}} = M'_{\hat{x}} \overset{\hat{\delta}\neq 0}{\iff} \Gamma = (M'_{\hat{x}} - M_{\hat{x}}) \cdot \hat{\delta}^{-1}.$$

So P_b must have guessed Γ which happens with probabilty $1/p$. Taking a union bound, we obtain total soundness error of at most $(n+1)/p$. □

Lemma 13 (Soundness of `ComInput`). *Let $[\![x_1]\!]_p^b, \dots, [\![x_n]\!]_p^b$ values committed to via $\mathcal{F}_{\mathsf{ZK}}^{p,b}$. If $\langle\!\langle x'_1 \rangle\!\rangle_p, \dots, \langle\!\langle x'_n \rangle\!\rangle_p \leftarrow \mathtt{ComInput}(p, \mathsf{P}_b; [\![x_1]\!]_p^b, \dots, [\![x_n]\!]_p^b)$ does not abort, then $x'_i = x_i$ for $i \in [n]$ except with probability n/p.*

Proof. Let $x'_i = x_i + \delta_i$ and suppose there exists a $j \in [n]$ such that $\delta_j \neq 0$. Moreover, let σ be the value P_b commits to in Step 2 and $\sigma' = \sigma + \delta_\sigma$ the value it inputs into $\mathcal{F}_{\mathsf{MPC}}^{p,q}$. In Step 4, we compute $\mu = \sigma + \sum_{i=1}^n \chi^i \cdot x_i$. The check in Step 5 passes iff $\mu = \sigma' + \sum_{i=1}^n \chi^i \cdot x'_i \iff 0 = \delta_\sigma + \sum_{i=1}^n \chi^i \cdot \delta_i$, i.e., when the polynomial $f(X) = \delta_\sigma + \sum_{i=1}^n X^i \cdot \delta_i$ of degree at most n has a root at χ. Since f has at most n roots and χ is sampled uniformly and independently at random after f is fixed, the probability that $f(\chi) = 0$ is at most n/p. □

10.4 Security Proof – Setup

Correctness

1. The MAC check Step 1 is equivalent to the check that is done by $\mathcal{F}_{\mathsf{DPF}}$.
2. In Step 2, P_1 samples msk using $\mathcal{F}_{\mathsf{ZK}}.\mathsf{Random}$, i.e., uniformly from $\mathbb{F}_p$. while $\mathcal{F}_{\mathsf{DPF}}$ samples it from $\mathbb{F}_p^*$. However, due to field size $p = \Omega(2^{2\lambda})$ this change is statistically indistinguishable and 0 is only sampled with negligible probability. Also at other places in the protocol, we assume that $\mathcal{F}_{\mathsf{ZK}}.\mathsf{Random}$ never returns 0.
3. P_1 computes the mpk according to definition in Step 3.
4. The consistency proof in Step 5 goes through: given $\mathsf{msk} = (\rho, r, a_0)$
   (a) $\alpha_0 = a_0 \cdot r^{-1}$ is checked directly.
   (b) To show that $H_s = r^s \cdot H = r^s \cdot \rho \cdot G$, we check
      i. $H_1 = r \cdot \rho \cdot G$,
      ii. $H_{i+1} = r \cdot H_i$ for all $i \in [S_a, -1) \cup [1, S_b)$,
      iii. and to bridge the gap at H_0, we check $H_1 = r^2 \cdot H_{-1}$.
5. Hence, the output $(\mathsf{st}_0^{\mathsf{DPF}}, \mathsf{st}_1^{\mathsf{DPF}})$ are correct in Step 8.
6. The remainder of the protocol just precomputes commitments for the key generation part. The correctness of the proven simple relations is confirmed by inspection.

Simulation. The simulator is given in Figure 22. We treat the cases where the environment $\mathcal{Z}$ corrupts P_0 and P_1 separately.

Simulation – Corrupted P_0. P_0 does send any messages besides sending its inputs – the keys Γ_Δ, K_Δ for the information-theoretic MAC on P_1's Δ – to $\mathcal{F}_{\mathsf{MPC}}^{p,q}$. The MAC check computed with $\mathcal{F}_{\mathsf{MPC}}^{p,q}$ in the protocol is identical to the check by $\mathcal{F}_{\mathsf{DPF}}$ in the simulation. P_0 has no influence during the computation of $\mathsf{Compound\text{-}DPF}.\mathsf{Setup}^{\mathcal{A},0}$, and the distribution of the $\mathsf{st}_0^{\mathsf{DPF}}$ computed by $\mathcal{F}_{\mathsf{DPF}}$ is the same that an honest P_1 would send.

Thus the environment $\mathcal{Z}$'s view in the simulation is statistically close to that in the real protocol execution, where the only difference stems from the tiny probability that P_1 samples an invalid msk and the real protocol thus aborts due to the $\mathsf{CheckNonZero}$ as noted in the correctness section above, whereas this would never happen in the simulation.

Simulation – Corrupted P_1. P_1 chooses its $\Delta, M_[\Delta]$ for the consistency check. Again, the MAC check computed with $\mathcal{F}_{\mathsf{MPC}}^{p,q}$ in the protocol is identical to the check by $\mathcal{F}_{\mathsf{DPF}}$ in the simulation. If the msk chosen by P_1 is invalid (has zero elements s.t. $\mathsf{Compound\text{-}PPRF}.\mathsf{CheckMSK}(\mathsf{msk}) = \bot$), the simulation always aborts, same as the real protocol due to the $\mathsf{CheckNonZero}$. Next, P_1 sends mpk'. If it is invalid, i.e., $\mathsf{mpk}' \neq \mathsf{Compound\text{-}PPRF}.\mathsf{ComputeMPK}(\mathsf{msk})$ for the extracted msk at least one of the consistency checks in Step 5 will catch it, and the simulator ensures that the corresponding simulated check fails as well. If P_1 tries to cheat during $\mathtt{ComInput}$ by inputting inconsistent values, the simulation will always simulate an abort, whereas in the protocol the check passes with probability $1/p$ (see Lemma 13) which is negligible for our choice of p. Finally, if P_1 tries to commit to invalid values in the remainder of the protocol, it will be caught in the protocol due to $\mathcal{F}_{\mathsf{ZK}}$ as well as in the simulation.

Thus the simulation of $\Pi_{\mathsf{DPF.Setup}}$ is statistically indistinguishable from the real protocol execution.

10.5 Security Proof – CHHK25-PPRF Keygen

Correctness. Let $\mathsf{interface}$ and spec of the PPRF F be as in CHHK25-PPRF.

1. In Step 2, P_1 generates the secret key $\mathsf{sk}_\ell = a_\ell$ as the product $a_\ell = a_0^B \cdot \prod_{i=1}^n a_{\ell,i}$. Since the $a_{\ell,i}$ are uniformly random, a_ℓ has the right distribution.
2. In Step 3, P_1 prepares two values for each $i \in [1, n]$ where $\alpha_{\ell,i}^{(0)} = a_{\ell,i}$ and $\alpha_{\ell,i}^{(1)} = a_{\ell,i} \cdot r^{2^{i-1}}$; thus $\alpha_{\ell,i}^{(b)} = a_{\ell,i} \cdot r^{b \cdot 2^{i-1}}$ for $b \in \{0, 1\}$. In the case of $i = 1$, the values $\alpha_{\ell,1}^{(b)}$ are additionally multiplicatively masked with a random value μ_ℓ. Moreover, it authenticates these values with information-theoretic MAC tags $M_{\alpha_{\ell,i}}^{(b)}$ such that for each pair of values, both MAC tags $M_{\alpha_{\ell,i}}^{(0)}$ and $M_{\alpha_{\ell,i}}^{(1)}$ are computed with the same MAC keys Γ_α (global for all i, ℓ) and $K_{\alpha_{\ell,i}}$.

Fig. 22: Simulator for Π_{DPF}

Setup – Corrupted P_0

1. Receive $\Gamma_\Delta, K_\Delta \in \mathbb{F}_p$ from P_0 in Step 1 and send them to $\mathcal{F}_{\mathsf{DPF}}$.
2. If $\mathcal{F}_{\mathsf{DPF}}$ aborts, simulate $\mathcal{F}_{\mathsf{MPC}}^{p,q}$.CheckZero failing in Step 1c and end the simulation.
3. Otherwise, receive mpk from $\mathcal{F}_{\mathsf{DPF}}$.
4. Simulate success of CheckNonZero in Step 2b and all the CheckZero and CheckDLog in the protocol.
5. If $\mathcal{Z}$ has not send abort, send continue to $\mathcal{F}_{\mathsf{DPF}}$.

Setup – Corrupted P_1

1. Extract Δ, M_Δ from P_1's messages to $\mathcal{F}_{\mathsf{MPC}}^{p,q}$.Input in Step 1 and send them to $\mathcal{F}_{\mathsf{DPF}}$.
2. If $\mathcal{F}_{\mathsf{DPF}}$ aborts, simulate $\mathcal{F}_{\mathsf{MPC}}^{p,q}$.CheckZero failing in Step 1c and end the simulation.
3. Extract $\mathsf{msk} = (\rho, r, a_0)$ from P_1's messages to $\mathcal{F}_{\mathsf{ZK}}^{p,1}$.Random in Step 2. If one of them is 0, simulate failure of CheckNonZero in Step 2b and abort.
4. Receive mpk from P_1 in Step 3. (NB: if $\mathsf{mpk} \neq \mathsf{ComputeMPK}(\mathsf{msk})$, we will abort later).
5. Simulate the consistency checks in Step 5 by keeping track of the values in the commitments. Simulate failing checks when the checked condition does not hold and simulate abort in this case (NB: if $\mathsf{mpk} \neq \mathsf{ComputeMPK}(\mathsf{msk})$, one of them wil fail).
6. Simulate `ComInput`:
   (a) Extract ρ', σ, σ' from P_1's messages to $\mathcal{F}_{\mathsf{ZK}}$ and $\mathcal{F}_{\mathsf{MPC}}^{p,q}$.
   (b) Send $\chi \xleftarrow{\$} \mathbb{F}_p$ to P_1.
   (c) If $\rho = \rho'$ and $\sigma = \sigma'$, simulate $\mathcal{F}_{\mathsf{ZK}}$.CheckZero succeeding; otherwise, simulate an abort.
7. Simulate consistency checks on extra commitments as in Step 5 above.
8. If $\mathcal{Z}$ has not send abort, send continue to $\mathcal{F}_{\mathsf{DPF}}$.

Keygen

1. For CHHK25-PPRF: Proceed as in Figure 23 if P_0 is corrupted, and as in Figure 24 if P_1 is corrupted.
2. For k-NR-PPRF: Proceed as in Figure 25.

3. When the OTOpen is invoked with the bit decomposition of $z \in [0, N)$ as P_0's choice bits, P_0 receives $\alpha_{\ell,i}^{(z_{\ell,i})}$ and $M_{\alpha_{\ell,i}}^{(z_{\ell,i})}$ for $i \in [n]$ where $z_\ell = \sum_{i=1}^n z_{\ell,i} \cdot 2^{i-1}$.
4. In Step 4, the `AuthCommit` macro is used for P_0 to commit to the received values $\alpha_{\ell,i}$ via $\mathcal{F}_{\mathsf{ZK}}^{p,0}$ while checking that the provided MAC tags supplied by P_0 verify against the MAC keys given by P_1. If both parties are honest, the check passes.
5. Next $\mathcal{F}_{\mathsf{ZK}}^{p,0}$ is used to obtain a commitment $[\![\beta_{\ell,0}]\!]_p^0$ to the product of these values together with α_0^B such that

$$\begin{aligned}\beta_{\ell,0} &= \alpha_0^B \cdot \prod_{i=1}^n \alpha_{\ell,i}^{(z_{\ell,i})} = \mu_\ell \cdot a_0^B \cdot r^{-B} \cdot \prod_{i=1}^n a_{\ell,i} \cdot r^{z_{\ell,i} \cdot 2^{i-1}} \\ &= \mu_\ell \cdot \left(a_0^B \cdot \prod_{i=1}^n a_{\ell,i} \right) \cdot r^{\sum_{i=1}^n z_{\ell,i} \cdot 2^{i-1} - B} = \mu_\ell \cdot a_\ell \cdot r^{z_\ell - B}.\end{aligned}$$

   Note that $\mathsf{psk}_\ell := (N, \alpha_\ell, z_\ell) = \mathsf{CHHK25\text{-}PPRF.Puncture}(\mathsf{msk}, \mathsf{sk}_\ell, z_\ell)$ and $\beta_{\ell,0} = \alpha_\ell \cdot \mu_\ell$, so P_0 has received the punctured key masked with μ_ℓ.
6. Then with `ComInput`, P_0 inputs the same $\beta_{\ell,0}$ into $\mathcal{F}_{\mathsf{MPC}}^{p,q}$.
7. P_1 first computes a commitment to $\beta_{\ell,1} := \rho \cdot \mu_\ell^{-1}$, and then with `ComInput` inputs $\beta_{\ell,1}$ into $\mathcal{F}_{\mathsf{MPC}}^{p,q}$.
8. Note that the evaluation of CHHK25-PPRF at the punctured point z_ℓ we need to compute

$$\begin{aligned}q_\ell &= \mathsf{CHHK25\text{-}PPRF.Eval}(\mathsf{msk}, \mathsf{sk}_\ell = a_\ell, z_\ell) \\ &= \mathsf{H}(a_\ell \cdot H_{z_\ell - B}) = \mathsf{H}(a_\ell \cdot r^{z_\ell - B} \cdot H) = \mathsf{H}(\alpha_\ell \cdot H) = \mathsf{H}(\alpha_\ell \cdot \rho \cdot H).\end{aligned}$$

9. Using $\mathcal{F}_{\mathsf{MPC}}^{p,q}$, we compute $\gamma_\ell := \beta_{\ell,0} \cdot \beta_{\ell,1} = \rho \cdot \alpha_\ell$ such that the factors μ_ℓ and μ_ℓ^{-1} cancel out such that γ_ℓ is the input to the hash function.

10. In Step 5 we use $\mathcal{F}_{\mathsf{MPC}}^{p,q}$ to securely compute

$$\gamma_\ell \mapsto q_\ell = \mathsf{H}(\gamma_\ell \cdot G) = \underbrace{(\underbrace{(\gamma_\ell \cdot G)}_{Q_\ell}.\mathsf{x} \cdot \overline{G})}_{Q'_\ell}.\mathsf{x}$$

with intermediate values $Q_\ell \in \mathbb{G}$ and $Q'_\ell \in \overline{\mathbb{G}}$:
   (a) To compute Q_ℓ we first let $\mathcal{F}_{\mathsf{MPC}}^{p,q}$ output an additive sharing $\gamma_{\ell,0} + \gamma_{\ell,1}$ of γ_ℓ.
   (b) Then each P_b locally computes $Q_{\ell,b} := \gamma_{\ell,b} \cdot G$ and commits to $Q_{\ell,b}$ using $\mathcal{F}_{\mathsf{ZK}}^{q,b}$ as a tuple over $\mathbb{F}_q$.
   (c) $\mathcal{F}_{\mathsf{ZK}}^{q,b}$ is used to prove that $Q_{\ell,b}$ is a valid point on the elliptic curve $\mathbb{G}$, which is the case for honest parties.
   (d) With `ComInput` these tuples are again input into $\mathcal{F}_{\mathsf{MPC}}^{p,q}$ over $\mathbb{F}_q$ and we use $\mathcal{F}_{\mathsf{MPC}}^{p,q}$ to securely evaluate the EC group addition formula to compute Q_ℓ (again a tuple over $\mathbb{F}_q$).
   (e) $\mathcal{F}_{\mathsf{MPC}}^{p,q}.\mathsf{CheckEC}$ is used to verify that $Q_\ell = \gamma_\ell \cdot G$ If both parties are honest, the check passes.
   (f) Defining $\gamma_\ell = Q_\ell.\mathsf{x}$ to be the first component of the tuple is a no-op.
   (g) The second part of the computation is to compute $q_\ell = Q'_\ell.\mathsf{x} = (\gamma_\ell \cdot \overline{G}).\mathsf{x}$ which works the same way except with $\mathbb{G}$ and $\overline{\mathbb{G}}$ swapped.
11. In Step 6, P_1 opens the commitment to the mask μ_ℓ such that P_0 can unmask $\mathsf{psk}_\ell = \alpha_\ell = \beta_{\ell,0} \cdot \mu_\ell^{-1}$.
12. Having the PPRF evaluation q_ℓ inside $\mathcal{F}_{\mathsf{MPC}}^{p,q}$ we compute the correction value $v_\ell = q_\ell - \Delta \cdot e_\ell$ in Step 7.

Simulation. We provide the simulator in Figures 23 and 24 and treat the cases when $\mathcal{Z}$ corrupts P_0 and P_1 separately.

Simulation – Corrupted P_0. Simulating the view of P_0 is relatively straight-forward, because all values it sees are uniformly field elements (sometimes excluding 0), because everything is masked until the psk_ℓ and v_ℓ are revealed.

Argue that the output $(\mathsf{k}_{0,(\ell)}^{\mathsf{DPF}}, \mathsf{k}_{1,(\ell)}^{\mathsf{DPF}})$ are correctly distributed w.r.t. P_0's behavior. The simulator first extracts P_0's inputs z_ℓ and e_ℓ for all $\ell \in [t]$ from P_0s messages to $\mathcal{F}_{\mathsf{ZK}}.\mathsf{OTOpen}$ and $\mathcal{F}_{\mathsf{MPC}}^{p,q}.\mathsf{Input}$ and forwards them to $\mathcal{F}_{\mathsf{DPF}}$. The functionality generates $(\mathsf{k}_{0,(\ell)}^{\mathsf{DPF}}, \mathsf{k}_{1,(\ell)}^{\mathsf{DPF}}) \leftarrow \mathsf{Compound\text{-}DPF}.\mathsf{Gen}((\mathsf{st}_0^{\mathsf{DPF}}, \mathsf{st}_1^{\mathsf{DPF}}), N, (z_\ell, \Delta \cdot e_\ell))$ and the simulator receives $\mathsf{k}_{0,(\ell)}^{\mathsf{DPF}}$.

During the protocol P_0 has limited options to cheat.

- If it tried to input invalid values (i.e., values it had not previously received from OTOpen) in `AuthCommit`, it will get caught with overwhelming probability in the protocol (see Lemma 12), whereas the simulation always aborts.
- Similarly, if it tried to input invalid values (i.e., values was not previously committed to via $\mathcal{F}_{\mathsf{ZK}}$) in `ComInput`, it will get caught with overwhelming probability in the protocol (see Lemma 13), whereas the simulation always aborts.

At the end of the simulation, the simulator sends exactly the right values as outputs of $\mathcal{F}_{\mathsf{ZK}}.\mathsf{Open}$ and $\mathcal{F}_{\mathsf{MPC}}^{p,q}.\mathsf{Output}$ such that P_0 computes as their output $\mathsf{k}_{0,(\ell)}^{\mathsf{DPF}}$. Thus the environment's view in the simulation is statistically close to the real protocol.

Simulation – Corrupted P_1. Similarly, all values that P_1 receives are random values. Here, the challenging part is to taking care that the choices of P_1 in the protocol are correctly translated into influence with the functionality $\mathcal{F}_{\mathsf{DPF}}$.

First there is the selected failure attack, and we need to extract the right index set I defined as the set of all index vectors $\boldsymbol{z}$ which would *not* lead to an abort of P_0 in the real protocol. P_1 has two options to trigger such an abort:

1. Using the selected failure of $\mathcal{F}_{\mathsf{ZK}}.\mathsf{OTOpen}$ such that the simulator obtains the index sets $I_\ell^{(1)}$.
2. Sending inconsistent information-theoretic MACs $M_{\alpha_{\ell,i}}^{(j)}$ as associated messages in $\mathcal{F}_{\mathsf{ZK}}.\mathsf{OTOpen}$. Here the abort depends on on which MAC keys $\Gamma_\alpha, K_{\alpha_{\ell,i}}$ P_1 provides during the `AuthCommit` in Step 4a, and the simulator can compute the index sets $I_\ell^{(2)}$.

Since the functionality $\mathcal{F}_{\mathsf{DPF}}$ only provides a single selected failure query, P_0 must not immediately abort if $\mathcal{F}_{\mathsf{ZK}}.\mathsf{OTOpen}$ returns $\perp$ instead proceeds with random values in `AuthCommit` such that this check fails with overwhelming probability. At this points, the simulator knows both index sets and can sends the intersection $I_\ell^{(1)} \cap I_\ell^{(2)}$ as query to $\mathcal{F}_{\mathsf{DPF}}$ which aborts of not, and $\mathcal{S}$ can correspondingly simulate `AuthCommit` failing or not.

Next, $\mathcal{F}_{\mathsf{DPF}}$ runs $\mathsf{Compound\text{-}DPF}.\mathsf{Gen}^{\mathcal{S},1}$ with the simulator. Here the influence is choosing sk_ℓ; so $\mathcal{S}$ passes the sk_ℓ it has extracted from $\mathcal{F}_{\mathsf{ZK}}.\mathsf{Random}$ in Step 2 to $\mathcal{F}_{\mathsf{DPF}}$. If $\mathcal{F}_{\mathsf{DPF}}$ aborts, then it is because $\mathsf{CheckSK}(\mathsf{msk}, N, \mathsf{sk}_\ell) = \perp$, which happens when $a_0 = 0$. In the protocol this would yield to P_0 computing $\beta_{\ell,0} = 0$ in Step 4b, so $\mathcal{S}$ continues the simulation until this point and then simulates P_0 aborting.

The only other point where P_1 can try to cheat is during `ComInput`, where it will be caught with overwhelming probability in the protocol. Hence, the view of the environment $\mathcal{Z}$ in the simulation is statistically close to the view in the real protocol.

Fig. 23: Simulator for $\Pi_{\mathsf{DPF.CHHK25}}$ (Part 1)

Corrupted P_0

1. Extract $(e_\ell)_{\ell\in[t]}$ from P_0's messages to $\mathcal{F}_{\mathsf{MPC}}^{p,q}.\mathsf{Input}$ in Step 1.
2. Simulate the oblivious puncturing, for $\ell \in [t]$:
   (a) Simulate the call to $\mathcal{F}_{\mathsf{ZK}}^{p,0}.\mathsf{OTOpen}$ in Step 3e for $i \in [n]$:
      i. Receive $z_{\ell,i} \in \{0,1\}$ from P_0.
      ii. Sample $\alpha_{\ell,i}, M_{\alpha_{\ell,i}} \xleftarrow{\$} \mathbb{F}_p$.
      iii. Send $(\alpha_{\ell,i}^{(z_{\ell,i})}, M_{\alpha_{\ell,i}}^{(z_{\ell,i})}) := (\alpha_{\ell,i}, M_{\alpha_{\ell,i}})$ to P_0.
   (b) Compute $z_\ell := \sum_{i=1}^n 2^{i-1} \cdot z_{\ell,i}$, and $\beta_{\ell,0} := \alpha_0^B \cdot \prod_{i=1}^n \alpha_{\ell,i}$.
3. Simulate `AuthCommit` in Step 4a:
   (a) Extract $(\alpha'_{\ell,i})_{\ell\in[t],i\in[n]}$ from P_0's messages to $\mathcal{F}_{\mathsf{ZK}}^{p,0}$.
   (b) Send $\chi \xleftarrow{\$} \mathbb{F}_p$ to P_0.
   (c) Receive $M_{\hat{\alpha}}$ from P_0.
   (d) If $\alpha'_{\ell,i} \neq \alpha_{\ell,i}$ for some $\ell \in [t], i \in [n]$ or $M_{\hat{\alpha}} \neq \sum_{\ell\in[t],i\in[n]} \chi^{(\ell-1)\cdot n+i} \cdot M_{\alpha_{\ell,i}}$ simulate $\mathcal{F}_{\mathsf{ZK}}^{p,0}.\mathsf{CheckZero}$ failing and simulate an abort. Otherwise, simulate $\mathsf{CheckZero}$ succeeding.
4. Simulate $\mathtt{ComInput}(p, \mathsf{P}_0, (\llbracket\beta_{\ell,0}\rrbracket_p^0)_{\ell\in[t]})$ in Step 4c:
   (a) Extract $(\beta'_{\ell,0})_{\ell\in[t]} \in \mathbb{F}_p^t$ from P_0's messages to $\mathcal{F}_{\mathsf{MPC}}^{p,q}$.
   (b) Extract $\sigma \in \mathbb{F}_p$ from P_0's message to $\mathcal{F}_{\mathsf{ZK}}^{p,0}$ and $\sigma' \in \mathbb{F}_p$ from its message to $\mathcal{F}_{\mathsf{ZK}}^{p,0}$.
   (c) Send $\chi \xleftarrow{\$} \mathbb{F}_p$ to P_0.
   (d) Simulate $\mathcal{F}_{\mathsf{ZK}}^{p,0}.\mathsf{Open}$ outputting μ.
   (e) If $\beta'_{\ell,0} \neq \beta'_{\ell,0}$ for some $\ell \in [t]$ or $\sigma' \neq \sigma$ simulate $\mathcal{F}_{\mathsf{MPC}}^{p,q}.\mathsf{CheckZero}$ failing and simulate an abort. Otherwise, simulate $\mathsf{CheckZero}$ succeeding.
5. Simulate $\mathtt{ComInput}(p, \mathsf{P}_1, (\llbracket\beta_{\ell,1}\rrbracket_p^1)_{\ell\in[t]})$ in Step 4e:
   (a) Receive $\chi \in \mathbb{F}_q$ from P_0.
   (b) Simulate $\mathcal{F}_{\mathsf{ZK}}^{q,1}.\mathsf{Open}$ by sending $\mu \xleftarrow{\$} \mathbb{F}_q$ to P_0.
   (c) Simulate $\mathcal{F}_{\mathsf{ZK}}^{q,1}.\mathsf{CheckZero}$ succeeding.
6. Simulate first part of the call to $\Pi_{\mathsf{DPF.Hash}}$ in Step 5:
   (a) Simulate $\mathcal{F}_{\mathsf{MPC}}^{p,q}.\mathsf{OutputShared}$ in Step 1 of $\Pi_{\mathsf{DPF.Hash}}$ by sending $\gamma_{\ell,0} \xleftarrow{\$} \mathbb{F}_p^*$ to P_0.
   (b) Compute $Q_{\ell,0} := \gamma_{\ell,0} \cdot G$ and extract $Q'_{\ell,0} \in \mathbb{F}_q^2$ from P_0's message to $\mathcal{F}_{\mathsf{ZK}}^{\mathbb{F}_q,0}$ in Step 2 of $\Pi_{\mathsf{DPF.Hash}}$.
   (c) If $Q'_{\ell,0} \in \mathbb{G}$, simulate $\mathcal{F}_{\mathsf{ZK}}^{\mathbb{F}_q,0}.\mathsf{CheckZero}$ in Step 3 of $\Pi_{\mathsf{DPF.Hash}}$ succeeding, otherwise simulate abort.
   (d) Simulate $\mathtt{ComInput}(q, \mathsf{P}_0, (\llbracket Q_{\ell,0}\rrbracket_q^0)_{\ell\in[t]})$ in Step 4 of $\Pi_{\mathsf{DPF.Hash}}$ $(b = 0)$ analogous to Step 4 of the simulation above (abort if P_0 tries to input something else than $(Q'_{\ell,0})_{\ell\in[t]}$).
   (e) Simulate $\mathtt{ComInput}(q, \mathsf{P}_1, (\llbracket Q_{\ell,1}\rrbracket_q^1)_{\ell\in[t]})$ in Step 4 of $\Pi_{\mathsf{DPF.Hash}}$ $(b = 1)$ analogous to Step 5 of the simulation above.
   (f) Simulate $\mathcal{F}_{\mathsf{MPC}}^{p,q}.\mathsf{CheckEC}$ in Step 6 of $\Pi_{\mathsf{DPF.Hash}}$ failing if $Q'_{\ell,0} \neq Q_{\ell,0}$ for some $\ell \in [t]$, and succeeding otherwise.
7. Simulate Steps 7 to 13 of $\Pi_{\mathsf{DPF.Hash}}$ analogous to Step 6 of the simulation above.

Continued in Figure 24.

Fig. 24: Simulator for $\Pi_{\mathsf{DPF.CHHK25}}$ (Part 2)

8. Simulate the key reveal:
   (a) Send (z_ℓ, e_ℓ) to $\mathcal{F}_{\mathsf{DPF}}$ and receive $\mathsf{k}^{\mathsf{DPF}}_{0,(\ell)} = \big(\mathsf{psk}_\ell = (N, \alpha_\ell, z_\ell), v_\ell\big) \in \mathbb{F}_p \times [0, N) \times \mathbb{F}_p$ for $\ell \in [t]$.
   (b) Compute $\mu_\ell := \beta_{\ell,0} \cdot \alpha_\ell^{-1}$ for $\ell \in [t]$.
   (c) Simulate $\mathcal{F}^{p,1}_{\mathsf{ZK}}.\mathsf{Open}$ in Step 6 by sending μ_ℓ to P_0 for $\ell \in [t]$.
   (d) Simulate $\mathcal{F}^{p,q}_{\mathsf{MPC}}.\mathsf{Output}$ in Step 7 by sending v_ℓ to P_0 for $\ell \in [t]$.
9. If $\mathcal{Z}$ has not send abort, send continue to $\mathcal{F}_{\mathsf{DPF}}$.

Corrupted P_1

1. Extract the $a_{\ell,i}$ for all $\ell \in [t], i \in [n]$ from P_1's messages to $\mathcal{F}^{p,1}_{\mathsf{ZK}}.\mathsf{Random}$ in Step 2.
2. Compute $\mathsf{sk}_\ell = a_\ell = a_0^B \cdot \prod_{i=1}^n a_{\ell,i}$ using a_0 from msk extracted during the simulation of the setup. (both keys contain the domain N as well.)
3. Continue simulating the protocol and verify with $\mathcal{F}_{\mathsf{ZK}}$ that the prepared commitments have the correct form.
4. Simulate the oblivious puncturing, for $\ell \in [t]$:
   (a) Simulate the call to $\mathcal{F}^{p,1}_{\mathsf{ZK}}.\mathsf{OTOpen}$ in Step 3e for $i \in [n]$:
      i. Extract $\alpha^{(b)}_{\ell,i}$ from the $\mathcal{F}^{p,1}_{\mathsf{ZK}}$ commitments for $b \in \{0,1\}$.
      ii. Receive $M^{(b)}_{\alpha_{\ell,i}} \in \mathbb{F}_p$ for $b \in \{0,1\}$ from P_1.
      iii. Receive selected-failure set $I^{(1)}_{\ell,i} \subseteq \{0,1\}$ from $\mathcal{Z}$.
   (b) $I^{(1)}_\ell := \left\{ z \in [0, 2^n) \;\middle|\; \exists \boldsymbol{z} \in I^{(1)}_{\ell,1} \times \cdots \times I^{(1)}_{\ell,n} \,.\, z = \sum_{i=1}^n 2^{i-1} \cdot z_i \right\}$.
   NB: if $z_\ell \notin I^{(1)}_\ell$, we need to fail later.
5. Simulate `AuthCommit` in Step 4a:
   (a) Receive $\chi \in \mathbb{F}_p$ from P_1.
   (b) Receive $\Gamma_\alpha, (K_{\alpha_{\ell,i}})_{\ell \in [t], i \in [n]} \in \mathbb{F}_p$ from P_1.
   (c) Define $I^{(2)}_{\ell,i} := \left\{ b \in \{0,1\} \;\middle|\; \Gamma_\alpha \cdot \alpha^{(b)}_{\ell,i} + K_{\alpha_{\ell,i}} = M^{(b)}_{\alpha_{\ell,i}} \right\}$, and $I^{(2)}_\ell$ analogous to $I^{(1)}_\ell$.
   (d) Send $I := \bigtimes_{\ell \in [t]} \left(I^{(1)}_\ell \cap I^{(2)}_\ell \right)$ to $\mathcal{F}_{\mathsf{DPF}}$.
   (e) If $\mathcal{F}_{\mathsf{DPF}}$ aborts, simulate failing $\mathcal{F}^{p,0}_{\mathsf{ZK}}.\mathsf{CheckZero}$; otherwise simulate $\mathsf{CheckZero}$ succeeding.
6. Send sk_ℓ to $\mathcal{F}_{\mathsf{DPF}}$ for $\ell \in [t]$.
7. If $\mathcal{F}_{\mathsf{DPF}}$ aborts due to $\mathsf{CheckSK}(\mathsf{msk}, N, \mathsf{sk}_\ell) = \bot$, we simulate P_0 aborting in Step 4b.
8. Simulate $\mathtt{ComInput}(p, \mathsf{P}_0, (\llbracket \beta_{\ell,0} \rrbracket^0_p)_{\ell \in [t]})$ in Step 4c analogously to Step 5 in the case of corrupted P_0.
9. Simulate $\mathtt{ComInput}(p, \mathsf{P}_1, (\llbracket \beta_{\ell,1} \rrbracket^1_p)_{\ell \in [t]})$ in Step 4e analogously to Step 4 in the case of corrupted P_0.
10. Simulate the call to $\Pi_{\mathsf{DPF.Hash}}$ (Step 5) analogously to Steps 6 and 7 of the simulation in the case of corrupted P_0 (this part of the protocol is symmetric).
11. The remainder of the simulation is trivial because P_1 does not receive any messages. If $\mathcal{Z}$ has not send abort, send continue to $\mathcal{F}_{\mathsf{DPF}}$.

10.6 Security Proof – k-NR-PPRF Keygen

Correctness. Let interface and spec of the PPRF F be as in k-NR-PPRF.

1. In Step 2, P_1 generates the secret key $\mathsf{sk}_\ell = \boldsymbol{a}_\ell = (a_{\ell,1}, \ldots, a_{\ell,n'})$ where $a_{\ell,1}$ is the product $a_{\ell,1} = \prod_{i=1}^{n} a_{\ell,1,i}$. Since the $a_{\ell,i}$ and $a_{\ell,1,i}$ are uniformly random, $\boldsymbol{a}_\ell$ has the right distribution.
2. Let $\boldsymbol{z}_\ell \in [0,k)^n$ be the k-ary decomposition of the index $z_\ell \in [0,N)$ such that $z_\ell = \sum_{i=1}^{n} z_{\ell,i} \cdot k^{i-1}$.

We first argue correctness of the punctured keys psk_ℓ that P_1 obtains. The punctured key $\mathsf{psk}_\ell := (N, \boldsymbol{\alpha}_\ell, z_\ell) = k\text{-NR-PPRF.Puncture}(\mathsf{msk}, \mathsf{sk}_\ell, z_\ell)$ has the form:

$$\mathsf{psk}_\ell = \boldsymbol{\alpha}_\ell = \left(\alpha_{\ell,1} = a_{\ell,1} \cdot r^{-\|\boldsymbol{z}_\ell\|^2}, \left(\alpha_{\ell,1+i} = a_{\ell,1+i} \cdot r^{2z_{\ell,i}}\right)_{i\in[n]}\right).$$

3. In Step 3a, P_1 samples random masks $\mu_{\ell,i}$ and prepares two k-tuples for each $i \in [1,n]$. For $j \in [0,k)$:
   (a) First, we have $\alpha^{(j)}_{\ell,1,i} = a_{\ell,1,j} \cdot r^{-j^2}$ for $i \in [n]$ (with $\alpha_{\ell,1,1}$ additionally multiplicatively masked by $\mu_{\ell,1}$).
   (b) Second, we have $\alpha^{(j)}_{\ell,1+i} = \mu_{\ell,1+i} \cdot a_{\ell,1+i} \cdot r^{2j}$ for $i \in [n]$ (all masked by $\mu_{\ell,1+i}$).
4. When the OTOpen is invoked in Step 3c with the $z_{\ell,i}$ as P_0's choices, P_0 receives
   (a) $\alpha^{(z_1)}_{\ell,1,1} = \mu_{\ell,1} \cdot a_{\ell,1,1} \cdot r^{-z^2_{\ell,1}}$.
   (b) $\alpha^{(z_i)}_{\ell,1,i} = a_{\ell,1,j} \cdot r^{-z^2_{\ell,i}}$ for $i \in [2,n]$.
   (c) $\beta_{\ell,1+i,0} := \alpha^{(z_i)}_{\ell,1+i} = \mu_{\ell,1+i} \cdot a_{\ell,1+i} \cdot r^{2z_{\ell,i}}$ for $i \in [n]$.
5. Observe that

$$\begin{aligned}\beta_{\ell,1,0} &:= \prod_{i=1}^{n} \alpha^{(z_i)}_{\ell,1,i} = \mu_{\ell,1} \cdot \prod_{i=1}^{n} a_{\ell,1,i} \cdot r^{-z^2_{\ell,i}} \\ &= \mu_{\ell,1} \cdot \left(\prod_{i=1}^{n} a_{\ell,1,i}\right) \cdot r^{\sum_{i=1}^{n} -z_i^2} = \mu_{\ell,1} \cdot a_{\ell,1} \cdot r^{-\|\boldsymbol{z}_\ell\|^2}.\end{aligned}$$

   Thus we have $\beta_{\ell,i,0} = \alpha_{\ell,i} \cdot \mu_{\ell,i}$ for $i \in [n']$, and P_0 has received the punctured key masked with $\boldsymbol{\mu}_\ell$.
6. In Step 6, P_1 opens the commitment to the masks $\mu_{\ell,i}$ such that P_0 can unmask $\mathsf{psk}_\ell = (N, \alpha_{\ell,i} = \beta_{\ell,i,0} \cdot \mu^{-1}_{\ell,i})_{i\in[n']}$.

Now we cycle back and show that the correction value is also computed correctly. We have $k\text{-NR-PPRF.Eval}(\mathsf{msk}, \mathsf{sk}_\ell, z_\ell) = \mathsf{H}(t \cdot H) = \mathsf{H}(t \cdot \rho \cdot G)$ where

$$t = a_0^{\|\boldsymbol{z}_\ell\|^2} \cdot a_1 \cdot \prod_{i=1}^{n} a_{1+i}^{z_i}.$$

7. In Step 3b, P_1 samples more random masks and prepares an additional k-tuples per $i \in [n]$. For $j \in [0,k)$ we have $\eta^{(j)}_{\ell,0,i} = \nu_{\ell,i} \cdot a_0^{j^2} \cdot a^j_{\ell,1+i}$, and P_1 also compute information-theoretic MACs on these values.
8. When the OTOpen is invoked in Step 3c P_0 receives (in addition to the values discussed above) $\eta^{(z_{\ell,i})}_{\ell,0,i} = \nu_{\ell,i} \cdot a_0^{z^2_{\ell,i}} \cdot a^{z_{\ell,i}}_{\ell,1+i}$, for $i \in [n]$ as wall as the corresponding MAC tag.
   Let $\nu_\ell := \prod_{i=1}^{n} \nu_{\ell,i}$. Observe that

$$\eta_{\ell,0} := \prod_{i=1}^{n} \eta^{(z_{\ell,i})}_{\ell,0,i} = \prod_{i=1}^{n} \left(\nu_{\ell,i} \cdot a_0^{z^2_{\ell,i}} \cdot a^{z_{\ell,i}}_{\ell,1+i}\right) = \nu_\ell \cdot a_0^{\|\boldsymbol{z}_\ell\|^2} \cdot \prod_{i=1}^{n} a^{z_{\ell,i}}_{\ell,1+i}.$$

   With $\eta_{\ell,1} := \rho \cdot a_0^{-1} \cdot \nu_\ell^{-1}$ we have $\eta_{\ell,0} \cdot \eta_{\ell,1} = \rho \cdot t$.

The remainder of the correctness argument is analogous to that of $\Pi_{\mathsf{DPF.CHHK25}}$.

Simulation. We give the simulator in Figure 25. The indistinguishability argument is analogous to the one of $\Pi_{\mathsf{DPF.kNR}}$. The only diference is that more values have to be handled at each step, since the sk and psk of k-NR-PPRF are tuples of n' field elements instead of single field elements as with CHHK25-PPRF.

Fig. 25: Simulator for Π_{DPF} – kNR-DPF Keygen Subprotocol

Corrupted P_0

1. Extract $(e_\ell)_{\ell\in[t]}$ from P_0's messages to $\mathcal{F}_{\mathsf{MPC}}^{p,q}.\mathsf{Input}$ in Step 1.
2. Simulate the oblivious puncturing, for $\ell \in [t]$:
   (a) Simulate the call to $\mathcal{F}_{\mathsf{ZK}}^{p,0}.\mathsf{OTOpen}$ in Step 3c for $i \in [n]$:
      i. Receive $z_{\ell,i} \in [0,k)$ from P_0.
      ii. Sample $\alpha_{\ell,1,i}, \alpha_{\ell,1+i}, \eta_{\ell,0,i}, M_{\eta_{\ell,0,i}} \xleftarrow{\$} \mathbb{F}_p$.
      iii. Send $(\alpha_{\ell,1,i}^{(z_{\ell,i})}, \alpha_{\ell,1+i}^{(z_{\ell,i})}, \eta_{\ell,0,i}^{(z_{\ell,i})}, M_{\eta_{\ell,0,i}}^{(z_{\ell,i})}) := (\alpha_{\ell,1,i}, \alpha_{\ell,1+i}, \eta_{\ell,0,i}, M_{\eta_{\ell,0,i}})$ to P_0.
   (b) Compute $z_\ell := \sum_{i=1}^{n} k^{i-1} \cdot z_{\ell,i}$.
   (c) Compute $\beta_{\ell,1,0} := \prod_{i=1}^{n} \alpha_{\ell,1,i}$ and $\beta_{\ell,1+i,0} := \alpha_{\ell,1+i}$.
3. Simulate Step 4 analogously to the simulation of $\Pi_{\mathsf{DPF.CHHK25}}$ in the case of corrupted P_0 (Steps 3 to 5).
4. Simulate Step 5 as in the simulation of $\Pi_{\mathsf{DPF.CHHK25}}$ in the case of corrupted P_0 (Steps 6 and 7).
5. Simulate the key reveal:
   (a) Send (z_ℓ, e_ℓ) to $\mathcal{F}_{\mathsf{DPF}}$ and receive $\mathsf{k}_{0,(\ell)}^{\mathsf{DPF}} = \left(\mathsf{psk}_\ell = (N, \boldsymbol{\alpha}_\ell, z_\ell), v_\ell\right) \in \mathbb{N} \times \mathbb{F}_p^{n'} \times [0,N) \times \mathbb{F}_p$ for $\ell \in [t]$.
   (b) Compute $\mu_{\ell,i} := \beta_{\ell,i,0} \cdot \alpha_{\ell,i}^{-1}$ for $\ell \in [t], i \in [n']$.
   (c) Simulate $\mathcal{F}_{\mathsf{ZK}}^{p,1}.\mathsf{Open}$ in Step 6 by sending $\mu_{\ell,i}$ to P_0 for $\ell \in [t], i \in [n']$.
   (d) Simulate $\mathcal{F}_{\mathsf{MPC}}^{p,q}.\mathsf{Output}$ in Step 7 by sending v_ℓ to P_0 for $\ell \in [t]$.
6. If $\mathcal{Z}$ has not send abort, send $\mathsf{continue}$ to $\mathcal{F}_{\mathsf{DPF}}$.

Corrupted P_1

1. Extract the $a_{\ell,1,i}, a_{\ell,1+i}$ for all $\ell \in [t], i \in [n]$ from P_1's messages to $\mathcal{F}_{\mathsf{ZK}}^{p,1}.\mathsf{Random}$ in Step 2.
2. Compute $\mathsf{sk}_\ell = \boldsymbol{a}_\ell = (a_{\ell,i})_{i\in[n']}$ for $a_{\ell,1} := \prod_{i=1}^{n} a_{\ell,1,i}$.
3. Continue simulating the protocol and verify with $\mathcal{F}_{\mathsf{ZK}}$ that the prepared commitments have the correct form.
4. Simulate the oblivious puncturing, for $\ell \in [t]$:
   (a) Simulate the call to $\mathcal{F}_{\mathsf{ZK}}^{p,1}.\mathsf{OTOpen}$ in Step 3c for $i \in [n]$:
      i. Extract $\alpha_{\ell,1,i}^{(j)}$ $\alpha_{\ell,1+i}^{(j)}$ $\eta_{\ell,0,i}^{(j)}$ from the commitments for $j \in [0,k)$.
      ii. Receive $M_{\eta_{\ell,0,i}}^{(j)} \in \mathbb{F}_p$ for $j \in [0,k)$ from P_1.
      iii. Receive selected-failure set $I_{\ell,i}^{(1)} \subseteq [0,k)$ from $\mathcal{Z}$.
   (b) $I_\ell^{(1)} := \left\{ z \in [0,k^n) \;\middle|\; \exists \boldsymbol{z} \in I_{\ell,1}^{(1)} \times \cdots \times I_{\ell,n}^{(1)} \,.\, z = \sum_{i=1}^{n} k^{i-1} \cdot z_i \right\}$.
   NB: if $z_\ell \notin I_\ell^{(1)}$, we need to fail later.
5. Simulate `AuthCommit` in Step 4a:
   (a) Receive $\chi \in \mathbb{F}_p$ from P_1.
   (b) Receive $\Gamma_\eta, (K_{\eta_{\ell,0,i}})_{\ell\in[t],i\in[n]} \in \mathbb{F}_p$ from P_1.
   (c) Define $I_{\ell,i}^{(2)} := \left\{ j \in [0,k) \;\middle|\; \Gamma_\eta \cdot \eta_{\ell,0,i}^{(b)} + K_{\eta_{\ell,0,i}} = M_{\eta_{\ell,0,i}}^{(j)} \right\}$, and $I_\ell^{(2)}$ analogous to $I_\ell^{(1)}$.
   (d) Send $I := \bigtimes_{\ell\in[t]} \left(I_\ell^{(1)} \cap I_\ell^{(2)}\right)$ to $\mathcal{F}_{\mathsf{DPF}}$.
   (e) If $\mathcal{F}_{\mathsf{DPF}}$ aborts, simulate failing $\mathcal{F}_{\mathsf{ZK}}^{p,0}.\mathsf{CheckZero}$; otherwise simulate $\mathsf{CheckZero}$ succeeding.
6. Send sk_ℓ to $\mathcal{F}_{\mathsf{DPF}}$ for $\ell \in [t]$.
7. Proceed as in simulation of $\Pi_{\mathsf{DPF.CHHK25}}$ for corr. P_1 (Steps 8–11).

11 Distributed PCF Key Generation

11.1 The Ideal Functionality $\mathcal{F}_{\mathsf{PCF}}$

The ideal PCF key generation functionality $\mathcal{F}_{\mathsf{PCF}}$ (Figure 26) takes $\Delta \in \mathbb{F}_{p^r}$ as input from P_1 and then runs the PCF.$\mathsf{Gen}(\Delta)$ algorithm. If P_b is corrupted, then the is ran interactively with the simulator $\mathcal{S}$ as PCF.$\mathsf{Gen}^{\mathcal{S},b}(\Delta)$ allowing adversarial influence.

Fig. 26: $\mathcal{F}_{\mathsf{PCF}}$

The functionality interacts with two parties $\mathsf{P}_0, \mathsf{P}_1$, and is parameterized by the security parameter λ and a VOLE-PCF PCF (Definition 20) over $\mathbb{F}_{p^r}$.
Let $\sigma = \bot$ if no party is corrupted, and otherwise let $\sigma \in \{0,1\}$ denote the index of the corrupted party.

Gen On input (PCFGen) from P_0 and (Gen, Δ) from P_1:
1. Run $k \leftarrow$ PCF.$\mathsf{Gen}^{\mathcal{S},\sigma}(1^\lambda, \Delta)$.
2. If $k = \bot$, send (PCFAbort) to both parties. Otherwise parse $(\mathsf{k}_0^{\mathsf{PCF}}, \mathsf{k}_1^{\mathsf{PCF}}) := k$
3. If $\sigma \neq \bot$, send $\mathsf{k}_\sigma^{\mathsf{PCF}}$ to $\mathcal{S}$. If $\mathcal{S}$ responds with (continue) or $\sigma = \bot$, send (PCFKey, $\mathsf{k}_b^{\mathsf{PCF}}$) to P_b for $b \in \{0,1\}$; otherwise send (PCFAbort) to both parties.

11.2 The Protocol Π_{PCF}

Our protocol is straight-forward: The parties start by invoking $\mathcal{F}_{\mathsf{VOLE}}$ to obtain the seed $[\![\mathbf{s}^{(0)}]\!]$. Then $\mathcal{F}_{\mathsf{DPF}}$ is first used to obtain the reusable setup for the DPF scheme, and then to generate $t^{(\ell)}$ many DPF keys for each level $\ell \in [L]$.

Fig. 27: Π_{PCF} – PCF Keygen Protocol

Let PCF = (Gen, Eval) be the VOLE-PCF from [BCMRS25] as in Figure 11 over $\mathbb{F}_{p^r}$ with parameters and $L, t^{(\ell)}, m^{(\ell)}$ as in Figure 11. The protocol is given in the $(\mathcal{F}_{\mathsf{VOLE}}^{p,r}, \mathcal{F}_{\mathsf{DPF}})$-hybrid model, where $\mathcal{F}_{\mathsf{DPF}}$ is parametrized with a DPF with reusable setup (Definition 17) that supports domain sizes $\{m^{(\ell)}/t^{(\ell)}\}_{\ell\in[L]}$ and has range $\mathbb{F}_{p^r}$

Gen On input (PCFGen) from P_0 and (Gen, Δ) from P_1:
1. P_0 sends (Init), and P_1 sends (Init, Δ) to $\mathcal{F}_{\mathsf{VOLE}}$.
2. Send (Sample, $m^{(0)}$) to $\mathcal{F}_{\mathsf{VOLE}}$ and receive $[\![\mathbf{s}]\!]$ with $\mathbf{s} \in \mathbb{F}_p^{m^{(0)}}$.
3. Authenticate Δ:
   (a) Send (Sample, r) to $\mathcal{F}_{\mathsf{VOLE}}$ and receive $[\![\boldsymbol{\Gamma}]\!]$ for $\boldsymbol{\Gamma} \in \mathbb{F}_p^r$.
   (b) Let $(1, X, \ldots, X^{r-1})$ be a basis of $\mathbb{F}_{p^r}$, and compute $[\![\Gamma_\Delta]\!] \leftarrow \sum_{i=1}^r X^{i-1} \cdot [\![\Gamma_i]\!]$ such that $\Gamma_\Delta \in \mathbb{F}_{p^r}$.
   (c) P_0 parses $[\![\Gamma_\Delta]\!]_0 = (\Gamma_\Delta, K_\Delta)$. and P_1 parses $[\![\Gamma_\Delta]\!]_1 = (\Delta, M_\Delta)$. // $M_\Delta = \Gamma_\Delta \cdot \Delta + K_\Delta$
4. P_0 sends (Setup, Γ_Δ, K_Δ) and P_1 sends (Setup, Δ, M_Δ) to $\mathcal{F}_{\mathsf{DPF}}$. If $\mathcal{F}_{\mathsf{DPF}}$ does not abort, P_b receives $\mathsf{st}_b^{\mathsf{DPF}}$ for $b \in \{0,1\}$.
5. For $\ell \in [L]$:
   (a) P_0 samples $\mathbf{e}^{(\ell)} = \left((\alpha_i^{(\ell)}, \beta_i^{(\ell)})\right)_{i\in[t^{(\ell)}]} \leftarrow \mathsf{Reg}_{t^{(\ell)}}(m^{(\ell)}, \mathbb{F}_p)$.
   (b) P_0 sends (Keygen, $m^{(\ell)}/t^{(\ell)}, t^{(\ell)}, \boldsymbol{\alpha}^{(\ell)}, \boldsymbol{\beta}^{(\ell)}$) and P_1 sends (Keygen, $m^{(\ell)}/t^{(\ell)}, t^{(\ell)}$) to $\mathcal{F}_{\mathsf{DPF}}$.
   (c) If $\mathcal{F}_{\mathsf{DPF}}$ aborts, then abort. Otherwise, P_b receives $(\mathsf{k}_{b,(\ell,i)}^{\mathsf{DPF}})_{i\in[t^{(\ell)}]}$ for $b \in \{0,1\}$.
6. P_b outputs $\mathsf{k}_b^{\mathsf{PCF}}$ for $b \in \{0,1\}$ and

$$\mathsf{k}_0^{\mathsf{PCF}} := \left([\![\mathbf{s}]\!]_0, \mathsf{st}_0^{\mathsf{DPF}}, \left(\mathbf{e}^{(\ell)}, (\mathsf{k}_{0,(\ell,i)}^{\mathsf{DPF}})_{i\in[t^{(\ell)}]}\right)_{\ell\in[L]}\right), \qquad \mathsf{k}_1^{\mathsf{PCF}} := \left([\![\mathbf{s}]\!]_1, \mathsf{st}_1^{\mathsf{DPF}}, \left((\mathsf{k}_{1,(\ell,i)}^{\mathsf{DPF}})_{i\in[t^{(\ell)}]}\right)_{\ell\in[L]}\right).$$

11.3 Security Proof

Theorem 14. *Let $\mathbb{F}_{p^r}$ be a finite field of size $|\mathbb{F}_{p^r}| = \Omega(2^\rho)$. Then the protocol Π_{PCF} (Figure 27) realizes with statistical security the functionality $\mathcal{F}_{\mathsf{PCF}}$ (Figure 26) in the $\mathcal{F}_{\mathsf{DPF}}$-hybrid model, where $\mathcal{F}_{\mathsf{PCF}}$ is parameterized by* PCF = Recursive-LPN-PCF *with* interface $= \{\lambda, \mathcal{X}, \mathbb{F}_{p^r}\}$ *and* spec $= \{\mathsf{DPF}, \dots\}$ *and $\mathcal{F}_{\mathsf{DPF}}$ is parameterized by* DPF.

Proof. We give the simulator in Figure 28.

Fig. 28: Simulator for Π_{PCF}

Gen – Corrupted P_0

1. Receive $[\![\mathbf{s}^{(0)}]\!]_0$ from P_0 and send $[\![\mathbf{s}^{(0)}]\!]_0$ to $\mathcal{F}_{\mathsf{PCF}}$.
2. Receive $[\![\boldsymbol{\Gamma}]\!]_0$ from P_0 and compute $(\Gamma_\Delta, K_\Delta) = \sum_{i=1}^{r} X^{i-1} \cdot [\![\Gamma_i]\!]_0$ as in the protocol.
3. Simulate $\mathcal{F}_{\mathsf{DPF}}$.Setup:
   (a) Receive $(\mathsf{Setup}, \Gamma'_\Delta, K'_\Delta)$ from P_0. If $\Gamma'_\Delta \neq \Gamma_\Delta$ or $K'_\Delta \neq K_\Delta$, simulate $\mathcal{F}_{\mathsf{DPF}}$ aborting.
   (b) Execute $\mathsf{DPF.Setup}^{\mathcal{S},0}$ with $\mathcal{F}_{\mathsf{PCF}}$ and forward the adversary's influence to the environment $\mathcal{Z}$.
   (c) Receive $\mathsf{st}_0^{\mathsf{DPF}}$ from $\mathcal{F}_{\mathsf{PCF}}$ and send it to P_0.
4. Simulate each invocation of $\mathcal{F}_{\mathsf{DPF}}$.Gen for $\ell \in [L]$:
   (a) Receive $(\mathsf{Keygen}, m^{(\ell)}/t^{(\ell)}, t^{(\ell)}, \boldsymbol{\alpha}^{(\ell)}, \boldsymbol{\beta}^{(\ell)})$ from P_0.
   (b) Send $\mathbf{e}^{(\ell)} = ((\alpha_i^{(\ell)}, \beta_i^{(\ell)}))_{i \in [t^{(\ell)}]}$ to $\mathcal{F}_{\mathsf{PCF}}$.
   (c) For $i \in [t^{(\ell)}]$:
      i. Execute $\mathsf{DPF.Gen}^{\mathcal{S},0}$ with $\mathcal{F}_{\mathsf{PCF}}$ and forward the adversary's influence to the environment $\mathcal{Z}$.
      ii. If $\mathcal{F}_{\mathsf{PCF}}$ aborts, simulate an abort. Otherwise receive $\mathsf{k}^{\mathsf{DPF}}_{0,(\ell,i)}$ from $\mathcal{F}_{\mathsf{PCF}}$.
   (d) If no abort happened, send $(\mathsf{k}^{\mathsf{DPF}}_{0,(\ell,i)})_{i \in [t^{(\ell)}]}$ as output of $\mathcal{F}_{\mathsf{DPF}}$ to P_0.
   (e) If $\mathcal{F}_{\mathsf{PCF}}$ aborts, simulate an abort. Otherwise receive $\mathsf{k}^{\mathsf{DPF}}_{0,(\ell,i)}$ from $\mathcal{F}_{\mathsf{PCF}}$ and send it as output of $\mathcal{F}_{\mathsf{DPF}}$ to P_0.
5. If $\mathcal{Z}$ has not send abort, send continue to $\mathcal{F}_{\mathsf{DPF}}$.

Gen – Corrupted P_1

1. Receive $[\![\mathbf{s}^{(0)}]\!]_1$ from P_1 and send $[\![\mathbf{s}^{(0)}]\!]_1$ to $\mathcal{F}_{\mathsf{PCF}}$.
2. Receive $[\![\boldsymbol{\Gamma}]\!]_1$ from P_1 and compute $(\Delta, M_\Delta) = \sum_{i=1}^{r} X^{i-1} \cdot [\![\Gamma_i]\!]_1$ as in the protocol.
3. Simulate $\mathcal{F}_{\mathsf{DPF}}$.Setup:
   (a) Receive $(\mathsf{Setup}, \Delta', M'_\Delta)$ from P_1. If $\Delta' \neq \Delta$ or $M'_\Delta \neq M_\Delta$, simulate $\mathcal{F}_{\mathsf{DPF}}$ aborting.
   (b) Execute $\mathsf{DPF.Setup}^{\mathcal{S},1}$ with $\mathcal{F}_{\mathsf{PCF}}$ and forward the adversary's influence to the environment $\mathcal{Z}$.
   (c) Receive $\mathsf{st}_1^{\mathsf{DPF}}$ from $\mathcal{F}_{\mathsf{PCF}}$ and send it to P_1.
4. Simulate each invocation of $\mathcal{F}_{\mathsf{DPF}}$.Gen for $\ell \in [L]$:
   (a) Receive $(\mathsf{Keygen}, m^{(\ell)}/t^{(\ell)}, t^{(\ell)})$ from P_1.
   (b) Receive selected-failure set I from P_1 and send it to $\mathcal{F}_{\mathsf{PCF}}$.
   (c) If $\mathcal{F}_{\mathsf{PCF}}$ aborts, simulate $\mathcal{F}_{\mathsf{DPF}}$ aborting.
   (d) For $i \in [t^{(\ell)}]$:
      i. Execute $\mathsf{DPF.Gen}^{\mathcal{S},1}$ with $\mathcal{F}_{\mathsf{PCF}}$ and forward the adversary's influence to the environment $\mathcal{Z}$.
      ii. If $\mathcal{F}_{\mathsf{PCF}}$ aborts, simulate an abort. Otherwise receive $\mathsf{k}^{\mathsf{DPF}}_{1,(\ell,i)}$ from $\mathcal{F}_{\mathsf{PCF}}$.
   (e) If no abort happened, send $(\mathsf{k}^{\mathsf{DPF}}_{1,(\ell,i)})_{i \in [t^{(\ell)}]}$ as output of $\mathcal{F}_{\mathsf{DPF}}$ to P_1.
5. If $\mathcal{Z}$ has not send abort, send continue to $\mathcal{F}_{\mathsf{DPF}}$.

In the simulation, the ideal functionality $\mathcal{F}_{\mathsf{PCF}}$ runs $\mathsf{PCF.Gen}^{\mathcal{S},b}(\Delta)$ interactively with the simulator $\mathcal{S}$ when P_b is corrupted. $\mathcal{S}$ simulates instances of $\mathcal{F}_{\mathsf{VOLE}}$ and $\mathcal{F}_{\mathsf{DPF}}$. It uses the influence and leakage is has during the execution of PCF.Gen to simulate the environment $\mathcal{Z}$'s influence and leakage during the interactions with $\mathcal{F}_{\mathsf{VOLE}}$ and $\mathcal{F}_{\mathsf{DPF}}$. It receives $[\![\mathbf{s}^{(0)}]\!]_b$ from P_b and forwards it to PCF.Gen. Similarly, it simulates P_0's influence on the noise vectors $\mathbf{e}^{(\ell)}$ and P_1's selected-failure query on $\mathbf{e}^{(\ell)}$.

For the execution of $\mathsf{DPF.Setup}^{\mathcal{S},b}$ and $\mathsf{DPF.Gen}^{\mathcal{S},b}$ the influence is forwarded such that $\mathcal{F}_{\mathsf{PCF}}$ actually runs $\mathsf{DPF.Setup}^{\mathcal{Z},b}$ and $\mathsf{DPF.Gen}^{\mathcal{Z},b}$ while $\mathcal{Z}$ thinks it is interacting the $\mathcal{F}_{\mathsf{DPF}}$.

Since the parties only interact with the ideal functionalities $\mathcal{F}_{\mathsf{VOLE}}$ and $\mathcal{F}_{\mathsf{DPF}}$, and do not exchange messages otherwise in the protocol, and the environments influence in the functionalities is perfectly simulated by $\mathcal{S}$ forwarding the influence to $\mathcal{F}_{\mathsf{PCF}}$, the views are identical.

There is one exception, though: To ensure the same Δ is used for both $\mathcal{F}_{\mathsf{VOLE}}$ and $\mathcal{F}_{\mathsf{DPF}}$, the parties use r additional samples to compute an information-theoretic MAC on Δ. If P_1 tries to use a different Δ', the MAC check by $\mathcal{F}_{\mathsf{DPF}}$ fails with probability $1/p^r \leq 2^{-\rho}$ in the protocol, while it always fails in the simulation. Thus the environment's view in the simulation is statistically close to its view in the real protocol. □

12 Two-Party Signing

12.1 The $\mathcal{F}_{\text{ro-VOLE}}$-Hybrid Model

The Ideal Functionality. The functionality $\mathcal{F}_{\text{ro-VOLE}}$ is presented in Figure 29. In spirit, our functionality can be considered a one-time corruptible VOLE functionality of (potentially super-polynomial) dimension $|\mathcal{X}|$. The rows, indexed by $x \in \mathcal{X}$, are accessible via the interface $(\mathsf{Sample},\ x)$. At initialization the ideal adversary $\mathcal{S}$ commits to all of its shares for all possible $x \in \mathcal{X}$ simultaneously by submitting a deterministic function f that the functionality uses to non-interactively compute the shares of the corrupted party. Most importantly, the (Sample) interface of $\mathcal{F}_{\text{ro-VOLE}}$ is non-interactive; an adversary can neither prevent the delivery of output nor adaptively decide the corrupted party shares once the initialization phase has completed. This directly reflects our PCF-based instantiation, in which $\mathcal{A}$ similarly cannot prevent sampling. The protocol $\Pi_{\text{ro-VOLE}}$ is presented in Figure 30, and is proven to UC-realize $\mathcal{F}_{\text{ro-VOLE}}$ in the $\mathcal{F}_{\mathsf{PCF}}$-hybrid model in Theorem 15.

Fig. 29: $\mathcal{F}_{\text{ro-VOLE}}(p, r, \mathcal{X})$, ro-VOLE Functionality

The $\mathcal{F}_{\text{ro-VOLE}}$ functionality is parameterized by a prime $p \in \mathbb{P}$, an integer $r \in \mathbb{N}$, and a set $\mathcal{X}$. The functionality maintains an initially empty dictionary $\mathsf{D} : \mathcal{X} \times \{\mathsf{S}, \mathsf{R}\} \to (\mathbb{F}_p \times \mathbb{F}_{p^r}) \times \mathbb{F}_{p^r}$, and interacts with the sender P_S, the receiver P_R and the ideal adversary $\mathcal{S}$ via the queries below, all of which implicitly include a session identifier sid. The functionality interacts with $\mathcal{S}$ on behalf of any corrupted party, and $\mathcal{S}$ can instruct the functionality to abort during initialization prior to output via a query ($\mathsf{VOLEabort}$), which is then forwarded as output to the parties.

Initialization. The init interface can be queried by each party a single time.

On ($\mathsf{VOLEInit}$) **from** $\mathsf{P}_\sigma \in \{\mathsf{P}_\mathsf{S}, \mathsf{P}_\mathsf{R}\}$**.** Send ($\mathsf{VOLEInit},\ \mathsf{P}_\sigma$) to $\mathcal{S}$.

On ($\mathsf{VOLEInit},\ \mathsf{inf}$) **from** $\mathcal{S}$**.** Ignore the query if ($\mathsf{VOLEInit}$) not queried by all honest $\mathsf{P}_\sigma \in \{\mathsf{P}_\mathsf{S}, \mathsf{P}_\mathsf{R}\}$.

1. If P_S is corrupted: parse $f := \mathsf{inf}$, where $f : \mathcal{X} \to \mathbb{F}_p \times \mathbb{F}_{p^r}$ is the description of a deterministic polynomial-time computable function.
2. If P_R is corrupted: parse $(\Delta, f) := \mathsf{inf}$, where $f : \mathcal{X} \to \mathbb{F}_{p^r}$ is the description of a deterministic polynomial-time computable function and $\Delta \in \mathbb{F}_{p^r}$.
3. Otherwise sample $\Delta \xleftarrow{\$} \mathbb{F}_{p^r}$.
4. Output ($\mathsf{Initialized}$) to P_S if P_S is honest, and ($\mathsf{Initialized},\ \Delta$) to P_R if P_R is honest.

Sample. The sample interface opens after ($\mathsf{Init},\ \cdot$) is called by both parties.

On ($\mathsf{Sample},\ x \in \mathcal{X}$) **from** $\mathsf{P}_\sigma \in \{\mathsf{P}_\mathsf{S}, \mathsf{P}_\mathsf{R}\}$**.**

1. If $x \notin \mathsf{D}$:
   (a) If P_S is corrupted:
   Evaluate $(y, \langle y \rangle_\mathsf{S}) := f(x)$. Define $\langle y \rangle_\mathsf{R} := \Delta \cdot y + \langle y \rangle_\mathsf{S}$.
   (b) Else if P_R is corrupted:
   Evaluate $\langle y \rangle_\mathsf{R} := f(x)$. Sample $y \xleftarrow{\$} \mathbb{F}_p$. Define $\langle y \rangle_\mathsf{S} := \Delta \cdot y - \langle y \rangle_\mathsf{R}$.
   (c) Else (both parties are honest):
   Sample $y \xleftarrow{\$} \mathbb{F}_p, \langle y \rangle_\mathsf{S} \xleftarrow{\$} \mathbb{F}_{p^r}$ and define $\langle y \rangle_\mathsf{R} := \Delta \cdot y - \langle y \rangle_\mathsf{S}$.
   (d) Set $\mathsf{D}[x][\mathsf{S}] := (y, \langle y \rangle_\mathsf{S})$, and $\mathsf{D}[x][\mathsf{R}] := \langle y \rangle_\mathsf{R}$.
2. Output $\mathsf{D}[x][\sigma]$ to P_σ.

Our PCF-Based Protocol. The protocol $\Pi_{\text{ro-VOLE}}$ presented in Figure 30 is fairly intuitive; each party $\mathsf{P}_\sigma \in \{\mathsf{P}_\mathsf{S}, \mathsf{P}_\mathsf{R}\}$ initializes $\mathcal{F}_{\mathsf{PCF}}$, P_R uniformly sampling its own $\Delta \xleftarrow{\$} \mathbb{F}_{p^r}$, and awaits the PCF key as output ($\mathsf{PCFKey},\ \mathsf{k}_\sigma^{\mathsf{PCF}}$). Subsequently, on any input ($\mathsf{Sample},\ x$) the party $\mathsf{P}_\sigma \in \{\mathsf{P}_\mathsf{S}, \mathsf{P}_\mathsf{R}\}$ simply outputs $\mathsf{Eval}(\sigma, \mathsf{k}_\sigma^{\mathsf{PCF}}, x)$.

Fig. 30: $\Pi_{\text{ro-VOLE}}(p, r, \mathcal{X})$, ro-VOLE Protocol

The protocol with session identifier sid executed between sender $\mathsf{P_S}$ and receiver $\mathsf{P_R}$ is parameterized by a prime p, $r \in \mathbb{N}$, and a set $\mathcal{X}$.

The protocol is written in the $\mathcal{F}_{\text{PCF}}$-hybrid model:

$\mathcal{F}_{\text{PCF}} := \mathcal{F}_{\text{PCF}}(\mathsf{pp}_{\text{PCF}})$ corresponds to $(\mathsf{Gen}^{\mathcal{A},\sigma}, \mathsf{Eval})$, a secure PCF for VOLE under adversarial influence with parameters $\mathsf{pp}_{\text{PCF}} = (p, r, \mathcal{X})$.

On $(\mathsf{VOLEInit})$ **Party** $\mathsf{P}_\sigma = \mathsf{P_R}$**:**
1. Sample $\Delta \xleftarrow{\$} \mathbb{F}_{p^r}$.
2. Query $(\mathsf{PCFGen},\ \Delta)$ to $\mathcal{F}_{\text{PCF}}$.
3. On $(\mathsf{PCFabort})$ from $\mathcal{F}_{\text{PCF}}$: output $(\mathsf{VOLEabort})$.
4. Await $(\mathsf{PCFKey},\ \mathsf{k}_\sigma^{\text{PCF}})$ from $\mathcal{F}_{\text{PCF}}$.
5. Output $(\mathsf{Initialized},\ \Delta)$.

On $(\mathsf{VOLEInit})$ **Party** $\mathsf{P}_\sigma = \mathsf{P_S}$**:**
1. Query (PCFGen) to $\mathcal{F}_{\text{PCF}}$.
2. On $(\mathsf{PCFabort})$ from $\mathcal{F}_{\text{PCF}}$: output $(\mathsf{VOLEabort})$.
3. Await $(\mathsf{PCFKey},\ \mathsf{k}_\sigma^{\text{PCF}})$ from $\mathcal{F}_{\text{PCF}}$.
4. Output $(\mathsf{Initialized})$.

On $(\mathsf{Sample},\ x \in \mathcal{X})$ **Party** $\mathsf{P}_\sigma \in \{\mathsf{P_S}, \mathsf{P_R}\}$**:** Output $\mathsf{Eval}(\sigma, \mathsf{k}_\sigma^{\text{PCF}}, x)$.

Theorem 15 ($\Pi_{\text{ro-VOLE}}$ Security). *For all $p, r \in \mathbb{N}$ with p prime, and for all $\mathcal{X}$ with an efficient description the protocol $\Pi_{\text{ro-VOLE}}(p, r, \mathcal{X})$ (Figure 30) UC-realizes the functionality $\mathcal{F}_{\text{ro-VOLE}}(p, r, \mathcal{X})$ (Figure 29) in the $\mathcal{F}_{\text{PCF}}$-hybrid model in the presence of a malicious adversary statically corrupting at most one party, assuming that $\mathsf{PCF} = (\mathsf{Gen}^{\mathcal{A},\sigma}, \mathsf{Eval})$ of $\mathcal{F}_{\text{PCF}}$ is a secure PCF for VOLE under adversarial influence (Definition 20).*

Fig. 31: $\mathcal{S}_{\text{ro-VOLE}}(p, r, \mathcal{X})$ – Simulator for $\Pi_{\text{ro-VOLE}}(p, r, \mathcal{X})$.

The simulator $\mathcal{S}_{\text{ro-VOLE}}$ interacts with functionality $\mathcal{F}_{\text{ro-VOLE}}(p, r, \mathcal{X})$ (Figure 29) and simulates execution of the protocol $\Pi_{\text{ro-VOLE}}$ (Figure 30) with session identifier sid executed by $\mathsf{P_S}, \mathsf{P_R}$ in the presence of adversary $\mathcal{A}$.

Hybrid Functionalities. The simulator emulates $\mathcal{F}_{\text{PCF}}(p, r, \mathcal{X})$ (Figure 26) for $\mathsf{P_S} = \mathsf{P}_0$ and $\mathsf{P_R} = \mathsf{P}_1$.

Protocol.

On $(\mathsf{VOLEInit},\ \mathsf{P}_\sigma)$ **from** $\mathcal{F}_{\text{ro-VOLE}}$**.**
Record $(\mathsf{VOLEInit},\ \mathsf{P}_\sigma)$. Send $(\mathsf{PCFGen},\ \mathsf{P}_\sigma)$ to $\mathcal{A}$ from $\mathcal{F}_{\text{PCF}}$.

On $(\mathsf{PCFabort})$ **from** $\mathcal{A}$ **to** $\mathcal{F}_{\text{PCF}}$**.**
Query $(\mathsf{VOLEabort})$ to $\mathcal{F}_{\text{ro-VOLE}}$.

On $(\mathsf{PCFGen},\ \mathsf{infl})$ **from** $\mathcal{A}$ **to** $\mathcal{F}_{\text{PCF}}$**.** Ignore unless $(\mathsf{PCFInit})$ was queried by all honest parties.
1. If both $\mathsf{P_S}, \mathsf{P_R}$ are honest: define $\mathsf{infl}^* := \perp$.
2. Else if $\mathsf{P}_0 = \mathsf{P_S}$ is corrupted:
   (a) Sample $\Delta \xleftarrow{\$} \mathbb{F}_{p^r}$.
   (b) As $\mathcal{F}_{\text{PCF}}$: execute $(\mathsf{k}_0^{\text{PCF}}, \mathsf{k}_1^{\text{PCF}}) \leftarrow \mathsf{Gen}^{\mathcal{A},0}(1^\lambda, \Delta)$ with $\mathcal{A}$.
   (c) Define $\mathsf{infl}^* := f(\cdot)$ where $f(\cdot) := \mathsf{Eval}(0, \mathsf{k}_0^{\text{PCF}}, \cdot)$.
3. Else ($\mathsf{P}_1 = \mathsf{P_R}$ is corrupted):
   (a) Parse $\Delta := \mathsf{infl}$.
   (b) As $\mathcal{F}_{\text{PCF}}$: execute $(\mathsf{k}_0^{\text{PCF}}, \mathsf{k}_1^{\text{PCF}}) \leftarrow \mathsf{Gen}^{\mathcal{A},1}(1^\lambda, \Delta)$ with $\mathcal{A}$.
   (c) Define $\mathsf{infl}^* := (\Delta, f(\cdot))$ where $f(\cdot) := \mathsf{Eval}(1, \mathsf{k}_1^{\text{PCF}}, \cdot)$.
4. Query $(\mathsf{VOLEInit},\ \mathsf{infl}^*)$ to $\mathcal{F}_{\text{ro-VOLE}}$.

Proof. The simulator $\mathcal{S}_{\texttt{ro-VOLE}}$ appears in Figure 31.

Hybrid H_0: *Real World.* This is the real execution of the protocol $\Pi_{\mathsf{ro\text{-}VOLE}}$ in which the environment $\mathcal{Z}$ runs the protocol between the parties $\mathsf{P_S}, \mathsf{P_R}$ and the (dummy) real-world adversary $\mathcal{A}$, all with access to ideal functionality $\mathcal{F}_{\mathsf{PCF}}$.

Hybrid H_1: *Ideal Layout.* The existing ITIs ($\mathsf{P_S}$, $\mathsf{P_R}$, $\mathcal{F}_{\mathsf{PCF}}$, $\mathcal{A}$) are grouped into a single node called the simulator $\mathcal{S}$ that forwards communication between $\mathcal{A}$ and $\mathcal{Z}$ without modification. The environment $\mathcal{Z}$ now interacts with new dummy party nodes for $\mathsf{P_S}$ and $\mathsf{P_R}$ that simply forward input/output to a new dummy functionality node $\mathcal{F}$ that again similarly forwards input/output to $\mathcal{S}$. This change is syntactic so the view of $\mathcal{Z}$ is unchanged.

Hybrid H_2: *Functionality Record Keeping & Corruptibility.* The simulator $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{PCF}}$ without modification, but generates a query $(\mathsf{VOLEInit},\ \mathsf{infl}^*)$ to $\mathcal{F}_{\mathsf{ro\text{-}VOLE}}$ that includes an influence string infl^* computed from the influence of $\mathcal{A}$ over the PCF submitted via the string infl and the emulated interaction between $\mathcal{A}$ and $\mathcal{F}_{\mathsf{PCF}}$.

- If $\mathsf{P_S} = \mathsf{P}_0$ is corrupted $\mathcal{S}$ follows Step 2 of Figure 31:
  $\mathcal{S}$ executes $(\mathsf{k}_0^{\mathsf{PCF}}, \mathsf{k}_1^{\mathsf{PCF}}) \leftarrow \mathsf{Gen}^{\mathcal{A},0}(1^\lambda, \Delta)$ for a random $\Delta \xleftarrow{\$} \mathbb{F}_{p^r}$. The simulator defines $\mathsf{infl}^* := f(\cdot)$ for $f(\cdot) = \mathsf{Eval}(0, \mathsf{k}_0^{\mathsf{PCF}}, \cdot)$ the PCF evaluation for P_0 with the extracted key.
- If $\mathsf{P_R} = \mathsf{P}_1$ is corrupted $\mathcal{S}$ follows Step 3 of Figure 31:
  $\mathcal{S}$ receives $\Delta^* = \mathsf{infl}$ from $\mathcal{A}$, executes $(\mathsf{k}_0^{\mathsf{PCF}}, \mathsf{k}_1^{\mathsf{PCF}}) \leftarrow \mathsf{Gen}^{\mathcal{A},1}(1^\lambda, \Delta^*)$, and defines $\mathsf{infl}^* := (\Delta^*, f(\cdot))$ where the function $f(\cdot) = \mathsf{Eval}(1, \mathsf{k}_1^{\mathsf{PCF}}, \cdot)$ is simply PCF evaluation for P_1 with the extracted key.
- If $(\mathsf{PCFabort})$ is received from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{PCF}}$ prior to output: send $(\mathsf{VOLEabort})$ to $\mathcal{F}$, which is forwarded as output to the parties.

In this hybrid, the output of interface $(\mathsf{VOLEInit})$ and $(\mathsf{Sample},\ \cdot)$ are still forwarded from $\mathcal{S}$ to any honest party. Specifically, as in H_1, when $\mathsf{P_R}$ is honest $\mathcal{S}$ still simulates the honest party as per the protocol and the randomly sampled Δ input to the emulated $\mathcal{F}_{\mathsf{PCF}}$ is sent to $\mathcal{F}$ and forwarded to the honest dummy $\mathsf{P_R}$. Also as in H_1 $\mathcal{S}$ chooses the output of any honest party P_σ on a query $(\mathsf{Sample},\ x)$ to $\mathcal{F}$, which is still forwarded to $\mathcal{S}$, to be $\mathsf{Eval}(\mathsf{k}_\sigma^{\mathsf{PCF}}, \sigma, x)$. Therefore, H_2 is perfectly indistinguishable from H_1.

Hybrid H_3: *Ideal Correlated Outputs for Honest* $\mathsf{P_S}$ *and Honest* $\mathsf{P_R}$. The only change in this hybrid is that the outputs of the (Sample) interface of $\mathcal{F}$ are sampled according to $\mathsf{VOLE}(\Delta)$ when both $\mathsf{P_S}$ and $\mathsf{P_R}$ are honest. In this hybrid, $\mathcal{S}$ still samples Δ as in Step 2a of Figure 31 and sends this as the output of $\mathcal{F}$ to an honest $\mathsf{P_R}$. The emulated $\mathcal{F}_{\mathsf{PCF}}$ does not produce output when both parties are honest, so no keys are computed. The simulator $\mathcal{S}$ samples the output of $(\mathsf{Sample},\ x) \xleftarrow{\$} \mathsf{VOLE}(\Delta)$ to be returned by $\mathcal{F}$ on each subsequent query $(\mathsf{Sample},\ x)$ for the same $x \in \mathcal{X}$.

H_3 is indistinguishable by the Strong VOLE-Correlated Pseudorandomness of PCF. The reduction is straightforward; an adversary $\mathcal{Z}$ that can distinguish between H_3 and H_2 with non-negligible probability implies an adversary $\mathcal{B}$ that wins the Strong VOLE-Correlated Pseudorandomness game $\mathsf{Exp}^{\mathsf{pr}}_{\mathcal{A},b}(1^\lambda, \Delta)$ in Figure 10.

Assume that there exists an $\mathcal{A}$ distinguishes H_3 from H_2 with non-negligible probability, i.e., that $\mathcal{A}$ outputs $b^* = 0$ after interacting with $(\mathcal{S}, \mathcal{F})$ in H_3 with non-negligibly greater probability than after interacting with $(\mathcal{S}, \mathcal{F})$ in H_2. Our Strong Correlated Pseudorandomness adversary $\mathcal{B}(1^\lambda)$ is then constructed as follows:

1. Query $((\cdot,\cdot),(\Delta,\cdot)) \leftarrow \mathcal{O}_b^{\mathsf{pr}}(x)$ for $x \xleftarrow{\$} \mathcal{X}$.
2. Simulate initialization as in H_2 and H_3, but send $(\mathsf{Initialized},\ \Delta)$ as the output of $\mathcal{F}$ to $\mathsf{P_R}$ instead of sampling the value as in Step 2a.
3. On $(\mathsf{Sample},\ x)$ from P_σ to $\mathcal{F}$: query $(\llbracket y \rrbracket_0, \llbracket y \rrbracket_1) \leftarrow \mathcal{O}_b^{\mathsf{pr}}(x)$ and output $\llbracket y \rrbracket$ to P_σ through $\mathcal{F}$.
4. When $\mathcal{A}$ outputs $b^* \in \{0,1\}$: Output $b' = b^*$.

When $b = 0$ the output of $\mathcal{O}_0^{\mathsf{pr}}(x) \xleftarrow{\$} \mathsf{VOLE}(\Delta)$ is a uniform VOLE correlation with fixed Δ, corresponding to H_3. When $b = 1$ the output of the oracle is PCF evaluation $\mathcal{O}_1^{\mathsf{pr}}(x) \in \mathsf{VOLE}(\Delta)$, corresponding to H_2. Therefore, the advantage of $\mathcal{B}$ is exactly the advantage of $\mathcal{A}$; $\mathcal{B}$ violates the Strong VOLE-Correlated Pseudorandomness of PCF assuming that $\mathcal{A}$ can distinguish between H_2 and H_3 with non-negligible probability.

Hybrid H_4: $\mathcal{F}$ *Samples* Δ *for Honest* $\mathsf{P_S}$ *and Honest* $\mathsf{P_R}$. In this hybrid, $\mathcal{F}$ uniformly samples $\Delta \xleftarrow{\$} \mathbb{F}_{p^r}$ as the output of $(\mathsf{Initialized},\ \Delta)$ to an honest $\mathsf{P_R}$ in the case that $\mathsf{P_S}$ is also honest. In H_3 $\mathcal{S}$ sampled the

output $\Delta \xleftarrow{\$} \mathbb{F}_{p^r}$ according to the same distribution, and therefore the output distribution of $\mathcal{F}$ in H_4 is identical to that in H_3. This change is perfectly indistinguishable. As of this hybrid, $\mathcal{F}$ and $\mathcal{S}$ are both fully implemented in the case that both parties are honest.

Hybrid H_5: *Ideal Correlated Outputs for a Single Corrupted Party.* The changes in this hybrid are to the outputs of (Sample) in the case that either $\mathsf{P_S}$ or $\mathsf{P_R}$ is corrupted, and to the output of (VOLEInit) in the case that $\mathsf{P_S}$ is corrupted (implying that $\mathsf{P_R}$ is honest). In this hybrid $\mathcal{S}$ still generates adversarial influence as in H_2, the only changes are to the output of $\mathcal{F}$.

When $\mathsf{P_R}$ is corrupted:

- On a query (Sample, x) from honest $\mathsf{P_S}$: $\mathcal{S}$ samples the output as $[\![y]\!]_0 \xleftarrow{\$} \mathsf{VOLE}(\Delta^*, \mathsf{Eval}(1, \mathsf{k}_1^{\mathsf{PCF}}, \cdot))$, where Δ^* was received from $\mathcal{A}$ via the query (PCFDeliverKeys, Δ^*).

When $\mathsf{P_S}$ is corrupted, $\mathcal{S}$ still samples $\Delta_{\mathcal{S}} \xleftarrow{\$} \mathbb{F}_{p^r}$ and emulates $\mathcal{F}_{\mathsf{PCF}}$ execution of $\mathsf{Gen}^{\mathcal{A},0}(1^\lambda, \Delta_{\mathcal{S}})$, but instead:

- $\mathcal{S}$ samples a new random $\Delta'_{\mathcal{S}}$ and $\mathcal{F}$ forwards this as the output (Initialized, $\Delta'_{\mathcal{S}}$) to the honest $\mathsf{P_R}$.
- On a query (Sample, x) from honest $\mathsf{P_R}$: $\mathcal{S}$ samples the output $[\![y]\!]_1 \xleftarrow{\$} \mathsf{VOLE}(\Delta'_{\mathcal{S}}, \mathsf{Eval}(0, \mathsf{k}_0^{\mathsf{PCF}}, \cdot))$.

H_5 is indistinguishable by the Strong PCF security of PCF. An adversary $\mathcal{A}$ that can distinguish with non-negligible probability implies an adversary $\mathcal{B}$ that wins the Strong PCF Security game $\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{B},b}(1^\lambda, \Delta, \Delta')$ in Figure 10. For all values of $(b, \sigma) \in \{0,1\}^2$ Table 3 details the output distribution of $\mathcal{O}^{\mathsf{sec}}$ in the Strong PCF Security game in Figure 10, and of $\mathcal{F}$ in H_4 and H_5.

Table 3. The distribution of the output for all values of $b, \sigma \in \{0,1\}^2$. Above, $\Delta, \Delta', \Delta_{\mathcal{S}}, \Delta'_{\mathcal{S}}, \Delta^* \in \mathbb{F}_{p^r}$, where Δ, Δ' are the input to $\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},b}$, Δ^* is chosen by $\mathcal{A}$, and $\Delta_{\mathcal{S}}, \Delta'_{\mathcal{S}}$ are sampled by $\mathcal{S}$. In the rows $\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},\cdot}$ are the distributions of the outputs in the Strong PCF Security game in Figure 10. In the rows $(\mathcal{S}, \mathcal{F})$ are the distribution of the outputs of Hybrids 4 and 5, as written.

		$b = b^* = 0$: Hybrid 5	$b = b^* = 1$: Hybrid 4
$\sigma = 0$	$\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},b}$	$\mathcal{O}^{\mathsf{sec}}_{0,0}(x) \xleftarrow{\$} \mathsf{VOLE}(\Delta', \mathsf{Eval}(0, \mathsf{k}_0^{\mathsf{PCF}}, x))$	$\mathcal{O}^{\mathsf{sec}}_{1,0}(x) = \mathsf{Eval}(1, \mathsf{k}_1^{\mathsf{PCF}}, x) \in \mathsf{VOLE}(\Delta)$
	$(\mathcal{S}, \mathcal{F})$	$\mathcal{F}(x) \xleftarrow{\$} \mathsf{VOLE}(\Delta'_{\mathcal{S}}, \mathsf{Eval}(0, \mathsf{k}_0^{\mathsf{PCF}}, x))$	$\mathcal{F}(x) = \mathsf{Eval}(1, \mathsf{k}_1^{\mathsf{PCF}}, x) \in \mathsf{VOLE}(\Delta_{\mathcal{S}})$
$\sigma = 1$	$\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},b}$	$\mathcal{O}^{\mathsf{sec}}_{0,1}(x) \xleftarrow{\$} \mathsf{VOLE}(\Delta^*, \mathsf{Eval}(1, \mathsf{k}_1^{\mathsf{PCF}}, x))$	$\mathcal{O}^{\mathsf{sec}}_{1,1}(x) = \mathsf{Eval}(0, \mathsf{k}_0^{\mathsf{PCF}}, x) \in \mathsf{VOLE}(\Delta^*)$
	$(\mathcal{S}, \mathcal{F})$	$\mathcal{F}(x) \xleftarrow{\$} \mathsf{VOLE}(\Delta^*, \mathsf{Eval}(1, \mathsf{k}_1^{\mathsf{PCF}}, x))$	$\mathcal{F}(x) = \mathsf{Eval}(0, \mathsf{k}_0^{\mathsf{PCF}}, x) \in \mathsf{VOLE}(\Delta^*)$

Assume that there exists an adversary $\mathcal{A}$ that distinguishes H_5 from H_4 with non-negligible probability, i.e., that $\mathcal{A}$ outputs $b^* = 0$ after interacting with $(\mathcal{S}, \mathcal{F})$ in H_5 with non-negligibly greater probability than after interacting with $(\mathcal{S}, \mathcal{F})$ in H_4. Our Strong PCF Security adversary $\mathcal{B}$ is constructed from $\mathcal{A}$ as follows:

1. On $(1^\lambda, \mathsf{pp}_{\mathsf{PCF}})$:
   Output $\sigma \in \{0,1\}$ in the game for whichever P_σ is corrupted by $\mathcal{A}$.
2. If $\mathsf{P}_\sigma = \mathsf{P_R}$ then on $(1^\lambda, \mathsf{pp}_{\mathsf{PCF}}, 1)$:
   Output Δ^* in the game, for (PCFDeliverKeys, Δ^*) received from $\mathcal{A}$.
3. In $\mathsf{Gen}^{\mathcal{B},\sigma}(1^\lambda, \cdot)$:
   Emulate $\mathcal{F}_{\mathsf{PCF}}$ to $\mathcal{A}$: execute $\mathsf{Gen}^{\mathcal{A},\sigma}(1^\lambda, \cdot)$ by forwarding messages between the game and $\mathcal{A}$.
4. On $(\sigma, \mathsf{k}_\sigma^{\mathsf{PCF}})$:
   (a) If $\mathsf{P}_\sigma = \mathsf{P_S}$:
      i. Query $(\Delta', \cdot) := [\![y]\!]_{1-\sigma} \leftarrow \mathcal{O}^{\mathsf{sec}}_{b,\sigma}(x)$ for $x \xleftarrow{\$} \mathcal{X}$.
      ii. Output (Initialized, Δ') to honest $\mathsf{P_R}$ as $\mathcal{F}$.
   (b) On (Sample, x) from $\mathsf{P}_{1-\sigma}$ to $\mathcal{F}$:
      i. Query $[\![y]\!]_{1-\sigma} \leftarrow \mathcal{O}^{\mathsf{sec}}_{b,\sigma}(x)$.

ii. Output $[\![y]\!]_{1-\sigma}$ as $\mathcal{F}$ to $\mathsf{P}_{1-\sigma}$.

(c) When $\mathcal{A}$ outputs $b^* \in \{0,1\}$: Output $b' = b^*$.

When $\mathsf{P_R}$ is corrupted, Δ^* chosen by $\mathcal{A}$ is fixed. When $\mathsf{P_S}$ is corrupted ($\sigma = 0$) the reduction $\mathcal{B}$ defines $\Delta_{\mathcal{S}} = \Delta$ when $b = 1$, or $\Delta'_{\mathcal{S}} := \Delta'$ when $b = 0$, which is computed from the output of the oracle (without knowledge of b, of course). In reference to Table 3, this makes the output distributions identical in all cases. Therefore, the advantage of $\mathcal{B}$ is exactly the advantage of $\mathcal{A}$, and $\mathcal{B}$ violates the Strong PCF security of PCF assuming that $\mathcal{A}$ can distinguish between H_5 and H_4 with non-negligible probability.

Hybrid H_6: *Non-Interactive Sampling.* In this hybrid, $\mathcal{F}$ computes the output of (Sample, x) to an honest party without interaction with $\mathcal{S}$. The outputs in H_6 are random in $\mathsf{VOLE}(\Delta)$, where Δ is still sent by $\mathcal{S}$. The output distribution of $\mathcal{F}$ in H_6 is identical to that in H_5; this change is perfectly indistinguishable. As of this hybrid, the (Sample) interface of $\mathcal{F}$ is fully implemented.

Hybrid H_7: *Functionality Samples* Δ *for Corrupted* $\mathsf{P_S}$ *and Honest* $\mathsf{P_R}$. In this hybrid the (VOLEInit) interface of $\mathcal{F}$ no longer forwards the output sampled by $\mathcal{S}$ to an honest $\mathsf{P_R}$ when $\mathsf{P_S}$ is corrupted, but instead samples $\Delta \xleftarrow{\$} \mathbb{F}_{p^r}$ uniformly. In this hybrid, Δ is sampled according to the same distribution by $\mathcal{F}$ as by $\mathcal{S}$ in H_6; the distribution of the outputs of $\mathcal{F}$ are the same and this change is perfectly indistinguishable. As of this hybrid $\mathcal{F}$ and $\mathcal{S}$ are fully implemented: $\mathcal{F} = \mathcal{F}_{\text{ro-VOLE}}$, and $\mathcal{S} = \mathcal{S}_{\text{ro-VOLE}}$.

Hybrid H_8: *Ideal World.* This hybrid corresponds to the ideal execution in which the environment $\mathcal{Z}$ invokes $\mathcal{F}_{\text{ro-VOLE}}$ with dummy parties $\mathsf{P_S}$ and $\mathsf{P_R}$, and interacts with $\mathcal{S}_{\text{ro-VOLE}}$. This change is perfectly indistinguishable.

□

12.2 Deterministic Two Party ECDSA

The Ideal Functionality. The two party ECDSA functionality $\mathcal{F}_{\mathsf{ECDSA}}$ is presented in Figure 32. We omit discussion of our functionality, as it is reasonably standard: P_A and P_B invoke $\mathcal{F}_{\mathsf{ECDSA}}$ for a signature, and (only) P_A receives the output signature σ at the discretion of the ideal adversary. Of particular note is the absence of a subsession identifier; our signing sessions are exclusively identified by m, the message to be signed.

The ECDSA Protocol. Our two party ECDSA protocol Π_{ECDSA} is in Figure 33. In the figure we present two variants of our protocol simultaneously: version 1 uses two VOLE instances and is highlighted in blue, version 2 uses three VOLE instances and is both highlighted in pink and boxed. All non-highlighted text is executed in both variants of the protocol. The two variants of our deterministic signing protocol are also discussed in the Technical Overview (see **Two-party ECDSA Signing**). We review some of the underlying intuition here.

Version Two. We start with the second version of our protocol, in which P_A and P_B are assumed access to three separate instances of $\mathcal{F}_{\text{ro-VOLE}}$ towards the purposes listed below:

1. $\mathcal{F}^A_{\text{ro-VOLE}}$ with P_A as receiver outputs shares sampled from $\mathsf{VOLE}(\mathsf{sk}_A)$.
   - Key Generation, Step 3d: parties sample $((w_A, \mathsf{sk}_A), (u_A, v_A)) \leftarrow \mathcal{F}^A(1)$, and the value $W_A := w_A \cdot G_{\mathsf{E}}$ is sent from P_A to P_B to authenticate $\mathsf{pk}_A = \mathsf{sk}_A \cdot G_{\mathsf{E}}$.
   - Signing, Step 1: parties sample $((w_A, \mathsf{sk}_A), (u_A, v_A)) \leftarrow \mathcal{F}^A(0\|m)$ as shares of 0. P_B randomizes its messages by adding its shares (u_A, v_A) to its secret nonce $k^{-1} - u_A := u$ and $k_B^{-1}(\mathsf{H_E}(m) + r \cdot \mathsf{sk}_B) - r \cdot v_A := v$ to its (almost) partial signature. Upon recovery of the values (u, v), P_A uses (w_A, sk_A) to cancel the shares of P_B, computing the signature as:

$$s := k_A^{-1} \cdot \underbrace{(v + \underbrace{r \cdot (w_A + \mathsf{sk}_A \cdot u)}_{= r \cdot (k_B^{-1} \cdot \mathsf{sk}_A + v_A)})}_{= k_B^{-1} \cdot (\mathsf{H_E}(m) + r(\mathsf{sk}_A + \mathsf{sk}_B))}$$

2. $\mathcal{F}^B_{\text{ro-VOLE}}$ with P_B as receiver outputs shares $\mathsf{VOLE}(\Delta_B)$.

Fig. 32: $\mathcal{F}_{\mathsf{ECDSA}}(\mathsf{H}_{\mathsf{E}}, \mathcal{G}_{\mathsf{E}}, \mathcal{X})$, ECDSA Signing Functionality

The $\mathcal{F}_{\mathsf{ECDSA}}$ functionality is parameterized by a function $\mathsf{H}_{\mathsf{E}} : \mathcal{X} \to \mathbb{F}_{q_{\mathsf{E}}}$, a tuple $\mathcal{G}_{\mathsf{E}} = (\mathbb{G}_{\mathsf{E}}, G_{\mathsf{E}}, q_{\mathsf{E}})$ for a group $\mathbb{G}_{\mathsf{E}}$ with generator G_{E} of order q_{E}, and a message space $\mathcal{X}$. The functionality interacts with parties P_A, P_B and the ideal adversary $\mathcal{S}$ via the queries below, which all implicitly include a session identifier sid. The functionality interacts with $\mathcal{S}$ on behalf of the corrupted party, and $\mathcal{S}$ can instruct the functionality to abort either key generation via an instruction of ($\mathsf{KEYabort}$), or any signing session via an instruction of ($\mathsf{SIGabort}$, m), which is then forwarded to the parties as output.

Key Generation. The key generation interface can be queried by each party a single time.

On ($\mathsf{ECDSAInit}$) **from** $\mathsf{P} \in \{\mathsf{P}_A, \mathsf{P}_B\}$**.** Send ($\mathsf{ECDSAInit}$, P) to $\mathcal{S}$. Once ($\mathsf{ECDSAInit}$) is queried by all honest $\mathsf{P} \in \{\mathsf{P}_A, \mathsf{P}_B\}$: define $\mathsf{pk} := \mathsf{sk} \cdot G_{\mathsf{E}}$ for $\mathsf{sk} \xleftarrow{\$} \mathbb{F}^*_{q_{\mathsf{E}}}$, record ($\mathsf{Keys}$, pk, sk), and send (PubKey, pk) to $\mathcal{S}$.

On ($\mathsf{ECDSADeliverKey}$) **from** $\mathcal{S}$**.** If (Keys, pk, sk) is recorded then output (PubKey, pk) to all honest $\mathsf{P} \in \{\mathsf{P}_A, \mathsf{P}_B\}$.

Two Party Signing. The signing interface opens after the key generation interface delivers output (PubKey, pk) to $\mathsf{P}_A, \mathsf{P}_B$. The signatures for each message are unique, so a single query for any m is accepted per party.

On ($\mathsf{ECDSASign}$, m) **from** $\mathsf{P} \in \{\mathsf{P}_A, \mathsf{P}_B\}$**.** On the first query for m define $R := k \cdot G_{\mathsf{E}}$ for $k \xleftarrow{\$} \mathbb{F}^*_{q_{\mathsf{E}}}$, and record ($\mathsf{Signature}$, m, σ), where:

$$\sigma := (r, s) \text{ for } r := R.\mathsf{x} \text{ and } s := k^{-1} \cdot (\mathsf{H}_{\mathsf{E}}(m) + r \cdot \mathsf{sk}).$$

If P_B is corrupted: send ($\mathsf{ECDSANonce}$, m, R) to $\mathcal{S}$.

Else if P_A is corrupted: Send ($\mathsf{ECDSASignature}$, m, σ) to $\mathcal{S}$.

Else: Send ($\mathsf{ECDSASign}$, m, P) to $\mathcal{S}$.

On ($\mathsf{ECDSADeliverSig}$, m) **from** $\mathcal{S}$**.** If all honest $\mathsf{P} \in \{\mathsf{P}_A, \mathsf{P}_B\}$ queried ($\mathsf{ECDSASign}$, m), and if P_A is honest then output the record ($\mathsf{Signature}$, m, σ) to P_A.

– Signing, Step 4d: parties sample $((w_B, \Delta_B), (k_A, v_B)) \leftarrow \mathcal{F}^B(m)$, and the value $V_B := v_B \cdot G_{\mathsf{E}}$ is used by P_B to authenticate the nonce $R_A := k_A \cdot G_{\mathsf{E}}$. Upon receipt of R_A, P_B computes $V_B := w_B - \Delta_B \cdot R_A$, a value that can be computed by P_A if $R_A = k_A \cdot G_{\mathsf{E}}$, which is then used as a one-time-pad for its response (u, v) to P_A.

3. $\mathcal{F}^C_{\text{ro-VOLE}}$ with P_A as receiver outputs shares $\mathsf{VOLE}(\Delta_C)$.
   – Signing, Step 4d: parties sample $((w_C, \Delta_C), (k_B, v_C)) \leftarrow \mathcal{F}^C(m)$, and the value $V_C := v_C \cdot G_{\mathsf{E}}$ is used by P_A to authenticate the nonce $R_B := k_B \cdot G_{\mathsf{E}}$. In a manner similar to the above, P_B uses V_C as a one-time-pad for its response to P_A. Upon receipt of R_B, P_A computes $V'_C := w_C - \Delta_C \cdot R_B$, which is exactly the value computed by P_B if $R_B = k_B \cdot G_{\mathsf{E}}$ sent by P_B.

As discussed, we optimize the protocol by having P_B xor these verification values V_B, V_C with its response $s_B := (u, v) \oplus \mathsf{H}_{\mathtt{com}}(V_B) \oplus \mathsf{H}_{\mathtt{com}}(V_C)$, such that P_A can only recover these values if the verification succeeds. With this optimization P_A sends to P_B a single group element $R_A \in \mathbb{G}_{\mathsf{E}}$, and P_B responds with R_B and s_B exactly same size as (u, v). Verification is essentially communication-free!

Version One. This variant of our protocol uses two instances of $\mathcal{F}_{\text{ro-VOLE}}$ for the same purposes above by equating $\mathcal{F}^C_{\text{ro-VOLE}} := \mathcal{F}^A_{\text{ro-VOLE}}$. To this end, the message space is halved as all sample queries are prefixed by a bit $(0\|m), (1\|m)$ to ensure that fresh shares are sampled for each purpose. Parties are additionally assumed access to a NIZKPoK Π_{DL}, discussed below.

1=3. $\mathcal{F}^A_{\text{ro-VOLE}}$ with P_A as receiver outputs shares $\mathsf{VOLE}(\mathsf{sk}_A)$.

- Key Generation Step 3d, and Signing Step 1: shares are used to authenticate pk_A, and randomize the response of P_B exactly as above.
- Signing, Step 4e and 5e: parties sample $((w_C, \mathsf{sk}_A), (k_B, v_C)) \leftarrow \mathcal{F}^A(1\|m)$ that are used by P_A to authenticate the nonce R_B. As above, P_B computes $R_B := k_B \cdot G_{\mathsf{E}}$ and $V_C := v_C \cdot G_{\mathsf{E}}$, but in contrast P_B computes a proof of knowledge $\pi := \mathsf{Prv}^{\mathsf{H}}_{\mathsf{DL}}((R_B, V_C), (k_B, v_C))$ as opposed to using the one-time-pad. Upon receipt of the response, P_A computes $V_C := v_C - \mathsf{sk}_A \cdot R_B$, recovers the message, and verifies the proof $\mathsf{Ver}^{\mathsf{H}}_{\mathsf{DL}}(\pi, (R_B, V_C))$.

Note that simply switching $\mathcal{F}^C_{\text{ro-VOLE}}$ with $\mathcal{F}^A_{\text{ro-VOLE}}$ is straightforwardly insecure. The commitment $\mathsf{pk}_A = \mathsf{sk}_A \cdot G_{\mathsf{E}}$ sent by P_A during key generation opens the possibility for a corrupted P_B to compute V^*_C for *any* R^*_B of its choosing. This motivates our inclusion of the proof of knowledge Π_{DL}. Intuitively, while P_B can compute $V^*_C := (k_B - k^*_B) \cdot \mathsf{pk}_A - v_C \cdot G_{\mathsf{E}}$ for any R^*_B, it cannot prove knowledge of the corresponding $v^*_C \neq v_C$ without *knowing* sk_A. Therefore, our proof of security for version 1 reduces a P_B that sends a message $(R^*_B, (u^*, v^*, \pi^*))$ that P_A would accept for $R^*_B \neq k_B \cdot G_{\mathsf{E}}$ to the existential unforgeability of ECDSA (Definition 2, Figure 2), given that Π_{DL} is a knowledge sound NIZKPoK for $\mathcal{R}^{\mathsf{DL}}_{\mathbb{G}_{\mathsf{E}}}$.

Fig. 33: $\Pi_{\mathtt{ECDSA}}(\mathsf{H}_{\mathsf{E}}, \mathcal{G}_{\mathsf{E}}, \mathcal{X})$ **– 2-Party ECDSA Protocol**

Note: protocol versions one and two are presented simultaneously.
The protocol executed between P_A and P_B is parameterized by the function H_{E}, group parameters $\mathcal{G}_{\mathsf{E}} = (\mathbb{G}_{\mathsf{E}}, G_{\mathsf{E}}, q_{\mathsf{E}})$ describing an additive group $\mathbb{G}_{\mathsf{E}}$ with generator G_{E} of order q_{E}, and the set $\mathcal{X} = \{0,1\}^N$. he signing protocol has message space $\{0,1\}^{N-1}$.
Parties P_A and P_B are assumed access to:
– $\mathcal{F}_{\mathtt{sec}}$: a pairwise secure channel with adversarial delay,
– $\mathsf{H}_{\mathtt{com}}(\cdot)$: an instance of $\mathcal{F}_{\mathtt{RO}}$ with output $\{0,1\}^{|\pi|+\,2\cdot\log(q_{\mathsf{E}})}$,
– $\mathcal{F}^A_{\text{ro-VOLE}} := \mathcal{F}_{\text{ro-VOLE}}(\mathbb{F}_{q_{\mathsf{E}}}, 1, \mathcal{X})$ with P_A as receiver
– $\mathcal{F}^B_{\text{ro-VOLE}} := \mathcal{F}_{\text{ro-VOLE}}(\mathbb{F}_{q_{\mathsf{E}}}, 1, \mathcal{X})$ with P_B as receiver
– $\mathcal{F}^C_{\text{ro-VOLE}} := \mathcal{F}_{\text{ro-VOLE}}(\mathbb{F}_{q_{\mathsf{E}}}, 1, \mathcal{X})$ with P_A as receiver
– $\Pi_{\mathsf{DL}} = (\mathsf{Prv}^{\mathsf{H}}_{\mathsf{DL}}, \mathsf{Ver}^{\mathsf{H}}_{\mathsf{DL}})$ a knowledge sound NIZKPoK for $\mathcal{R}^{\mathsf{DL}}_{\mathbb{G}_{\mathsf{E}}}$.

On (ECDSAInit), **Party** $\mathsf{P} \in \{\mathsf{P}_A, \mathsf{P}_B\}$**:**
1. Query (VOLEInit) to $\mathcal{F}^A_{\text{ro-VOLE}}$, $\mathcal{F}^B_{\text{ro-VOLE}}$, $\mathcal{F}^C_{\text{ro-VOLE}}$.

– On (VOLEabort) from any instance: output (KEYabort).

– If $\mathsf{P}_A = \mathsf{P}$: await (Initialized, sk_A), (Initialized), (Initialized, Δ_C).

– If $\mathsf{P}_B = \mathsf{P}$: await (Initialized), (Initialized, Δ_B), (Initialized)

2. Query (Sample, 1) to $\mathcal{F}^A_{\text{ro-VOLE}}$. $w_A = u_A \cdot \mathsf{sk}_A + v_A$
   – If $\mathsf{P}_A = \mathsf{P}$: await w_A.
   – If $\mathsf{P}_B = \mathsf{P}$: await (u_A, v_A).
3. If $\mathsf{P}_B = \mathsf{P}$:
   (a) $\mathsf{pk}_B := \mathsf{sk}_B \cdot G_{\mathsf{E}}$ for $\mathsf{sk}_B \xleftarrow{\$} \mathbb{F}^*_{q_{\mathsf{E}}}$
   (b) Send (BPubKey, pk_B) to P_A.
   (c) Await (APubKey, pk_A, W_A) from P_A.
   (d) If $W_A \neq u_A \cdot \mathsf{pk}_A + v_B \cdot G_{\mathsf{E}}$: output (KEYabort).
4. If $\mathsf{P}_A = \mathsf{P}$:
   (a) Await (BPubKey, pk_B) from P_B.
   (b) $\mathsf{pk}_A := \mathsf{sk}_A \cdot G_{\mathsf{E}}$, $W_A := w_A \cdot G_{\mathsf{E}}$.
   (c) Send (APubKey, pk_A, W_A) to P_B.
5. Output (PubKey, $\mathsf{pk} := \mathsf{pk}_A + \mathsf{pk}_B$).

On (Sign, m) **Party** $\mathsf{P} \in \{\mathsf{P}_A, \mathsf{P}_B\}$**:**

1. Send (Sample, $0\|m$) to $\mathcal{F}^A_{\text{ro-VOLE}}$ and $\mathcal{F}^B_{\text{ro-VOLE}}$, respectively.
   – If $\mathsf{P}_A = \mathsf{P}$: await w_A and (k_A, v_B). $w_A = u_A \cdot \mathsf{sk}_A + v_A$
   – If $\mathsf{P}_B = \mathsf{P}$: await (u_A, v_A) and w_B. $w_B = k_A \cdot \Delta_B + v_B$
2. Send (Sample, $1\|m$) to $\mathcal{F}^{A\ C}_{\text{ro-VOLE}}$ $w_C = k_B \ \mathsf{sk}_A \ \Delta_C + v_C$
   – If $\mathsf{P}_A = \mathsf{P}$: await w_C.
   – If $\mathsf{P}_B = \mathsf{P}$: await (k_B, v_C).
3. If $\mathsf{P}_A = \mathsf{P}$:
   (a) Send (ANonce, $m, R_A := k_A \cdot G_{\mathsf{E}}$) to P_B.
4. If $\mathsf{P}_B = \mathsf{P}$:
   (a) Await (ANonce, m, R_A) from P_A.
   (b) $r := (k_B \cdot R_A).\mathsf{x}$, $R_B := k_B \cdot G_{\mathsf{E}}$,
   (c) $u := k_B^{-1} - u_A$, $v := k_B^{-1} \cdot (\mathsf{H}_{\mathsf{E}}(m) + r \cdot \mathsf{sk}_B) - r \cdot v_A$
   (d) $V_B := w_B \cdot G_{\mathsf{E}} - \Delta_B \cdot R_A$, $V_C := v_C \cdot G_{\mathsf{E}}$
   (e) $\pi \leftarrow \mathsf{Prv}^{\mathsf{H}}_{\mathsf{DL}}((R_B, V_C), (k_B, v_C))$
   (f) $s_B := (u, v, \pi) \oplus \mathsf{H}_{\mathsf{com}}(V_B) \oplus \mathsf{H}_{\mathsf{com}}(V_C)$
   (g) Send (BSig, m, R_B, s_B)to P_A.
5. If $\mathsf{P}_A = \mathsf{P}$:
   (a) Await (BSig, m, R_B, s_B) from P_B.
   (b) $V_B := v_B \cdot G_{\mathsf{E}}$
   (c) $V_C := w_C \cdot G_{\mathsf{E}} - \mathsf{sk}_A \ \Delta_C \cdot R_B$
   (d) $(u, v, \pi) := s_B \oplus \mathsf{H}_{\mathsf{com}}(V_B) \oplus \mathsf{H}_{\mathsf{com}}(V_C)$
   (e) If $1 \neq \mathsf{Ver}^{\mathsf{H}}_{\mathsf{DL}}(\pi, (R_B, V_C))$: output (SIGabort, m).
   (f) $r := (k_A \cdot R_B).\mathsf{x}$
   (g) Signature: $\sigma := (r, s)$ for $s := k_A^{-1} \cdot (v + r \cdot (w_A + \mathsf{sk}_A \cdot u))$

(h) If $1 \neq \mathsf{Verify}(\mathsf{pk}, m, \sigma)$: output $(\mathsf{SIGabort},\ m)$.
(i) Output $(\mathsf{Signature},\ m, \sigma)$.

Theorem 16 ($\Pi_{\mathtt{ECDSA}}$ Security). *Let $\lambda \in \mathbb{N}$. Let $\mathcal{G}_{\mathsf{E}} = (\mathbb{G}_{\mathsf{E}}, G_{\mathsf{E}}, q_{\mathsf{E}})$ be a cryptographic elliptic curve, and $\mathsf{H}_{\mathsf{E}} : \{0,1\}^* \to \mathbb{F}_{q_{\mathsf{E}}}$ be a hash function used in ECDSA. For all sets $\mathcal{X}$, protocol $\Pi_{\mathtt{ECDSA}}(\mathsf{H}_{\mathsf{E}}, \mathcal{G}_{\mathsf{E}}, \mathcal{X})$ (Figure 33) UC-realizes functionality $\mathcal{F}_{\mathtt{ECDSA}}(\mathsf{H}_{\mathsf{E}}, \mathcal{G}_{\mathsf{E}}, \mathcal{X})$ (Figure 32) in the $(\mathcal{F}_{\mathtt{RO}}, \mathcal{F}_{\text{ro-VOLE}}, \mathcal{F}_{\mathtt{sec}})$-hybrid model in the presence of a malicious adversary statically corrupting at most one party. For Version 1 of $\Pi_{\mathtt{ECDSA}}$, security is under the assumption that the ECDSA signature is existentially unforgeable, and that Π_{DL} is a knowledge sound non-interactive proof of knowledge of discrete logarithm in $\mathbb{G}_{\mathsf{E}}$.*

Fig. 34: $\mathcal{S}_{\mathtt{ECDSA}}$ – Simulator for $\Pi_{\mathtt{ECDSA}}(\mathsf{H}_{\mathsf{E}}, \mathbb{G}_{\mathsf{E}}, \mathcal{X})$.

Note: the simulator for protocol versions one and two are presented simultaneously.

The simulator $\mathcal{S}_{\mathtt{ECDSA}}$ for an execution of $\Pi_{\mathtt{ECDSA}}$ between P_A and P_B with session identifier sid receives as input the function $\mathsf{H}_{\mathsf{E}} : \mathcal{X} \to \mathbb{F}_{q_{\mathsf{E}}}$, parameters $\mathcal{G}_{\mathsf{E}} = (\mathbb{G}_{\mathsf{E}}, G_{\mathsf{E}}, q_{\mathsf{E}})$ for a group $\mathbb{G}_{\mathsf{E}}$ with generator G_{E} of order q_{E}, and a set $\mathcal{X}$. The simulator is given oracle access to real-world adversary $\mathcal{A}$, and for all messages sent between parties via $\mathcal{F}_{\mathtt{sec}}$, routes all messages through $\mathcal{A}$ for delivery with adversarial-delay.

The simulator executes the instructions below on the invocations listed below, emulates the instances:
- $\mathcal{F}_{\mathtt{sec}}$: a pairwise secure channel with adversarial delay,
- $\mathsf{H}_{\mathtt{com}}(\cdot) := \mathcal{F}_{\mathtt{RO}}$ with output bitstrings of length $\ell_{\mathtt{com}} = |\pi| + 2 \cdot \mathsf{log}(q_{\mathsf{E}})$,
- $\mathcal{F}^{A}_{\text{ro-VOLE}} := \mathcal{F}_{\text{ro-VOLE}}(\mathbb{F}_{q_{\mathsf{E}}}, 1, \mathcal{X})$ with P_A as receiver
- $\mathcal{F}^{B}_{\text{ro-VOLE}} := \mathcal{F}_{\text{ro-VOLE}}(\mathbb{F}_{q_{\mathsf{E}}}, 1, \mathcal{X})$ with P_B as receiver
- $\mathcal{F}^{C}_{\text{ro-VOLE}} := \mathcal{F}_{\text{ro-VOLE}}(\mathbb{F}_{q_{\mathsf{E}}}, 1, \mathcal{X})$ with P_A as receiver

and runs the zero knowledge simulator $\mathcal{S}^{\mathsf{H}}_{\mathsf{DL}}$ for $\Pi_{\mathsf{DL}} = (\mathsf{Prv}^{\mathsf{H}}_{\mathsf{DL}}, \mathsf{Ver}^{\mathsf{H}}_{\mathsf{DL}})$.

Key Generation.

1. On $(\mathsf{ECDSAInit},\ \mathsf{P})$ from $\mathcal{F}_{\mathtt{ECDSA}}$:
   (a) For $I \in \{A, B, C\}$: Send $(\mathsf{VOLEInit},\ \mathsf{P})$ to $\mathcal{A}$ from $\mathcal{F}^{I}_{\text{ro-VOLE}}$.
2. On $(\mathsf{VOLEInit},\ \mathsf{inf}_A)$, and $(\mathsf{VOLEInit},\ \mathsf{inf}_B)$, and $(\mathsf{VOLEInit},\ \mathsf{inf}_C)$ from $\mathcal{A}$ to instances $\mathcal{F}^{A}_{\text{ro-VOLE}}$, $\mathcal{F}^{B}_{\text{ro-VOLE}}$, $\mathcal{F}^{C}_{\text{ro-VOLE}}$:
   (a) If P_A is corrupted:
      – Parse $(\mathsf{sk}_A, f_A(\cdot)) := \mathsf{inf}_A$, $f_B(\cdot) := \mathsf{inf}_B$, $(\Delta_C, f_C(\cdot)) := \mathsf{inf}_C$
      – $\Delta_B \xleftarrow{\$} \mathbb{F}_{q_{\mathsf{E}}}$.
   (b) If P_B is corrupted:
      – Parse $f_A(\cdot) := \mathsf{inf}_A$, $(\Delta_B, f_B(\cdot)) := \mathsf{inf}_B$, $f_C(\cdot) := \mathsf{inf}_C$
      – $\Delta_A \xleftarrow{\$} \mathbb{F}_{q_{\mathsf{E}}}$, $\Delta_C \xleftarrow{\$} \mathbb{F}_{q_{\mathsf{E}}}$
3. On $(\mathsf{VOLEabort})$ to $\mathcal{F}^{I}_{\text{ro-VOLE}}$ for any $I \in \{A, B, C\}$ from $\mathcal{A}$:
   – If $(\mathsf{VOLEInit},\ \cdot)$ has not been queried by $\mathcal{A}$ to $\mathcal{F}^{I}_{\text{ro-VOLE}}$: send $(\mathsf{KEYabort})$ to $\mathcal{F}_{\mathtt{ECDSA}}$.
4. On $(\mathsf{PubKey},\ \mathsf{pk})$ from $\mathcal{F}_{\mathtt{ECDSA}}$:
   (a) If P_A is corrupted:
      i. $\mathsf{pk}_A := \mathsf{sk}_A \cdot G_{\mathsf{E}}$.
      ii. $\mathsf{pk}_B := \mathsf{pk} - \mathsf{pk}_A$.
      iii. $w_A := f_A(1)$
      iv. $W_A := w_A \cdot G_{\mathsf{E}}$.

v. Send $(\mathsf{BPubKey},\ \mathsf{pk}_B)$ to P_A from P_B.

vi. Await $(\mathsf{APubKey},\ \mathsf{pk}_A^*, W_A^*)$ from P_A to P_B:

A. If $\mathsf{pk}_A^* \neq \mathsf{pk}_A$ or $W_A^* \neq W_A$: send $(\mathsf{KEYabort})$ to $\mathcal{F}_{\mathsf{ECDSA}}$.

B. Else: send $(\mathsf{ECDSADeliverKey})$ to $\mathcal{F}_{\mathsf{ECDSA}}$.

(b) Else if P_B is corrupted:

i. Await $(\mathsf{BPubKey},\ \mathsf{pk}_B^*)$ from P_B.

ii. $\mathsf{pk}_A := \mathsf{pk} - \mathsf{pk}_B$.

iii. $(u_A, v_A) := f_A(1)$

iv. $W_A := u_A \cdot \mathsf{pk}_A + v_A$

v. Send $(\mathsf{APubKey},\ \mathsf{pk}_A, W_A)$ to P_B from P_A.

vi. Send $(\mathsf{ECDSADeliverKey})$ to $\mathcal{F}_{\mathsf{ECDSA}}$.

(c) Else ($\mathsf{P}_A, \mathsf{P}_B$ honest):

i. $\mathsf{sk}_A \xleftarrow{\$} q_{\mathsf{E}}$, $W_A \xleftarrow{\$} \mathbb{G}_{\mathsf{E}}$, $W_B \xleftarrow{\$} \mathbb{G}_{\mathsf{E}}$

ii. $\mathsf{pk}_A := \mathsf{sk}_A \cdot G_{\mathsf{E}}$

iii. $\mathsf{pk}_B := \mathsf{pk} - \mathsf{pk}_A$

iv. Send $(\mathsf{BPubKey},\ \mathsf{pk}_B)$ to P_A from P_B.

v. Await $(\mathsf{BPubKey},\ \cdots)$ released/delivered by $\mathcal{A}$.

vi. Send $(\mathsf{APubKey},\ \mathsf{pk}_A, W_A)$ to P_B from P_A.

vii. Await $(\mathsf{APubKey},\ \cdots)$ released/delivered by $\mathcal{A}$.

viii. Send $(\mathsf{ECDSADeliverKey})$ to $\mathcal{F}_{\mathsf{ECDSA}}$.

Signing: Honest P_A, Honest P_B. The protocol is executed over $\mathcal{F}_{\mathsf{sec}}$, so the simulator $\mathcal{S}_{\mathsf{ECDSA}}$ simply sends random messages of the appropriate size on behalf of the simulated honest parties in the pattern indicated by the protocol, as below.

5. On $(\mathsf{ECDSAInit},\ m, \mathsf{P}_A)$ from $\mathcal{F}_{\mathsf{ECDSA}}$:
   - Sample $M \xleftarrow{\$} \mathbb{G}_{\mathsf{E}}$.
   - Send $(\mathsf{ANonce},\ m, M)$ from P_A to P_B.
6. On $(\mathsf{ECDSAInit},\ m, \mathsf{P}_B)$ from $\mathcal{F}_{\mathsf{ECDSA}}$, and $(\mathsf{ANonce},\ m, \cdot)$ from P_A released/delivered to P_B:
   - Sample $M \xleftarrow{\$} \mathbb{G}_{\mathsf{E}}$, $M' \xleftarrow{\$} \{0,1\}^{\ell_{\mathsf{com}}}$.
   - Send $(\mathsf{BSign},\ m, M, M')$ from P_B to P_A.
7. Once $(\mathsf{BSign},\ m, \cdots)$ from P_B released/delivered to P_A by $\mathcal{A}$: send $(\mathsf{ECDSADeliverSig},\ m)$ to $\mathcal{F}_{\mathsf{ECDSA}}$.

Signing: Honest P_A, Corrupted P_B. In this case, from Steps 2b and 4b, $\mathcal{S}_{\mathsf{ECDSA}}$ knows only the public information about keys $\mathsf{pk}, \mathsf{pk}_A, \mathsf{pk}_B$, and knows the entire output of each emulated instance of $\mathcal{F}_{\mathsf{ro\text{-}VOLE}}^I$, which can can be computed from Δ_I and $f_I(\cdot)$, i.e., the shares of P_B.

8. On $(\mathsf{Nonce},\ m, R)$ from $\mathcal{F}_{\mathsf{ECDSA}}$:
   (a) Compute $(u_A, v_A) := f_A(0\|m)$, $w_B := f_B(0\|m)$, $(k_B, v_C) := f_{\boxed{A}\ \boxed{C}}(1\|m)$
   (b) $R_B := k_B \cdot G_{\mathsf{E}}$
   (c) $R_A := k_B^{-1} \cdot R$
   (d) Send $(\mathsf{ANonce},\ m, R_A)$ from P_A to P_B.
9. On $(\mathsf{BSig},\ m, R_B^*, s_B^*)$ to P_A from P_B:
   (a) $V_B := \Delta_B \cdot R_A + w_B \cdot G_{\mathsf{E}}$
   (b) If $R_B^* = R_B$:
      i. $V_C := v_C \cdot G_{\mathsf{E}}$
      ii. $(u^*, v^*, \boxed{\pi^*}) := s_B^* \oplus \mathsf{H}_{\mathsf{com}}(V_B)\ \boxed{\oplus \mathsf{H}_{\mathsf{com}}(V_C)}$.

iii. If $1 \neq \mathsf{Ver}^{\mathsf{H}}_{\mathsf{DL}}(\pi^*, (R^*_B, V_C))$: Send $(\mathsf{SIGabort},\ m)$ to $\mathcal{F}$.

(c) Else if $R^*_B \neq R_B$: Send $(\mathsf{SIGabort},\ m)$ to $\mathcal{F}_{\mathsf{ECDSA}}$.

(d) If any of the below hold: send $(\mathsf{SIGabort},\ m)$ to $\mathcal{F}_{\mathsf{ECDSA}}$.

i. $u^* \neq k_B^{-1} - u_A$

ii. $v^* \cdot G_{\mathsf{E}} \neq k_B^{-1} \cdot (\mathsf{H}_{\mathsf{E}}(m) \cdot G_{\mathsf{E}} + r \cdot \mathsf{pk}_B) - r \cdot v_A \cdot G_{\mathsf{E}}$

(e) Else: Send $(\mathsf{ECDSADeliverSig},\ m)$ to $\mathcal{F}_{\mathsf{ECDSA}}$.

Signing: Corrupted P_A*, Honest* P_B. In this case, from Steps 2a and 4a, $\mathcal{S}_{\mathtt{ECDSA}}$ knows the public information about the keys $\mathsf{pk}, \mathsf{pk}_A, \mathsf{pk}_B$, the secret key sk_A, and the entire output of each emulated instance of $\mathcal{F}^I_{\mathsf{ro\text{-}VOLE}}$, which can can be computed from Δ_I and $f_I(\cdot)$, i.e., the shares of P_A.

10. On $(\mathsf{ECDSASignature},\ m, \sigma)$ from $\mathcal{F}_{\mathsf{ECDSA}}$, and $(\mathsf{ANonce},\ m, R^*_A)$ from P_A to P_B:

(a) Parse the signature $(r, s) := \sigma$

(b) Compute the shares of P_B: $w_A := f_A(0\|m)$, $(k_A, v_B) := f_B(0\|m)$, $w_C := f_{A\ C}(1\|m)$

(c) Recover $R \in \mathbb{G}_{\mathsf{E}}$ such that $r = R.\mathsf{x}$.

(d) Define $R_B := k_A^{-1} \cdot R$

(e) If $R^*_A = k_A \cdot G_{\mathsf{E}}$:

i. $V_B := v_B \cdot G_{\mathsf{E}}$

ii. $V_C := w_C \cdot G_{\mathsf{E}} - \mathsf{sk}_A\ \Delta_C \cdot R_B$

iii. $\pi \leftarrow \mathcal{S}^{\mathsf{H}}_{\mathsf{DL}}((R_B, V_C))$

iv. $u \xleftarrow{\$} \mathbb{F}_{q_{\mathsf{E}}}$

v. $v := k_A \cdot s - r \cdot (w_A - \mathsf{sk}_A \cdot u)$

vi. Define $s_B := (u, v, \pi) \oplus \mathsf{H}_{\mathtt{com}}(V_B) \oplus \mathsf{H}_{\mathtt{com}}(V_C)$.

(f) Else: Sample $s_B \xleftarrow{\$} \{0,1\}^{\ell_{\mathtt{com}}}$.

(g) Send $(\mathsf{BSig},\ m, R_B, s_B)$ to $\mathcal{A}$ for P_B.

Proof. The simulator $\mathcal{S}_{\mathtt{ECDSA}}$ appears in Figure 34. We begin by showing correctness of the protocol, then proceed with a hybrid argument.

Correctness. The output (r, s) of P_A in $\Pi_{\mathtt{ECDSA}}$ is a verifying ECDSA signature under the key $\mathsf{pk} = (\mathsf{sk}_A + \mathsf{sk}_B) \cdot G_{\mathsf{E}}$, as shown in Eq 26.

$$
\begin{aligned}
s :=& k_A^{-1} (v + r \cdot (w_A + \mathsf{sk}_A \cdot u)) \\
=& k_A^{-1} \left(k_B^{-1} \cdot (\mathsf{H}_{\mathsf{E}}(m) + r \cdot \mathsf{sk}_B) - r \cdot v_A + r \cdot \left(w_A + \mathsf{sk}_A \cdot \left(k_B^{-1} - u_A\right)\right)\right) \\
=& k_A^{-1} \cdot k_B^{-1} \cdot (\mathsf{H}_{\mathsf{E}}(m) + r \cdot \mathsf{sk}_B) \\
& - k_A^{-1} \cdot r \cdot v_A + k_A^{-1} \cdot r \cdot \left(w_A + \mathsf{sk}_A \cdot \left(k_B^{-1} - u_A\right)\right) \\
=& k^{-1} \cdot (\mathsf{H}_{\mathsf{E}}(m) + r \cdot \mathsf{sk}_B) + k_A^{-1} \cdot r \cdot (w_A - \mathsf{sk}_A \cdot u_A - v_A) + k^{-1} \cdot r \cdot \mathsf{sk}_A \\
=& k^{-1} \cdot (\mathsf{H}_{\mathsf{E}}(m) + r \cdot (\mathsf{sk}_A + \mathsf{sk}_B))
\end{aligned}
\tag{26}
$$

Hybrid H_0*: Real World.* This is the real execution of the protocol $\Pi_{\mathtt{ECDSA}}$, in which the environment $\mathcal{Z}$ runs the protocol between the parties P_A and P_B with access to ideal functionalities $(\mathcal{F}_{\mathsf{ro\text{-}VOLE}}, \mathcal{F}_{\mathtt{RO}}, \mathcal{F}_{\mathtt{sec}})$, along with the real world adversary $\mathcal{A}$.

Hybrid H_1*: Ideal Layout.* The existing instances $(\mathsf{P}_A, \mathsf{P}_B, \mathcal{F}_{\mathsf{ro\text{-}VOLE}}, \mathcal{F}_{\mathtt{RO}}, \mathcal{F}_{\mathtt{sec}}, \mathcal{A})$ are grouped into a single node called the simulator $\mathcal{S}$ that forwards communication between $\mathcal{A}$ and $\mathcal{Z}$ without modification. The environment $\mathcal{Z}$ now interacts with new dummy party nodes for P_A and P_B that simply forward input/output to a new dummy functionality node $\mathcal{F}$ that similarly forwards input/output to $\mathcal{S}$. This change is syntactic; H_1 is perfectly indistinguishable from that in H_0.

Hybrid H_2: *Functionality Record Keeping & Output Queries.* In this hybrid, $\mathcal{S}$ issues queries and $\mathcal{F}$ maintains records in all of the signing sessions. The simulator issues queries to $\mathcal{F}$ that include the output to be delivered, corresponding to aborts or outputs in the protocol simulation: (ECDSADeliverKey), (KEYabort), (ECDSADeliverSig), and (SIGabort) queries. The functionality now maintains records for the signatures and party queries, but still forwards input/outputs between P_A, P_B and $\mathcal{S}$. This change does not impact the view of $\mathcal{Z}$; H_2 is perfectly indistinguishable from H_1.

Hybrid H_3: *Full Simulation for Honest* P_A *and Honest* P_B. In this hybrid when both P_A and P_B are honest $\mathcal{S}$ and $\mathcal{F}$ are modified in the following ways:

– *Changes to* $\mathcal{F}$ as in $\mathcal{F}_{\text{ECDSA}}$, Figure 32:
  - Samples $\mathsf{sk} \leftarrow \mathbb{F}^*_{q_\mathsf{E}}$ and defines $\mathsf{pk} := \mathsf{sk} \cdot G_\mathsf{E}$.
  - Public key is released to $\mathcal{S}$ via (PubKey, pk), but sk is not released to $\mathcal{S}$.
  - Generates all signatures once queried by both P_A and P_B.
  - Honest party signing queries released to $\mathcal{S}$ via (ECDSASign, m, P) but the signature is not included.

– *Changes to* $\mathcal{S}$ as in $\mathcal{S}_{\text{ECDSA}}$, Figure 34:
  - Key generation is simulated as in Step 4c; the secret key of P_A is sampled at random, the public key of P_B is then computed from pk and sk_A.
  - Signing is simulated as in Step 5, 6, and 7; the honest party messages are randomly sampled to be of the correct length.

As the protocol is written in the $\mathcal{F}_{\mathsf{sec}}$-hybrid model only the message length is leaked to $\mathcal{A}$. This case simply requires correctness of the protocol, which was shown above. Therefore, H_3 is perfectly indistinguishable from H_2. As of H_3, both $\mathcal{S}$ and $\mathcal{F}$ are fully implemented in the case that both P_A and P_B are honest.

Hybrid H_4: *Simulated Proofs from Simulated* P_B. This hybrid behaves identically to H_3, except in version 1 of the protocol when P_A is corrupted. In this case, the simulator invokes the proof system simulator $\pi \leftarrow \mathcal{S}^{\mathsf{H}}((R_A, V_C))$ to simulate the proof π sent to P_A by P_B; H_4 is indistinguishable from H_3 by the zero knowledge of Π_{DL}.

Hybrid H_5: *Simulated* V_C *for Corrupted* P_A *from Honest* P_B. This hybrid behaves identically to H_4, except that when P_A is corrupted $\mathcal{S}$ computes the value V_C as in Step 10(e)ii of $\mathcal{S}_{\text{ECDSA}}$, using only the information of the corrupted P_A. Specifically, $V_C := w_C \cdot G_\mathsf{E} -$ sk_A Δ_C $\cdot R_B$ is computed from the shares extracted from $\mathcal{F}^A_{\text{ro-VOLE}}$ or $\mathcal{F}^C_{\text{ro-VOLE}}$. In both H_4 and H_5 the shares of P_A are fixed and R_B is uniform in $\mathbb{G}_\mathsf{E}$; V_C is identically distributed in H_5 and H_4.

Hybrid H_6: *Simulated* V_B *for Corrupted* P_A *from Honest* P_B. In this hybrid when P_A is corrupted $\mathcal{S}$ precomputes the message expected from P_A by extracting $(k_A, v_B) := f_B(0\|m)$ from $\mathcal{F}^B_{\text{ro-VOLE}}$ and computing $R_A := k_A \cdot G_\mathsf{E}$. Then, $\mathcal{S}$ proceeds as below upon receipt of (ANonce, m, R^*_A) from $\mathcal{A}$:

– *Changes to* $\mathcal{S}$ *when* P_A *is corrupted:*
  - Extract $(k_A, v_B) := f_B(0\|m)$
  - Compute $R_A := k_A \cdot G_\mathsf{E}$
  - Await (ANonce, m, R^*_A) from $\mathcal{A}$.
  - ⋆ *Changed:* If $R^*_A = R_A$: define $V_B := v_B \cdot G_\mathsf{E}$, as in Step 10(e)i of $\mathcal{S}_{\text{ECDSA}}$.
  - ⋆ *Changed:* Else if $R^*_A \neq R_A$: sample $V_B \xleftarrow{\$} \mathbb{G}_\mathsf{E}$ uniformly.

If $R^*_A = R_A$ the simulated response from P_B in H_6 is identically distributed to the response in H_5; $V_B = v_B \cdot G_\mathsf{E} = w_B \cdot G_\mathsf{E} - \Delta_B \cdot R^*_A = v_B \cdot G_\mathsf{E} = w_B \cdot G_\mathsf{E} - \Delta_B \cdot R_A$.

A distinguisher between H_6 and H_5 must send $R^*_A \neq R_A$ and compute the corresponding $V^*_B = v_B \cdot G_\mathsf{E} = w_B \cdot G_\mathsf{E} - \Delta_B \cdot R^*_A$. In H_5 this V^*_B would unmask a verifying tuple $(u, v,$ π $) := s_B \oplus \mathsf{H}_{\mathtt{com}}(V_B)$ $\oplus\mathsf{H}_{\mathtt{com}}(V_C)$, whereas in H_6 the unmasked tuple $s_B \oplus \mathsf{H}_{\mathtt{com}}(V_B)$ $\oplus\mathsf{H}_{\mathtt{com}}(V_C)$ is uniform in $\{0,1\}^{\ell_{\mathtt{com}}}$.

We argue that $\mathcal{Z}$ can at best guess this value randomly, then verify the unmasked tuple. Observe that in both H_6 and H_5 a system of linear equations over $\mathbb{F}_{q_\mathsf{E}}$ can be computed from the view of $\mathcal{Z}$, in which all values are indexed by m to indicate that value in signing session m. In H_6 the relevant system is $F = \{v_{B,m} = Y_m - k_{A,m} \cdot X\}_{m \in \mathcal{X}}$ in the variables $(X, \{Y_m\}_{m \in \mathcal{X}})$ corresponding to $(\Delta_B, w_{B,m})_{m \in \mathcal{X}}$. In H_6 the simulated output of P_B is otherwise independent of these values; in H_6 the system F cannot be solved by $\mathcal{Z}$.

In H_5 the output of P_B has additional dependency on $X = \Delta_B$ when $R^*_A \neq R_A$ via the hashed value $V^*_B := v^*_B \cdot G_{\mathsf{E}} = w_B \cdot G_{\mathsf{E}} - \Delta_B \cdot R^*_A$. This dependency corresponds to equations $F^* = \{Z_m = Y_m - k^*_{A,m} \cdot X\}_{m \in Q'}$ involving the variables relevant to signing sessions $Q' \subseteq Q_{\mathsf{sig}} \subseteq \mathcal{X}$ in which the inequality $R^*_A \neq R_A$ holds. As $\mathsf{H}_{\mathsf{com}}$ is an instance of $\mathcal{F}_{\mathtt{RO}}$, the output $\mathsf{H}_{\mathsf{com}}(Z_m \cdot G_{\mathsf{E}})$ is independent of the variables X, Y_m, Z_m and despite the known structure of the unmasked message sent by the simulated P_B the unmasked values (u, v) are unpredictable to $\mathcal{Z}$ prior to unmasking.Therefore, in H_5 the $2 \cdot |Q'|$ equations in $F \cup F^*$ in the $2 \cdot |Q'| + 1$ variables $(X, \{Y_m, Z_m\}_{m \in Q'})$ for $Q' \subseteq Q_{\mathsf{sig}}$ the subset of signing sessions in which $R^*_A \neq R_A$ cannot be solved, just as in H_6.

A distinguishing $\mathcal{Z}$ can therefore only guess $V^*_B \xleftarrow{\$} \mathbb{G}_{\mathsf{E}}$ and check the unmasked tuple. Such a $\mathcal{Z}$ makes Q_{com} guesses at $Q' \subseteq Q_{\mathsf{sig}}$ values in $\mathbb{G}_{\mathsf{E}}$, and is successful with probability at most $q_{\mathsf{E}}^{-1} \cdot |Q_{\mathsf{sig}}| \cdot |Q_{\mathsf{com}}|$, which is negligible when $Q_{\mathsf{sig}} = Q_{\mathsf{sig}}(\lambda)$, $Q_{\mathsf{com}} = Q_{\mathsf{com}}(\lambda)$ are polynomial in λ and q_{E} is exponential in λ.

Hybrid H_7: *Random Responses to Corrupted* P_A. In this hybrid, $\mathcal{S}$ again changes the responses to a corrupted P_A from a simulated P_B. In H_7 the response is computed as below.

– *Changes to* $\mathcal{S}$ *when* P_A *is corrupted:*
  - Extract $(k_A, v_B) := f_B(0 \| m)$.
  - ⋆ *Changed:* Compute a signature (r, s) under $\mathsf{sk} = \mathsf{sk}_A + \mathsf{sk}_B$ with nonce shares $k = k_A \cdot k_B$ for $k_B \xleftarrow{\$} \mathbb{F}^*_{q_{\mathsf{E}}}$.
  - Compute $R_A := k_A \cdot G_{\mathsf{E}}$
  - Await $(\mathsf{ANonce},\ m, R^*_A)$ from $\mathcal{A}$.
  - ⋆ *Changed:* If $R^*_A = R_A$ simulate as in Step 10e of $\mathcal{S}_{\mathtt{ECDSA}}$: compute V_B, V_C as in H_6, sample $u \xleftarrow{\$} \mathbb{F}_{q_{\mathsf{E}}}$, and compute v such that P_A will recover s.
  - ⋆ *Changed:* Else if $R^*_A \neq R_A$: sample $s_B \xleftarrow{\$} \{0,1\}^{\ell_{\mathtt{com}}}$ uniformly.

In the case that $R^*_A \neq R_A$ the value V_B is uniform as of H_6, because $\mathsf{H}_{\mathsf{com}}$ is a random oracle the output $\mathsf{H}_{\mathsf{com}}(V_B)$ is uniform in $\{0,1\}^{\ell_{\mathtt{com}}}$, and therefore the message s_B is uniform. In this case, the distribution of s_B in H_6 is exactly that in H_7.

In the case that $R^*_A = R_A$ the values $(u_A, v_A) \xleftarrow{\$} \mathsf{VOLE}(\mathsf{sk}_A, w_A)$ and therefore the distribution of u is uniform. There is a single value v any each pair u, k_B, which is computed in the simulated computation. Therefore, (u, v) is identically distributed in H_7 and H_6.

Hybrid H_8: *Full Simulation for Corrupted* P_A *and Honest* P_B. In this hybrid, $\mathcal{F}$ now samples the secret values for P_A and P_B—specifically, the secret key sk and the signatures (r, s). As of H_8, $\mathcal{F}$ and $\mathcal{S}$ are both fully implemented in the case that P_A is corrupted; this hybrid is identical to H_7 except for the following changes when P_A is corrupted:

– *Changes to* $\mathcal{F}$ as in $\mathcal{F}_{\mathsf{ECDSA}}$, Figure 32:
  - Sample $\mathsf{sk} \leftarrow \mathbb{F}^*_{q_{\mathsf{E}}}$ and define $\mathsf{pk} := \mathsf{sk} \cdot G_{\mathsf{E}}$.
  - Release $(\mathsf{PubKey},\ \mathsf{pk})$ to $\mathcal{S}$ (sk is not released).
  - Generate $\sigma = (r, s)$ once queried by P_B.
  - Release $(\mathsf{ECDSASignature},\ m, \sigma)$ to $\mathcal{S}$ (secret nonce is not released).

– *Changes to* $\mathcal{S}$ as in $\mathcal{S}_{\mathtt{ECDSA}}$, Figure 34:
  - Key generation is simulated as in Step 4a:
    - sk_A is received from $\mathcal{A}$ via $\mathcal{F}^A_{\mathsf{ro\text{-}VOLE}}$
    - ⋆ *Changed:* $(\mathsf{PubKey},\ \mathsf{pk})$ is received from $\mathcal{F}$ once queried by P_B
    - ⋆ *Changed:* $\mathsf{pk}_B := \mathsf{pk} - \mathsf{sk}_A \cdot G_{\mathsf{E}}$ is sent to P_A
  - Nonce generation is simulated as in Step 10d of $\mathcal{S}_{\mathtt{ECDSA}}$:
    - ⋆ *Changed:* $(\mathsf{ECDSASignature},\ m, (r, s))$ received from $\mathcal{F}$.
    - k_A extracted from $\mathcal{F}^B_{\mathsf{ro\text{-}VOLE}}$
    - ⋆ *Changed:* $R_B := (k_A)^{-1} \cdot R$
  - ⋆ *Changed:* (u, v) sampled as in Step 10(e)v of $\mathcal{S}_{\mathtt{ECDSA}}$ to be consistent with (r, s).

As of H_7, $\mathcal{S}$ did not use any of the secret values of P_B to simulate signing sessions to a corrupted P_A except to generate a valid signature (r, s). In H_8 this signature is received from $\mathcal{F}$ and is identically distributed to the signature in H_7 as $R \in \mathbb{G}_{\mathsf{E}}$ is uniform in both hybrids. Therefore, the view of $\mathcal{Z}$ in the signing sessions is identically distributed in H_8 and H_7.

In H_7 the value sk_A was received from a corrupted P_A, and pk_B was sampled by $\mathcal{S}$ to be uniform. This still holds in H_8, as $\mathcal{F}$ samples pk uniformly. Therefore, H_8 is perfectly indistinguishable from H_7.

Hybrid H_9: *Simulated* V_B *for Corrupted* P_B *from Honest* P_A. In this hybrid, when P_B is corrupted $\mathcal{S}$ computes the verification value V_B as in as in Step 9a of $\mathcal{S}_{\text{ECDSA}}$, using only the information of P_B. Specifically, $V_B := w_B \cdot G_{\mathsf{E}} - \Delta_B \cdot R_A$ for the share (Δ_B, w_B) extracted from $\mathcal{F}^B_{\text{ro-VOLE}}$. In both H_8 and H_9 the shares (Δ_B, w_B) of P_B extracted from $\mathcal{F}^B_{\text{ro-VOLE}}$ are fixed and V_B only depends on R_A sent by $\mathcal{S}$; this change in computation does not result in any change to the view of $\mathcal{Z}$.

Hybrid H_{10}: *Deterministic Nonces from Corrupted* P_B *in Case 2*. In this hybrid when P_B is corrupted $\mathcal{S}$ computes a portion of the message expected from P_B and then computes the value V_C depending on the actual received message, as below.

– *Changes to* $\mathcal{S}$ *when* P_B *is corrupted:*
  - Await $(\mathsf{BSig}, m, R^*_B, s^*_B)$ from $\mathcal{A}$
  - Extract $(k_B, v_C) := f_C(1\|m)$
  - ⋆ *Changed:* If $R^*_B = k_B \cdot G_{\mathsf{E}}$: Define $V_C := v_C \cdot G_{\mathsf{E}}$.
  - ⋆ *Changed:* Else in Case 2 Send $(\mathsf{SIGabort}, m)$ to $\mathcal{F}$.

If $R^*_B = R_B := k_B \cdot G_{\mathsf{E}}$ then the simulated computation of V_C using only values known by P_B recovers the same value as in H_9; H_{10} is identical to H_9.

What remains is when $R^*_B \neq R_B$. A distinguisher between H_9 and H_{10} must send $R^*_B \neq R_B$ and compute the corresponding $V^*_C = w_C \cdot G_{\mathsf{E}} - \Delta_C \cdot R^*_B$; in H_9, the simulated P_A would recover the same V^*_C and unmask the message $(u^*, v^*) := s^*_B \oplus \mathsf{H}_{\mathtt{com}}(V_B) \oplus \mathsf{H}_{\mathtt{com}}(V^*_C)$ for subsequent verification, whereas in H_{10} sending an unexpected R^*_B will always result in $(\mathsf{SIGabort}, m)$ as output.

Observe that the only output of P_A that depends on Δ_C is whether or not a signing session aborts, leaking information to $\mathcal{Z}$ in the case that $R^*_B \neq R_B$; $\mathcal{Z}$ learns nothing about Δ_C except whether a signature was successfully reconstructed by P_A as this is the only output of P_A that depends on Δ_C. Effectively, in H_9 $\mathcal{Z}$ has $|Q_{\mathsf{sig}}|$ guesses for Δ_C, whereas in H_{10} there is no possibility for $\mathcal{Z}$ to guess Δ_C. Therefore, $\mathcal{Z}$ has $q_{\mathsf{E}}^{-1} \cdot |Q_{\mathsf{sig}}|$ chance in guessing the correct Δ_C in H_9; the advantage of $\mathcal{Z}$ in distinguishing H_9 from H_{10} is negligible when $Q_{\mathsf{sig}} = Q_{\mathsf{sig}}(\lambda)$ is polynomial in λ and q_{E} is exponential in λ.

Hybrid H_{11}: *Full Simulation for Honest* P_A *and Corrupted* P_B *in Case 1*. In this hybrid when P_B is corrupted in case 1 $\mathcal{S}$ verifies that a message received from P_B has the exact structure that would result in verification by an honest P_A. Notably, the simulator needs neither sk_A nor sk_B to verify the structure of the received messages. The functionality is changed to account for the fact that $\mathcal{S}$ no longer needs to know anything other than the secrets of P_B, as below:

– *Changes to* $\mathcal{F}$ *when* P_B *is corrupted in Case 1 as in* $\mathcal{F}_{\text{ECDSA}}$, *Figure 32:*
  - Sample $\mathsf{sk} \leftarrow \mathbb{F}^*_{q_{\mathsf{E}}}$ and define $\mathsf{pk} := \mathsf{sk} \cdot G_{\mathsf{E}}$.
  - Release $(\mathsf{PubKey}, \mathsf{pk})$ to $\mathcal{S}$ (sk is not released).
  - Generate $\sigma = (r, s)$ once queried by P_B, and release $(\mathsf{ECDSANonce}, m, R)$ to $\mathcal{S}$ for $R = k \cdot G_{\mathsf{E}}$.

The first set of changes to $\mathcal{S}$ when P_B is corrupted are exactly as in the simulation of key generation in Step 4b by $\mathcal{S}_{\text{ECDSA}}$. In this case $\mathcal{S}$ receives pk from $\mathcal{F}$ and pk_B from P_B, defines $\mathsf{pk}_A := \mathsf{pk} - \mathsf{pk}_B$, and simulates W_A with the shares of P_B extracted from $\mathcal{F}^A_{\text{ro-VOLE}}$.

– *Changes to* $\mathcal{S}$ *when* P_B *is corrupted in Case 1, Key Generation, as in Step 4b of* $\mathcal{S}_{\text{ECDSA}}$.
  - Receive $(\mathsf{PubKey}, \mathsf{pk}_A)$ from $\mathcal{F}$.
  - Emulate all instances of $\mathcal{F}_{\text{ro-VOLE}}$.
  - Await $(\mathsf{BPubKey}, \mathsf{pk}_B)$ from P_B.
  - Define $\mathsf{pk} := \mathsf{pk}_A + \mathsf{pk}_B$
  - Send $(\mathsf{AddPubKey}, \mathsf{pk}_B)$ to $\mathcal{F}$.
  - Extract $(u_A, v_A) := f_A(1)$
  - Set $W_A := u_A \cdot \mathsf{pk}_B + v_A \cdot G_{\mathsf{E}}$
  - Send $(\mathsf{APubKey}, \mathsf{pk}_A, W_A)$ to P_B.
  - Send $(\mathsf{ECDSADeliverKey})$ to $\mathcal{F}$.

These changes are perfectly indistinguishable to $\mathcal{Z}$; $\mathsf{pk}, \mathsf{pk}_A$ are identically distributed in H_{11} and H_{10}, and there is a single valid choice of W_A per pk_A.

– *Changes to* $\mathcal{S}$ *when* P_B *is corrupted in Case 1, Signing, as in Step 9b of* $\mathcal{S}_{\text{ECDSA}}$:
  - On (Nonce, m, R):
    - ∗ Extract $(k_B, v_C) := f_A(1\|m)$.

    * Send $(\mathsf{ANonce},\ m, (k_B)^{-1} \cdot R)$ to P_B.
- Await $(\mathsf{BSig},\ m, R_B^*, s_B^*)$ from $\mathcal{A}$
- Extract $(u_A, v_A) := f_A(0\|m)$.
- Compute V_B as in H_9
- Unmask $(u^*, v^*, \pi^*) := s_B^* \oplus \mathsf{H}_{\mathsf{com}}(V_B)$
- ⋆ *Changed:* If $R_B^* \neq k_B \cdot G_{\mathsf{E}}$: send $(\mathsf{SIGabort},\ m)$ to $\mathcal{F}$.
- ⋆ *Changed:* Else:
    * Define $V_C^* := v_C \cdot G_{\mathsf{E}}$.
    * $r^* := \left((u^* - u_A)^{-1} \cdot R_A\right).\mathsf{x}$
    * If any of the following hold send $(\mathsf{SIGabort},\ m)$ to $\mathcal{F}$:
        · $(u^* - u_A) \cdot R_B^* \neq G_{\mathsf{E}}$
        · $(v^* + r^* \cdot v_A) \cdot R_B^* - \mathsf{H}_{\mathsf{E}}(m) \cdot G_{\mathsf{E}} \neq r^* \cdot \mathsf{pk}_B$
        · $1 \neq \mathsf{Ver}_{\mathsf{DL}}^{\mathsf{H}}(\pi^*, V_C^*)$
- Else: Query $(\mathsf{ECDSADeliverSig},\ m)$ to $\mathcal{F}$.

Claim 1: For fixed public key $\mathsf{pk} = \mathsf{sk}_A \cdot G_{\mathsf{E}} + \mathsf{pk}_B$ and message $(\mathsf{BSig},\ m, R_B^*, s_B^*)$ received from P_B in signing session m, if $1 = \mathsf{Verify}(\mathsf{pk}, m, (r^*, s))$ for the signature (r^*, s) recovered by P_A as above then the pair (u^*, v^*) satisfies Eq 27.

$$\begin{aligned} u^* &= (k_B^*)^{-1} - u_A^* \\ v^* &= (k_B^*)^{-1} \cdot (\mathsf{H}_{\mathsf{E}}(m) + r^* \cdot \mathsf{sk}_B) - r^* \cdot v_A^* \\ w_A &= u_A \cdot \mathsf{sk}_A + v_A = u_A^* \cdot \mathsf{sk}_A + v_A^* \end{aligned} \tag{27}$$

where $k_B^* \in \mathbb{F}_{q_{\mathsf{E}}}^*$ is such that $k_B^* \cdot G_{\mathsf{E}} = R_B^*$, $r^* := (k_B^* \cdot R_A).\mathsf{x}$, $w_A = u_A \cdot \mathsf{sk}_A + v_A$ are the outputs of $\mathcal{F}_{\mathsf{ro\text{-}VOLE}}^{A}$ for $(1\|m)$, and $(u_A^*, v_A^*) \in \mathsf{VOLE}(\mathsf{sk}_A, w_A)$ are arbitrary.

To argue the claim we observe that there always exists some $X \in \mathbb{F}_{q_{\mathsf{E}}}$ such that $u^* = (k_B^*)^{-1} + X$, and that if reconstruction results in a verifying signature (r, s) then the following is implied:

$$\begin{aligned} s &:= k_A^{-1} \cdot (v^* + r^* \cdot (w_A + \mathsf{sk}_A \cdot u^*)) = (k_A \cdot k_B^*)^{-1} (\mathsf{H}_{\mathsf{E}}(m) + r^* \cdot (\mathsf{sk}_A + \mathsf{sk}_B)) \\ v^* &= \underbrace{(k_B^*)^{-1} (\mathsf{H}_{\mathsf{E}}(m) + r^* \cdot \mathsf{sk}_B)}_{\text{Part 1}} + \underbrace{(k_B^*)^{-1} (r^* \cdot \mathsf{sk}_A) - r^* \cdot (w_A + \mathsf{sk}_A \cdot u^*)}_{\text{Part 2}} \\ &= \text{Part 1} + (k_B^*)^{-1} \cdot r^* \cdot \mathsf{sk}_A - r^* \cdot \left(w_A + \mathsf{sk}_A \cdot \underbrace{\left((k_B^*)^{-1} + X\right)}_{=u^*} \right) \\ &= \text{Part 1} + (k_B^*)^{-1} \cdot r^* \cdot \mathsf{sk}_A - r^* \cdot w_A + r^* \cdot \mathsf{sk}_A \cdot (k_B^*)^{-1} + r^* \cdot \mathsf{sk}_A \cdot X \\ &= \text{Part 1} - r^* \cdot \underbrace{(w_A - \mathsf{sk}_A \cdot X)}_{=v_A^*} \end{aligned}$$

For any assignment $X = u_A^*$ we see that $(u_A^*, v_A^*) \in \mathsf{VOLE}(\mathsf{sk}_A, w_A)$, as was claimed.

Given the structure of the message received from P_B, it is clear that the two cases that $\mathcal{S}$ queries $(\mathsf{SIGabort},\ m)$ in H_{11} but not in H_{10} is when $\mathcal{A}$ computes:

1. $(u_A, v_A) \neq (u_A, v_A) := f_A(0\|m)$, or
2. compute a proof π^* of knowledge of $(k_B^*, v_C^*) = \left(\log_{G_{\mathsf{E}}}(V_C^*), \log_{G_{\mathsf{E}}}(R_B^*)\right)$ such that $(k_B^*, v_C^*) \neq (k_B, v_C) := f_A(1\|m)$, and $k_B^* \cdot \mathsf{sk}_A + v_C^* = k_B \cdot \mathsf{sk}_A + v_C$.

An adversary $\mathcal{A}$ with non-negligible advantage in distinguishing H_{10} from H_{11} implies an adversary $\mathcal{B}$ with non-negligible advantage in breaking the existential unforgeability of ECDSA (Definition 2) by way of extracting the signing key sk from $\mathcal{A}$. The adversary $\mathcal{B}$ is as follows:

1. Receive $\mathsf{pk} \in \mathbb{G}_{\mathsf{E}}$ from $\mathsf{Exp}_{\mathcal{B}}^{\mathsf{EUF\text{-}CMA}}$.
2. Maintain initially empty sets Q, Q^*.
3. Sample $b \xleftarrow{\$} \{0, 1\}$.
4. If $b = 0$: execute H_{10} as $\mathcal{S}, \mathcal{F}$.

– Upon receipt of a message from P_B such that π^* verifies and the reconstructed signature s verifies, if either $R_B^* \neq k_B \cdot G_{\mathsf{E}}$ or $(u_A^*, v_A^*) \neq (u_A, v_A)$ then add m to Q^*.

5. If $b = 1$: execute H_{11} as $\mathcal{S}, \mathcal{F}$.
   (a) Simulate generation of public key pk as in H_{11}.
   (b) Upon input to simulated P_A to sign a message m, until Q^* non-empty:
   – Query $(r, s) \leftarrow \mathcal{O}^{\mathsf{sig}}(m)$. Add m to Q.
   – Simulate the first round message of P_A as in H_{11}.
   – Await R_B^*, s_B^* from $\mathcal{A}$.
   – If $R_B^* \neq k_B \cdot G_{\mathsf{E}}$:
   • $(u^*, v^*, \pi^*) := s_B^* \oplus \mathsf{H}_{\mathtt{com}}(V_B)$
   • Extract $((R_B^*, V_C^*), (k_B^*, v_C^*)) \leftarrow \mathcal{E}_{\mathsf{DL}}^{\mathsf{H},\mathcal{A}}(\pi^*)$ (via rewinding)
   ∗ If extraction fails send $(\mathsf{SIGabort},\ m)$ to $\mathcal{A}$ to abort session m.
   • Verify $k_B \cdot \mathsf{pk}_A + v_C \cdot G_{\mathsf{E}} = k_B^* \cdot \mathsf{pk}_A + v_C^* \cdot G_{\mathsf{E}}$
   • $u_A^* := (k_B^*)^{-1} - u^*$
   • $r^* := (k_B^* \cdot R_A).\mathsf{x}$
   • Verify $k_B^* \cdot (v^* - r^* \left((u_A - u_A^*)\, \mathsf{pk}_A + v_A G_{\mathsf{E}}\right)) = \mathsf{H}_{\mathsf{E}}(m) \cdot G_{\mathsf{E}} + r^* \mathsf{pk}_B$
   • Add m to Q^*.
   • Send $(\mathsf{SIGabort},\ m)$ to $\mathcal{A}$.
   – Else:
   • Verify the received message as in H_{11}.
   • If the signature verifies send the signature to $\mathcal{A}$.
   (c) Once Q^* is non-empty, then retrieve the relevant values for $m \in Q^*$, and:
   – Compute $\mathsf{sk}_A := (k_B^* - k_B) \cdot (v_C - v_C^*)^{-1}$
   – Compute $u_A^* := u^* - (k_B^*)^{-1}$
   – Compute $v_A^* := \mathsf{sk}_A \cdot (u_A - u_A^*) + v_A$
   – Compute $r^* := (k_B^* \cdot R_A)\,.\mathsf{x}$.
   – Compute $\mathsf{sk}_B := (r^*)^{-1} \cdot (k_B^* \cdot (v^* - r^* \cdot v_A^*) - \mathsf{H}_{\mathsf{E}}(m))$
   – Compute $\mathsf{sk} := \mathsf{sk}_A + \mathsf{sk}_B$ to $\mathcal{C}$.
6. Await $\mathcal{A}$ to output b^*.
7. If $b^* = b = 1$ and Q^* is non-empty:
   – Sample $m^* \xleftarrow{\$} \{0,1\}^{N-1} \setminus Q$.
   – Compute $\sigma^* := (r^*, s^*) \leftarrow \mathsf{Sign}\,(\mathsf{sk}, m)$.
   – Output (m^*, σ^*) to $\mathsf{Exp}_{\mathcal{B}}^{\mathsf{EUF\text{-}CMA}}$.
8. Else, output $\perp$.

By assumption the advantage of $\mathcal{A}$ in distinguishing H_{11} from H_{10} is $\frac{1}{2} + \epsilon(\cdot)$ for some non-negligible function $\epsilon(\cdot)$.

$$\begin{aligned}\mathsf{Adv}(\mathcal{A}) = \frac{1}{2} + \epsilon(\lambda) &= \Pr\left[b^* = b\right] \\ &= \Pr\left[b^* = b \wedge Q^* \neq \emptyset\right] + \Pr\left[b^* = b \wedge Q^* = \emptyset\right] \\ &= \frac{1}{2} \cdot \Pr\left[Q^* = \emptyset\right] + \Pr\left[b^* = b \wedge Q^* \neq \emptyset\right]\end{aligned}$$

Above, the first equation is by the law of total probability, and the second is because if $Q^* = \emptyset$ then b^* is independent of $b \xleftarrow{\$} \{0,1\}$. Rearranging, we compute the probability that $\mathcal{A}$ is correct and $Q^* \neq \emptyset$ by rearranging the above and replacing $\Pr[Q^* = \emptyset] = 1 - \Pr[Q^* \neq \emptyset]$, as:

$$\begin{aligned}\Pr\left[b^* = b \wedge Q^* \neq \emptyset\right] &= \frac{1}{2} + \epsilon(\lambda) - \frac{1}{2} \cdot \Pr\left[Q^* = \emptyset\right] \\ &= \frac{1}{2} \Pr\left[Q^* \neq \emptyset\right] + \epsilon(\lambda)\end{aligned}$$

Again by the law of total probability and substitutions, we have that the probability that $\mathcal{B}$ can output a verifying forgery is non-negligible:

$$\begin{aligned}
\mathsf{Adv}_{\mathsf{EUF}}(\mathcal{B}) &:= \Pr\left[b^* = b \wedge Q^* \neq \emptyset \wedge b = 1\right] \\
&= \Pr\left[b^* = b \wedge Q^* \neq \emptyset\right] \\
&\qquad - \Pr\left[b^* = b \wedge Q^* \neq \emptyset \wedge b = 0\right] \\
&= \frac{1}{2}\Pr\left[Q^* \neq \emptyset\right] + \epsilon(\lambda) \\
&\qquad - \Pr\left[b^* = b \wedge Q^* \neq \emptyset | b = 0\right] \cdot \Pr\left[b = 0\right] \\
&= \frac{1}{2}\left(\Pr\left[Q^* \neq \emptyset\right] - \Pr\left[b^* = b \wedge Q^* \neq \emptyset | b = 0\right]\right) + \epsilon(\lambda) \\
&= \frac{1}{2}\left(\Pr\left[Q^* \neq \emptyset \wedge b^* \neq b\right]\right) + \epsilon(\lambda) \\
&\geq \epsilon(\lambda).
\end{aligned}$$

Therefore, under the assumption that ECDSA is existentially unforgeable and that Π_{DL} satisfies Definition 3.4, no adversary $\mathcal{A}$ that can distinguish H_{11} from H_{10} exists. We note that as of H_{11}, $\mathcal{S} = \mathcal{S}_{\text{ECDSA}}$ and $\mathcal{F} = \mathcal{F}_{\mathsf{ECDSA}}$ are fully implemented in case 1.

Hybrid H_{12}: *Full Simulation for Corrupted* P_B *in Case 2*. As of this hybrid, $\mathcal{S}$ does not sample any secrets of P_A and simulates the protocol without using any leakage from $\mathcal{F}$ other than that allowed by $\mathcal{F}_{\mathsf{ECDSA}}$. First, we review the changes to $\mathcal{F}$.

– *Changes to* $\mathcal{F}$ *when* P_B *is corrupted in Case 2 as in* $\mathcal{F}_{\mathsf{ECDSA}}$*, Figure 32:*
  - Sample $\mathsf{sk} \leftarrow \mathbb{F}^*_{q_{\mathsf{E}}}$ and define $\mathsf{pk} := \mathsf{sk} \cdot G_{\mathsf{E}}$.
  - Release $(\mathsf{PubKey},\ \mathsf{pk})$ to $\mathcal{S}$.
  - Generate $\sigma = (r, s)$ once queried by P_B, and release $(\mathsf{ECDSANonce},\ m, R)$ to $\mathcal{S}$ for $R = k \cdot G_{\mathsf{E}}$.

The first set of changes to $\mathcal{S}$ when P_B is corrupted are exactly as in Step 4b of $\mathcal{S}_{\text{ECDSA}}$. In this case $\mathcal{S}$ receives pk from $\mathcal{F}$ and pk_B from P_B, defines $\mathsf{pk}_A := \mathsf{pk} - \mathsf{pk}_B$, and simulates the value W_A with the shares of P_B extracted from $\mathcal{F}^A_{\text{ro-VOLE}}$.

– *Changes to* $\mathcal{S}$ *when* P_B *is corrupted in Case 2, Key Generation, as in Step 4b of* $\mathcal{S}_{\text{ECDSA}}$.
  - Receive $(\mathsf{PubKey},\ \mathsf{pk}_A)$ from $\mathcal{F}$.
  - Emulate all instances of $\mathcal{F}_{\text{ro-VOLE}}$.
  - Await $(\mathsf{BPubKey},\ \mathsf{pk}_B)$ from P_B.
  - Define $\mathsf{pk} := \mathsf{pk}_A + \mathsf{pk}_B$
  - Send $(\mathsf{AddPubKey},\ \mathsf{pk}_B)$ to $\mathcal{F}$.
  - Extract $(u_A, v_A) := f_A(1)$
  - Set $W_A := u_A \cdot \mathsf{pk}_B + v_A \cdot G_{\mathsf{E}}$
  - Send $(\mathsf{APubKey},\ \mathsf{pk}_A, W_A)$ to P_B.
  - Send $(\mathsf{ECDSADeliverKey})$ to $\mathcal{F}$.

These changes are perfectly indistinguishable to $\mathcal{Z}$; $\mathsf{pk}, \mathsf{pk}_A$ are identically distributed in H_{12} and H_{11}, and tthere is a single valid choice of W_A per pk_A.

– *Changes to* $\mathcal{S}$ *when* P_B *is corrupted in Case 2, Signing Session* m*, as in Step 9c of* $\mathcal{S}_{\text{ECDSA}}$:
  - Await $(\mathsf{BSig},\ m, R^*_B, s^*_B)$ from $\mathcal{A}$
  - Extract $(u_A, v_A) := f_A(0\|m)$, $(k_B, v_C) := f_C(1\|m)$
  - If $k_B \cdot G_{\mathsf{E}} \neq R^*_B$: send $(\mathsf{SIGabort},\ m)$ to $\mathcal{F}$ as in H_{10}
  - Compute V_B and V_C as in H_9 and H_{10}
  - Unmask $(u^*, v^*) := s^*_B \oplus \mathsf{H}_{\mathtt{com}}(V_B) \oplus \mathsf{H}_{\mathtt{com}}(V_B)$
  - ⋆ *Changed:* Otherwise, if the verification below does not hold: send $(\mathsf{SIGabort},\ m)$ to $\mathcal{F}$.
    - Define $U^*_A := \left(u^* - k_B^{-1}\right) \cdot \mathsf{pk}_A$
    - Define $V^*_A := v_A \cdot G_{\mathsf{E}} = r^{-1} \cdot \left(v^* \cdot G_{\mathsf{E}} - k_B^{-1} \cdot \left(\mathsf{H}_{\mathsf{E}}(m) \cdot G_{\mathsf{E}} + r \cdot \mathsf{pk}_B\right)\right)$
    - Verify $u_A \cdot \mathsf{pk}_A + v_A \cdot G_{\mathsf{E}} = U^*_A + V^*_A$
  - Else: send $(\mathsf{ECDSADeliverSig},\ m)$ to $\mathcal{F}$.

The only change of $\mathcal{S}$ in this case is that verification of the messages from corrupted P_B is done without knowledge of the secrets of P_A. By Claim 1, signature reconstruction and verification is equivalent to verifying that the received messages have the structure seen in H_{11}, which is exactly what is done by $\mathcal{S}$ above. Therefore, there is no change in the view of $\mathcal{Z}$. As of this hybrid, $\mathcal{F} = \mathcal{F}_{\mathsf{ECDSA}}$ and $\mathcal{S} = \mathcal{S}_{\mathsf{ECDSA}}$ are fully implemented in case 2 when P_B is corrupted.

Hybrid H_{13}: *Ideal World.* This hybrid corresponds to the ideal execution in which the environment $\mathcal{Z}$ invokes $\mathcal{F}_{\mathsf{ECDSA}}$ with dummy parties P_A and P_B and interacts with $\mathcal{S}_{\mathsf{ECDSA}}$. This change is perfectly indistinguishable because $\mathcal{F} = \mathcal{F}_{\mathsf{ECDSA}}$ and $\mathcal{S} = \mathcal{S}_{\mathsf{ECDSA}}$ as of:

- H_3 when both P_A and P_B are honest,
- H_8 when P_A is corrupted,
- H_{11} when P_B is corrupted in Case 1, and
- H_{12} when P_B is corrupted in Case 2.

□

12.3 Deterministic Two Party Schnorr

Our Deterministic two-party Schnorr Signature protocol $\Pi_{\mathtt{Sch}}$ is presented in Figure 35. This protocol is simple; P_A and P_B use two instances of $\mathcal{F}_{\mathsf{ro\text{-}VOLE}}$ and a single random oracle instance for optimization:

1. $\mathcal{F}^{A}_{\mathsf{ro\text{-}VOLE}}$ with P_A as receiver outputs k_B and authenticates $R_B = k_B \cdot G_{\mathsf{S}}$,
2. $\mathcal{F}^{B}_{\mathsf{ro\text{-}VOLE}}$ with P_B as receiver outputs k_A and authenticates $R_A = k_A \cdot G_{\mathsf{S}}$.

We incorporate a similar optimization to that in version two of $\Pi_{\mathtt{ECDSA}}$; P_B uses the R_A received from P_A and its output from the instances of $\mathcal{F}_{\mathsf{ro\text{-}VOLE}}$ to compute verification values $V_A := v_A \cdot G_{\mathsf{S}}$ and $V_B := w_B \cdot G_{\mathsf{S}} - \Delta_B \cdot R_A$ that can be recomputed by P_A if the exchanged nonces R_A and R_B were computed from the output of $\mathcal{F}_{\mathsf{ro\text{-}VOLE}}$, then sends the partial signature $(k_B + e \cdot \mathsf{sk}_B) \oplus \mathsf{H}_{\mathtt{com}}(V_A) \oplus \mathsf{H}_{\mathtt{com}}(V_B)$. As both functionality instances correspond to a secret Δ that is never publicly committed to by the receiver—as was the case when $\Delta_A = \mathsf{sk}_A$ in our ECDSA $\Pi_{\mathtt{ECDSA}}$—this protocol does not require a NIZKPoK. Our approach to Schnorr is discussed extensively in our Technical Overview (see **Two-party Schnorr signing**).

Schnorr as a signature scheme (Figure 2) is significantly simpler than ECDSA and, therefore, much simpler to distribute. The primary difference between Schnorr and ECDSA is that ECDSA computes a non-linear operation over a multiplicatively shared secret k, requiring the parties to perform a secure multiplication. There is no such requirement for Schnorr; the first component of the signature $s = k + e \cdot \mathsf{sk}$ is linear in the secret key sk and the nonce k. The first component of the signature $s = k + e \cdot \mathsf{sk}$ is linear in the secret key sk and the nonce k. In our Schnorr protocol the nonce $R = R_A + R_B$ is additively shared, in contrast to the multiplicative sharing in ECDSA, just as $\mathsf{pk} = \mathsf{pk}_A + \mathsf{pk}_B$. Therefore, security of our Schnorr protocol $\Pi_{\mathtt{Sch}}$ follows by a similar (but simpler) analysis to that of $\Pi_{\mathtt{ECDSA}}$.

Fig. 35: $\Pi_{\mathsf{Sch}}(\mathsf{H}_\mathsf{S}, \mathcal{G}_\mathsf{S}, \mathcal{X})$ **2-Party Schnorr Protocol**

The protocol executed between P_A and P_B is parameterized by the function H_S, a group $\mathcal{G}_\mathsf{S} = (\mathbb{G}_\mathsf{S}, G_\mathsf{S}, q_\mathsf{S})$, and message space $\mathcal{X}$. Parties P_A and P_B execute the protocol with access to random oracle $\mathsf{H}_{\mathsf{com}} : \{0,1\}^* \to \mathbb{F}_{q_\mathsf{S}}$, and:

- $\mathcal{F}^A_{\mathsf{ro\text{-}VOLE}} := \mathcal{F}_{\mathsf{ro\text{-}VOLE}}(q_\mathsf{S}, 1, \mathcal{X})$ with P_A as receiver, and
- $\mathcal{F}^B_{\mathsf{ro\text{-}VOLE}} := \mathcal{F}_{\mathsf{ro\text{-}VOLE}}(q_\mathsf{S}, 1, \mathcal{X})$ with P_B as receiver,

On (SCHNORRInit), **Party** $\mathsf{P} \in \{\mathsf{P}_A, \mathsf{P}_B\}$**:**

1. Query (VOLEInit) to $\mathcal{F}^A_{\mathsf{ro\text{-}VOLE}}$, $\mathcal{F}^B_{\mathsf{ro\text{-}VOLE}}$.
   - On (VOLEabort) from either: output (KEYabort).
   - If $\mathsf{P}_A = \mathsf{P}$: await (Initialized, Δ_A) and (Initialized), respectively.
   - If $\mathsf{P}_B = \mathsf{P}$: await (Initialized) and (Initialized, Δ_B), respectively.
2. If $\mathsf{P}_A = \mathsf{P}$:
   (a) $\mathsf{pk}_A := \mathsf{sk}_A \cdot G_\mathsf{S}$ for $\mathsf{sk}_A \xleftarrow{\$} \mathbb{F}_{q_\mathsf{S}}$
   (b) Send (AKeyCommit, $C := \mathsf{H}_{\mathsf{com}}(\mathsf{pk}_A)$) to P_B.
3. If $\mathsf{P}_B = \mathsf{P}$:
   (a) Await (AKeyCommit, C) from P_A.
   (b) $\mathsf{pk}_B := \mathsf{sk}_B \cdot G_\mathsf{S}$ for $\mathsf{sk}_B \xleftarrow{\$} \mathbb{F}^*_{q_\mathsf{S}}$
   (c) Send (BPubKey, pk_B) to P_A.
4. If $\mathsf{P}_A = \mathsf{P}$:
   (a) Await (BPubKey, pk_B) from P_B.
   (b) Send (APubKey, pk_A) to P_B.
5. If $\mathsf{P}_B = \mathsf{P}$:
   (a) Await (APubKey, pk_A) from P_A.
   (b) If $C \neq \mathsf{H}_{\mathsf{com}}(\mathsf{pk}_A)$: output (KEYabort).
6. Output (PubKey, pk) for $\mathsf{pk} := \mathsf{pk}_A + \mathsf{pk}_B$.

On (SCHNORRSign, m), **Party** $\mathsf{P} \in \{\mathsf{P}_A, \mathsf{P}_B\}$**:**

1. Send (Sample, m) to $\mathcal{F}^A_{\mathsf{ro\text{-}VOLE}}$ and $\mathcal{F}^B_{\mathsf{ro\text{-}VOLE}}$.
   - If $\mathsf{P}_A = \mathsf{P}$: await w_A and (k_A, v_B). $\quad w_A = k_B \cdot \Delta_A + v_A$
   - If $\mathsf{P}_B = \mathsf{P}$: await (k_B, v_A) and w_B. $\quad w_B = k_A \cdot \Delta_B + v_B$
2. If $\mathsf{P}_A = \mathsf{P}$:
   (a) $R_A := k_A \cdot G_\mathsf{S}$.
   (b) Send (ANonce, m, R_A) to P_B.
3. If $\mathsf{P}_B = \mathsf{P}$:
   (a) Await (ANonce, m, R_A) from P_A.
   (b) $R_B := k_B \cdot G_\mathsf{S}$
   (c) $V_A := v_A \cdot G_\mathsf{S}$, $V_B := w_B \cdot G_\mathsf{S} - \Delta_B \cdot R_A$
   (d) $e \leftarrow \mathsf{H}_\mathsf{S}(\mathsf{pk}, R, m)$ for $R := R_A + R_B$
   (e) Define $s_B := (k_B + e \cdot \mathsf{sk}_B) \oplus \mathsf{H}_{\mathsf{com}}(V_A) \oplus \mathsf{H}_{\mathsf{com}}(V_B)$.
   (f) Send (BSig, m, R_B, s_B) to P_A.
4. If $\mathsf{P}_A = \mathsf{P}$:
   (a) Await (BSig, m, R_B, s_B) from P_B.
   (b) $V_A := w_A \cdot \mathbb{G}_\mathsf{S} - \Delta_A \cdot R_B$, $V_B := v_B \cdot \mathbb{G}_\mathsf{S}$
   (c) $s' := s_B \oplus \mathsf{H}_{\mathsf{com}}(V_A) \oplus \mathsf{H}_{\mathsf{com}}(V_B)$
   (d) If $1 \neq \mathsf{Verify}(\mathsf{pk}_B, m, (R_B, s'))$: Output (SIGabort, m).
   (e) $e \leftarrow \mathsf{H}_\mathsf{S}(\mathsf{pk}, R, m)$ for $R := R_A + R_B$
   (f) $s := s' + k_A + e \cdot \mathsf{sk}_A$
   (g) Output (Signature, $m, (R, s)$).

13 Parameters

To use the PCF of [BCMRS25] (see Section 7.2) in our signing protocols, we need to decide the number of levels L, as well as the LPN parameters (Section 13.1) and the choice on how to instantiate the DPF (i.e., the specification of the DPF) on each level $\ell \in [L]$. Our optimization goals are 1. fast distributed key generation protocol, 2. fast evaluation, and 3. small key sizes. We explore different choices for $L \in \{3, 4, 5\}$.

13.1 LPN Parameters

We start with the parameters from [BCMRS25] and set $m^{(\ell)} = s \cdot m^{(\ell-1)}$ for $\ell \in [L]$ with $s = 2^5$ and $m^{(0)} = 2^{15}$.

Number Noise Coordinates t. In [BCMRS25], the authors used the same number $t = 2700$ of noise coordinates in every level $\ell \in [L]$ of their PCF. This was chosen such that the LPN instance on the first level $\ell = 1$ yields $\lambda = 128$ bit of security.

Since for higher levels $\ell > 1$, the ratio $s = \frac{m^{(\ell)}}{m^{(\ell-1)}}$ stays constant, a lower value of t would be sufficient for the same security level. The choice of t is – as long as the LPN instance is secure – a tradeoff between number of DPFs and their domain size (which influence the evaluation cost of the DPF). In our distributed setup protocols, the main costs are the number of secure multiplications performed in MPC, which scale linearly with the number of DPF instances that are based on Naor-Reingold. Hence, to obtain a faster distributed setup, we can choose a lower $t^{(\ell)}$ on level ℓ.

Table 4 show the minimum $t^{(\ell)}$ on each level $\ell \in [L]$ to obtain a secure LPN instance according to the LPN estimator[32] of [LWYY24]. For $\ell > 3$, we take $t^{(\ell)} = t^{(3)}$.

Table 4. Minimum noise $t^{(\ell)}$ for each level $\ell \in [3]$ with $m^{(\ell-1)} = 2^{15+(\ell-1)\cdot 5}$ and $m^{(\ell)} = 2^{20+(\ell-1)\cdot 5}$ to achieve at least $\lambda = 128$ bits of security according to the LPN estimator.

ℓ	$m^{(\ell-1)}$	$m^{(\ell)}$	$t^{(\ell)}$
1	2^{15}	2^{20}	$\geq$2700
2	2^{20}	2^{25}	$\geq$1600
3	2^{25}	2^{30}	$\geq$1300

Matrix Sparsity k. To obtain a stateless construction, the PCF relies on the filtered LPN assumption (Definition 12) on the outermost level L. where the adversary can query at most 2^λ rows of the matrix. To choose $k^{(L)}$ we follow the methodology used by [BCMRS25] to select their optimistic parameter sets: we choose the lowest $k^{(L)}$ where we can prove a non-trivial lower bound on the dual distance of a $2^\lambda \times n$ matrix. We use the dual distance analysis from [BCMRS25] (see Section 4.2 for more details), and report the obtained values in Table 5. For the matrix $\mathbf{B}$, which has $m^{(L)}$ columns, choose the sparsity k^B according to row $L + 1$ of Table 5.

13.2 Choosing the DPF

Given a PCF instantiation with L levels and $t^{(\ell)}$ noise coordinates in every level $\ell \in [L]$, we turn to selecting, for each level ℓ, a PPRF candidate in {GGM-PPRF, CHHK25-PPRF, k-NR-PPRF} over a domain of size $m^{(\ell)}/t^{(\ell)}$. For each candidate with master public key mpk and secret key sk, we measure the total key size as $|\mathsf{mpk}| + t^{(\ell)} \cdot |\mathsf{sk}|$. For each candidate, we also estimate the runtime of the PPRF evaluation, as well as the runtime of the DKG for the corresponding KI-DPF. All quantities are computed as a function of the domain size $m^{(\ell)}/t^{(\ell)}$. We used a script to exhaustively try all possible combinations of candidates (including all possible choices of the parametrization of k-NR-PPRF) and sort the best-performing combinations according either to setup runtime, evaluation time, or total key size.

[32] https://lpnestimator.com/

Table 5. Minimum sparsity $k^{(L)}$ to obtain a non-trivial lower bound D on the dual distance of a $2^\lambda \times n$ matrix with $\lambda = 128$, such that D holds with probability at least $1 - 2^{-\rho}$.

L	$n = m^{(L-1)}$	$k^{(L)}$		
		$\rho = 40$	$\rho = 64$	$\rho = 80$
3	2^{25}	14	16	16
4	2^{30}	12	13	13
5	2^{35}	10	11	11
6	2^{40}	8	9	10

Table 6. LPN parameter for our instantiation of Recursive-LPN-PCF with $\lambda = 128$, $\rho = 40$, field size $|\mathbb{F}_p| \approx 2^{256}$. All parameter sets start with secret size $n = 2^{15}$ and expand by a factor $s = 32$ on each level.

L	$m^{(L)}$	$t^{(\ell)}$	$k^{(\ell)}$	k^B
3	2^{30}	$(2700, 1600, 1300)$	$(7, 6, 14)$	12
4	2^{35}	$(2700, 1600, 1300, 1300)$	$(7, 6, 5, 12)$	10
5	2^{40}	$(2700, 1600, 1300, 1300, 1300)$	$(7, 6, 5, 5, 10)$	8

Estimating the Runtime of the Distributed Setup. To estimate the runtime of the distributed setup, we either rely on published benchmarks for common cryptographic primitives and protocols, or run our own benchmarks (all our benchmarks were run on one core of a CPU Intel Core Ultra 9 285H (Arrow Lake), 16C/24T, at 2.3 GHz). In more details, we used the following:

- For VOLE-based zero-knowledge, we use the QuickSilver protocol [YSWW21]. The authors report in [YSWW21, Table 1] a throughput of 4.8M multiplications/s (over an Amazon EC2 machine m5.2xlarge) over a 61-bit field. We extrapolate the throughput by estimating a 4× slowdown over a 256-bit field with the same statistical security.
- For maliciously-secure end-to-end VOLE generation, we rely on Wolverine [WYKW21]. In Table 2 and Table 4, the authors report 10.8M VOLEs produced at a rate of 130ns/VOLE (over an Amazon EC2 instance type m5.4xlarge) over a 61-bit field. We extrapolate a comparable 4× slowdown over a larger field.
- For authenticated Beaver triples, we rely on Overdrive [KPR18, Table 2], who report 60,000 authenticated triples per second over a 64-bit field, and 30,000 over a 128-bit field. We follow the same linear scale extrapolation (in line with their numbers) and estimate 15,000 authenticated triples per second over a 256-bit field (the authors used i7-4790 and i7-3770S CPUs with 16 to 32 GB of RAM).
- For all elliptic curve operations, we ran the benchmarks locally using Bitcoin's libsecp256k1 library.[33].
- For hashing with SHA-256, we ran the benchmarks locally using OpenSSL's implementation.
- For AES operations, as in previous works [BCMRS25], we relied on the numbers reported in [MSY21] that show a throughput of 1.3cycles/byte using AES-NI on an Intel Core i7-980X.
- Eventually, we use SoftSpokenOT [Roy22] to get benchmarks for oblivious transfer extension.

Parameter Sets for Different Optimization Goals. Table 7 reports the combinations of KI-DPFs selected by our script according to the estimated benchmarks. Numbers reported in the introduction correspond to the last row of the $L = 4$ part of the table. For each value of L, the first row reports a pure GGM implementation, which provides unacceptably slow runtimes for setup, but can give excellent performances in settings where a trusted dealer is distributing the keys (e.g., when a client is delegating its signing tasks to two servers).

[33] https://github.com/bitcoin-core/secp256k1

Table 7. Key sizes and runtime estimates for the Recursive-LPN-PCF instantiate with various number of levels L and combinations of KI-DPFs

PPRF Combination	Key Sizes (KiB)			Runtime Estimates	
	$\lvert\mathsf{k}_0^{\mathsf{PCF}}\rvert$	$\lvert\mathsf{k}_1^{\mathsf{PCF}}\rvert$	$\sum$	Eval (ms)	Setup (s)
$L = 3$, $m^{(L)} = 2^{30}$					
GGM-GGM-GGM	3 568	1 112	4 680	0.007	79.5
GGM-GGM-96NR	4 170	1 254	5 424	1.545	20.9
GGM-GGM-32NR	3 485	1 294	4 779	1.778	13.6
GGM-GGM-16NR	3 441	1 335	4 776	2.011	11.8
GGM-GGM-8NR	3 498	1 416	4 914	2.478	11.0
$L = 4$, $m^{(L)} = 2^{35}$					
GGM-GGM-GGM-GGM	4 161	1 132	5 293	0.025	2 541.1
GGM-GGM-GGM-72NR	4 487	1 315	5 801	1.501	97.1
GGM-GGM-GGM-32NR	4 047	1 355	5 403	1.696	91.7
GGM-GGM-GGM-18NR	3 992	1 396	5 388	1.890	89.7
GGM-GGM-96NR-72NR	4 459	1 457	5 916	3.039	36.9
GGM-GGM-32NR-72NR	4 283	1 498	5 781	3.272	30.9
GGM-GGM-32NR-32NR	3 844	1 538	5 382	3.467	25.5
GGM-GGM-16NR-18NR	3 829	1 619	5 449	3.895	22.0
GGM-GGM-10NR-12NR	3 883	1 701	5 584	4.322	20.5
$L = 5$, $m^{(L)} = 2^{40}$					
GGM-GGM-GGM-GGM-GGM	4 856	1 152	6 009	0.103	81 311.3
GGM-GGM-GGM-18NR-19NR	4 420	1 721	6 141	3.614	100.7
GGM-GGM-96NR-18NR-19NR	4 951	1 863	6 814	10.023	41.9
GGM-GGM-6NR-5NR-4NR	4 842	2 594	7 437	17.100	28.0
GGM-GGM-4NR-3NR-4NR	5 125	2 879	8 004	20.016	27.6

Acknowledgements. This research was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement number 101115790 (OBELiSC), and the French Agence Nationale de la Recherche (ANR) under the France 2030 ANR Project ANR-22-PECY-003 SecureCompute.

References

[ABCM25] Benny Applebaum, Dung Bui, Geoffroy Couteau, and Nikolas Melissaris. *Structured-Seed Local Pseudorandom Generators and Their Applications*. In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2025, Berkeley, CA, USA, August 11-13, 2025*. 2025. DOI: 10.4230/LIPICS.APPROX/RANDOM.2025.63. URL: https://doi.org/10.4230/LIPIcs.APPROX/RANDOM.2025.63.

[ACGMY26] Michael Adjedj, Geoffroy Couteau, Arik Galansky, Nikolaos Makriyannis, and Oren Yomtov. *Stateless 2PC Signatures for Internet-Scale Authentication and Authorization*. ASIACCS. 2026. URL: https://eprint.iacr.org/2025/1475.

[ADINZ17] Benny Applebaum, Ivan Damgård, Yuval Ishai, Michael Nielsen, and Lior Zichron. *Secure Arithmetic Computation with Constant Computational Overhead*. In: *Advances in Cryptology – CRYPTO 2017, Part I*. Aug. 2017. DOI: 10.1007/978-3-319-63688-7_8.

[Adj+24] Michael Adjedj, Constantin Blokh, Geoffroy Couteau, Arik Galansky, Antoine Joux, and Nikolaos Makriyannis. *Two-Round 2PC ECDSA at the Cost of 1 OLE*. Cryptology ePrint Archive, Report 2024/1950. 2024. URL: https://eprint.iacr.org/2024/1950.

[AHI11] Benny Applebaum, Danny Harnik, and Yuval Ishai. *Semantic Security under Related-Key Attacks and Applications*. In: *ICS 2011: 2nd Innovations in Computer Science*. Jan. 2011.

[Ale03] Michael Alekhnovich. *More on Average Case vs Approximation Complexity*. In: *44th Annual Symposium on Foundations of Computer Science*. Oct. 2003. DOI: 10.1109/SFCS.2003.1238204.

[ANOSS22] Damiano Abram, Ariel Nof, Claudio Orlandi, Peter Scholl, and Omer Shlomovits. *Low-Bandwidth Threshold ECDSA via Pseudorandom Correlation Generators*. In: *2022 IEEE Symposium on Security and Privacy*. May 2022. DOI: 10.1109/SP46214.2022.9833559.

[ARSTZ15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. *Ciphers for MPC and FHE*. In: *Advances in Cryptology – EUROCRYPT 2015, Part I*. Apr. 2015. DOI: 10.1007/978-3-662-46800-5_17.

[AS22] Damiano Abram and Peter Scholl. *Low-Communication Multiparty Triple Generation for SPDZ from Ring-LPN*. In: *PKC 2022: 25th International Conference on Theory and Practice of Public Key Cryptography, Part I*. Mar. 2022. DOI: 10.1007/978-3-030-97121-2_9.

[ASM06] Man Ho Au, Willy Susilo, and Yi Mu. *Constant-Size Dynamic k-TAA*. In: *SCN 06: 5th International Conference on Security in Communication Networks*. Sept. 2006. DOI: 10.1007/11832072_8.

[BB04] Dan Boneh and Xavier Boyen. *Short Signatures Without Random Oracles*. In: *Advances in Cryptology – EUROCRYPT 2004*. May 2004. DOI: 10.1007/978-3-540-24676-3_4.

[BBMRS21] Carsten Baum, Lennart Braun, Alexander Munch-Hansen, Benoît Razet, and Peter Scholl. *Appenzeller to Brie: Efficient Zero-Knowledge Proofs for Mixed-Mode Arithmetic and Z2k*. In: *ACM CCS 2021: 28th Conference on Computer and Communications Security*. Nov. 2021. DOI: 10.1145/3460120.3484812.

[BBMS22] Carsten Baum, Lennart Braun, Alexander Munch-Hansen, and Peter Scholl. *Moz$\mathbb{Z}_{2^k}$arella: Efficient Vector-OLE and Zero-Knowledge Proofs over $\mathbb{Z}_{2^k}$*. In: *Advances in Cryptology – CRYPTO 2022, Part IV*. Aug. 2022. DOI: 10.1007/978-3-031-15985-5_12.

[BBS04] Dan Boneh, Xavier Boyen, and Hovav Shacham. *Short Group Signatures*. In: *Advances in Cryptology – CRYPTO 2004*. Aug. 2004. DOI: 10.1007/978-3-540-28628-8_3.

[BCGI18] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. *Compressing Vector OLE*. In: *ACM CCS 2018: 25th Conference on Computer and Communications Security*. Oct. 2018. DOI: 10.1145/3243734.3243868.

[BCMPR24] Dung Bui, Geoffroy Couteau, Pierre Meyer, Alain Passelègue, and Mahshid Riahinia. *Fast Public-Key Silent OT and More from Constrained Naor-Reingold*. In: *Advances in Cryptology – EUROCRYPT 2024, Part VI*. May 2024. DOI: 10.1007/978-3-031-58751-1_4.

[BCMRS25] Lennart Braun, Geoffroy Couteau, Kelsey Melissaris, Mahshid Riahinia, and Elahe Sadeghi. *Fast Pseudorandom Correlation Functions from Sparse LPN*. In: *Advances in Cryptology – ASIACRYPT 2025, Part VII*. Dec. 2025. DOI: 10.1007/978-981-95-5122-4_14.

[BCMRS26] Lennart Braun, Geoffroy Couteau, Kelsey Melissaris, Mahshid Riahinia, and Elahe Sadeghi. *Succinct Two-Round Two-Party Signing from PCFs*. In: 2026. DOI: 10.1007/978-3-032-35374-0_4.

[BCPW15] Fabrice Benhamouda, Geoffroy Couteau, David Pointcheval, and Hoeteck Wee. *Implicit Zero-Knowledge Arguments and Applications to the Malicious Setting*. In: *Advances in Cryptology – CRYPTO 2015, Part II*. Aug. 2015. DOI: 10.1007/978-3-662-48000-7_6.

[BGI14] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. *Functional Signatures and Pseudorandom Functions*. In: *PKC 2014: 17th International Conference on Theory and Practice of Public Key Cryptography*. Mar. 2014. DOI: 10.1007/978-3-642-54631-0_29.

[BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. *Function Secret Sharing: Improvements and Extensions*. In: *ACM CCS 2016: 23rd Conference on Computer and Communications Security*. Oct. 2016. DOI: 10.1145/2976749.2978429.

[BHLS25] Dan Boneh, Iftach Haitner, Yehuda Lindell, and Gil Segev. *Exponent-VRFs and Their Applications*. In: *Advances in Cryptology – EUROCRYPT 2025, Part VII*. May 2025. DOI: 10.1007/978-3-031-91098-2_8.

[BMRS21] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. *Mac'n'Cheese: Zero-Knowledge Proofs for Boolean and Arithmetic Circuits with Nested Disjunctions*. In: *Advances in Cryptology – CRYPTO 2021, Part IV*. Aug. 2021. DOI: 10.1007/978-3-030-84259-8_4.

[Bou+] Cyril Bouvier, Guilhem Castagnos, Dario Catalano, Quentin Combal, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. *TECLA: Two-party ECDSA from CLAss groups. Two-Party ECDSA from Linearly Homomorphic Encryption over Class Groups*. Preview Writeup submitted to NIST Multi-Party Threshold Cryptography Call.

[Boy+19a] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. *Efficient Two-Round OT Extension and Silent Non-Interactive Secure Computation*. In: *ACM CCS 2019: 26th Conference on Computer and Communications Security*. Nov. 2019. DOI: 10.1145/3319535.3354255.

[Boy+19b] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. *Efficient Pseudorandom Correlation Generators: Silent OT Extension and More*. In: *Advances in Cryptology – CRYPTO 2019, Part III*. Aug. 2019. DOI: 10.1007/978-3-030-26954-8_16.

[Boy+20a] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. *Correlated Pseudorandom Functions from Variable-Density LPN*. In: *61st Annual Symposium on Foundations of Computer Science*. Nov. 2020. DOI: 10.1109/FOCS46700.2020.00103.

[Boy+20b] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. *Correlated Pseudorandom Functions from Variable-Density LPN*. Cryptology ePrint Archive, Paper 2020/1417. 2020. URL: https://eprint.iacr.org/2020/1417.

[Boy+20c] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. *Efficient Pseudorandom Correlation Generators from Ring-LPN*. In: *Advances in Cryptology – CRYPTO 2020, Part II*. Aug. 2020. DOI: 10.1007/978-3-030-56880-1_14.

[Boy+22] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Nicolas Resch, and Peter Scholl. *Correlated Pseudorandomness from Expand-Accumulate Codes*. In: *Advances in Cryptology – CRYPTO 2022, Part II*. Aug. 2022. DOI: 10.1007/978-3-031-15979-4_21.

[BW13] Dan Boneh and Brent Waters. *Constrained Pseudorandom Functions and Their Applications*. In: *Advances in Cryptology – ASIACRYPT 2013, Part II*. Dec. 2013. DOI: 10.1007/978-3-642-42045-0_15.

[CD23] Geoffroy Couteau and Clément Ducros. *Pseudorandom Correlation Functions from Variable-Density LPN, Revisited*. In: *PKC 2023: 26th International Conference on Theory*

and Practice of Public Key Cryptography, Part II. May 2023. DOI: 10.1007/978-3-031-31371-4_8.

[CDS94] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. *Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols*. In: *Advances in Cryptology – CRYPTO'94*. Aug. 1994. DOI: 10.1007/3-540-48658-5_19.

[CGGMP20] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. *UC Non-Interactive, Proactive, Threshold ECDSA with Identifiable Aborts*. In: *ACM CCS 2020: 27th Conference on Computer and Communications Security*. Nov. 2020. DOI: 10.1145/3372297.3423367.

[CHHK25] Geoffroy Couteau, Carmit Hazay, Aditya Hegde, and Naman Kumar. *Breaking the $1/\lambda$-Rate Barrier for Arithmetic Garbling*. In: *Advances in Cryptology – EUROCRYPT 2025, Part VI*. May 2025. DOI: 10.1007/978-3-031-91095-1_7.

[CL04] Jan Camenisch and Anna Lysyanskaya. *Signature Schemes and Anonymous Credentials from Bilinear Maps*. In: *Advances in Cryptology – CRYPTO 2004*. Aug. 2004. DOI: 10.1007/978-3-540-28628-8_4.

[CRR21] Geoffroy Couteau, Peter Rindal, and Srinivasan Raghuraman. *Silver: Silent VOLE and Oblivious Transfer from Hardness of Decoding Structured LDPC Codes*. In: *Advances in Cryptology – CRYPTO 2021, Part III*. Aug. 2021. DOI: 10.1007/978-3-030-84252-9_17.

[Dam+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. *Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits*. In: *ESORICS 2013: 18th European Symposium on Research in Computer Security*. Sept. 2013. DOI: 10.1007/978-3-642-40203-6_1.

[DD13] Vasily Dolmatov and Alexey Degtyarev. *Gost r 34.10-2012: Digital signature algorithm*. Tech. rep. 2013.

[DKLs18] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. *Secure Two-party Threshold ECDSA from ECDSA Assumptions*. In: *2018 IEEE Symposium on Security and Privacy*. May 2018. DOI: 10.1109/SP.2018.00036.

[DKLs19] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. *Threshold ECDSA from ECDSA Assumptions: The Multiparty Case*. In: *2019 IEEE Symposium on Security and Privacy*. May 2019. DOI: 10.1109/SP.2019.00024.

[DKLs24] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. *Threshold ECDSA in Three Rounds*. In: *2024 IEEE Symposium on Security and Privacy*. May 2024. DOI: 10.1109/SP54263.2024.00178.

[DNP25] Hila Dahari-Garbian, Ariel Nof, and Luke Parker. *Trout: Two-Round Threshold ECDSA from Class Groups*. In: *ACM CCS 2025: 32nd Conference on Computer and Communications Security*. Oct. 2025. DOI: 10.1145/3719027.3765192.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. *Multiparty Computation from Somewhat Homomorphic Encryption*. In: *Advances in Cryptology – CRYPTO 2012*. Aug. 2012. DOI: 10.1007/978-3-642-32009-5_38.

[Ds17] Jack Doerner and abhi shelat. *Scaling ORAM for Secure Computation*. In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Oct. 2017. DOI: 10.1145/3133956.3133967.

[Duc24] Clément Ducros. *Multiparty Computation from the Hardness of Coding Theory*. PhD thesis. Université Paris Cité, 2024.

[Fen+25] Qi Feng, Kang Yang, Kaiyi Zhang, Xiao Wang, Yu Yu, and Xiang Xie. *Stateless Deterministic Multi-party EdDSA Signatures with Low Communication*. In: *PKC 2025: 28th International Conference on Theory and Practice of Public Key Cryptography, Part V*. May 2025. DOI: 10.1007/978-3-031-91832-2_9.

[FHKRS25] Sebastian Faust, Carmit Hazay, David Kretzler, Leandro Rometsch, and Benjamin Schlosser. *Non-interactive Threshold BBS+ from Pseudorandom Correlations*. In: *Topics in Cryptology – CT-RSA 2025*. Apr. 2025. DOI: 10.1007/978-3-031-88661-4_9.

[FS87] Amos Fiat and Adi Shamir. *How to Prove Yourself: Practical Solutions to Identification and Signature Problems*. In: *Advances in Cryptology – CRYPTO'86*. Aug. 1987. DOI: 10.1007/3-540-47721-7_12.

[GGM84] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. *How to Construct Random Functions (Extended Abstract)*. In: *25th Annual Symposium on Foundations of Computer Science*. Oct. 1984. DOI: 10.1109/SFCS.1984.715949.

[GI14] Niv Gilboa and Yuval Ishai. *Distributed Point Functions and Their Applications*. In: *Advances in Cryptology – EUROCRYPT 2014*. May 2014. DOI: 10.1007/978-3-642-55220-5_35.

[GIKM98] Yael Gertner, Yuval Ishai, Eyal Kushilevitz, and Tal Malkin. *Protecting Data Privacy in Private Information Retrieval Schemes*. In: *30th Annual ACM Symposium on Theory of Computing*. May 1998. DOI: 10.1145/276698.276723.

[Gil99] Niv Gilboa. *Two Party RSA Key Generation*. In: *Advances in Cryptology – CRYPTO'99*. Aug. 1999. DOI: 10.1007/3-540-48405-1_8.

[GKMN21] François Garillot, Yashvanth Kondi, Payman Mohassel, and Valeria Nikolaenko. *Threshold Schnorr with Stateless Deterministic Signing from Standard Assumptions*. In: *Advances in Cryptology – CRYPTO 2021, Part I*. Aug. 2021. DOI: 10.1007/978-3-030-84242-0_6.

[GLSY04] Rosario Gennaro, Darren Leigh, R. Sundaram, and William S. Yerazunis. *Batching Schnorr Identification Scheme with Applications to Privacy-Preserving Authorization and Low-Bandwidth Communication Devices*. In: *Advances in Cryptology – ASIACRYPT 2004*. Dec. 2004. DOI: 10.1007/978-3-540-30539-2_20.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. *A Digital Signature Scheme Secure Against Adaptive Chosen-message Attacks*. In: *SIAM Journal on Computing* 2 (Apr. 1988).

[GOR11] Vipul Goyal, Adam O'Neill, and Vanishree Rao. *Correlated-Input Secure Hash Functions*. In: *TCC 2011: 8th Theory of Cryptography Conference*. Mar. 2011. DOI: 10.1007/978-3-642-19571-6_12.

[GS22] Jens Groth and Victor Shoup. *On the Security of ECDSA with Additive Key Derivation and Presignatures*. In: *Advances in Cryptology – EUROCRYPT 2022, Part I*. May 2022. DOI: 10.1007/978-3-031-06944-4_13.

[Guo+26] Xiaojie Guo, Hanlin Liu, Zhicong Huang, Hongrui Cui, Wenhao Zhang, Cheng Hong, Xiao Wang, Kang Yang, and Yu Yu. *Dory: Streaming PCG with Small Memory*. In: *2026 IEEE Symposium on Security and Privacy (SP)*. May 2026. DOI: 10.1109/SP63933.2026.00079.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. *Extending Oblivious Transfers Efficiently*. In: *Advances in Cryptology – CRYPTO 2003*. Aug. 2003. DOI: 10.1007/978-3-540-45146-4_9.

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. *The elliptic curve digital signature algorithm (ECDSA)*. In: *International journal of information security* 1 (2001).

[KG20] Chelsea Komlo and Ian Goldberg. *FROST: Flexible Round-Optimized Schnorr Threshold Signatures*. In: *SAC 2020: 27th Annual International Workshop on Selected Areas in Cryptography*. Oct. 2020. DOI: 10.1007/978-3-030-81652-0_2.

[Kon26] Yashvanth Kondi. *Two-Party ECDSA Signing at Constant Communication Overhead*. In: *Advances in Cryptology – EUROCRYPT 2026, Part I*. May 2026. DOI: 10.1007/978-3-032-25291-3_12.

[KOR23] Yashvanth Kondi, Claudio Orlandi, and Lawrence Roy. *Two-Round Stateless Deterministic Two-Party Schnorr Signatures from Pseudorandom Correlation Functions*. In: *Advances in Cryptology – CRYPTO 2023, Part I*. Aug. 2023. DOI: 10.1007/978-3-031-38557-5_21.

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru. *Overdrive: Making SPDZ Great Again*. In: *Advances in Cryptology – EUROCRYPT 2018, Part III*. Apr. 2018. DOI: 10.1007/978-3-319-78372-7_6.

[KPRR25] Vladimir Kolesnikov, Stanislav Peceny, Srinivasan Raghuraman, and Peter Rindal. *Stationary Syndrome Decoding for Improved PCGs*. In: *Advances in Cryptology – CRYPTO 2025, Part I*. Aug. 2025. DOI: 10.1007/978-3-032-01855-7_10.

[KPTZ13] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. *Delegatable pseudorandom functions and applications*. In: *ACM CCS 2013: 20th Conference on Computer and Communications Security*. Nov. 2013. DOI: 10.1145/2508859.2516668.

[Lin17] Yehuda Lindell. *Fast Secure Two-Party ECDSA Signing*. In: *Advances in Cryptology – CRYPTO 2017, Part II*. Aug. 2017. DOI: 10.1007/978-3-319-63715-0_21.

[LWYY24] Hanlin Liu, Xiao Wang, Kang Yang, and Yu Yu. *The Hardness of LPN over Any Integer Ring and Field for PCG Applications*. In: *Advances in Cryptology – EUROCRYPT 2024, Part VI*. May 2024. DOI: 10.1007/978-3-031-58751-1_6.

[Mak22] Nikolaos Makriyannis. *On the Classic Protocol for MPC Schnorr Signatures*. Cryptology ePrint Archive, Report 2022/1332. 2022. URL: https://eprint.iacr.org/2022/1332.

[Mor+25] Hiraku Morita, Erik Pohle, Kunihiko Sadakane, Peter Scholl, Kazunari Tozawa, and Daniel Tschudi. *MAESTRO: Multi-Party AES Using Lookup Tables*. In: *USENIX Security 2025: 34th USENIX Security Symposium*. Aug. 2025. URL: https://www.usenix.org/conference/usenixsecurity25/presentation/morita.

[MSY21] Jean-Pierre Münch, Thomas Schneider, and Hossein Yalame. *VASA: Vector AES Instructions for Security Applications*. Cryptology ePrint Archive, Report 2021/1493. 2021. URL: https://eprint.iacr.org/2021/1493.

[MYG24] Nikolaos Makriyannis, Oren Yomtov, and Arik Galansky. *Practical Key-Extraction Attacks in Leading MPC Wallets*. In: *ACM CCS 2024: 31st Conference on Computer and Communications Security*. Oct. 2024. DOI: 10.1145/3658644.3670359.

[Nat19] National Institute of Standards and Technology. *Digital Signature Standard (DSS)*. Federal Information Processing Standards Publication 186-5 (Draft). National Institute of Standards and Technology, 2019. DOI: 10.6028/NIST.FIPS.186-5-draft. URL: https://doi.org/10.6028/NIST.FIPS.186-5-draft.

[NR95] Moni Naor and Omer Reingold. *Synthesizers and Their Application to the Parallel Construction of Pseudo-Random Functions*. In: *36th Annual Symposium on Foundations of Computer Science*. Oct. 1995. DOI: 10.1109/SFCS.1995.492474.

[NRSW20] Jonas Nick, Tim Ruffing, Yannick Seurin, and Pieter Wuille. *MuSig-DN: Schnorr Multi-Signatures with Verifiably Deterministic Nonces*. In: *ACM CCS 2020: 27th Conference on Computer and Communications Security*. Nov. 2020. DOI: 10.1145/3372297.3417236.

[OSY21] Claudio Orlandi, Peter Scholl, and Sophia Yakoubov. *The Rise of Paillier: Homomorphic Secret Sharing and Public-Key Silent OT*. In: *Advances in Cryptology – EUROCRYPT 2021, Part I*. Oct. 2021. DOI: 10.1007/978-3-030-77870-5_24.

[PS16] David Pointcheval and Olivier Sanders. *Short Randomizable Signatures*. In: *Topics in Cryptology – CT-RSA 2016*. Feb. 2016. DOI: 10.1007/978-3-319-29485-8_7.

[Roy22] Lawrence Roy. *SoftSpokenOT: Quieter OT Extension from Small-Field Silent VOLE in the Minicrypt Model*. In: *Advances in Cryptology – CRYPTO 2022, Part I*. Aug. 2022. DOI: 10.1007/978-3-031-15802-5_23.

[RRT23] Srinivasan Raghuraman, Peter Rindal, and Titouan Tanguy. *Expand-Convolute Codes for Pseudorandom Correlation Generators from LPN*. In: *Advances in Cryptology – CRYPTO 2023, Part IV*. Aug. 2023. DOI: 10.1007/978-3-031-38551-3_19.

[Sch90] Claus-Peter Schnorr. *Efficient Identification and Signatures for Smart Cards*. In: *Advances in Cryptology – CRYPTO'89*. Aug. 1990. DOI: 10.1007/0-387-34805-0_22.

[Sch91] Claus-Peter Schnorr. *Efficient Signature Generation by Smart Cards*. In: *Journal of Cryptology* 3 (Jan. 1991). DOI: 10.1007/BF00196725.

[SGRR19] Phillipp Schoppmann, Adrià Gascón, Leonie Reichert, and Mariana Raykova. *Distributed Vector-OLE: Improved Constructions and Implementation*. In: *ACM CCS 2019: 26th Conference on Computer and Communications Security*. Nov. 2019. DOI: 10.1145/3319535.3363228.

[Sho97] Victor Shoup. *Lower Bounds for Discrete Logarithms and Related Problems*. In: *Advances in Cryptology – EUROCRYPT'97*. May 1997. DOI: 10.1007/3-540-69053-0_18.

[SW14] Amit Sahai and Brent Waters. *How to use indistinguishability obfuscation: deniable encryption, and more*. In: *46th Annual ACM Symposium on Theory of Computing*. May 2014. DOI: 10.1145/2591796.2591825.

[WRK17] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. *Authenticated Garbling and Efficient Maliciously Secure Two-Party Computation*. In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Oct. 2017. DOI: 10.1145/3133956.3134053.

[WYKW21] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. *Wolverine: Fast, Scalable, and Communication-Efficient Zero-Knowledge Proofs for Boolean and Arithmetic Circuits*. In: *2021 IEEE Symposium on Security and Privacy*. May 2021. DOI: 10.1109/SP40001.2021.00056.

[Xue+23] Haiyang Xue, Man Ho Au, Mengling Liu, Kwan Yin Chan, Handong Cui, Xiang Xie, Tsz Hon Yuen, and Chengru Zhang. *Efficient Multiplicative-to-Additive Function from Joye-Libert Cryptosystem and Its Application to Threshold ECDSA*. In: *ACM CCS 2023: 30th Conference on Computer and Communications Security*. Nov. 2023. DOI: 10.1145/3576915.3616595.

[YSWW21] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. *QuickSilver: Efficient and Affordable Zero-Knowledge Proofs for Circuits and Polynomials over Any Field*. In: *ACM CCS 2021: 28th Conference on Computer and Communications Security*. Nov. 2021. DOI: 10.1145/3460120.3484556.

[ZLLZH24] C Zhang, Y Liu, F Leng, Q Zhao, and Z He. *RFC 9563: SM2 Digital Signature Algorithm for DNSSEC*. 2024.