Verifiable Oblivious Pseudorandom Functions from Lattices: Practical-ish and Thresholdisable[⋆]

Martin R. Albrecht[1] and Kamil Doruk Gur[2⋆⋆]

[1] King's College London and SandboxAQ
`martin.albrecht@{kcl.ac.uk,sandboxaq.com}`
[2] Department of Computer Science, University of Maryland
`dgur1@cs.umd.edu`

Abstract. We revisit the lattice-based verifiable oblivious PRF construction from PKC'21 and remove or mitigate its central three sources of inefficiency. First, applying Rényi divergence arguments, we eliminate one superpolynomial factor from the ciphertext modulus q, allowing us to reduce the overall bandwidth consumed by RLWE samples by about a factor of four. This necessitates us introducing intermediate unpredictability notions to argue PRF security of the final output in the Random Oracle model. Second, we remove the reliance on the $\mathsf{1D\text{-}SIS}$ assumption, which reduces another superpolynomial factor, albeit to a factor that is still superpolynomial. Third, by applying the state-of-the-art in zero-knowledge proofs for lattice statements, we achieve a reduction in bandwidth of several orders of magnitude for this material. Finally, we give a t-out-of-n threshold variant of the VOPRF for constant t and with trusted setup, based on a n-out-of-n distributed variant of the VOPRF (and without trusted setup).

1 Introduction

An oblivious pseudorandom function (OPRF) is a two-party protocol between a client and a server allowing the client to derive a pseudorandom output based on their input. In particular, an OPRF allows a client to receive a pseudorandom function (PRF) evaluation on an input x from a server with key k. The security of the protocol then refers to (i) the server not learning anything about the input x and (ii) the client not learning anything besides the PRF evaluation of x under k. An OPRF is additionally *verifiable* if the client is guaranteed that the output received is indeed evaluated under a committed key. (V)OPRFs recently gained considerable popularity with important applications including but not limited to secure keyword search [FIPR05], private set intersection [JL09], secure data de-duplication [KBR13], password-protected secret sharing [JKK14,JKKX16], and more popularly password-authenticated key exchange(PAKE) [JKX18] and private lightweight authentication mechanisms [DGS+18]. A systematisation of knowledge of OPRFs is given in [CHL22].

[⋆] This is the full version of [AG24] which appeared in Asiacrypt 2024.
[⋆⋆] Work partially done while working for SandboxAQ.

Unfortunately, while VOPRFs have useful practical applications and an abundant number of constructions, these are insecure in the post-quantum setting, i.e. in the presence of a quantum adversary, since they rely on classical assumptions. Hence, it is required to design VOPRFs relying on plausibly quantum-resistant assumptions so that the world of functionalities afforded by VOPRFs can also be realised in this new era. Yet, like many other functionalities in a post-quantum setting, post-quantum secure VOPRFs are scarce. We give an overview of the current state of the art in Table 1, extending a table from [ADDG24].

Table 1: Post-quantum (V)OPRF candidates in the literature

work	assumption	r	communication	model
		plain		
[ADDS21]	R(LWE), SIS	2	$\approx$ 2MB	semi-honest, QROM
[SHB21]	Legendre PRF	3	$\approx \lambda \cdot$ 13K	semi-honest, *pp*, ROM
[BKW20]	CSIDH	3	424KB	malicious client
[HMR23]	CSIDH	2	21KB	semi-honest, *ts*
[HMR23]	CSIDH	4	35KB	malicious client, *ts*
[HMR23]	CSIDH	258	25KB	semi-honest
[DGH$^+$21]	[BIP$^+$18]	2	80B	semi-honest, *pp*
[FOO23]	AES	?	4746KB	malicious client, *pp*
[ADDG24]	lattices, [BIP$^+$18]	2	2.5MB + 10KB	malicious client, ROM
[ADDG24]	lattices, [BIP$^+$18]	2	15MB + 5.3KB	malicious client, ROM
[APRR24]	[APRR24]	2	4.8B + 114.5B	semi-honest, *pp*, wPRF
		verifiable		
[ADDS21]	R(LWE), SIS	2	> 128GB	malicious, QROM
[ADDG24]	lattices, [BIP$^+$18]	2	2.8MB + 110KB	malicious, ROM
[ADDG24]	lattices, [BIP$^+$18]	2	15MB + 60KB	malicious, ROM
[BDFH24]	Legendre PRF	9	911KB	malicious, ROM
[Bas24]	[Bas24]	2	28.9kB	malicious, ROM
this work, $Q = 2^{16}$	R(LWE), SIS	2	108.3kB + 188.6kB	malicious, ROM
this work, $Q = 2^{64}$	R(LWE), SIS	2	221.5kB + 315.9kB	malicious, ROM

The column "r" gives the number of rounds. ROM is the random oracle model, QROM the quantum random oracle model, "pp" stands for "preprocessing", "ts" for "trusted setup", "wPRF" for PRFs that accept random inputs. When bandwidth is reported as a sum, this is for a one-time offline cost and online costs per query respectively, where online costs are amortised over 64 queries for [ADDG24]. See Table 2 for details on our parameters.

In particular, VOPRFs based on lattices have only a limited number of constructions. The first known round-optimal post-quantum VOPRF in [ADDS21] is more of a feasibility result rather than a practical proposal due to the required zero-knowledge proofs causing the communication to be in GBs. Even ignoring this cost, bandwidth per query was estimated at about 2MB in the semi-honest setting. More recently [ADDG24] adapted the "Crypto Dark Matter" PRF [BIP$^+$18] to the lattice setting using fully-homomorphic encryption [CGGI20]. The resulting construction achieves practical sizes but is slow to evaluate for the server, and rather complex to implement by relying on the full machinery of fully homomor-

Table 2: Example parameters

$\log Q$	λ	$(d, \log q, \log \sigma')$	$\lvert\mathbf{c}\rvert$	$\lvert\mathbf{c}_x\rvert$	$\lvert\mathbf{d}_x\rvert$	size
4	100	(4096, 137, 35)	(68.5, 36.1)	(68.5, 73.2)	(1.8, 38.8)	(104.6, 182.3)
16	95	(4096, 143, 41)	(71.5, 36.8)	(71.5, 75.5)	(1.8, 39.8)	(108.3, 188.6)
32	90	(4096, 151, 49)	(75.5, 38.0)	(75.5, 79.4)	(1.8, 41.0)	(113.5, 197.7)
64	167	(8192, 169, 67)	(169.0, 52.5)	(169.0, 88.6)	(1.8, 56.5)	(221.5, 315.9)

Q is the number of queries supported, λ the dRLWE security level, d, q are dRLWE dimension and modulus and σ' the size of the drowning noise. All sizes are in kB, (x, y) means size of value and proof except in the the last column which gives the offline and online sizes. We always target a correctness error of 2^{-100}.

phic encryption. Moreover, in addition to the PRF assumption in [BIP+18], the construction relies on a heuristic argument for verifiability.

For the threshold setting, the concurrent work of [KCM24] is most similar to ours. There, the authors propose four different distributed OPRFs based on the Legendre PRF in a setting where client-server communication is round optimal but servers communicate between client evaluations. Our construction does not have this requirement after the initial setup. Similar to our construction, the construction in [KCM24] is only efficient for small (t, n) pairs. Unlike our constructions, however, [KCM24] requires n servers to be available for evaluation for different settings whereas we only require t for security with aborts. The constructions also have weaker security guarantees compared to ours. Out of the four, only the last constructions promises security against malicious servers with a dishonest majority, which relies on the security guarantees of the underlying MPC operations. This causes issues with some security assumptions, as neither client privacy nor verifiability is provided against $\geq t$ or n corrupted servers respectively. Our construction provides these guarantees also when n servers are corrupted. On other hand, [KCM24] achieves low bandwidth between clients and servers of roughly $n \cdot \lambda^2$ bits, which is much smaller than our construction.

1.1 Technical Overview

To present our contributions, we begin with a high-level overview of the construction from [ADDS21] and highlight its main bottlenecks. The VOPRF construction is based on the ring instantiation of the PRF by Banerjee and Peikert [BP14]

$$F_k(x) = \left\lfloor \frac{p}{q} \cdot \mathbf{a}^F(x) \cdot k \right\rceil \tag{1}$$

where $k \in \mathcal{R}_q$ is the key with small coefficients represented in $\{-q/2, \ldots, q/2\}$ and $\mathbf{a}^F(x)$ is essentially a hash function processing the client input x. Security of the construction can be reduced to the hardness of RLWE. The construction in [ADDS21] instantiates this framework with uniformly random public vectors $\mathbf{a}_0, \mathbf{a}_1 \in \mathcal{R}_q^{1\times\ell}$ and a bit decomposition function G^{-1}. Given a public $\mathbf{a} \in \mathcal{R}_q^{1\times\ell}$ the high-level protocol is then:

1. The server publishes a commitment $\mathbf{c} := \mathbf{a} \cdot k + \mathbf{e}$ to a small key $k \in \mathcal{R}$.

2. For input x, the client chooses a small $s \in \mathcal{R}$ and $\mathbf{e}_{\mathbb{C}} \in \mathcal{R}^{1\times\ell}$, and computes $\mathbf{c}_x \coloneqq \mathbf{a} \cdot s + \mathbf{e}_{\mathbb{C}} + \mathbf{a}^F(x) \bmod q$.
3. Using k, the server sends $\mathbf{d}_x \coloneqq \mathbf{c}_x \cdot k + \mathbf{e}_{\mathbb{S}} \bmod q$ for small $\mathbf{e}_{\mathbb{S}} \in \mathcal{R}^{1\times\ell}$.
4. The client finally outputs $\mathbf{y} = \left\lfloor \frac{p}{q} \cdot (\mathbf{d}_x - \mathbf{c} \cdot s) \right\rceil$.

Since $\mathbf{d}_x = \mathbf{a}\cdot s\cdot k + \mathbf{a}^F(x)\cdot k + \mathbf{e}_{\mathbb{C}}\cdot k + \mathbf{e}_{\mathbb{S}}$, if $\mathbf{e}_{\mathbb{S}}$ is chosen from a distribution that hides the presence of additive terms $\mathbf{e}_{\mathbb{C}}\cdot k$, $\mathbf{e}\cdot s$ and the absence of the additive term $\mathbf{e}_x$ (which follow some narrow distribution $\mathcal{E}_{\mathbf{a}_0,\mathbf{a}_1,x,\sigma}$) then it is indistinguishable from $\mathbf{d}'_x = (\mathbf{a}\cdot k + \mathbf{e})\cdot s + \mathbf{e}_{\mathbb{S}} + (\mathbf{a}^F(x)\cdot k + \mathbf{e}_x) = \mathbf{c}\cdot s + (\mathbf{a}^F(x)\cdot k + \mathbf{e}_x) + \mathbf{e}_{\mathbb{S}}$. Then if $\mathbf{e}_x$ is chosen from a proper distribution [BP14], $\mathbf{a}^F(x)\cdot k + \mathbf{e}_x$ and consequently $\mathbf{d}_x$ leaks nothing about k by the RLWE assumption. Similarly, if s chosen from a proper RLWE secret distribution and $\mathbf{e}$ is from a discrete Gaussian, the client message $\mathbf{c}_x = \mathbf{a}\cdot s + \mathbf{e} + \mathbf{a}^F(x)$ is also indistinguishable from uniform by RLWE.

Correctness is satisfied with high probability regardless of the choice of k by the one-dimensional short integer solution ($\mathsf{1D\text{-}SIS}$) assumption [BV15]. Verifiability is then achieved with the help of non-interactive zero-knowledge arguments of knowledge showing $\mathbf{c}$,$\mathbf{c}_x$, and $\mathbf{d}_x$ are computed correctly.

The above construction is intuitive in following well-established pre-quantum Diffie-Hellmann blueprints. Moreover, its simple algebraic nature (and instantiation in the standard model, except potentially for zero-knowledge proofs) allows for extensions such as threshold variants.

However, the concrete instantiation is highly inefficient due to three reasons.

First, the correctness of the PRF adds a superpolynomial factor to the modulus q to ensure correct rounding which in the end results in large parameters. Indeed, to thwart adversaries that maliciously sample k such that $\mathbf{a}^F(x)\cdot k$ produces a rounding error for a target value x, [ADDS21] relies on the $\mathsf{1D\text{-}SIS}$ assumption as just mentioned. This assumption requires $q \gg 2^{2\lambda}$, i.e. more than what we would naively expect to have correct rounding with overwhelming probability.[3]

Second, to hide the additive terms $\mathbf{e}_{\mathbb{C}}\cdot k$, $\mathbf{e}\cdot s$ and $\mathbf{e}_x$, the $\mathbf{e}_{\mathbb{S}}$ has to have superpolynomial size in the norm of these terms. This allows for an argument based on statistical distance to go through.

Third, the NIZKAoKs required for verifiability and to protect against malicious clients add further overheads as these relations require non-trivial statements. In particular, the proof that $\mathbf{c}_x$ is correctly computed has to show $\mathbf{c}_x$ indeed contains $\mathbf{a}^F(x)$ without revealing the secrets x, s, or $\mathbf{e}_{\mathbb{C}}$. Since $\mathbf{a}^F(x)$ is highly irregular with calls to bit decompositions and two different public vectors, [ADDS21] used the NIZKAoK construction from [YAZ+19] which proves *rank-1 constraints* (R1CS) over $\mathbb{Z}_q$, breaking the native structure of the protocol. Combined with large parameters this causes bandwidth in the GBs.

[3] Concretely, [ADDS21] picks $\log(q) \approx 256$ and a ring dimension of 2^{14} for the semi-honest setting, i.e. without considering maliciously chosen k. This leads to a communication cost of 2MB already; relying on the $\mathsf{1D\text{-}SIS}$ assumption would require $\log(q) \approx 2048$ based on SIS estimates provided by the lattice estimator [APS15].

1.2 Contributions

In this work, we resolve or reduce the above-mentioned sources of inefficiency.

First, we avoid relying on the $\mathsf{1D\text{-}SIS}$ assumption, by borrowing a trick from the non-interactive key exchange in [GdKQ+23]. Instead of defining the PRF output as $\lfloor \frac{p}{q} \cdot (\mathbf{a}^F(x) \cdot k) \rceil$, we define it as $\lfloor \frac{p}{q} \cdot (\mathbf{a}^F(x) \cdot k + \mathbf{r}) \rceil$ where $\mathbf{r}$ is the output of some Random Oracle called on x and $\mathbf{c}$: $\mathbf{r} \coloneqq \mathsf{H}_{\mathbf{r}}(x, \mathbf{c})$. In the Random Oracle model, $\mathbf{r}$ is independent of k and thus $\lfloor \frac{p}{q} \cdot (\mathbf{a}^F(x) \cdot k + \mathbf{r} + \mathbf{e}_{\mathbb{C}} \cdot k + \mathbf{e}_{\mathbb{S}}) \rceil$ will round to the correct value $\lfloor \frac{p}{q} \cdot (\mathbf{a}^F(x) \cdot k + \mathbf{r}) \rceil$ with a probability to $\approx 1 - \|\mathbf{e}_{\mathbb{C}} \cdot k + \mathbf{e}_{\mathbb{S}}\|_\infty / (q/p)$. This still requires a superpolynomial gap between q and $\|\mathbf{e}_{\mathbb{C}} \cdot k + \mathbf{e}_{\mathbb{S}}\|_\infty$ but this gap is comparable to that in the semi-honest setting of [ADDS21].

Second, we change the way how we analyse $\mathbf{e}_{\mathbb{S}}$ and remove the superpolynomial dependency on the norm of additive terms. To achieve this, we use a Rényi divergence based approach instead of the statistical distance. However, for this, we have to replace the simulation-based security in the standard model in [ADDS21] with a game-based notion in the Random Oracle model. In more detail, except in rather particular circumstances, we cannot apply Rényi divergence arguments to decision problems [BLR+18]. To work around this, we first show that our construction based on [ADDS21] achieves the notion of unpredictability, which we then upgrade to PRF security. Overall, this leads to a bandwidth improvement of roughly an order of magnitude when compared with the semi-honest parameters of [ADDS21] (and without NIZKAoK).

Third, we replace the NIZKAoK [YAZ+19] with that from [LNP22] compressed with LaBRADOR [BS23] and also work in larger rings $\mathcal{R}_q$ with lattice statements. This improves bandwidth by several orders of magnitude.

Overall, we obtain the sizes reported in Table 2. Compared with [ADDS21], our work allows for practical-*ish* parameters. Compared with [ADDG24], our bandwidth requirements are smaller if few evaluations are required. In terms of computational burden, we note that [ADDG24] has an expensive computation on the server side (TFHE bootstrapping) whereas we have an expensive computation on the client side (proving well-formedness with a complex statement).

Finally, we extend the functionality of the VOPRF and build multiparty protocols. We use n-out-of-n and t-out-of-n *threshold VOPRFs* which consist of n servers jointly evaluating the input x and n (respectively t) servers are required to generate the output. The n-out-of-n construction is immediate from the key-homomorphic properties of the VOPRF. To achieve the more interesting t-out-of-n setting, we exploit that in the VOPRF setting, we expect t to be quite small, i.e. constant. Moreover, we assume a trusted setup. While this is a significant limitation of this work, we think this assumption is justified in the VOPRF setting, where one entity may aim to avoid single points of failure, rather than multiple parties coming together to, say, validate some statement, i.e. the threshold signature setting. In our approach, we essentially output $\binom{n}{t}$ copies of the n-out-of-n setting. We use rejection sampling to enforce that these are all well-distributed. To achieve verifiability in the t-out-of-n case we rely on an additional cut-and-choose type argument to be able to use weaker NIZKAoKs.

2 Preliminaries

For integers a and b where $a < b$ we use $[a, b]$ to represent the set $\{a, a+1, \ldots, b-1, b\}$. If $a = 0$ and $b = n-1$ we use the notation $[n]$ instead. For a vector b we use $b[i]$ as the indexing operator. We denote the output of probabilistic algorithms with $\leftarrow$ and deterministic ones with $\coloneqq$. Similarly for a distribution $\mathcal{D}$ or a bounded set S if an element x is sampled according to distribution $\mathcal{D}$ or uniformly random from S we denote it as $x \leftarrow \mathcal{D}$ and $x \leftarrow S$ respectively. For two distributions, we use $\approx_c$ to denote they are computationally indistinguishable. A PPT algorithm is a probabilistic algorithm with running time polynomial in the security parameter λ. We say a function is negligible in λ if $\lambda^{-\omega(1)}$ and write $r_1 \gg r_2$ as short-hand for $r_1 \geq \lambda^{\omega(1)} \cdot r_2$. We denote the ℓ_x norm of a vector with $\|\cdot\|_x$. If $x = 2$ and clear from the context, we omit the subscript. A distribution $\mathcal{D}$ is B bounded if $\Pr[\|\mathbf{x}\| \geq B \colon \mathbf{x} \leftarrow \mathcal{D}] < \delta$ for a negligible δ. We also consider the rounding operation $\lfloor \cdot \rceil$ to the nearest integer (rounding down if there is a tie) and $\lfloor x \rceil_{q'} \coloneqq \lfloor q'/q \cdot x \rceil$ from $\mathbb{Z}_q$ to $\mathbb{Z}_{q'}$ for $q' < q$ and $x \in \mathbb{Z}_q$. We use lowercase letters to denote ring elements and boldface lowercase letters to denote vectors.

We use power of two cyclotomic rings in this work. For a modulus $q \in \mathbb{Z}$, we consider the polynomial ring $\mathcal{R} = \mathbb{Z}[X]/\langle X^N + 1\rangle$ and $\mathcal{R}_q \coloneqq \mathcal{R}/q\mathcal{R}$ for a power-of-two N. The set $\mathcal{R}_{\leq c}$ is then the set of all elements of $\mathcal{R}$ with coefficients that have an absolute value of at most c. Norms of ring elements are defined over the coefficient vectors of the said elements and norms of vectors of ring elements are norms of the concantenation of the coefficient vectors.

Define $\mathbf{G} : \mathcal{R}_q^{\ell\times\ell} \to \mathcal{R}_q^{1\times\ell}$ to be the linear operation corresponding to left multiplication by $(1, 2, \ldots, 2^{\ell-1})$. Further, define $G^{-1} : \mathcal{R}_q^{1\times\ell} \to \mathcal{R}_q^{\ell\times\ell}$ to be the bit decomposition operation that essentially inverts $\mathbf{G}$ i.e. the i^{th} column of $G^{-1}(\mathbf{a})$ is the bit decomposition of $a_i \in \mathcal{R}_q$ into binary polynomials.

For $\mathbf{a}_0, \mathbf{a}_1 \in \mathcal{R}_q^{1\times\ell}$, $x \in \{0,1\}^L$, and $i \in [L]$ define

$$\mathbf{a}_{x\setminus i} \coloneqq G^{-1}\left(\mathbf{a}_{x_{i+1}} \cdot G^{-1}\left(\mathbf{a}_{x_{i+2}} \cdot G^{-1}\left(\cdots\left(\mathbf{a}_{x_{L-2}} \cdot G^{-1}\left(\mathbf{a}_{x_{L-1}}\right)\right)\cdots\right)\right)\right) \in \mathcal{R}_q^{\ell\times\ell}.$$

Now, for a client input $x \in \{0,1\}^L$, let $\mathcal{E}_{\mathbf{a}_0,\mathbf{a}_1,x,\sigma}$ be the distribution of all $\mathbf{e}_x$ computed as $\mathbf{e} = \sum_{i=0}^{L-2} \mathbf{e}_i \cdot \mathbf{a}_{x\setminus i} + \mathbf{e}_{L-1}$ where $\forall i \in [L] \colon \mathbf{e}_i \leftarrow \mathcal{R}_{\chi_\sigma}^{1\times\ell}$.

2.1 Discrete Gaussian Distributions over Polynomial Rings

The discrete Gaussian distribution over $\mathcal{R}$ is defined as follows:

Definition 1. *For $\mathbf{x} \in \mathcal{R}^m$ let $\rho_{\mathbf{v},s}(\mathbf{x}) = \exp(-\pi\|\mathbf{x}-\mathbf{v}\|_2^2/s^2)$ be the Gaussian function of parameters $\mathbf{v} \in \mathcal{R}^m$ and $s \in \mathbb{R}$. Then the discrete Gaussian distribution $\mathcal{D}_{\mathbf{v},s}^m$ centered at $\mathbf{v}$ is*

$$\mathcal{D}_{\mathbf{v},s}^m = \rho_{\mathbf{v},s}(x)/\rho_{\mathbf{v},s}(\mathcal{R}^m) \text{ where } \rho_{\mathbf{v},s}(\mathcal{R}^m) = \sum_{\mathbf{x}\in\mathcal{R}^m} \rho_{\mathbf{v},s}(x).$$

When there is only a single element in a vector, we omit the superscript and if $\mathbf{v}$ is the zero-vector, we omit the subscript $\mathbf{v}$. When s exceeds the *smoothing* *parameter* $\eta_\varepsilon(\mathcal{R}^m) \leq \omega(\sqrt{\log(mN)})$, $\mathcal{D}^m_s$ behaves like a continuous Gaussian of standard deviation of $\sigma = s/2\pi$. The following lemmas will be useful when we discuss our key generation algorithm for our threshold construction where we need to argue about the distribution of key shares based on properties of Gaussians.

Lemma 1. *[MP13, Theorem 3.3]* Let s be a parameter exceeding the smoothing parameter by a factor of at least $\sqrt{2}$ and $\mathbf{x}_i$ for $i \in [n]$ be independent samples from $\mathcal{D}^m_s$. Then the distribution of $\mathbf{x} \coloneqq \sum_i \mathbf{x}_i$ is statistically close to $\mathcal{D}^m_{s\sqrt{n}}$.

For our threshold construction, we rely on *rejection sampling* [Lyu12] to guarantee each partial key adheres to a particular distribution. The following lemma shows for a vector of ring elements $\mathbf{x}$ sampled from a Gaussian, the ℓ_2 norm is bounded for all but negligible probability:

Lemma 2. *[Lyu12, Lemma 4.4 adapted]* For any $\gamma > 1$, we have

$$\Pr\left[\|\mathbf{x}\|_2 > \gamma \cdot \sigma \cdot \sqrt{m \cdot N} \colon \mathbf{x} \leftarrow \mathcal{D}^m_s\right] < \gamma^{m\,N} \cdot e^{m\,N \cdot (1-\gamma^2)/2}.$$

To be able to argue that our key shares in the t-out-of-n setting are no different compared with a key sampled from an independent Gaussian distribution, we have the following lemma that allows us to decide on a σ and the expected number of repetitions M in rejection sampling:

Lemma 3. *[Lyu12, Lemma 4.5 adapted]* For a $V \subseteq \mathcal{R}^m$, let $T = \max_{\mathbf{v} \in V} \|\mathbf{v}\|_2$. For a fixed t with $t = \omega(\sqrt{\log(mN)})$ and $t = o(\log(mN))$ if $\sigma = \alpha \cdot T$ for any positive α then:

$$\Pr\left[M \geq \mathcal{D}^m_s(\mathbf{x})/\mathcal{D}^m_{\mathbf{v},s}(\mathbf{x}) \colon \mathbf{x} \leftarrow \mathcal{D}^m_s\right] \geq 1 - \epsilon$$

where $M = e^{t/\alpha + 1/(2(\alpha^2))}$ and $\epsilon = 2e^{-t^2/2}$.

For practical set of parameters, M grows slowly which is useful for arguing that rejection sampling does not need too many trials to "clean up" a small centre $\mathbf{v}$. For the remainder of the work we use $\mathcal{R}_{\chi_\sigma}$ to denote distribution of elements in $\mathcal{R}_q$ which have coefficients distributed according to the a discrete Gaussian with parameter χ_σ.

2.2 Rényi Divergence

For any two discrete probability distributions ϕ and ϕ' such that $\mathsf{Supp}(\phi) \subseteq \mathsf{Supp}(\phi')$ and an $\alpha \in (1, +\infty)$. The Rényi divergence of order α can be defined as:

$$R_\alpha(\phi \| \phi') \coloneqq \left(\sum_{x \in \mathsf{Supp}(\phi')} \frac{\phi(x)^\alpha}{\phi'(x)^{\alpha-1}} \right)^{\frac{1}{\alpha-1}}.$$

The Rényi divergence R_α has the following properties for probability distributions ϕ, ϕ', ϕ'' with $\mathsf{Supp}(\phi) \subseteq \mathsf{Supp}(\phi') \subseteq \mathsf{Supp}(\phi'')$:

– **Log. Positivity:** $R_\alpha(\phi \| \phi') \geq R_\alpha(\phi \| \phi) = 1$.
– **Data Processing Inequality:** $R_\alpha(\phi^f \| \phi'^f) \leq R_\alpha(\phi \| \phi')$ for any function f where ϕ^f denotes the distribution of $f(y)$ where $y \leftarrow \phi$.
– **Multiplicativity:** Assume ϕ and ϕ' are two distributions for a pair of random variables (Y_0, Y_1). For $i \in \{0,1\}$, let ϕ_i denote the marginal distribution of Y_i under ϕ, and let $\phi_{1|0}(\cdot|y_0)$ denote the conditional distribution of Y_1 given $Y_0 = y_0$. Then
  - $R_\alpha(\phi \| \phi') = R_\alpha(\phi_0 \| \phi'_0) \cdot R_\alpha(\phi_1 \| \phi'_1)$ if Y_0 and Y_1 are independent for α in given interval.
  - $R_\alpha(\phi \| \phi') \leq R_\infty(\phi_0 \| \phi'_0) \cdot \max_{y_0 \in X} R_\alpha(\phi_{1|0}(\cdot|y_0) \| \phi'_{1|0}(\cdot|y_0))$
– **Probability Preservation:** Let $E \subseteq \mathsf{Supp}(\phi')$ be an arbitrary event. For given interval of α, $\phi'(E) \geq \phi(E)^{\frac{\alpha}{\alpha-1}} / R_\alpha(\phi \| \phi')$. Furthermore

$$\phi'(E) \geq \phi(E)/R_\infty(\phi \| \phi').$$

Additionally, we rely on the following lemma to argue about the Rènyi divergence between Gaussians with different centers.

Lemma 4. *[LSS14]* Let P and Q be distributions corresponding to Gaussians $\mathcal{D}^m_{\mathbf{c},s}$ and $\mathcal{D}^m_{\mathbf{c}',s}$ with centers $\mathbf{c}$ and $\mathbf{c}'$, and $s \geq \eta(\mathcal{R}^m)$. Then for any $\alpha \in (1, +\infty)$:

$$R_\alpha(P||Q) \leq \exp\left(\alpha \cdot \pi \cdot \frac{\|\mathbf{c} - \mathbf{c}'\|_2^2}{s^2}\right)$$

Remark 1. Note that the Rényi divergence grows exponentially with m, since $\|\mathbf{c} - \mathbf{c}'\|_2^2$ grows linearly with m. Similarly, the Rényi divergence for z samples grows exponentially in z by the multiplicative property.

2.3 Hardness Assumptions

We rely on the standard decisional RLWE problem for the security of the VOPRF function.

Definition 2 ([SSTX09,LPR10]). *Let $\mathcal{R}_q, m, \sigma_s, \sigma_e > 0$ depend on security parameter λ for integers q, N and m and $\mathcal{R}_q := \mathbb{Z}_q[x]/(X^N + 1)$. The decision Ring Learning with Errors ($\mathsf{dRLWE}_{\mathcal{R}_q,m,\sigma_s,\sigma_e}$ for short) problem is to distinguish between*

$$(a_i, a_i \cdot s + e_i)_{i \in [m]} \in (\mathcal{R}_q)^2 \text{ and } (a_i, u_i)_{i \in [m]} \in (\mathcal{R}_q)^2$$

for $a_i, u_i \leftarrow \mathcal{R}_q$; $s, \leftarrow \mathcal{R}_{\chi_{\sigma_s}} e_i \leftarrow \mathcal{R}_{\chi_{\sigma_e}}$.

When the number of samples m is implicit, we omit the subscript.

Remark 2. We note the trivial hierarchy that $\mathsf{dRLWE}_{\mathcal{R}_q,m,\sigma_0,\sigma}$ is at least as hard as $\mathsf{dRLWE}_{\mathcal{R}_q,m,\sigma_1,\sigma}$ when $\sigma_0 = \sqrt{k} \cdot \sigma_1$ for any integer $k > 1$ and $\sigma_1 \geq \sqrt{2} \cdot \eta_\varepsilon(\mathcal{R})$. Given samples from the latter (a_i, b_i) submit $(a_i, b_i + a_i \cdot \delta)$ to the distinguisher for the former, where $\delta \leftarrow \mathcal{R}_{\chi_{\sqrt{k-1}\sigma_0}}$. By a simple corollary of Lemma 1, we have that $\delta + s$ is correctly distributed if $s \leftarrow \mathcal{R}_{\chi_{\sigma_0}}$. Since $\sum_{i=0}^{k-1} \chi_{\sigma_0} \approx_s \chi_{\sigma_1}$ and $\sum_{i=0}^{k-2} \chi_{\sigma_0} \approx_s \chi_{\sqrt{k-1} \cdot \sigma_0}$, we have $\chi_0 + \chi_{\sqrt{k-1}} \approx_s \chi_{\sigma_1}$. Here, "$\approx_s$" indicates that two distributions are statistically close.

2.4 Non-Interactive Zero-Knowledge Arguments of Knowledge (NIZKAoK)

We use the standard definitions regarding zero-knowledge (ZK) proof systems and arguments of knowledge (AoK). Informally, a ZK proof system for a language $\mathcal{L}$ allows a prover $\mathbb{P}$ to convince a verifier $\mathbb{V}$ some x is in $\mathcal{L}$ and not reveal anything else. A ZKAoK then provides a stronger guarantee where $\mathbb{P}$ also convinces $\mathbb{V}$ that they hold a witness w attesting to the fact. Formally the definition is as follows:

Definition 3. *For a prover $\mathbb{P}$, a verifier $\mathbb{V}$, a language $\mathcal{L}$ with accompanying predicate $P_{\mathcal{L}}(\cdot, \cdot)$, a witness set $\mathcal{W}_{\mathcal{L}}(\cdot)$ such that for all $x \in \mathcal{L}$ and $w \in \mathcal{W}_{\mathcal{L}}$ $P_{\mathcal{L}}(x, w) = 1$, a NIZKAoK is a tuple of algorithms $(\mathsf{Setup}, \mathbb{P}, \mathbb{V})$ such that:*

- $\mathsf{Setup}(1^{\lambda})$*: On input 1^{λ} outputs a common reference string* crs.
- $\mathbb{P}(\mathsf{crs}, x, w)$*: On input of a common reference string* crs*, a statement $x \in \mathcal{L}$, and a witness $w \in \mathcal{W}_{\mathcal{L}}$ outputs a proof $\pi \in \{0,1\}^*$ polynomial in λ.*
- $\mathbb{V}(\mathsf{crs}, x, \pi)$*: On input of a common reference string* crs*, a statement x, and a proof $\pi \in \{0,1\}^*$ outputs $b \in \{0,1\}$.*

The security of a NIZKAoK holds as long as the following definitions hold

Definition 4 (Completeness). *For $x \in \mathcal{L}$, $w \in \mathcal{W}_{\mathcal{L}}(x)$ with $P_{\mathcal{L}}(x, w,) = 1$:*

$$\Pr\left[1 \leftarrow \mathbb{V}(\mathsf{crs}, x, \pi) \colon \begin{array}{c} \mathsf{crs} \leftarrow \mathsf{Setup}(1^{\lambda}) \\ \pi \leftarrow \mathbb{P}(\mathsf{crs}, x, w) \end{array}\right] \geq 1 - \epsilon$$

where ϵ is negligible in λ.

Definition 5 (Computational Knowledge Extraction (Extractability)). *The NIZKAoK is said to have computational knowledge extraction (or is extractable) with knowledge error ϵ_{extr} if for any malicious prover $\mathbb{P}^*$ with auxiliary information* aux *there exists an extraction algorithm* $\mathsf{Extract}$ *and a polynomial p such that for any x:*

$$\Pr\left[1 \leftarrow P_{\mathcal{L}}(x, w') \colon w' \leftarrow \mathsf{Extract}(\mathbb{P}^*(\mathsf{crs}, x, \mathsf{aux}))\right] \geq \frac{\epsilon_{\mathsf{Vfy}} - \epsilon_{\mathsf{extr}}}{p(|x|)}$$

where ϵ_{Vfy} is the probability that $\mathbb{V}(\mathsf{crs}, x, \mathbb{P}^*(\mathsf{crs}, x, \mathsf{aux}))$ outputs 1.

Definition 6 (Computational zero-knowledge). *There exists a simulated setup algorithm* $\mathsf{SimSetup}$ *which on input 1^{λ} outputs crs_{Sim} and a trapdoor $\mathcal{T}$ along with a PPT simulator* Sim *where for all $x \in \mathcal{L}$ and $w \in \mathcal{W}_{\mathcal{L}}(x)$:*

$$\left\{ \begin{array}{c} \mathsf{crs} \leftarrow \mathsf{Setup}(1^{\lambda}) \\ \pi \leftarrow \mathbb{P}(\mathsf{crs}, x, w) \end{array} \right\} \approx_c \left\{ \begin{array}{c} \mathsf{crs}' \\ \pi_{\mathsf{Sim}} \leftarrow \mathsf{Sim}(\mathsf{crs}', \mathcal{T}, x) \end{array} \colon (\mathsf{crs}', \mathcal{T}) \leftarrow \mathsf{SimSetup}(1^{\lambda}) \right\}$$

2.5 Verifiable Oblivious Pseudorandom Functions

A verifiable oblivious pseudorandom function (VOPRF) for a keyed function F is a two-party protocol between a client $\mathbb{C}$ and a server $\mathbb{S}$ consisting of the following algorithms:

- $\mathsf{Init_S}$ is a protocol run by $\mathbb{S}$ which on input 1^λ outputs a secret key sk and its public commitment pk.
- $\mathsf{Init_C}$ is a protocol run by $\mathbb{C}$ which on input pk outputs a *state* indicating acceptance/rejection of public commitment.
- $\mathsf{Query_C}$ is a protocol run by $\mathbb{C}$ which on input client input x and *state* outputs a blinded message $\bar{x}$ and a state ρ.
- $\mathsf{Query_S}$ is a protocol run by $\mathbb{S}$ which on input of client's blinded message $\bar{x}$ and a secret key sk outputs a blinded evaluation y_x.
- $\mathsf{Finalize}$ is a protocol run by $\mathbb{C}$ which on input server's blinded evaluation y_x, public commitment pk and a state ρ outputs the PRF output y.

We define security of VOPRF based on corresponding games. This is not common for OPRF protocols and only a handful instantiations in literature use game-based notion of security. A protocol $\mathsf{PRF} = (\mathsf{Init_S}, \mathsf{Init_C}, \mathsf{Query_C}, \mathsf{Query_S}, \mathsf{Finalize})$ with inputs $x \in \{0,1\}^*$ and $sk \in \mathcal{K}$ is a VOPRF protocol corresponding to a keyed function F if the following hold:

Definition 7 (Correctness). *For every pair of inputs x, sk:*

$$\Pr\left[\mathsf{PRF}(x, sk) \neq F_{sk}(x)\right] \leq \epsilon$$

where ϵ is negligible in security parameter λ.

Definition 8 (Obliviousness [Leh19]). PRF *is said to be oblivious if for any PPT adversary $\mathcal{A}$ the probability of the obliviousness experiment depicted in Figure 1a outputting* 1 *is* $1/2 + \epsilon$ *where ϵ negligible in λ.*

In the obliviousness game of Definition 8 security is still based on an indistinguishability notion. For malicious $\mathbb{C}$, since we will rely on Rènyi based argument, we cannot use such an indistinguishability-based notion for security unless a specific set of conditions are met [BLR+18]. We instead define a search-based security game and upgrade this to indistinguishability of the output in the Random Oracle model (ROM).

Definition 9 (One-more unpredictability [ECS+15]). PRF *is said to be one-more unpredictable if for any PPT adversary $\mathcal{A}$ the probability of the one-more unpredictability experiment depicted in Figure 1c outputting* 1 *is negligible in λ.*

Note that the queries to the oracle $\mathcal{O}^{\mathsf{PRF}}$ include receiving blinded inputs from $\mathcal{A}$ as it is not guaranteed that $\mathcal{A}$ outputs a correctly computed $\bar{x}$. The definition is similar to unforgeability definitions for signature schemes. The intuition here is once we can argue that the interaction between $\mathbb{C}$ and $\mathbb{S}$ has an unpredictable output, the security of the VOPRF can be shown in the ROM. To achieve this we use a different notion of security called *one-more PRF security.*

```
$\underline{\mathsf{Exp}^{\mathsf{OBV}}_{\mathsf{PRF}}(\mathcal{A})}$
$pk \leftarrow \mathcal{A}(1^\lambda)$; $st \leftarrow \mathsf{PRF.Init_C}(pk)$
$\textbf{assert } st \neq \mathsf{abort}$
$x_0, x_1 \in \{0,1\}^L, \mathit{\Gamma} \leftarrow \mathcal{A}(st)$
$b \leftarrow \{0,1\}$
$\bar{x_b}, \rho \leftarrow \mathsf{PRF.Query_C}(pk, x_b)$
$b' \leftarrow \mathcal{A}(\mathit{\Gamma}, \bar{x_b})$
$\textbf{return } 1 \textbf{ if } b = b'$
```

(a) The experiment $\mathsf{Exp}^{\mathsf{OBV}}_{\mathsf{PRF}}(\mathcal{A})$

```
$\underline{\mathsf{Exp}^{\mathsf{VER}}_{\mathsf{PRF}}(\mathcal{A})}$
$pk, sk \leftarrow \mathcal{A}(1^\lambda)$; $st \leftarrow \mathsf{PRF.Init_C}(pk)$
$\textbf{assert } st \neq \mathsf{abort}$
$x \in \{0,1\}^L, \mathit{\Gamma} \leftarrow \mathcal{A}(st)$
$\bar{x}, \rho \leftarrow \mathsf{PRF.Query_C}(pk, x)$
$y_x \leftarrow \mathcal{A}(\mathit{\Gamma}, \bar{x})$; $y \leftarrow F_{sk}(x)$
$y' \leftarrow \mathsf{PRF.Finalize}(pk, y_x, \rho)$
$\textbf{return } 1 \textbf{ if } y' \neq \perp \wedge y \neq y'$
```

(b) The experiment $\mathsf{Exp}^{\mathsf{VER}}_{\mathsf{PRF}}(\mathcal{A})$

```
$\underline{\mathsf{Exp}^{om\text{-}\mathsf{UNPRED}}_{\mathsf{PRF}}(\mathcal{A})}$
$ctr := 0$
$pk, sk \leftarrow \mathsf{PRF.Init_S}(1^\lambda)$
$(x^{(0)}, y^{(0)}), \ldots, (x^{(Q)}, y^{(Q)}) \leftarrow \mathcal{A}(\mathcal{O}^{\mathsf{PRF}}, pk)$
$\textbf{if } \exists\, i \neq j \text{ s.t. } x^{(i)} = x^{(j)} \textbf{ return false}$
$\textbf{return } \wedge_i (y^{(i)} = F_{sk}(x^{(i)})) \wedge (ctr \leq Q)$

$\underline{\mathcal{O}^{\mathsf{PRF}}(\bar{x})}$
$ctr := ctr + 1$
$\textbf{return } \mathsf{PRF.Query_S}(sk, \bar{x})$
```

(c) The experiment $\mathsf{Exp}^{om\text{-}\mathsf{UNPRED}}_{\mathsf{PRF}}(\mathcal{A})$

Fig. 1: Obliviousness, Verifiability and Unpredictability

Definition 10 (One-more PRF security [ECS+15]). *A* PRF *is said to be one-more pseudorandom if for any PPT adversary $\mathcal{A}$ the probability of the one-more pseudorandomness depicted in Figure 2 outputting* 1 *is negligible in* λ.

It is easy to see that unpredictability implies one-more PRF security in the ROM when Finalize includes y_x as an input to an oracle H.

Lemma 5. *Let F be a keyed one-more unpredictable function and* H *be a hash function modeled as a random oracle. Then the function* PRF *corresponding to F has one-more PRF security.*

Proof. We can construct an adversary $\mathcal{A}$ against unpredictability using an adversary $\mathcal{B}$ against one-more PRF security as a subroutine. Let $\mathcal{B}$ be an adversary who outputs a tuple $(i_1, \ldots, i_Q)$ and a bit b'. If $\mathcal{B}$ wins the one-more PRF security game then $\mathcal{B}$ distinguished Q real or random PRF executions with $ctr < Q$ queries to the PRF oracle and $q \geq Q$ queries to the real or random oracle for the specific input. Even if $\mathcal{B}$ submits all of its PRF oracle queries to the real or random oracle, since $q > ctr$ this means there exists at least one oracle answer

```
$\mathsf{Exp}^{om\text{-}\mathsf{PRF}}_{\mathsf{PRF},\mathsf{H}}(\mathcal{A})$
$ctr, q \coloneqq 0, 0$
$pk, sk \leftarrow \mathsf{PRF.Init}_{\mathsf{S}}(1^\lambda)$
$(i_1, \ldots, i_Q, b') \leftarrow \mathcal{A}(\mathsf{H}, \mathcal{O}^{\mathsf{PRF}}, \mathcal{O}^{RoR})$
if $\exists\, \alpha$ s.t. $i_\alpha \notin [Q]$ return false
if $Q > q$ or $ctr \geq Q$ return false
if $\exists\, \alpha \neq \beta$ s.t. $i_\alpha = i_\beta$ return false
return $b' \coloneqq \bigoplus_{\alpha=1}^{Q} b[i_\alpha]$

$\mathcal{O}^{RoR}(x_i)$
$q \coloneqq q + 1,\ b[q] \leftarrow \{0,1\}$
$y_0 \leftarrow \{0,1\}^*$
$y_1 \leftarrow \mathsf{H}(x_i, F_{sk}(x_i))$
return $y_{b[q]}$

$\mathcal{O}^{\mathsf{PRF}}(\bar{x})$
$ctr \coloneqq ctr + 1$
return $\mathsf{PRF.Query}_{\mathsf{S}}(sk, \bar{x})$
```

Fig. 2: The experiment $\mathsf{Exp}^{om\text{-}\mathsf{PRF}}_{\mathsf{PRF},\mathsf{H}}(\mathcal{A})$

$x_{i'}, y_{i'}$ that has not been queried to the PRF oracle. The adversary $\mathcal{A}$ then can use $x_{i'}$ to find $F_{\mathsf{sk}}(x_{i'})$ such that $y_{i'} = \mathsf{H}(x_{i'}, F_{\mathsf{sk}}(x_{i'}))$. The adversary $\mathcal{A}$ then submits $x_{i'}, F_{\mathsf{sk}}(x_{i'})$ along with queried x_i as the answer to the unpredictability game. Since the queried x_i have been generated by the PRF oracle, the answers are valid. On the other hand, $x_{i'}$ was never queried to the PRF oracle yet is a valid answer which means $\mathcal{A}$ will win the unpredictability game.

It also has been shown in prior work that one-more PRF security is the strict strengthening of the traditional PRF security [ECS+15].

Finally, we define the notion of *verifiability* which assures $\mathbb{C}$ the output y is indeed $F_{sk}(x)$.

Definition 11 (Verifiability [ADDG24]). PRF *is said to be verifiable if for any PPT adversary $\mathcal{A}$ the probability of the verifiability experiment depicted in Figure 1b outputting 1 is negligible in λ.*

2.6 Lattice (VO)PRFs

We use the construction from [ADDS21] which adapts the lattice PRF $F_k(x) = \lfloor \frac{p}{q} \cdot \mathbf{a}^F(x) \cdot k \rceil$ from [BP14] into ring setting. The construction can be thought of an instantiation of Figure 3 where $n = 1$, $\mathsf{H}_r(\cdot,\cdot) \coloneqq 0$ and $\mathsf{H}(\cdot, y_x) \coloneqq y_x$.[4]

Remark 3. We note that there is a generic transformation upgrading any VOPRF to a *partial* VOPRF, where part of the client's input is in the clear. The server computes $\mathsf{sk}_t \coloneqq \mathsf{H}_K(\mathsf{sk}, t)$ for its master secret key sk and public client input or tag t and then proceeds with sk_t [CHL22,JKR18]. In the verifiable case, the server is also required to output a commitment to sk_t, to have something to verify against. We forego discussing partial variants of (V)OPRFs for the remainder of this work.

[4] In [ADDS21] the protocol is defined for vectors of ring elements of length ℓ rather than single ring elements. We give the variant here, already mentioned in [ADDS21], that only considers a single ring element, for performance.

3 Construction

We first define the languages $\mathcal{L}_0$,$\mathcal{L}_1$, $\mathcal{L}_2$ with corresponding predicates $P_{\mathcal{L},0}$, $P_{\mathcal{L},1}$, $P_{\mathcal{L},2}$ for our NIZKAoKs:

$$\mathcal{L}_0 := \left\{ P_{\mathcal{L},0}(x,w) = 1 \,\middle|\, \begin{array}{c} x := (c_i) \ \wedge \ w := (k_i, e_i)\colon \\ \|k_i\|_2, \|e_i\|_2 \le B_0 \ \wedge \ c_i = a \cdot k_i + e_i \mod q \end{array} \right\}$$

$$\mathcal{L}_1 := \left\{ P_{\mathcal{L},1}(x,w) = 1 \,\middle|\, \begin{array}{c} x := (c_x) \ \wedge \ w := (x = (x_0, \ldots, x_{L-1}), s, e_{\mathbb{C}})\colon \\ \|s\|_2, \|e_{\mathbb{C}}\| \le B_1 \\ \wedge \ \mathbf{a}_x = \mathbf{a}_{x_0} \cdot G^{-1}(\ldots(\mathbf{a}_{x_{L-2}} \cdot G^{-1}(\mathbf{a}_{x_{L-1}}))\ldots) \\ \wedge \ c_x = a \cdot s + e_{\mathbb{C}} + \mathbf{a}_x[0] \mod q \end{array} \right\}$$

$$\mathcal{L}_2 := \left\{ P_{\mathcal{L},2}(x,w) = 1 \,\middle|\, \begin{array}{c} x := (c_i, c_x, d_{x,i}) \ \wedge \ w := (k_i, e_i, e_{\mathbb{S},i})\colon \\ \|k_i\|_2, \|e_i\|_2 \le B_0 \ \wedge \ \|e_{\mathbb{S},i}\|_2 \le B_2 \\ \wedge \ c_i = a \cdot k_i + e_i \mod q \\ \wedge \ d_{x,i} = c_x \cdot k_i + \mathbf{e}_{\mathbb{S},i} \mod q \end{array} \right\}$$

for reference strings $\mathsf{crs}_0 = (a, B_0)$, $\mathsf{crs}_1 = (a, \mathbf{a}_0, \mathbf{a}_1, B_1)$, and $\mathsf{crs}_2 = (a, B_2)$.[5] Our construction, which is a mild variant of the construction given in [ADDS21], is given in Figure 3 when $n = 1$. For the rest of this work, we set $B_0 = \sigma \cdot \sqrt{N}$, $B_1 = \sigma \cdot \sqrt{N}$, and $B_2 = \sigma' \cdot \sqrt{N}$.

We start by proving that the protocol is secure against a malicious $\mathbb{S}$ i.e. is oblivious and verifiable. The first proof is almost exactly the same as the malicious server proof of [ADDS21] and given for completeness; the second proof is similar but drops the need for invoking the 1D-SIS assumption.

Theorem 1. *Let σ and N be $\mathsf{poly}(\lambda)$. Let $\mathsf{dRLWE}_{q,N,\sigma,\sigma}$ be hard. Let $(\mathbb{P}_0, \mathbb{V}_0)$, $(\mathbb{P}_1, \mathbb{V}_1)$ be NIZKAoKs for languages $\mathcal{L}_0, \mathcal{L}_1$, then the protocol in Figure 3 with $n = 1$ is oblivious against any PPT adversary $\mathcal{A}$ controlling $\mathbb{S}$.*

Proof. We show $\mathcal{A}$ controlling $\mathbb{S}^*$ cannot distinguish client input c_x through a series of hybrids.

Hybrid_0 This is the base obliviousness game where client $\mathbb{C}$ interacts with an adversary $\mathcal{A}$ controlling $\mathbb{S}^*$. The client $\mathbb{C}$ interacts with $\mathbb{S}^*$ through the PRF protocol after verifying server's commitment and $\mathcal{A}$ outputs two inputs x_0, x_1 of length L. The client $\mathbb{C}$ then chooses one of these inputs x_b and continues with PRF execution as before. In the end, if $\mathcal{A}$ finds b with non-negligible probability then it wins the obliviousness game. The advantage of $\mathcal{A}$ is its advantage in the obliviousness game.

Hybrid_1 $\mathbb{C}$ now does not compute π_1 honestly and uses the associated proof simulator for $\mathbb{P}_1, \mathbb{V}_1$ and sends a simulated proof π_1' instead. The rest proceeds as before. We have that Hybrid_0 and Hybrid_1 are indistinguishable by the ZK property of the underlying ZKAoK.

[5] We discuss how to instantiate these proof systems in Section 5.

CRS SetUp:

- $\mathbf{a}_0, \mathbf{a}_1 \leftarrow R_q^{1\times \ell}$.
- $a \leftarrow R_q$, sample $\bar{\mathsf{crs}}_0$ for $\mathbb{P}_0$, $\mathsf{crs}_0 := (\bar{\mathsf{crs}}_0, a)$.
- Sample crs_1, crs_2 for $\mathbb{P}_1, \mathbb{P}_2$.

Initialisation:

- $\mathsf{Init}_{\mathsf{S}}$: Server $\mathbb{S}_i$ for $i \in [n]$ executes
  - Choose σ_i such that $\sigma_i \leq \sigma$
  - $k_i \leftarrow R_{\chi_{\sigma_i}}$, $e_i \leftarrow R_{\chi_\sigma}$.
  - $c_i \leftarrow a \cdot k_i + e_i \mod q$.
  - $\pi_{0,i} \leftarrow \mathbb{P}_0(k_i, e_i : \mathsf{crs}_0, c_i)$

  and broadcasts $c_i, \pi_{0,i}$.
- $\mathsf{Init}_{\mathsf{C}}$: $\mathbb{C}$ on input of $\{(c_i, \pi_{0,i})\}_{i\in[n]}$ executes
  - $b_i \leftarrow \mathbb{V}_0(\mathsf{crs}_0, c_i, \pi_{0,i})$.
  - Output abort with i if $b_i = 0$, otherwise store c_i.

Query:

1. $\mathsf{Query}_{\mathsf{C}}$: $\mathbb{C}$ executes the following with the input $(x \in \{0,1\}^L, \mathsf{crs}_1, \mathsf{crs}_2)$
   - $s \leftarrow R_{\chi_\sigma}$, $e_{\mathbb{C}} \leftarrow R_{\chi_\sigma}$.
   - $\mathbf{a}_x := \mathbf{a}_{x_0} \cdot G^{-1}(\ldots(\mathbf{a}_{x_{L-2}} \cdot G^{-1}(\mathbf{a}_{x_{L-1}}))\ldots) \mod q$.
   - $c_x \leftarrow a \cdot s + e_{\mathbb{C}} + \mathbf{a}_x[0] \mod q$.
   - $\pi_1 \leftarrow \mathbb{P}_1(x, s, e_{\mathbb{C}} : \mathsf{crs}_1, c_x, a, \mathbf{a}_0, \mathbf{a}_1)$.

   and broadcasts (c_x, π_1) to all $\mathbb{S}_i$.
2. $\mathsf{Query}_{\mathsf{S}}$: $\mathbb{S}_i$ executes the following after receiving (c_x, π_1)
   - $b \leftarrow \mathbb{V}_1(\mathsf{crs}_1, c_x, \mathbf{a}_0, \mathbf{a}_1, \pi_1)$, output abort if $b = 0$.
   - $e_{\mathbb{S},i} \leftarrow R_{\chi_{\sigma'}}$.
   - $d_{x,i} := c_x \cdot k_i + e_{\mathbb{S},i} \mod q$.
   - $\pi_{2,i} \leftarrow \mathbb{P}_2(k_i, e_{\mathbb{S},i}, e_i : \mathsf{crs}_2, c_i, d_{x,i}, c_x, a)$.

   and sends $(d_{x,i}, \pi_{2,i})$ to $\mathbb{C}$ and outputs $\perp$.
3. $\mathsf{Finalize}$: $\mathbb{C}$ finally executes the following after receiving $(d_{x,i}, \pi_{2,i})$ from all $\mathbb{S}_i$.
   - $b_i \leftarrow \mathbb{V}_2(\mathsf{crs}_0, \mathsf{crs}_2, c_i, d_{x,i}, c_x, \pi_{2,i})$, output abort with i if $b_i = 0$.
   - $d_x := \sum_{i\in[n]} d_{x,i}$, $c := \sum_{i\in[n]} c_i$.
   - $r \leftarrow \mathsf{H}_r(x, c) \in \mathcal{R}_q$ // $\mathsf{H}_r(\cdot,\cdot) := 0$ in [ADDS21]
   - $y_x := \lfloor d_x + r - c \cdot s \rceil_p$.
   - $y \leftarrow \mathsf{H}(x, y_x)$ // $\mathsf{H}(\cdot, y_x) := y_x$ in [ADDS21]

   and outputs y.

Fig. 3: n-out-of-n VOPRF Construction.

Hybrid_2 Instead of honestly computing c_x, $\mathbb{C}$ now samples a uniformly random $u_x \leftarrow \mathcal{R}_q$. The rest of the experiment proceeds as before. We have Hybrid_1 and Hybrid_2 are then indistinguishable by the $\mathsf{dRLWE}_{q,N,\sigma,\sigma}$ assumption.

In Hybrid_2 $\mathbb{C}$'s reply u_x does not rely on $\mathcal{A}$'s chosen values x_0 or x_1 anymore, hence $\mathcal{A}$'s advantage cannot be greater than 1/2. This concludes the proof □

.

Theorem 2. *Let σ and N be* $\mathsf{poly}(\lambda)$. *Let $\beta = 2\,\sigma^2 \cdot N + \sigma' \cdot \sqrt{N}$ and $q/p \gg \beta$. Let $(\mathbb{P}_0, \mathbb{V}_0), (\mathbb{P}_2, \mathbb{V}_2)$ be NIZKAoKs for languages $\mathcal{L}_0, \mathcal{L}_2$, H_r be a random oracle, and Q_H be number of queries made to such oracle. Then the protocol in Figure 3 with $n = 1$ is verifiable against any PPT adversary $\mathcal{A}$ controlling $\mathbb{S}$ in the ROM.*

Proof. We show $\mathcal{A}$ corrupting a server $\mathbb{S}^*$ cannot have a significant advantage by biasing the output derived by $\mathbb{C}$ and force an incorrect evaluation.

If $\mathbb{S}^*$'s reply does not have a valid proof for the setup or the final round then the client will abort. If not, we extract a key k^* from π_0 with $\|k^*\|_\infty \le \sigma \cdot \sqrt{N}$. Let s and $e_\mathbb{C}$ be sampled as it is in the protocol for an honest client therefore $\|s\|_\infty \le \|s\|_2 \le \sigma \cdot \sqrt{N}$ and $\|e_\mathbb{C}\|_\infty \le \|e_\mathbb{C}\|_2 \le \sigma \cdot \sqrt{N}$. Observe that an honest client has

$$\frac{p}{q} \cdot \Big(d_x + r - c \cdot s \Big) = \frac{p}{q} \cdot \mathbf{a}_x[0] \cdot k^* + \frac{p}{q} \cdot r + \frac{p}{q} \cdot \Big(e_\mathbb{C} \cdot k^* - e \cdot s + e_\mathbb{S} \Big)$$

Each k^*, $e_\mathbb{S}$ are correctly computed according to the protocol hence $\|k^*\|_\infty \le \|k^*\|_2 \le \sigma \cdot \sqrt{N}$ and $\|e_\mathbb{S}\|_\infty \le \|e_\mathbb{S}\|_2 \le \sigma' \cdot \sqrt{N}$.

If every coefficient of $\frac{p}{q} \cdot \mathbf{a}_x[0] \cdot k^* + \frac{p}{q} \cdot r$ is further away from $\mathbb{Z} + 1/2$ than $\left\| \frac{p}{q} \cdot (e_\mathbb{C} \cdot k^* - e \cdot s + e_\mathbb{S}) \right\|_\infty$, the evaluation is correct. The adversary can query H_r to find r^* such that evaluation would be incorrect. Note that in the Random Oracle Model r^* is independent of k^*, since H_r takes a commitment to k^* as one of its inputs, therefore the only way $\mathcal{A}$ can find a satisfying r^* is by finding an x^* such that $r^* := \mathsf{H}_r(x^*, c)$. This probability is negligible in the Random Oracle Model as long as $2^\lambda \gg Q_\mathsf{H}$. It follows that $\mathcal{A}$ can only force an incorrect evaluation with probability proportional to $\|e_\mathbb{C} \cdot k^* - e \cdot s + e_\mathbb{S}\|_\infty/(q/p)$. We then have

$$\begin{aligned}\|e_\mathbb{C} \cdot k^* - e \cdot s + e_\mathbb{S}\|_\infty/(q/p) &\le (\|e_\mathbb{C}\|_\infty \cdot \|k^*\|_\infty + \|e\|_\infty \cdot \|s\|_\infty + \|e_\mathbb{S}\|_\infty)/(q/p) \\ &\le (2\sigma^2 \cdot N + \sigma'\sqrt{N})/(q/p).\end{aligned}$$

Since $q/p \gg \beta$ with $\beta = 2\,\sigma^2 \cdot N + \sigma' \cdot \sqrt{N}$ the probability above is negligible. □

4 Using Rènyi Divergence for Smaller Parameters

The security of [ADDS21] relies on $\mathbf{e}_\mathbb{S}$ chosen from a distribution $\mathcal{R}_{\chi_{\sigma'}}^{1\times\ell}$ with $\sigma' \gg \max(L \cdot \ell \cdot \sigma \cdot N^{3/2}, \sigma^2 \cdot N^2)$. This superpolynomial gap has a significant impact on communication costs. In the malicious client proof of [ADDS21], the security argument relies on the statistical distance between

$$\mathbf{c} \cdot s + \mathbf{e}_{\mathbb{C}} \cdot k + \mathbf{a}_x \cdot k + \mathbf{e}_{\mathbb{S}} \text{ and } \mathbf{c} \cdot s + \hat{\mathbf{e}_{\mathbb{S}}} + (\mathbf{a}_x \cdot k + \mathbf{e}_x)$$

where $\mathbf{e}_{\mathbb{C}} \leftarrow \mathcal{R}^{1\times\ell}_{\chi_\sigma}$, $\mathbf{e}_{\mathbb{S}}, \hat{\mathbf{e}_{\mathbb{S}}} \leftarrow \mathcal{R}^{1\times\ell}_{\chi_{\sigma'}}$, and $\mathbf{e}_x \leftarrow \mathcal{E}_{\mathbf{a}_0,\mathbf{a}_1,x,\sigma}$.

Here, instead, we use a Rènyi divergence based argument to prove that the protocol given in Figure 3 with $n = 1$, $\mathsf{H}_r(\cdot,\cdot) := 0$, and $\mathsf{H}(x, y_x) := y_x$ is unpredictable. We note, though, that Theorem 3 requires a bound Q on the permitted number of queries, this explains the four different rows in Table 2.

Theorem 3. *Assume that σ and N are $\mathsf{poly}(\lambda)$, and $p|q$. Let $\mathsf{dRLWE}_{q,N,\sigma,\sigma}$ hard and $\frac{q}{2p} \gg \sigma' \geq (L \cdot \sqrt{N} + 2 \cdot \sigma) \cdot \sigma \cdot N \cdot \sqrt{Q \cdot N}$ for a number of queries made Q. Let $(\mathbb{P}_0, \mathbb{V}_0), (\mathbb{P}_1, \mathbb{V}_1), (\mathbb{P}_2, \mathbb{V}_2)$ be NIZKAoKs for languages $\mathcal{L}_0, \mathcal{L}_1, \mathcal{L}_2$, then the VOPRF protocol defined in Figure 3 with $n = 1$, $\mathsf{H}_r(\cdot,\cdot) := 0$, and $\mathsf{H}(x, y_x) := y_x$ is unpredictable against any PPT adversary $\mathcal{A}$ controlling $\mathbb{C}$.*

Proof. We show $\mathcal{A}$ controlling $\mathbb{C}^*$ cannot find an unqueried request-response pair $(x^{(\tau)}, y_x^{(\tau)})$ with all but negligible probability in λ.

Hybrid_0*:* This is the real execution of the protocol where $\mathcal{A}$ makes Q queries to $\mathbb{S}$. The server $\mathbb{S}$ samples a key k and outputs a commitment c. For $\tau \in [Q]$, $\mathcal{A}$ sends a query $(c_x^{(\tau)}, \pi_1^{(\tau)})$ based on x for which $\mathbb{S}$ computes $(d_x^{(\tau)}, \pi_2^{(\tau)})$ if $\pi_1^{(\tau)}$ verifies and aborts otherwise. The adversary $\mathcal{A}$ then computes $y_x^{(\tau)}$ based on $d_x^{(\tau)}$ and $x^{(\tau)}$ (resp. $y_x^{(\tau)}$) is added to the set $\mathcal{X}$ (resp. $\mathcal{Y}$). At the end, $\mathcal{A}$ outputs (x^*, y_x^*) and wins the game if $x^* \notin \mathcal{X}$ and c_x^* generated on x^* evaluates to y_x^*. The advantage of $\mathcal{A}$ is the probability of $\mathcal{A}$ winning in the unpredictability game.

Hybrid_1*:* Hybrid_1 is the same as Hybrid_0 except how the proofs by the server are computed. Instead of honestly generating crs_0 and crs_2, and computing π_0 and $\pi_2^{(\tau)}$, $\mathbb{S}$ calls the simulator for the relative proof systems. Hybrid_1 is then indistinguishable from Hybrid_0 by the ZK property of the underlying ZKAoKs.

Hybrid_2*:* Hybrid_2 is the same as Hybrid_1 except that in the **Query** phase we have that after $\mathbb{S}$ receives $(c_x^{(\tau)}, \pi_1^{(\tau)})$, it calls the extractor for the underlying ZKAoK to obtain $(x^{(\tau)}, e_{\mathbb{C}}^{(\tau)}, s^{(\tau)})$ and aborts if it fails to do so. By the extractability of the underlying ZKAoK Hybrid_1 is exactly like Hybrid_2 as long as the extraction does not fail. Then Hybrid_1 and Hybrid_2 are indistinguishable.

Hybrid_3*:* $\mathbb{S}$ changes how $d_x^{(\tau)}$ are computed. Upon receiving $c_x^{(\tau)}$, $\mathbb{S}$ samples $e'_{\mathbb{S}} \leftarrow \mathcal{R}_{\chi_{\sigma'}}$ and $e_x^{(\tau)} \leftarrow \mathcal{E}_{\mathbf{a}_0,\mathbf{a}_1,x,\sigma}$ based on the extracted $x^{(\tau)}$. The server $\mathbb{S}$ then sends $d_x^{(\tau)} := c_x^{(\tau)} \cdot k + e'_{\mathbb{S}} + e_x^{(\tau)} - e_{\mathbb{C}}^{(\tau)} \cdot k + e \cdot s^{(\tau)}$.

Since $(s^{(\tau)}, e_{\mathbb{C}}^{(\tau)}, x^{(\tau)})$ were extracted from $\pi_1^{(\tau)}$, it is possible for $\mathbb{S}$ to sample $e_x^{(\tau)}$ based on $x^{(\tau)}$ and compute $e_{\mathbb{C}}^{(\tau)} \cdot k$ and $e \cdot s^{(\tau)}$. To bound the probability of the adversary winning when going from Hybrid_2 to Hybrid_3, we will show that if $\mathcal{A}$ making Q queries can win the game in Hybrid_2 with probability ρ then its probability of winning in Hybrid_3 is also polynomial in ρ. Let $\mathcal{D}_2^{(\tau)}$ and $\mathcal{D}_3^{(\tau)}$ denote the distributions of $(d_x^{(\tau)}, \pi_2^{(\tau)})$ in Hybrid_2 and Hybrid_3 respectively.

In both Hybrid_2 and Hybrid_3, $(\pi_2^{(\tau)})$ are simulated and hence are distributed exactly the same. In Hybrid_2, $d_x^{(\tau)}$ is computed as $d_x^{(\tau)} = c_x^{\tau} \cdot k + e_{\mathbb{S}}$ whereas in Hybrid_3 we have $d_x^{(\tau)} = c_x^{(\tau)} \cdot k + e'_{\mathbb{S}} + e_x^{(\tau)} - e_{\mathbb{C}}^{(\tau)} \cdot k + e \cdot s^{(\tau)}$ for $e_{\mathbb{S}}, e'_{\mathbb{S}} \in \mathcal{R}_{\chi_{\sigma'}}$ The distribution of $d_x^{(\tau)}$ in two hybrids can then be considered as two Gaussians with different centres. Outside $\{(d_x^{(\tau)}, \pi_2^{(\tau)})\}_{\tau \in Q}$, $\mathcal{A}$'s view in both hybrids consists of

$$\mathsf{crs}_0, \mathsf{crs}_1, \mathsf{crs}_2, a, \mathbf{a}_1, \mathbf{a}_0, c_x^{(\tau)}$$

and consequently $c_x^{(\tau)}$. Here, we have that $\mathsf{crs}_0, \mathsf{crs}_1, \mathsf{crs}_2, a, \mathbf{a}_1, \mathbf{a}_0$ and c are sampled independently from $\mathcal{A}$, therefore are fixed in both views. $x^{(\tau)}$ and consequently $c_x^{(\tau)}$ however are chosen by adversary which means $x^{(\tau)}$ (consequently $\mathbf{a}_x^{(\tau)}$), $s^{(\tau)}$, $e_{\mathbb{C}}^{(\tau)}$ and therefore $c_x^{(\tau)}$ can be adaptively chosen. However, note that each $c_x^{(\tau)}$ is associated with a proof $\pi_1^{(\tau)}$ proving $c_x^{(\tau)}$ is correctly computed. Since $\mathbb{S}$ does not abort, each $\pi_1^{(\tau)}$ has to verify therefore each $c_x^{(\tau)}$ corresponds to the same distribution.

The RD between $\mathcal{D}_2^{(\tau)}$ and $\mathcal{D}_3^{(\tau)}$ is then by Lemma 4:

$$\begin{aligned} R_\alpha(\mathcal{D}_2^{(\tau)} || \mathcal{D}_3^{(\tau)}) &\leq 1 \cdot \exp\left(\frac{\alpha \cdot \pi \cdot \|e_x - e_{\mathbb{C}} \cdot k + e \cdot s\|_2^2}{\sigma'^2} \right) \\ &\leq \exp\left(\frac{\alpha \cdot \pi \cdot \left(\sqrt{N} \|e_{\mathbb{C}}\|_2 \cdot \|k\|_2 + \sqrt{N} \|e\|_2 \cdot \|s\|_2 + \|e_x\|_2 \right)^2}{\sigma'^2} \right) \end{aligned}$$

Since $\tau \in Q$, we can define the distributions $\mathcal{D}_2$ and $\mathcal{D}_3$ for the distribution of $(d_x^{(\tau)}, \pi_2^{(\tau)})$ for the entire hybrids. We have:

$$R_\alpha(\mathcal{D}_2 || \mathcal{D}_3) \leq \exp\left(\frac{\alpha \cdot \pi \cdot Q \cdot \left(\sqrt{N} \|e_{\mathbb{C}}\|_2 \cdot \|k\|_2 + \sqrt{N} \|e\|_2 \cdot \|s\|_2 + \|e_x\|_2 \right)^2}{\sigma'^2} \right)$$

Let ψ, ψ' denote the views of $\mathcal{A}$ in Hybrid_2 and Hybrid_3 respectively. By data processing inequality of RD we then have:

$$R_\alpha(\psi || \psi') \leq R_\alpha(\mathcal{D}_2 || \mathcal{D}_3)$$

Let E be the event that $\mathcal{A}$ outputs a successful prediction. By our assumption we then have $\mathcal{D}_2(E) = \rho$. Following the probability preservation property of RD:

$$\psi'(E) \geq \frac{\rho^{\frac{\alpha}{\alpha-1}}}{R_\alpha(\psi || \psi')}$$

By assumption $\sigma' \geq (L \cdot \sqrt{N} + 2 \cdot \sigma) \cdot \sigma \cdot N \cdot \sqrt{Q \cdot N}$, $\|e\|_2, \|e_{\mathbb{C}}\|_2 \leq \sigma\sqrt{N}$, $\|k\|_2, \|s\|_2 \leq \sigma\sqrt{N}$, and $\|e_x\|_\infty \leq L \cdot \sigma \cdot N^{3/2}$ [ADDS21, Lemma 4]. This means $R_\alpha(\psi||\psi') \leq \exp(\pi \cdot \alpha)$ and consequently $\psi'(E) \geq \rho^{\frac{\alpha}{\alpha-1}} \cdot \exp(-\alpha\pi)$ which is non-negligible if and only if ρ is non-negligible.

Hybrid_4*:* $\mathbb{S}$ stops using key material k for replies to $\mathbb{C}^*$. The server $\mathbb{S}$ maintains a received list for $(x^{(\tau)}, y_q)$. After receiving and verifying $c_x^{(\tau)}$, it checks if the extracted $x^{(\tau)}$ has been queried before. If $(x^{(\tau)}, y_q)$ exists in $\mathsf{received}$, $\mathbb{S}$ retrieves y_q from the list, samples $\bar{e_{\mathbb{S}}}^{(\tau)} \leftarrow \mathcal{R}_{\chi_{\sigma'}}$ and returns $\bar{d}_x^{(\tau)} = c \cdot s^{(\tau)} + \bar{e_{\mathbb{S}}}^{(\tau)} + y_q$. If $x^{(\tau)}$ is queried for the first time, $\mathbb{S}$ first samples a PRF output y and then uniformly samples a y_q such that $y_q \leftarrow \mathcal{R}_q \cap (q/p \cdot y + \mathcal{R}_{\leq q/2p})$. $\mathbb{S}$ records $(x^{(\tau)}, y_q)$ and computes $\bar{d}_x^{(\tau)}$ the same. First, note that we can rewrite $d_x^{(\tau)}$ in Hybrid_3 as

$$\begin{aligned}
d_x^{(\tau)} &= c_x^{(\tau)} \cdot k + e'_{\mathbb{S}} + e_x^{(\tau)} - e_{\mathbb{C}}^{(\tau)} \cdot k + e \cdot s^{(\tau)} \\
&= a \cdot s^{(\tau)} \cdot k + e_{\mathbb{C}}^{(\tau)} \cdot k + \mathbf{a}_x^{(\tau)}[0] \cdot k + e'_{\mathbb{S}} + e_x^{(\tau)} - e_{\mathbb{C}}^{(\tau)} \cdot k + e \cdot s^{(\tau)} \\
&= a \cdot s^{(\tau)} \cdot k + e \cdot s^{(\tau)} + e_{\mathbb{C}}^{(\tau)} \cdot k + \mathbf{a}_x^{(\tau)}[0] \cdot k + e'_{\mathbb{S}} + e_x^{(\tau)} - e_{\mathbb{C}}^{(\tau)} \cdot k \\
&= c \cdot s^{(\tau)} + e_x^{(\tau)} + \left(\mathbf{a}_x^{(\tau)}[0] \cdot k + e'_{\mathbb{S}}\right)
\end{aligned}$$

We have that $\mathbf{a}_x^{(\tau)}[0] \cdot k + e_x^{(\tau)}$ is indistinguishable from some uniform $u_x^{(\tau)}$ by opening up the proof of [ADDS21, Lemma 3].[6] Said lemma holds under the hardness of $\mathsf{dRLWE}_{q,N,\sigma,\sigma}$ where $\mathbf{a}_x^{(\tau)}[0] \cdot k + e_x^{(\tau)}$ can be decomposed as multiple samples of the form $a_i \cdot k + e_i$ for uniform $a_i \in \mathcal{R}_q$ and small $e_i \in \mathcal{R}_q$. Multiple queries for $\mathbf{a}_x^{(\tau)}[0] \cdot k + e_x^{(\tau)}$ can then be considered as increased number of samples for $a_i \cdot k + e_i$. Hence, by the hardness of $\mathsf{dRLWE}_{q,N,\sigma,\sigma}$, $c \cdot s^{(\tau)} + \mathbf{a}_x^{(\tau)}[0] \cdot k + e_x^{(\tau)} + e'_{\mathbb{S}}$ is indistinguishable from $d_x^{(\tau)} = c \cdot s^{(\tau)} + u_x^{(\tau)} + e'_{\mathbb{S}}$ for some uniform $u_x^{(\tau)}$. Since y is a PRF output, y_q is a uniformly chosen element of a uniformly chosen interval, it is also indistinguishable from $u_x^{(\tau)}$. Finally $\bar{e_{\mathbb{S}}}^{(\tau)}$ is sampled from the same distribution as $e'_{\mathbb{S}}$, $d_x^{(\tau)}$ and $\bar{d}_x^{(\tau)}$ therefore Hybrid_3 and Hybrid_4 are indistinguishable.

Hybrid_5*:* Now that the VOPRF answer does not rely on k, $\mathbb{S}$ stops sampling a k altogether and samples a uniformly random $c \leftarrow \mathcal{R}_q$ instead. By the hardness of $\mathsf{dRLWE}_{q,N,\sigma,\sigma}$, c in Hybrid_4 and Hybrid_5 are indistinguishable.

Now that every reply to $\mathcal{A}$ is freshly generated and independent from any secret material, they are unpredictable. This concludes the proof. □

From the unpredictable function, we can define a VOPRF. We first define random oracles $\mathsf{H}\colon \{0,1\}^L \times \mathcal{R}_q \to \{0,1\}^\lambda$ and $\mathsf{H}_r\colon \{0,1\}^L \times \mathcal{R}_q \to \mathcal{R}_q$ which

[6] In [ADDS21], it is argued that $\mathbf{a}_x^{(\tau)}[0] \cdot k + e_x^{(\tau)}$ and $u_x^{(\tau)}$ are indistinguishable directly by a lemma implicit in the underlying PRF [BP14]. This is incorrect as is because the lemma does not consider multiple queries.

then are used to generate the VOPRF output on the client side. The new VOPRF protocol is depicted in Figure 3 with $n = 1$, and new definitions of H and H_r.

The transformation is rather standard and we immediately follow that the VOPRF protocol has one-more PRF security.

Corollary 1. *Assume that σ and N are $\mathsf{poly}(\lambda)$, and $p|q$. Let H, H_r be hash functions modeled as random oracles, and Q denote the number of queries made to the VOPRF. Let $\mathsf{dRLWE}_{q,N,\sigma,\sigma}$ hard and $\frac{q}{2p} \gg \sigma' \geq (L\cdot\sqrt{N}+2\cdot\sigma)\cdot\sigma\cdot N\cdot\sqrt{Q\cdot N}$. Let $(\mathbb{P}_0,\mathbb{V}_0), (\mathbb{P}_1,\mathbb{V}_1), (\mathbb{P}_2,\mathbb{V}_2)$ be NIZKAoKs for languages $\mathcal{L}_0, \mathcal{L}_1, \mathcal{L}_2$. Then if the VOPRF protocol defined in Figure 3 for $n = 1$ is unpredictable, it also has one-more PRF security in random oracle model against any PPT adversary $\mathcal{A}$ controlling $\mathbb{C}$.*

5 Efficient Lattice-Based NIZKAoK Instantiations

We discuss the required NIZKAoK constructions for instantiating $\mathbb{P}_0, \mathbb{P}_1, \mathbb{P}_2$.

The proof systems $\mathbb{P}_0, \mathbb{P}_2$ are similar in nature as both statements prove the knowledge of short elements (x_1, x_2) such that $y = a \cdot x_1 + x_2$ for public a, y. For these types of relations we can still use the argument systems given in [ADDS21] but we use more recent lattice-based NIZKs [LNP22]. Proof $\mathbb{P}_0$ shows that commitment c to the secret key k is correctly computed where $c \coloneqq a \cdot k + e$, and $\|k\|_2 \leq \sigma\sqrt{N}$ and $\|e\|_2 \leq \sigma\sqrt{N}$ for public a, c. Similarly, $\mathbb{P}_2$ shows the $\mathbb{S}$ output d_x is computed correctly with $d_x \coloneqq c_x \cdot k + e_{\mathbb{S}}$ for public c_x, d_x where $\|e_{\mathbb{S}}\|_2 \leq \sigma'\sqrt{N}$. These relations can easily be instantiated with proofs from [LNP22].

A major reason for [ADDS21] not being practical is the type of statements that have to be proven as part of the protocol execution. In particular, $\mathbb{P}_1$ requires $\mathbb{C}$ to show their message c_x is well-formed, which includes non-standard arguments. In [ADDS21], this is addressed using the NIZKAoK construction in [YAZ+19] which proves generic rank-1 constraints over $\mathbb{Z}_q$. This breaks the element structure in protocol therefore is highly inefficient. Here, we take a different approach and try to prove relationships while preserving the structure of the elements used throughout the protocol. Moreover, we prove these statements using more recent lattice-based proof systems [BS23].

We start by looking at the type of relations we have to consider for $\mathbb{P}_1$ in [ADDS21]. The first client message c_x is computed as:

$$\begin{aligned} \mathbf{a}_x &\coloneqq \mathbf{a}_{x_0} \cdot G^{-1}(\ldots(\mathbf{a}_{x_{L-1}} \cdot G^{-1}(\mathbf{a}_{x_{L-1}}))\ldots) \mod q \\ c_x &\leftarrow a \cdot s + e_{\mathbb{C}} + \mathbf{a}_x[0] \mod q. \end{aligned}$$

The relations to be proven can then be broken down to

$$\begin{aligned} \mathbf{B}_i &= G^{-1}(\mathbf{a}_{x_i} \cdot \mathbf{B}_{i+1}) \quad \text{for } i \in 0, \ldots, L-2 \\ \mathbf{B}_{L-1} &= G^{-1}(\mathbf{a}_{x_{L-1}}) \\ c_x &= a \cdot s + e_{\mathbb{C}} + (\mathbf{G} \cdot \mathbf{B}_0)[0] \end{aligned}$$

where $\mathbf{B}_i \in \mathcal{R}_2^{\ell \times \ell}$ are $\ell \times \ell$ matrices of binary decompositions. For a gadget vector $\mathbf{g}^\top = (1, 2, \ldots, 2^{\ell-1}) \in \mathcal{R}_q^{1\times\ell}$ the relations can be rewritten as :

$$\begin{aligned}
\mathbf{g}^\top \cdot \mathbf{B}_{L-1} &= \mathbf{a}_0 \cdot (1 - x_{L-1}) + \mathbf{a}_1 \cdot x_{L-1} \\
\mathbf{g}^\top \cdot \mathbf{B}_i &= \mathbf{a}_0 \cdot (1 - x_i) \cdot \mathbf{B}_{i+1} + \mathbf{a}_1 \cdot x_i \cdot \mathbf{B}_{i+1} \quad \text{for } i \in 0, \ldots, L-2 \\
c_x &= a \cdot s + e_{\mathbb{C}} + (\mathbf{g}^\top \cdot \mathbf{B}_0)[0]
\end{aligned}$$

In [ADDS21] this is then broken down to be represented as R1CS constraints over $\mathbb{Z}_q$. However, the relations above can be represented as dot products. We then use LaBRADOR [BS23] which efficiently proves statements of the form

$$f(\mathbf{s}) = \sum_{1\le i,j\le r} a_{i,j}\langle \mathbf{s}_i, \mathbf{s}_j\rangle + \sum_{i=1}^{r}\langle \psi_i, \mathbf{s}_i\rangle + b = 0$$

where $\mathbf{s}_i, \psi_i \in \mathcal{R}_q^n$, $a_{i,j}, b \in \mathcal{R}_q$ and the short solution $\mathbf{s} = \mathbf{s}_1, \ldots, \mathbf{s}_r$ is the witness.

We then define the witness vector in our construction as the concatenation of all the ring elements of the witness for the relation which includes $\mathbb{C}$ input $x = (x_0, \ldots, x_{L-1})$, matrices $\{\mathbf{B}_i\}_{i\in[L-1]}$, client short secret s, and the error $e_{\mathbb{C}}$. Each x_i is an element in $\mathcal{R}_2$ and $\mathbf{B}_i$ is an $\ell \times \ell$ matrix in $\mathcal{R}_2$, all of which will be treated as elements in $\mathcal{R}_q$. Finally, s and $e_{\mathbb{C}}$ are single elements in $\mathcal{R}_q$. Note that even though we truncate $\mathbf{a}_x$ in favor of using the first ring element, the relations we prove do not change until the final line. Thus the witness size is:

$$L + L \cdot \ell \cdot \ell + 2$$

All of these elements are over the ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^N + 1)$, whereas for LaBRADOR we need elements of in the ring $\mathcal{R}'_q = \mathbb{Z}_q[X]/(X^{64} + 1)$. For completeness, we give details on how to prove statements in $\mathcal{R}_q$ using vectors over $\mathcal{R}'_q$ in Appendix A. The effective witness vector size is then:

$$\lceil N/64 \rceil \cdot (L + L \cdot \ell \cdot \ell + 2)$$

We also need the Euclidean norm bound on the witness vector for LaBRADOR. Since x_i, $\mathbf{B}_i$ are in $\mathcal{R}_2$, and $\|s\|_2 \le \sigma\sqrt{N}$ and $\|e_{\mathbb{C}}\|_2 \le \sigma\sqrt{N}$ the norm bound is:

$$\sqrt{L + L \cdot \ell \cdot \ell + \sigma^2 \cdot N + \sigma^2 \cdot N} = \sqrt{(\ell^2 + 1) \cdot L + 2 \cdot \sigma^2 \cdot N}$$

This results in π_1 smaller than [ADDS21] in orders of magnitude. We discuss the exact parameters and sizes in Section 6.

6 Bandwidth Estimate

We give rough bandwidth estimates in Table 2. We adapt the estimation scripts from [ADDG24] and give our adapted scripts in Appendix B. Our size estimates for ring elements are simply $d \cdot \log q$ bits, except for d_x where we use (a) that we can drop lower order bits since those are drowned by σ' anyway and (b) that we only use d_x additively when computing $d_x + r - c \cdot s$, allowing us to drop many coefficients of d_x since we only want to extract from λ many. It would be possible to amortise the zero-knowledge proofs as in [ADDG24], but we forego this optimisation here.

While these are somewhat rough estimates, they suffice to make good on our claim that the parameters we obtain are practical-*ish*.

7 Threshold Lattice (V)OPRF

As a result of the nice homomorphic properties of [ADDS21], we give lattice-based threshold/distributed VOPRFs for both n-out-of-n and t-out-of-n thresholds, when t is constant.

7.1 Threshold Verifiable Oblivious Pseudorandom Functions

A (t, n) threshold VOPRF is an extension to VOPRFs where instead of having a single server $\mathbb{S}$, there are n servers $\mathbb{S}_0, \ldots, \mathbb{S}_{n-1}$ where any $t \leq n$ servers can collectively generate the PRF output. If $t = n$ i.e. the threshold scheme is n-out-of-n we call it a *distributed* scheme, but we may also call it *full threshold* so that we can discuss n-out-of-n and t-out-of-n together as "threshold". Based on the setting, the initialisation phase can be done by either each $\mathbb{S}_i$ individually or by an outside trusted authority. A threshold verifiable oblivious pseudorandom function (VOPRF) for a keyed function F is then an $n+1$ party protocol between a client $\mathbb{C}$ and n servers $\mathbb{S}_0, \ldots, \mathbb{S}_{n-1}$ consisting of following algorithms:

- Init_S is a protocol run by each $\mathbb{S}_i$ (or a trusted authority), which on input 1^λ outputs a partial secret key sk_i and its public commitment pk_i (or the combined commitment pk).
- Init_C is a protocol run by $\mathbb{C}$, which on input $\{pk\}_{i \in [n]}$ (respectively pk) outputs a *state* indicating acceptance/rejection of the public commitment.
- $\mathsf{Query}_\mathsf{C}$ is a protocol run by $\mathbb{C}$, which on input client input x and *state* outputs a blinded message $\bar{x}$ and a state ρ.
- $\mathsf{Query}_\mathsf{S}$ is a protocol run by $\mathbb{S}_i$ which on input of client's blinded message $\bar{x}$, a subset of participating users $\mathcal{U}$ and a partial key sk_i outputs a blinded partial evaluation $y_{x,i}$.
- $\mathsf{Finalize}$ is a protocol run by $\mathbb{C}$ which on input server's blinded evaluation y_x and public commitments $\{pk_i\}_{i \in \mathcal{U}}$ (or a single pk) and a state ρ outputs the PRF output y.

Some definitions of VOPRF security are not sufficient for the threshold case as $\mathcal{A}$ can corrupt a subset of servers $\mathcal{C}$ along with the client. Similarly $\mathcal{A}$ can engage in concurrent queries for the same input. So we extend unpredictability and one-more PRF security to accommodate a set of corrupted servers and concurrent $\mathsf{Query}_\mathsf{C}$ executions and again define the corresponding games. We also introduce a new algorithm Comb that takes a set of partial output shares and outputs a single combined output.

```
$\mathsf{Exp}^{om\text{-}\mathsf{TUNPRED}}_{\mathsf{PRF}}(\mathcal{A})$
$ctr \coloneqq 0$; $\mathcal{C}, \Gamma \leftarrow \mathcal{A}(\cdot)$; $\mathcal{H} \coloneqq [n] \setminus \mathcal{C}$;
$pk, \{sk_i\}_{i \in [n]} \leftarrow \mathsf{PRF.Init}_{\mathsf{S}}(1^\lambda)$; $\Gamma \leftarrow \mathcal{A}(\Gamma, pk)$
$\forall i \in \mathcal{H}\colon pk_i, sk_i \leftarrow \mathsf{PRF.Init}_{\mathsf{S}}(1^\lambda)$; $\Gamma, \forall i \in \mathcal{C}\colon pk_i \leftarrow \mathcal{A}(\Gamma, \{pk_j\}_{j \in \mathcal{H}})$
$(x^{(0)}, y^{(0)}), \ldots, (x^{(Q)}, y^{(Q)}) \leftarrow \mathcal{A}(\Gamma, \mathcal{O}^{\mathsf{PRF}}, pk, \{sk_i\}_{i \in \mathcal{C}})$
$(x^{(0)}, y^{(0)}), \ldots, (x^{(Q)}, y^{(Q)}) \leftarrow \mathcal{A}(\Gamma, \mathcal{O}^{\mathsf{PRF}}, \{pk_i\}_{i \in [n]}, \{sk_i\}_{i \in \mathcal{C}})$
if $|\mathcal{C}| \geq t$ return false;    if $\exists\, i \neq j$ s.t. $x^{(i)} = x^{(j)}$ return false
return $\wedge_i (y^{(i)} = F_{sk}(x^{(i)})) \wedge (ctr \leq Q)$

$\mathcal{O}^{\mathsf{PRF}}(\bar{x}, \mathcal{U}^{(\tau)})$
if $|\mathcal{U}^{(\tau)}| < t$ return false
$ctr \coloneqq ctr + 1$; $\mathcal{H}_{\mathcal{U}} \coloneqq \mathcal{U}^{(\tau)} \cap \mathcal{H}$; $\forall i \in \mathcal{H}_{\mathcal{U}}\colon y_i \coloneqq \mathsf{PRF.Query}_{\mathsf{S}}(sk_i, \bar{x})$
return $\{y_i\}_{i \in \mathcal{H}_{\mathcal{U}}}$
```

Fig. 4: The experiment $\mathsf{Exp}^{om\text{-}\mathsf{TUNPRED}}_{\mathsf{PRF}}(\mathcal{A})$. Lines in grey are executed if each $\mathbb{S}_i$ generates their own key. Lines in dashed boxes are executed if there is a trusted authority setting up the keys.

Definition 12 (Threshold unpredictability). *A threshold* PRF *is said to be unpredictable if for any PPT adversary $\mathcal{A}$ the probability of the one-more unpredictability depicted in Figure 4 outputting* 1 *is negligible in* λ.

Definition 13 (Threshold one-more TPRF security). *A threshold* PRF *is said to be pseudorandom if for any PPT adversary $\mathcal{A}$ the probability of the one-more pseudorandomness depicted in Figure 5 outputting* 1 *is negligible in* λ.

For both unpredictability and pseudorandomness, we assume a rushing adversary where honest parties send their messages first. The relationship between unpredictability and one-more PRF security also translates into the threshold setting where the unpredictable $\mathcal{A}$ controls the same parties as the PRF adversary $\mathcal{B}$. However, we must take that that unlike in the plain one-more PRF game, here both adversaries have partial inside access to evaluating parties. However, note that $\mathcal{A}$ is queried for its corrupt shares independent of the oracles decision regarding real or random value which prevents $\mathcal{A}$ from trivially winning the game.

Since we are in a multiparty setting, we need NIZKAoKs that provide concurrent security as well as prevent exponential tightness loss as part of the extractability. For which, we rely on NIZKAoKs that are *straight-line extractable* i.e. can extract the witness without rewinding. For some systems, this can be achieved by using the generic transform by Katsumata [Kat21]. However, while the transform is straightforward to use for proofs of $\mathcal{L}_0$ and $\mathcal{L}_2$, this is not the

```
$\mathsf{Exp}^{om\text{-}\mathsf{TPRF}}_{\mathsf{PRF,H}}(\mathcal{A})$
$ctr,\ q \coloneqq 0;\ \mathcal{C}, \mathit{\Gamma} \leftarrow \mathcal{A}(\cdot);\ \mathcal{H} \leftarrow [n] \setminus \mathcal{C}$
$\boxed{pk, \{sk_i\}_{i\in[n]} \leftarrow \mathsf{PRF.Init}_{\mathsf{S}}(1^\lambda);\ \mathit{\Gamma} \leftarrow \mathcal{A}(\mathit{\Gamma}, pk)}$
$\forall i \in \mathcal{H} \colon pk_i, sk_i \leftarrow \mathsf{PRF.Init}_{\mathsf{S}}(1^\lambda)$
$\mathit{\Gamma}, \forall i \in \mathcal{C} \colon pk_i \leftarrow \mathcal{A}(\mathit{\Gamma}, \{pk_j\}_{j\in\mathcal{H}})$
$(i_1, \dots, i_Q, b') \leftarrow$
    $\boxed{\mathcal{A}(\mathit{\Gamma}, pk, \{sk_i\}_{i\in\mathcal{C}}, \mathsf{H}, \mathcal{O}^{\mathsf{PRF}}, \mathcal{O}^{RoR})}$
    $\mathcal{A}(\mathit{\Gamma}, \{pk_i\}_{i\in[n]}, \{sk_i\}_{i\in\mathcal{C}}, \mathsf{H}, \mathcal{O}^{\mathsf{PRF}}, \mathcal{O}^{RoR})$
if $|\mathcal{C}| \geq t$ return false
if $\exists \alpha$ s.t. $i_\alpha \notin [Q]$ return false
if $Q > q$ or $ctr \geq Q$ return false
if $\exists\, \alpha \neq \beta$ s.t. $i_\alpha = i_\beta$ return false
return $b' \coloneqq \bigoplus_{\alpha=1}^{Q} b[i_\alpha]$

$\mathcal{O}^{RoR}(x^{(\tau)}, \mathcal{U}^{(\tau)})$
if $|\mathcal{U}^{(\tau)}| < t$ return false
$q \coloneqq q + 1,\ b[q] \leftarrow \{0,1\}$
$sk \coloneqq \mathsf{Comb}(\mathcal{U}^{(\tau)}, \{sk_i\}_{i\in\mathcal{U}^{(\tau)}})$
$y_0 \leftarrow \{0,1\}^*$
$y_1 \leftarrow \mathsf{H}(x^{(i)}, F_{sk}(x^{(i)}))$
return $y_{b[q]}$

$\mathcal{O}^{\mathsf{PRF}}(\bar{x}, \mathcal{U}^{(\tau)})$
if $|\mathcal{U}^{(\tau)}| < t$ return false
$ctr \coloneqq ctr + 1;\ \mathcal{H}_{\mathcal{U}} \coloneqq \mathcal{U}^{(\tau)} \cap \mathcal{H}$
$\forall i \in \mathcal{H}_{\mathcal{U}} : y_i \coloneqq \mathsf{PRF.Query}_{\mathsf{S}}(sk_i, \bar{x})$
return $\{y_i\}_{i\in\mathcal{H}_{\mathcal{U}}}$
```

Fig. 5: The experiment $\mathsf{Exp}^{om\text{-}\mathsf{TPRF}}_{\mathsf{PRF,H}}(\mathcal{A})$

case for $\mathcal{L}_1$ since we here we rely on LaBRADOR [BS23]. Thus, straight-line extractability for client messages is open. We note that it seems plausible we can apply the "encryption-to-the-sky" paradigm here, see e.g. [AKSY22], encrypting only that small part of the witness that we need extract to avoid blowing up bandwidth costs.

7.2 Case 1: n-out-of-n

We start with the easy case where all n participants are required to generate the pseudorandom output. The n-out-of-n distributed setting can be considered as a *multikey* application of the base scheme. This is also the reason why we used the n-out-of-n protocol with $n = 1$ in the previous sections. Instead of setting a single σ however, we allow each party to choose $\sigma_i \leq \sigma$ for publicly known σ.[7] We have multiple servers $\mathbb{S}_0, \dots, \mathbb{S}_{n-1}$ interacting with single client $\mathbb{C}$, Figure 3 depicts the protocol. For the sake of simplicity of exposure, though, we assume $\sigma_i = \sigma$ for all i for the rest of this section. Correctness follows from the underlying protocol

[7] We will expand on this idea when we discuss our t-out-of-n construction.

where $\lfloor r + d_x - c \cdot s \rceil_p$

$$
\begin{aligned}
&= \left\lfloor r + \sum d_{x,i} - \sum c_i \cdot s \right\rceil_p \\
&= \left\lfloor r + c_x \cdot \sum k_i + \sum e_{\mathbb{S},i} - s \cdot \sum (a \cdot k_i + e_i) \right\rceil_p \\
&= \left\lfloor r + a \cdot s \sum k_i + e_{\mathbb{C}} \sum k_i + \mathbf{a}_x[0] \sum k_i + \sum e_{\mathbb{S},i} - a \cdot s \sum k_i - s \cdot \sum e_i \right\rceil_p \\
&= \left\lfloor r + \mathbf{a}_x[0] \sum k_i + e_{\mathbb{C}} \sum k_i + \sum e_{\mathbb{S},i} - s \sum e_i \right\rceil_p \\
&= \left\lfloor r + \frac{p}{q} \cdot \mathbf{a}_x[0] \cdot k \right\rceil_p
\end{aligned}
$$

with $k = \sum\limits_{i \in [n]} k_i$. Correctness then follows from Theorem 2. Note however, since $k = \sum\limits_{i \in [n]} k_i$ for $k_i \in \mathcal{R}_{\chi_{\sigma_i}}$ we have to scale the combined parameter by a factor of $\sqrt{n}$. We then have the following lemma.

Lemma 6. *Let $k_i \leftarrow \mathcal{R}_{\chi_{\sigma_i}}$, $k = \sum\limits_{i \in [n]} k_i$, and $q \gg p \cdot \sigma \cdot \sqrt{L \cdot n} \cdot N$. Then the function $F_k(x) := \lfloor \mathbf{a}_x[0] \cdot k + r \rceil_p$ is a PRF under $\mathsf{dRLWE}_{q,N,\sigma\sqrt{n},\sigma\sqrt{n}}$ assumption.*

Security against malicious servers i.e. obliviousness and verifiability is immediate. Each $\mathbb{S}_i$ receives the same client input, which is the same as in single party case. For completeness we state them here:

Theorem 4. *Let σ and N be $\mathsf{poly}(\lambda)$. Let $\mathsf{dRLWE}_{q,N,\sigma,\sigma}$ be hard. Let $(\mathbb{P}_0, \mathbb{V}_0)$, $(\mathbb{P}_1, \mathbb{V}_1)$ be straight-line extractable NIZKAoKs for languages $\mathcal{L}_0, \mathcal{L}_1$, then the protocol in Figure 3 is oblivious against any PPT adversary $\mathcal{A}$ controlling all $\mathbb{S}_i$.*

Theorem 5. *Let σ and N be $\mathsf{poly}(\lambda)$. Let $\beta = 2\, n \cdot \sigma^2 \cdot N + \sigma' \cdot \sqrt{n \cdot N}$ and $q/p \gg \beta$. Let $(\mathbb{P}_0, \mathbb{V}_0), (\mathbb{P}_2, \mathbb{V}_2)$ be straight-line extractable NIZKAoKs for languages $\mathcal{L}_0, \mathcal{L}_2$, H_r be a random oracle, and Q_{H} be number of queries made to such oracle. Then the protocol in Figure 3 is verifiable against any PPT adversary $\mathcal{A}$ controlling all $\mathbb{S}_i$.*

The interesting security goal is threshold unpredictability when there is a collusion between the malicious client and some subset of servers. We show how the protocol given in Figure 3 is unpredictable. We note that we implicitly assume that a malicious $\mathbb{C}$ sends the same message to honest servers. This assumption can be removed with an initial round of consistency check among the servers which we omit here.

Theorem 6. *Let σ and N be $\mathsf{poly}(\lambda)$. Let $\mathsf{dRLWE}_{q,N,\sigma,\sigma}$ be hard, and $\frac{q}{2p} \gg \sigma' \geq (L \cdot \sqrt{N} + (\sqrt{n} + 1) \cdot \sigma) \cdot \sigma \cdot N \cdot \sqrt{Q \cdot N}$ for a number of queries made Q. Let $(\mathbb{P}_0, \mathbb{V}_0), (\mathbb{P}_1, \mathbb{V}_1), (\mathbb{P}_2, \mathbb{V}_2)$ be straight-line extractable NIZKAoKs for languages $\mathcal{L}_0, \mathcal{L}_1, \mathcal{L}_2$, then the distributed VOPRF protocol defined in Figure 3 is threshold unpredictable against malicious clients controlling up to $n - 1$ servers.*

Proof. In the Random Oracle model, if the input of H at the end of the protocol is unpredictable, then so is the output of the protocol. Hence, we show that the client-derived input to H is unpredictable. Similar to the proof of Theorem 3, we show $\mathcal{A}$ controlling $\mathbb{C}^*$ and a subset of servers $\mathcal{C}$ cannot find an unqueried request-response pair $(x^{(\tau)}, y_x^{(\tau)})$ with all but negligible probability in λ.

Hybrid_0*:* This is the real execution of the protocol where the $\mathcal{A}$ makes Q queries to servers. The adversary $\mathcal{A}$ first corrupts a set of servers $\mathcal{C}$ with fewer than $t = n$ elements and the rest of the servers denoted with $\mathcal{H}$ behave honestly. The honest parties sample a key share k_i and each party outputs a commitment c_i alongside a proof of correct computation. For $\tau \in [Q]$, $\mathcal{A}$ sends a query $(c_x^{(\tau)}, \pi_1^{(\tau)})$ based on some x for which honest servers compute $(d_x^{(\tau)}, \pi_2^{(\tau)})$ if $\pi_1^{(\tau)}$ verifies and aborts otherwise. For corrupted servers, $\mathcal{A}$ can send arbitrary shares as long as $\pi_2^{(\tau)}$ verifies for each of them. The adversary $\mathcal{A}$ then computes $y_x^{(\tau)}$ based on $d_{x,i}^{(\tau)}$ and $x^{(\tau)}$ (resp. $y_x^{(\tau)}$) is added to the set $\mathcal{X}$ (resp. $\mathcal{Y}$). At the end, $\mathcal{A}$ outputs (x^*, y_x^*) and wins the game if $x^* \notin \mathcal{X}$ and c_x^* generated on x^* evaluates to y_x^*. The advantage of $\mathcal{A}$ is the probability of $\mathcal{A}$ winning in the threshold unpredictability game where $\mathcal{U}$ are all n servers.

Hybrid_1*:* Hybrid_1 is exactly like Hybrid_0 except how proofs by the server computed. Instead of honestly generating $\mathsf{crs}_{0,j}, \mathsf{crs}_{2,j}$ for $j \in \mathcal{H}$, and computing and $\pi_{0,j}, \pi_{2,j}^{(\tau)}$ each honest server calls the simulator for relative proof systems. Hybrid Hybrid_1 is then indistinguishable from Hybrid_0 by the ZK property of the underlying ZKAoKs.

Hybrid_2*:* Hybrid_2 is exactly like Hybrid_1 except the honest parties try to extract a witness for the corrupted parties. During Init_S, after the honest parties extract $\{k_i, e_i\}_{i \in \mathcal{C}}$ from $\pi_{0,i}$ using the extractor for the underlying ZKAoK, aborts if the extraction fails. Similarly during the **Query** phase after honest servers receive $c_x^{(\tau)}, \pi_1^{(\tau)}$, it calls the extractor to obtain $(x^{(\tau)}, e_{\mathbb{C}}^{(\tau)}, s^{(\tau)})$. By the extractability of the underlying ZKAoKs Hybrid_1 is exactly like Hybrid_2 unless the extraction fails, and Hybrid_1 and Hybrid_2 are indistinguishable.

Hybrid_3*:* $\mathbb{S}_{i'}$ changes how $d_{x,i'}^{(\tau)}$ is computed for $i' \in \mathcal{H}$. Upon receiving $c_x^{(\tau)}$, fix an index i'. For every other honest party, the computation continues as before. The server $\mathbb{S}_{i'}$ then derives the combined key $k = \sum_{i \in [n]} k_i$ and error $e = \sum_{i \in [n]} e_i$ and samples $e_x^{(\tau)} \leftarrow \mathcal{E}_{\mathbf{a}_0, \mathbf{a}_1, x, \sigma}$ based on the extracted $x^{(\tau)}$. The server $\mathbb{S}_{i'}$ finally computes $\bar{d}_{x,i'}^{(\tau)} = c_x^{(\tau)} \cdot k_{i'} + e_{\mathbb{S},i'}^{(\tau)} + e_x^{(\tau)} - e_{\mathbb{C}}^{(\tau)} \cdot k + e \cdot s^{(\tau)}$ sends $\bar{d}_{x,i'}^{(\tau)}$ as its share. The rest follows as before.

The difference between Hybrid_2 and Hybrid_3 is in the error term of $\mathbb{S}_{i'}$'s share where there is an added term of $e_x^{(\tau)} - e_{\mathbb{C}}^{(\tau)} \cdot k + e \cdot s^{(\tau)}$. Using the same Rènyi argument in Hybrid_3 of Theorem 3, we conclude if $\mathcal{A}$ has a winning probability in Hybrid_2, it also does a winning probability polynomial of said probability in

Hybrid_3. Since $\|k\|_2 \leq \sigma\sqrt{n}$ however, σ' has an increased factor compared to the single party case.[8]

Hybrid_4*:* We stop using combined key k for deriving $\bar{d}_x^{(\tau)}$. Each honest server maintains a received list for $(x^{(\tau)}, y_q)$. After receiving and verifying $c_x^{(\tau)}$ checks if the extracted $x^{(\tau)}$ has been queried before. If $(x^{(\tau)}, y_q)$ exists in $\mathsf{received}$, $\mathbb{S}_{i'}$ retrieves y_q from the list and samples $\bar{e}_{\mathbb{S},i'}^{(\tau)} \leftarrow \mathcal{R}_{\chi_{\sigma'}}$ and returns $\bar{d}_{x,i'}^{(\tau)} = c \cdot s^{(\tau)} - c_x^{(\tau)} \cdot \sum_{j \neq i'} k_j + \bar{e}'^{(\tau)} + y_q$. If $x^{(\tau)}$ is queried for the first time, $\mathbb{S}_{i'}$ first samples an output y and then uniformly samples a y_q such that $y_q \leftarrow \mathcal{R}_q \cap (q/p \cdot y + \mathcal{R}_{\leq q/2p})$. Each server records $(x^{(\tau)}, y_q)$ and computes $\bar{d}_{x,i}^{(\tau)}$ the same. In Hybrid_3 $\bar{d}_{x,i'}^{(\tau)}$ can be rewritten as:

$$\begin{aligned}
\bar{d}_{x,i'}^{(\tau)} &= c_x^{(\tau)} \cdot k_{i'} + e_{\mathbb{S},i'}^{(\tau)} + e_x^{(\tau)} - e_{\mathbb{C}}^{(\tau)} \cdot k + e \cdot s^{(\tau)} \\
&= c_x^{(\tau)} \cdot k - c_x^{(\tau)} \cdot \sum_{j \neq i'} k_j + e_{\mathbb{S},i'}^{(\tau)} + e_x^{(\tau)} - e_{\mathbb{C}}^{(\tau)} \cdot k + e \cdot s^{(\tau)} \\
&= a \cdot s^{(\tau)} \cdot k + e_{\mathbb{C}}^{(\tau)} \cdot k + \mathbf{a}_x^{(\tau)}[0] \cdot k - c_x^{(\tau)} \cdot \sum_{j \neq i'} k_j + e_{\mathbb{S},i'}^{(\tau)} + e_x^{(\tau)} - e_{\mathbb{C}}^{(\tau)} \cdot k + e \cdot s^{(\tau)} \\
&= c \cdot s^{(\tau)} - c_x^{(\tau)} \cdot \sum_{j \neq i'} k_j + \mathbf{a}_x^{(\tau)}[0] \cdot k + e_x^{(\tau)} + e_{\mathbb{S},i'}^{(\tau)}
\end{aligned}$$

Using the same argument in Hybrid_4 for Theorem 3, Hybrid_3 and Hybrid_4 are indistinguishable.

Hybrid_5*:* We modify honest parties' shares so that each of them includes additional error terms $e_i'^{(\tau)} \leftarrow \mathcal{R}_{\chi_\sigma}$ and $e_{x,i}'^{(\tau)} \leftarrow \mathcal{E}_{\mathbf{a}_0,\mathbf{a}_1,x,\sigma}$, and the adjusted share $\bar{d}_{x,i'}^{(\tau)}$ includes the substraction of these shares $-\sum_{i \neq i' \in \mathcal{H}} (e_i'^{(\tau)} + e_{x,i}'^{(\tau)})$. The rest follows as before. In Hybrid_4 each honest party outside i' computes their share as $d_{x,i}^{(\tau)} = c_x^{(\tau)} \cdot k_i + e_{\mathbb{S},i}^{(\tau)}$ whereas in Hybrid_5 $d_{x,i}^{(\tau)} = c_x^{(\tau)} \cdot k_i + e_{\mathbb{S},i}^{(\tau)} + e_i'^{(\tau)} + e_{x,i}'^{(\tau)}$. The difference is then how error terms are distributed for two Gaussians of parameter σ' with two different centers. Using a similar argument to Hybrid_3,[9] we conclude if $\mathcal{A}$ can win in Hybrid_4 with some probability, it also has a winning probability polynomial in the said probability in Hybrid_5.

Hybrid_6*:* We remove the dependency on partial key shares for honest parties. Except i', each honest server samples a uniformly random $u_i^{(\tau)} \leftarrow \mathcal{R}_q$ and computes their share as $\bar{d}_{x,i}^{(\tau)} := u_i^{(\tau)} + e_{\mathbb{S},i}^{(\tau)}$ instead. Similarly, $\mathbb{S}_{i'}$ defines its share as $\bar{d}_{x,i'}^{(\tau)} = c \cdot s^{(\tau)} - c_x^{(\tau)} \cdot \sum_{j \in \mathcal{C}} k_j + \bar{e}_{\mathbb{S},i'}^{(\tau)} + y_q - \sum_{i \neq i' \in \mathcal{H}} u_i^{(\tau)}$. The rest proceeds as before.

[8] Note that this hybrid also changes the combined $d_x^{(\tau)}$ since the additional error term carries over. By Lemma 1 the error term is statistically close to a Gaussian with parameter $\sigma'\sqrt{n}$ using a similar Rènyi argument as above, but with easier to satisfy parameters, already satisfied by the parameters considered in the main text.

[9] Note that we do not need to consider the combined $d_x^{(\tau)}$ as $\mathbb{S}_{i'}$ adjusts its share based on the added error terms.

In Hybrid_5 after the addition of noise terms, each partial evaluation is $d_{x,i}^{(\tau)} = c_x^{(\tau)} \cdot k_i + e_{\mathbb{S},i}^{(\tau)} + e_i'^{(\tau)} + e_{x,i}'^{(\tau)} = (a \cdot s^{(\tau)} + e_{\mathbb{C}}^{(\tau)}) \cdot k_i + e_i'^{(\tau)} + \mathbf{a}_x^{(\tau)}[0] \cdot k_i + e_{x,i}'^{(\tau)} + e_{\mathbb{S},i}^{(\tau)}$. Using the same argument to Hybrid_4 of Theorem 3, $\mathbf{a}_x^{(\tau)}[0] \cdot k_i + e_{x,i}'^{(\tau)}$ is indistinguishable from uniform. Replacing these terms with uniform ones, $d_{x,i}^{(\tau)}$ in Hybrid_5 and $u_i^{(\tau)} + e_{\mathbb{S},i}^{(\tau)}$, consequently Hybrid_5 and Hybrid_6 are indistinguishable.

Hybrid_7*:* Now that the function evaluation does not rely on the combined key honest servers stop deriving key shares k_j altogether. During initialization each server samples random $c_i \leftarrow \mathcal{R}_q$. Then by the hardness of $\mathsf{dRLWE}_{q,N,\sigma\sqrt{n},\sigma\sqrt{n}}$, c in Hybrid_6 and Hybrid_7 are indistinguishable.

Since every reply to $\mathcal{A}$ is freshly generated and independent from the secret combined key k and the honest key shares, they are unpredictable. Thus we conclude the proof. $\square$

Now that the protocol is threshold unpredictable, we can also argue it has threshold one-more PRF security.

Corollary 2. *Let σ and N be $\mathsf{poly}(\lambda)$. Let $\mathsf{dRLWE}_{q,N,\sigma,\sigma}$ be hard and $\frac{q}{2p} \gg \sigma' \geq (L \cdot \sqrt{N} + (\sqrt{n}+1) \cdot \sigma) \cdot \sigma \cdot N \cdot \sqrt{Q \cdot N}$ for a number of queries made Q. Let $(\mathbb{P}_0, \mathbb{V}_0), (\mathbb{P}_1, \mathbb{V}_1), (\mathbb{P}_2, \mathbb{V}_2)$ be straight-line extractable NIZKAoKs for languages $\mathcal{L}_0, \mathcal{L}_1, \mathcal{L}_2$ and H, H_r be hash functions modeled as random oracles, then if the distributed VOPRF protocol defined in Figure 6 is threshold unpredictable, it also has threshold one-more PRF security against any PPT adversary $\mathcal{A}$ controlling $\mathbb{C}$ and a subset of servers $\mathcal{C}$ of size at most $t-1$.*

7.3 Case 2: t-out-of-n

We now switch to the more interesting case of arbitrary thresholds i.e. t-out-of-n with $t \leq n$. We cannot directly use the additive homomorphism of the underlying operation but tweak it to our setting. Dealing with t-out-of-n shares in a lattice setting is not trivial against malicious adversaries, and we here work around known issues by assuming a trusted setup. As mentioned above, we consider this a realistic assumption for OPRFs as most use cases of threshold OPRFs utilise the functionality to prevent a single point of failure during execution rather than to achieve execution among untrusted parties. Still, this is a limitation of this work.

Similarly, we assume that in the context of distributed OPRFs, neither n nor $\binom{n}{t}$ is large. Hence we can consider a separate set of keys for different thresholds of servers since we only have $\binom{n}{t}$ of such sets. However, trivially combining all partial keys will result in $\binom{n}{t}$ different combined keys and consequently public commitments which now (i) requires the client to know which servers are replying for correctness (ii) the client will receive different PRF outputs for the same input for different threshold sets.

Instead, we make use of the setting we are in and consider a different way of representing these $\binom{n}{t}$ sets. We delegate key generation to a trusted authority which in return allows us to use different *additive shares* of the same combined key. On a high level, key generation proceeds as follows: The trusted authority first samples a combined key from the combined distribution of individual keys. For every threshold set $\mathcal{T}$, the authority fixes indices i_a and i_b. For every server i outside i_a and i_b in the set, it samples partial keys from a smaller distribution. For i_b, it samples a partial key from a wider distribution. The trusted authority then computes the key of i_a as the difference of the combined key and $t-1$ partial keys and rejects the key with a certain probability. If rejected, the process starts again for the threshold set. If not the trusted authority proceeds to the next set.

The first key idea is that we can choose distributions of keys for differing parameters as long as each of them are bounded from below for RLWE security and above for correctness and noise drowning. Hence, choosing a key with some $\sigma_{\mathsf{L}} > \sigma$ allows us to control the rejection probability. The second key idea is that while the last share is not exactly the same as the sampled keys, if rejected correctly it is statistically indistinguishable from the distribution of i_b's key and thus still secure. This rejection is similar to the inefficient variant Dilithium-G signature discussed in [DOTT22]. Since the last key share is computed as $k - \sum_{i \neq i_a} k_{i,\mathcal{T}} = k - \sum_{i \neq i_a, i_b} k_{i,\mathcal{T}} - k_{i_b,\mathcal{T}}$, we can treat it as a Gaussian centered around $k - \sum_{i \neq i_a, i_b} k_{i,\mathcal{T}}$ and use Lemmas 1 to 3 to find correct M, t, T to make the last share within a negligible statistical distance of a sampled Gaussian with parameter σ_{L} by Lemma 3. For $\sigma = 3.2$ and $N = 4096$, a threshold of 5 parties can have $\sigma_{\mathsf{L}} = 1024$ with $M = 131$ repetitions per key with all but negligible probability 2^{-102}. We emphasise that for large sizes of $\binom{n}{t}$ this can result in long key generation times but does not affect the actual PRF evaluation. For security argument, we will assume $\binom{n}{t}$ is $\mathsf{poly}(\lambda)$.

Remark 4. Our approach can be considered as a variant of *replicated secret sharing* [ISN89] using qualified sets. Hence, we could consider algorithms such as in [CDI05] for key generation instead. However, we highlight some key differences: (a) Secret sharing is done for long term keys rather than online randomness which renders the $\binom{n}{t}$ overhead more acceptable. (b) Our final shares are with overwhelming probability from specific Gaussian distributions rather than uniformly random in order to preserve the structure of the protocol. (c) Since shares of Shamir secret sharing are arbitrarily large any advantage regarding easy conversion into Shamir secret sharing is not relevant in our context.

One downside to this approach, we cannot show verifiability for individual partial evaluations as public commitments $k_{i,\mathcal{T}}$ for all subsets $\mathcal{T}$ of size t do not exist. Hence it is not possible for $\mathbb{S}_i$ to prove $d_{x,i}$ is computed correctly with respect to a partial key $k_{i,\mathcal{U}}$. One solution to this for the trusted authority to publish public commitments for each $k_{i,\mathcal{T}}$ which however would require $t \cdot \binom{n}{t}$ commitments to be published and for $\mathbb{C}$ to know which subset of users are participating in PRF execution. This is worse than the trivial construction of having $\binom{n}{t}$ different combined keys.

Instead, we combine the *cut-and-choose* type of approach in [ADDS21] with a weaker proof system. The intuition is while we cannot prove that correct $k_{i,\mathcal{U}}$

is used for generating $d_{x,i}$, we can verify that a small $k_{i,\mathcal{U}}$ is used consistently across multiple evaluations. If one of these evaluations points can be checked with respect to a publicly known value, we can argue that every evaluation used the correct combined key. This does not guarantee each individual partial evaluation is done correctly but assures $\mathbb{C}$ the final output is correct.

We change the protocol as follows. During setup there is a public fixed input x' and its evaluation $y_{x'}$ (under the key k) known both to the client and the servers. During $\mathsf{Query_C}$, instead of a single input x, $\mathbb{C}$ blinds two inputs x_0, x_1. To do that, the client first decides on a random bit b'. For $i \neq b'$, the client uses the private input $x_i = x$ for some $x \in \{0,1\}^L$ and for $i = b'$, $x_i = x'$. The client $\mathbb{C}$ then runs the computation for two c_x values and sends $c_{x,i}, \pi_{1,i}$ pairs. Each server $\mathbb{S}_j$ then runs partial evaluations on each of them and sends a proof $\pi_{2,j}$ to prove that the same short $k_{j,\mathcal{U}}$ have been used for computing all $c_{x,i}$ values. During $\mathsf{Finalize}$, $\mathbb{C}$ first verifies $\pi_{2,i}$ and then computes two different y_x values and checks if $y_{x,b'} = y_{x'}$. If everything verifies, $\mathbb{C}$ uses $y_{x,i}$ for $i \neq b'$ as its output.

We depict this t-out-of-n VOPRF construction with a trusted setup in Figure 6. Note that $\pi_{2,i}$ are computed with a different proof system $\mathbb{P}'_2, \mathbb{V}'_2$ for language $\mathcal{L}'_2$ since it's slightly different from $\mathbb{P}_2, \mathbb{V}_2$. It can however still be initiated with the same proof systems discussed in Section 5.

Correctness follows from the linearity of the additive secret sharing. Obliviousness is once again immediate as the client's input to the t servers do not change. We first show that the protocol described has verifiability:

Theorem 7. *Let σ, N, and $\binom{n}{t}$ be $\mathsf{poly}(\lambda)$. Let $\beta = 2\,t \cdot \sigma^2 \cdot N + \sigma' \cdot \sqrt{t \cdot N}$ and $q/p \gg \beta$. Let $(\mathbb{P}'_2, \mathbb{V}'_2)$ be straight-line extractable NIZKAoK for language $\mathcal{L}'_2$, H_r be a random oracle, and Q_H be number of queries made to such oracle. Let*

$$N \cdot \left(\log q - \log\left(\sigma \cdot \sqrt{(t+1) \cdot N}\right) - \log\left(\frac{q}{2p}\right)\right) > \lambda.$$

Then the protocol in Figure 6 is verifiable against any PPT adversary $\mathcal{A}$ controlling a subset of servers $\mathcal{C}$ of size at most $t-1$.

Proof. In verifiability game, the challenger will abort and $\mathcal{A}$ will trivially lose if the checks during $\mathsf{Finalize}$ fail. For $\mathcal{A}$ to win, $y_{x,i}$ for $i \neq b'$ derived by $\mathbb{C}$ must be different from the actual PRF. Similar to Theorem 2, $\mathcal{A}$ can only find an x^* that would cause r^* to force an incorrect evaluation only with negligible probability. If $\pi_{2,j}$ verifies for each $j \in \mathcal{U}$, then there exists short $\{k_{j,\mathcal{U}}\}_{j \in \mathcal{U}}$ used in each of the $\{d_{x,j,i}\}_{i \in \{0,1\}, j \in \mathcal{U}}$. Then if $y_{x,b'} = y_{x'}$ each server $\mathbb{S}_j$ knows $k^*_{j,\mathcal{U}}$ for some $k^\star$ where $k^\star = \sum_{j \in \mathcal{U}} k^*_{j,\mathcal{U}}$.

Since $\pi_{2,j}$ verifies, we have $\|k^*_{j,\mathcal{U}}\|_\infty \leq \|k^*_{j,\mathcal{U}}\|_2 \leq \sigma \cdot \sqrt{N}$ consequently $\|k^*\|_\infty \leq \sigma \cdot \sqrt{t \cdot N}$, and $\|e_{\mathbb{S},j,i}\|_\infty \leq \|e_{\mathbb{S},j,i}\|_2 \leq \sigma' \cdot \sqrt{N}$. Since $q/p \gg 2\,t \cdot$

CRS SetUp:

- $\mathbf{a}_0, \mathbf{a}_1 \leftarrow \mathcal{R}_q^{1\times\ell}$.
- $a \leftarrow \mathcal{R}_q^{1\times\ell}$.
- Sample crs_1, crs'_2 for $\mathbb{P}_1, \mathbb{P}'_2$.
- Fix an input $x' \in \{0,1\}^L$

Initialisation:

- Init_S: A trusted authority executes:
  - $k \leftarrow \mathcal{R}_{\chi_\sigma}$, $e \leftarrow \mathcal{R}_{\chi_\sigma}$.
  - $c \leftarrow a \cdot k + e \mod q$, $r' \leftarrow \mathsf{H}_r(x', c) \in \mathcal{R}_q^{1\times\ell}$.
  - $\mathbf{a}_{x'} := \mathbf{a}_{x_0} \cdot G^{-1}(\ldots(\mathbf{a}_{x_{L-2}} \cdot G^{-1}(\mathbf{a}_{x_{L-1}}))\ldots) \mod q$.
  - $y_{x'} := \mathbf{a}_{x'}[0] \cdot k + r'$.
  - For every threshold set $\mathcal{T}$ of size t:
    - $*$ Fix indices $i_a, i_b \in \mathcal{T}$.
    - $*$ For $i = i_b, k_{i_b,\mathcal{T}} \leftarrow \mathcal{R}\chi_{\sigma_\mathsf{L}}$
    - $*$ For $i \in \mathcal{T} \setminus \{i_a, i_b\}$, $k_{i,\mathcal{T}} \leftarrow \mathcal{R}\chi_\sigma$.
    - $*$ $k_{i_a,\mathcal{T}} := k - \sum\limits_{i \neq i_a \in \mathcal{T}} k_{i,\mathcal{T}}$
    - $*$ With a probability $1 - \min(1, D_{\sigma_\mathsf{L}}(k_{i_a,\mathcal{T}}))/D_{k-\sum k_{i,\mathcal{T}},\sigma_\mathsf{L}}(k_{i_a,\mathcal{T}})$ repeat the process for $\mathcal{T}$.

  Send $k_{i,\mathcal{T}}$ to server $\mathbb{S}_i$ for every $\mathcal{T}$ that $\mathbb{S}_i$ is part of, broadcast $c, y_{x'}$.

Query:

1. $\mathsf{Query}_\mathsf{C}$: $\mathbb{C}$ executes the following with the input $(x \in \{0,1\}^L, \mathsf{crs}_1, \mathsf{crs}'_2)$
   - $b' \leftarrow \{0,1\}$.
   - For each index $i \in \{0,1\}$:
     - If $i = b'$, $x_i = x'$ otherwise $x_i = x$
     - $s_i \leftarrow \mathcal{R}_{\chi_\sigma}$, $e_{\mathbb{C},i} \leftarrow \mathcal{R}_{\chi_\sigma}^{1\times\ell}$.
     - $\mathbf{a}_{x,i} := \mathbf{a}_{i,x_0} \cdot G^{-1}(\ldots(\mathbf{a}_{i,x_{L-2}} \cdot G^{-1}(\mathbf{a}_{i,x_{L-1}}))\ldots) \mod q$.
     - $c_{x,i} \leftarrow a \cdot s_i + e_{\mathbb{C},i} + \mathbf{a}_{x,i}[0] \mod q$.
     - $\pi_{1,i} \leftarrow \mathbb{P}_1(x_i, s_i, e_{\mathbb{C},i} : \mathsf{crs}_1, c_{x,i}, a, \mathbf{a}_0, \mathbf{a}_1)$.

   and broadcasts $\{(c_{x,i}, \pi_{1,i})\}_{i\in\{0,1\}}$ to every $\mathbb{S}_j$.
2. $\mathsf{Query}_\mathsf{S}$: $\mathbb{S}_j \in \mathcal{U}$ executes the following for each index i after receiving $\{(c_{x,i}, \pi_{1,i})\}_{i\in\{0,1\}}$
   - $b \leftarrow \mathbb{V}_1(\mathsf{crs}_1, c_{x,i}, \mathbf{a}_0, \mathbf{a}_1, \pi_{1,i})$, output abort if $b = 0$.
   - $e_{\mathbb{S},j,i} \leftarrow \mathcal{R}_{\chi_{\sigma'}}$, $d_{x,j,i} := c_x \cdot k_{j,\mathcal{U}} + e_{\mathbb{S},j,i} \mod q$.
   - $\pi_{2,j} \leftarrow \mathbb{P}'_2(k_{j,\mathcal{U}}, e_{\mathbb{S},i}, : \mathsf{crs}'_2, \{d_{x,j,i}, c_{x,i}\}_{i\in M}, a)$.

   and sends $(\{(d_{x,j,i})\}_{i\in\{0,1\}}, \pi_{2,j})$ to $\mathbb{C}$ and outputs $\perp$.
3. $\mathsf{Finalize}$: $\mathbb{C}$ finally executes the following after receiving $(\{(d_{x,j,i})\}_{i\in\{0,1\}})$ from $\mathbb{S}_j \in \mathcal{U}$.
   - $b_j \leftarrow \mathbb{V}'_2(\mathsf{crs}_2, \{d_{x,j,i}, c_{x,i}\}_{i\in\{0,1\}}, \pi_{2,j})$, output abort with i if $b_j = 0$.
   - $d_{x,i} := \sum\limits_j d_{x,j,i}$, $r_i \leftarrow \mathsf{H}_r(x_i, c) \in \mathcal{R}_q^{1\times\ell}$, $y_{x,i} := \lfloor d_{x,i} + r_i - c \cdot s_i \rceil_p$.
   - If $y_{x,i} \neq y_{x'}$ for $i = b'$ abort.
   - $y \leftarrow \mathsf{H}(x_i, y_{x,i})$ for $i \neq b'$.

   and outputs y.

Fig. 6: t-out-of-n VOPRF Construction with Trusted Setup

$\sigma^2 \cdot N + \sigma' \cdot \sqrt{t \cdot N}$, we have:

$$\left\lfloor \sum_j d_{x,j,b'} + r' - c \cdot s \right\rceil_p = \left\lfloor \mathbf{a}_x[0] \cdot k^* + r' + \left(e_{\mathbb{C}} \cdot k^* - e \cdot s + \sum_j e_{\mathbb{S},j,b'} \right) \right\rceil_p$$
$$= \lfloor \mathbf{a}_x[0] \cdot k^* + r' \rceil_p$$

with overwhelming probability. Then, $\mathcal{A}$ can only win if it can find $k^* \neq k$ such that $\lfloor \mathbf{a}_{x'}[0] \cdot k^* + r' \rceil_p = \lfloor \mathbf{a}_{x'}[0] \cdot k + r' \rceil_p$. Rearranging the terms we get $[\mathbf{a}_{x'}[0] \mid 1] \cdot \begin{bmatrix} k^* - k \\ e' \end{bmatrix} = \mathbf{0} \mod q$ for some $e' \in \mathcal{R}_q$, $\|e'\|_\infty \leq q/(2p)$. By our assumption we have that there are $\left(2\,\sigma \cdot \sqrt{t \cdot N}\right)^N \cdot (q/(2p))^N$ possible choices for $(k^* - k, e')$ but over the randomness of $\mathbf{a}_0[0], \mathbf{a}_1[0]$, the probability of obtaining 0 is $1/q^N$. Thus, with high probability such a k^* does not exist. Hence if the evaluation for $i = b'$ is correct, $k^* = k$.

Since the same $k^*_{j,\mathcal{U}}$ and consequently the same $k^* = k$ are used for computing $y_{x,i}$, $i \neq b'$; the evaluation must also be correct if $y_{x,b'}$ is correct. This concludes the proof. $\square$

Remark 5. Our proof above relies on the absences of any SIS solution to $[\mathbf{a}_{x'} \mid 1]$. First, our bound is rather loose, by first extracting a worst-case ℓ_∞ bound from the ℓ_2 bound established by the NIZKAoK and then constructing a box of solutions with this ℓ_∞ bound. A tighter approximation would be accomplished by bounding the number of integer points inside the ℓ_2 ball established by the NIZKAoK directly. Moreover, an alternative approach, giving smaller parameters, is to instead rely on a computational SIS assumption wrt the infinity norm and with unbalanced entries. This problem was considered in [ZYF+20,ESZ22]. Indeed, even assuming an infinity norm bound of $q/4$ for all components, the difficulty of the resulting SIS instance is comparable to λ as given in Table 2 according to the lattice estimator [APS15].

We now show how the protocol has threshold unpredictability.

Theorem 8. *Let σ, N, and $\binom{n}{t}$ be $\mathsf{poly}(\lambda)$. Let $t' = o(\log(N))$, $T \leq \sigma\sqrt{(t-1)N}$ and $\alpha = T/\sigma_{\mathsf{L}}$. Let $M = \exp(t'/\alpha + 1/2 \cdot \alpha^{-2})$. Let $\mathsf{dRLWE}_{q,N,\sigma,\sigma}$ be hard, and $\frac{q}{2p} \gg \sigma' \geq (L \cdot \sqrt{N} + 2 \cdot \sigma) \cdot \sigma \cdot N \cdot \sqrt{Q \cdot N}$ for the number of queries made Q. Let $(\mathbb{P}_1, \mathbb{V}_1), (\mathbb{P}'_2, \mathbb{V}'_2)$ be straight-line extractable NIZKAoKs for language $\mathcal{L}_1$, $\mathcal{L}'_2$. Then the (t, n) threshold OPRF protocol defined in Figure 6 is threshold unpredictable against malicious clients controlling up to $t - 1$ servers.*

Proof. We can think of our t-out-of-n construction as an n-out-of-n construction with $n = t$, here we adapt the proof of Theorem 6 to our case. We once again assume honest $\mathbb{S}_i$ receive the same input. We discuss what we change in each hybrid.

Hybrid_0*:* Since $\mathcal{U}$ are a set of t users for each query, for each query we define $\mathcal{C}_{\mathcal{U}} := \mathcal{U}^{(\tau)} \cap \mathcal{C}$ and $\mathcal{H}_{\mathcal{U}} := \mathcal{U}^{(\tau)} \cap \mathcal{H}$. The rest is the same.

Hybrid_1*:* Since the keys are distributed by a trusted authority, and there is no $\pi_{0,j}$ and only $\pi_{2,j}^{(\tau)}$ has to be simulated; Hybrid_1 is exactly the same as Hybrid_0 by the zero knowledge property of the underlying NIZKAoK.

Hybrid_2*:* Since the key setup is handled by a trusted authority (i) there is only a single commitment and (ii) all key shares are accessible via the trusted authority. Then the honest parties are not required to extract key shares of the corrupted parties to access them. The input of $\mathbb{C}$ however still needs to be extracted from $\pi_1^{(\tau)}$ which has the same argument as Hybrid_2 for n-out-of-n.

Hybrid_3*:* To fix $d_{x,i'}^{(\tau)}$ for $i' \in \mathcal{H}_{\mathcal{U}}$ (note that i' is not necessarily the same as i_a or i_b), instead of all corrupted partial keys only the ones in set $\mathcal{U}$ are used. The rest is the same. Also note that since k is sampled from $\mathcal{R}_{\chi_\sigma}$, the bound for σ' is the same as the single party case.

Hybrid_4*:* Again, we use the set specific partial keys for adjusting the share, the rest is exactly the same as Hybrid_4. Since the distribution of $\mathbf{a}_x$ and e_x are the same, the argument is the same.

Hybrid_5*:* Hybrid_5 is exactly the same with the exact argument. Note however that the honest parties can include i^a and i^b which requires the error to be sampled with parameter σ_{L}.

Hybrid_6*:* Hybrid_6 is exactly the same with the exact argument. Again, since honest parties can include i^a and i^b, we rely on the security of $\mathsf{dRLWE}_{q,N,\sigma_{\mathsf{L}},\sigma_{\mathsf{L}}}$ and $\mathsf{dRLWE}_{q,N,\sigma,\sigma}$ instead.

Hybrid_7*:* Since there is only one commitment $\mathbf{c}$ to the combined key k we replace it with a uniform value and give garbage key shares for each threshold $\mathcal{C}$ is part of. The argument is the same except we use $\mathsf{dRLWE}_{N,q,\sigma,\sigma}$ since k is sampled from $\mathcal{R}_{\chi_\sigma}$. Again, since every reply to $\mathcal{A}$ is freshly generated and independent from the secret combined key k and the honest key shares, they are unpredictable. Thus we conclude the proof. □

Finally, since the protocol is threshold unpredictable, we can also argue it has threshold one-more PRF security.

Corollary 3. *Let σ, N, and $\binom{n}{t}$ be $\mathsf{poly}(\lambda)$. Let $t' = o(\log(N))$, $T \leq \sigma\sqrt{(t-1)N}$ and $\alpha = T/\sigma_{\mathsf{L}}$. Let $M = \exp(t'/\alpha + 1/2 \cdot \alpha^{-2})$. Let $\mathsf{dRLWE}_{q,N,\sigma,\sigma}$ be hard, and $\frac{q}{2p} \gg \sigma' \geq (L \cdot \sqrt{N} + 2 \cdot \sigma) \cdot \sigma \cdot N \cdot \sqrt{Q \cdot N}$ for the number of queries made Q. Let $(\mathbb{P}_1, \mathbb{V}_1), (\mathbb{P}'_2, \mathbb{V}'_2)$ be straight-line extractable NIZKAoKs for language $\mathcal{L}_1, \mathcal{L}'_2$ and H, H_r be hash functions modeled as random oracles then if the (t, n) threshold OPRF protocol defined in Figure 6 is threshold unpredictable, it also has threshold one-more PRF security against any PPT adversary $\mathcal{A}$ controlling $\mathbb{C}$ and a subset of servers $\mathcal{C}$ of size at most $t-1$.*

Acknowledgements

We thank reviewers for their valuable comments and pointing out the issue with both [ADDS21] and our unpredictability proofs. This work was supported in part by UKRI grant EP/Y02432X/1. Part of this work was done while Kamil Doruk Gur was at SandboxAQ and supported in part by NSF award CNS-2154705.

References

ADDG24. Martin R. Albrecht, Alex Davidson, Amit Deo, and Daniel Gardham. Crypto dark matter on the torus - oblivious PRFs from shallow PRFs and TFHE. In Marc Joye and Gregor Leander, editors, *EUROCRYPT 2024, Part VI*, volume 14656 of *LNCS*, pages 447–476. Springer, Cham, May 2024. 1, 1, 1, 1.2, 11, 6

ADDS21. Martin R. Albrecht, Alex Davidson, Amit Deo, and Nigel P. Smart. Round-optimal verifiable oblivious pseudorandom functions from ideal lattices. In Juan Garay, editor, *PKC 2021, Part II*, volume 12711 of *LNCS*, pages 261–289. Springer, Cham, May 2021. 1, 1, 1.1, 1.1, 1.1, 3, 1.2, 2.6, 4, 3, 3, 3, 4, 4, 4, 6, 5, 7, 7.3, 7.3

AG24. Martin R. Albrecht and Kamil Doruk Gür. Verifiable oblivious pseudorandom functions from lattices: Practical-ish and thresholdisable. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part IV*, volume 15487 of *LNCS*, pages 205–237. Springer, Singapore, December 2024. ⋆

AKSY22. Shweta Agrawal, Elena Kirshanova, Damien Stehlé, and Anshu Yadav. Practical, round-optimal lattice-based blind signatures. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 39–53. ACM Press, November 2022. 7.1

APRR24. Navid Alamati, Guru-Vamsi Policharla, Srinivasan Raghuraman, and Peter Rindal. Improved alternating-moduli PRFs and post-quantum signatures. Cryptology ePrint Archive, Report 2024/582, 2024. 1

APS15. Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of Learning with Errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015. 3, 5

Bas24. Andrea Basso. POKE: A framework for efficient PKEs, split KEMs, and OPRFs from higher-dimensional isogenies. Cryptology ePrint Archive, Report 2024/624, 2024. 1

BDFH24. Ward Beullens, Lucas Dodgson, Sebastian Faller, and Julia Hesse. The 2Hash OPRF framework and efficient post-quantum instantiations. Cryptology ePrint Archive, Report 2024/450, 2024. 1

BIP$^+$18. Dan Boneh, Yuval Ishai, Alain Passelègue, Amit Sahai, and David J. Wu. Exploring crypto dark matter: New simple PRF candidates and their applications. In Amos Beimel and Stefan Dziembowski, editors, *TCC 2018, Part II*, volume 11240 of *LNCS*, pages 699–729. Springer, Cham, November 2018. 1, 1

BKW20. Dan Boneh, Dmitry Kogan, and Katharine Woo. Oblivious pseudorandom functions from isogenies. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part II*, volume 12492 of *LNCS*, pages 520–550. Springer, Cham, December 2020. 1

BLR$^+$18. Shi Bai, Tancrède Lepoint, Adeline Roux-Langlois, Amin Sakzad, Damien Stehlé, and Ron Steinfeld. Improved security proofs in lattice-based cryptography: Using the Rényi divergence rather than the statistical distance. *Journal of Cryptology*, 31(2):610–640, April 2018. 1.2, 2.5

BP14. Abhishek Banerjee and Chris Peikert. New and improved key-homomorphic pseudorandom functions. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 353–370. Springer, Berlin, Heidelberg, August 2014. 1.1, 1.1, 2.6, 6

BS23. Ward Beullens and Gregor Seiler. LaBRADOR: Compact proofs for R1CS from module-SIS. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part V*, volume 14085 of *LNCS*, pages 518–548. Springer, Cham, August 2023. 1.2, 5, 7.1

BV15. Zvika Brakerski and Vinod Vaikuntanathan. Constrained key-homomorphic PRFs from standard lattice assumptions - or: How to secretly embed a circuit in your PRF. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *TCC 2015, Part II*, volume 9015 of *LNCS*, pages 1–30. Springer, Berlin, Heidelberg, March 2015. 1.1

CDI05. Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 342–362. Springer, Berlin, Heidelberg, February 2005. 4

CGGI20. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, January 2020. 1

CHL22. Sílvia Casacuberta, Julia Hesse, and Anja Lehmann. SoK: Oblivious pseudorandom functions. In *2022 IEEE European Symposium on Security and Privacy*, pages 625–646. IEEE Computer Society Press, June 2022. 1, 3

DGH$^+$21. Itai Dinur, Steven Goldfeder, Tzipora Halevi, Yuval Ishai, Mahimna Kelkar, Vivek Sharma, and Greg Zaverucha. MPC-friendly symmetric cryptography from alternating moduli: Candidates, protocols, and applications. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 517–547, Virtual Event, August 2021. Springer, Cham. 1

DGS$^+$18. Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy pass: Bypassing internet challenges anonymously. *PoPETs*, 2018(3):164–180, July 2018. 1

DOTT22. Ivan Damgård, Claudio Orlandi, Akira Takahashi, and Mehdi Tibouchi. Two-round n-out-of-n and multi-signatures and trapdoor commitment from lattices. *Journal of Cryptology*, 35(2):14, April 2022. 7.3

ECS$^+$15. Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. The pythia PRF service. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 547–562. USENIX Association, August 2015. 9, 10, 2.5

ESZ22. Muhammed F. Esgin, Ron Steinfeld, and Raymond K. Zhao. MatRiCT$^+$: More efficient post-quantum private blockchain payments. In *2022 IEEE Symposium on Security and Privacy*, pages 1281–1298. IEEE Computer Society Press, May 2022. 5

FIPR05. Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 303–324. Springer, Berlin, Heidelberg, February 2005. 1

FOO23. Sebastian Faller, Astrid Ottenhues, and Johannes Ottenhues. Composable oblivious pseudo-random functions via garbled circuits. Cryptology ePrint Archive, Report 2023/1176, 2023. 1

GdKQ$^+$23. Phillip Gajland, Bor de Kock, Miguel Quaresma, Giulio Malavolta, and Peter Schwabe. Swoosh: Practical lattice-based non-interactive key exchange. Cryptology ePrint Archive, Report 2023/271, 2023. 1.2

HMR23. Lena Heimberger, Fredrik Meisingseth, and Christian Rechberger. Oprfs from isogenies: Designs and analysis. Cryptology ePrint Archive, Paper 2023/639, 2023. https://eprint.iacr.org/2023/639. 1

ISN89. Mitsuru Ito, Akira Saito, and Takao Nishizeki. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 72(9):56–64, 1989. 4

JKK14. Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 233–253. Springer, Berlin, Heidelberg, December 2014. 1

JKKX16. Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 276–291, 2016. 1

JKR18. Stanislaw Jarecki, Hugo Krawczyk, and Jason Resch. Threshold partially-oblivious PRFs with applications to key management. Cryptology ePrint Archive, Report 2018/733, 2018. 3

JKX18. Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 456–486. Springer, Cham, April / May 2018. 1

JL09. Stanislaw Jarecki and Xiaomin Liu. Efficient oblivious pseudorandom function with applications to adaptive OT and secure computation of set intersection. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 577–594. Springer, Berlin, Heidelberg, March 2009. 1

Kat21. Shuichi Katsumata. A new simple technique to bootstrap various lattice zero-knowledge proofs to QROM secure NIZKs. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 580–610, Virtual Event, August 2021. Springer, Cham. 7.1

KBR13. Sriram Keelveedhi, Mihir Bellare, and Thomas Ristenpart. DupLESS: Server-aided encryption for deduplicated storage. In Samuel T. King, editor, *USENIX Security 2013*, pages 179–194. USENIX Association, August 2013. 1

KCM24. Novak Kaluderovic, Nan Cheng, and Katerina Mitrokotsa. A post-quantum distributed OPRF from the legendre PRF. Cryptology ePrint Archive, Report 2024/544, 2024. 1

Leh19. Anja Lehmann. ScrambleDB: Oblivious (chameleon) pseudonymization-as-a-service. *PoPETs*, 2019(3):289–309, July 2019. 8

LNP22. Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Maxime Planccon. Lattice-based zero-knowledge proofs and applications: Shorter, simpler, and more general. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 71–101. Springer, Cham, August 2022. 1.2, 5, B.3

LPR10. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23. Springer, Berlin, Heidelberg, May / June 2010. 2

LSS14. Adeline Langlois, Damien Stehlé, and Ron Steinfeld. GGHLite: More efficient multilinear maps from ideal lattices. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 239–256. Springer, Berlin, Heidelberg, May 2014. 4

Lyu12. Vadim Lyubashevsky. Lattice signatures without trapdoors. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 738–755. Springer, Berlin, Heidelberg, April 2012. 2.1, 2, 3

MP13. Daniele Micciancio and Chris Peikert. Hardness of SIS and LWE with small parameters. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 21–39. Springer, Berlin, Heidelberg, August 2013. 1

S^+23. William Stein et al. *Sage Mathematics Software Version 10.2*. The Sage Development Team, 2023. http://www.sagemath.org. A

SHB21. István András Seres, Máté Horváth, and Péter Burcsi. The legendre pseudorandom function as a multivariate quadratic cryptosystem: Security and applications. Cryptology ePrint Archive, Report 2021/182, 2021. 1

SSTX09. Damien Stehlé, Ron Steinfeld, Keisuke Tanaka, and Keita Xagawa. Efficient public key encryption based on ideal lattices. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 617–635. Springer, Berlin, Heidelberg, December 2009. 2

YAZ^+19. Rupeng Yang, Man Ho Au, Zhenfei Zhang, Qiuliang Xu, Zuoxia Yu, and William Whyte. Efficient lattice-based zero-knowledge arguments with standard soundness: Construction and applications. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 147–175. Springer, Cham, August 2019. 1.1, 1.2, 5

ZYF^+20. Jiang Zhang, Yu Yu, Shuqin Fan, Zhenfeng Zhang, and Kang Yang. Tweaking the asymmetry of asymmetric-key cryptography on lattices: KEMs and signatures of smaller sizes. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020, Part II*, volume 12111 of *LNCS*, pages 37–65. Springer, Cham, May 2020. 5

A Proving Statements in $\mathbb{Z}_q[X]/(X^N+1)$ using $\mathbb{Z}_q[X]/(X^{64}+1)$

We would like to prove statements in some ring $\mathcal{R}_q \coloneqq \mathbb{Z}_q[X]/(X^d+1)$ using LaBRADOR where we would like to use $\mathcal{R}'_q \coloneqq \mathbb{Z}_q[X]/(X^z+1)$ for $z = 64$. We thus have to express $c = a \cdot b$ for $a, b \in \mathcal{R}_q$ as arithmetic over $\mathcal{R}'_q$. Additions are immediate. We write $a \coloneqq \sum a_i \cdot X^i$ as a vector of dimension d/z, i.e. $\in {\mathcal{R}'_q}^{d/z}$ by setting:

$$\mathbf{a}_i = \sum_{j=0}^{z-1} a_{j\cdot\frac{d}{z}+i} \cdot X^j.$$

We also define functions performing (negacyclic) ring $\mathsf{rotr} : \mathcal{R}'_q \to \mathcal{R}'_q$ and vector $\mathsf{rotv} : (\mathcal{R}'_q)^{d/z} \to (\mathcal{R}'_q)^{d/z}$ rotations as

$$\mathsf{rotr}(a) \coloneqq X \cdot a \bmod X^z + 1 \equiv a_{z-1} + \sum_{i=0}^{z-2} a_i \cdot X^{i+1}.$$

and

$$\mathsf{rotv}(\mathbf{a}) \coloneqq (\mathsf{rotr}(\mathbf{a}_{d/z-1}), \mathbf{a}_0, \ldots, \mathbf{a}_{d/z-2}).$$

Finally, we define the multiplication matrix $\in (\mathcal{R}'_q)^{d/z\times d/z}$ of $a \in \mathcal{R}_q$ as the matrix where the first column is $\mathbf{a}$, the second column is $\mathsf{rotv}(\mathbf{a})$, the third column in $\mathsf{rotv}(\mathsf{rotv}(\mathbf{a}))$ etc. Then it holds that $\mathbf{A} \cdot \mathbf{b} \equiv \mathbf{c} \bmod q$. We given an example implemented in SageMath [S+23] below.

```
def tovec(a, d, z):
    r = [sum(a[j*d//z+i]*X^j for j in range(z)) for i in range(d//z)]
    return vector(R, r)

def rotr(e, z):
    return -e[z-1] + sum(e[i]*X^(i+1) for i in range(z-1))

def rotv(v, d, z):
    return vector([rotr(v[-1], z)] + list(v[:-1]))

def tomat(a, d, z):
    avec = tovec(a, d, z)
    amat = matrix(R, d//z, d//z)
    for i in range(d//z):
        amat[i,:] = avec
        avec = rotv(avec, d, z)
    return amat.T

d = 128
z = 8
P = PolynomialRing(QQ, ["a%d"%i for i in range(d)] +  ["b%d"%i for i in range(d)])
R.<X> = PolynomialRing(P)

a = sum(P.gens()[:d][i]*X^i for i in range(d))
b = sum(P.gens()[d:][i]*X^i for i in range(d))
c = a*b % (X^d + 1)

cres = (tomat(a, d, z)*tovec(b, d, z)) % (X^z + 1)
tovec(c, d, z) - cres == 0
```

B Code

B.1 Parameter Selection

The script is also attached.

```
"""
OPRF Parameter Selection.
"""

from sage.all import ceil, log, sqrt

from utils import lwe_security_level


class HashableDict(dict):
    def __hash__(self):
        return hash(frozenset(self.items()))


base_sigma = 3.2


class OPRFParams:
    def __init__(
        self,
        d=4096,
        sigma_s=base_sigma,
        sigma_e=base_sigma,
        # Other
        L=128,
        epsilon=100,
        Q=2**32,
    ):
        """
        :param d: RLWE dimension (power of two)
        :param sigma_s: secret standard deviation
        :param sigma_e: error standard deviation
        :param L: PRF value length
        :param epsilon: statistical correctness: 1-2^-ε
        :param Q: number of permitted queries

        """

        self.d = d
        self.sigma_s = sigma_s
        self.sigma_e = sigma_e
        self.L = L
        # we aim for 1-2^-ε overall
        self.Q = Q
        self.epsilon = epsilon

        self.ell = self.epsilon  # temporary

        self.sigma_ = (
            (self.L * sqrt(self.d) + 2 * sigma_e)
            * sigma_s
            * self.d
            * sqrt(self.Q * self.d)
        )

        # q//4 > σ' · 2^ε
        self.ell = ceil(log(self.sigma_, 2) + self.epsilon + 2)
        self.q = 2**self.ell

    def __call__(
        self,
        verbose=False,
        check_security=True,
    ):
        if check_security:
            level = self.check_assumptions(verbose=verbose)
            return level
        else:
            print(self.list_assumptions())
            return None

    def __repr__(self):
        return (
            f"OPRF(d={self.d}, q=2^{self.ell}, "
            + "Q=2^{float(log(self.Q,2.)):.1f})"
        )

    def check_assumptions(self, verbose=False):
        from estimator.estimator import LWE, ND, SIS

        Xs = ND.DiscreteGaussian(self.sigma_s)
        Xe = ND.DiscreteGaussian(self.sigma_e)

        lwe = LWE.Parameters(n=self.d, q=self.q, Xs=Xs, Xe=Xe)
        lwe_level = lwe_security_level(
            lwe,
            verbose=verbose,
        )

        return lwe_level

    def list_assumptions(self):
        s = []
        s.append(
            f"LWE.Parameters(n={self.d}, q=2^{float(log(self.q,2)):.0f}, "
            + f" Xs=ND.DiscreteGaussian(2^{float(log(self.sigma_s,2)):.1f}))"
            + f" Xe=ND.DiscreteGaussian(2^{float(log(self.sigma_e,2)):.1f}))"
        )
        return "\n".join(s)
```

B.2 Size Estimates

The script is also attached.

```
"""
OPRF Size Estimates.
"""

from sage.all import sqrt, ceil, log, cached_function, round
from lnp import LNP
from labrador import LaBRADOR
from parameters import OPRFParams as Parameters

OPRF_Signal = Parameters(d=4096, Q=2**4)
OPRF_4096_16 = Parameters(d=4096, Q=2**16)
OPRF_4096_32 = Parameters(d=4096, Q=2**32)
OPRF_8192_64 = Parameters(d=8192, Q=2**64)

argsv = [
    (OPRF_Signal, "-Signal"),
    (OPRF_4096_16, "-4096-16"),
    (OPRF_4096_32, "-4096-32"),
    (OPRF_8192_64, "-8192-64"),
]


def _kb(v):
    """
    Convert bits to kilobytes.
    """
    return round(float(v / 8.0 / 1024.0), 1)


def _mb(v):
    """
    Convert bits to megabytes.
    """
    return round(float(v / 8.0 / 1024.0 / 1024.0), 1)


def oprf_com_sizekb(params):
    """
    The size of the OPRF commitment in KB.

    :param params: OPRF parameters

    """

    # size of `c`
    return _kb(params.d * params.ell)


@cached_function
def oprf_com_proof_sizekb(params, lnp_params=None, labrador_params=False):
    """
    Well-formedness proof of the OPRF commitment using LNP22.

    :param params: OPRF parameters
    :param lnp_params: Parameters passed to LNP22 proof system
    :param labrador_params: Parameters passed to LaBRADOR proof system

    """
    if lnp_params is None:
        lnp_params = {"logq1": params.ell, "d": 32}
    lnp = LNP(**lnp_params)

    sigma = max(params.sigma_s, params.sigma_e)
    sizekb, q, b_d = lnp(
        alpha=params.sigma_s * sqrt(params.d),
        alpha_e=sqrt(
            params.sigma_s**2 * params.d
            + params.sigma_e**2 * params.d * params.ell
            + params.d
        ),
        alpha_d=1,
        ell=0,
        m1=params.d / lnp.d,
        ce=(params.ell + 1) * params.d / lnp.d + 1,
        k_bin=0,
        bounds_to_prove=[
            sqrt(sigma**2 * params.d + sigma**2 * params.d * params.ell)
        ],
        do_labrador=labrador_params,
        approximate_norm_proof=False,
    )

    return sizekb


def oprf_ct0_sizekb(params):
    """
    OPRF request size in kilobytes.
    """

    # size of ‘c_x‘
    return _kb(params.d * params.ell)


# @cached_function
def oprf_ct0_proof_sizekb(
    params, lnp_params=None, labrador_params=(True, 10, 4), labrador_only=False
):
    """
    Well-formedness proof of OPRF request per req. with LNP22 and LaBRADOR.

    :param params: OPRF parameters
    :param labrador_params: Parameters passed to LaBRADOR proof system

    """
    L = params.L
    ell = params.ell
    d = params.d
    sigma = max(params.sigma_s, params.sigma_e)

    if labrador_only:
        greedy, base, length = labrador_params
        labrador = LaBRADOR(logq=params.ell, d=1)
        beta = sqrt(L + L * ell**2 + 2 * sigma**2 * d)

        n = d / labrador.d * (L + L * ell * ell + 2)

        if greedy:
            f = labrador.greedy
        else:
            f = labrador

        sizekb, recursion = f(
            n=n,
            r=d / labrador.d,
            beta=beta,
            base=base,
            length=length,
            verbose=None,
        )
        return round(sizekb, 1)
    else:
        if lnp_params is None:
            lnp_params = {"logq1": params.ell, "d": 128}
        lnp = LNP(**lnp_params)

        # Exact norm bounds we need to prove
        ve = L + L * ell * ell + 2

        # Treating the entire x_0, ..., x_L, B_0, ..., B_{L-1}, s, e_\C, and the quadratic
        # terms x_0 B_1, ..., x_{L-2} B_{L-1}
        alpha = sqrt(
            L
            + L * ell * ell
            + sigma**2 * d
            + sigma**2 * d * ell
            + (L - 1) * ell * ell
        )

        alpha_e = sqrt(
            L
            + L * ell * ell
            + sigma**2 * d
            + sigma**2 * d * ell
            + (L - 1) * ell * ell
            + lnp.d * ve
            + sigma**2 * d
        )

        sizekb, q, b_d = lnp(
            alpha=alpha,
            alpha_e=alpha_e,
            alpha_d=1,
            ell=0,
            # same as Labrador witness size + the quadratic terms
            m1=d / lnp.d * (L + L * ell * ell + 2 + 2 * ell * ell),
            ce=(L + L * ell * ell + 2 + (L - 1) * ell * ell) * d / lnp.d + ve,
            k_bin=0,
            bounds_to_prove=[
                1,  # L + L* params.ell * params.ell times
                sigma * sqrt(d),  # for s
                sigma * sqrt(d),  # for e_client
            ],
            do_labrador=labrador_params,
            approximate_norm_proof=False,
        )
        return round(sizekb, 1)


def oprf_ct1_sizekb(params, compress_ct1=True):
    """
    Response size in kilobytes.
    """

    # size of ‘d_x‘

    # there’s not much point in sending anything but the top 10 bits of the
    # noise σ’

    # we use ‘d_x‘ additively only, so we may simply send the first ‘L‘
    # components
    if compress_ct1:
        return _kb(params.L * (params.ell - (log(params.sigma_, 2) - 10)))
    else:
        return _kb(params.d * params.ell)


@cached_function
def oprf_ct1_proof_sizekb(params, lnp_params=None, labrador_params=False):
    """
    Well-formedness proof of c1 [LNP22].

    :param params: OPRF parameters
    :param lnp_params: Parameters passed to LNP22 proof system
    :param labrador_params: Parameters passed to LaBRADOR proof system

    """
    if lnp_params is None:
        lnp_params = {"logq1": 12 + params.ell, "d": 32}
    lnp = LNP(**lnp_params)

    sigma = max(params.sigma_s, params.sigma_e)

    sizekb, q, b_d = lnp(
        alpha=params.sigma_s * sqrt(params.d),
        alpha_e=sqrt(
            sigma**2 * params.d
            + params.sigma_**2 * params.d * params.ell
            + params.d
        ),
        alpha_d=1,
        ell=0,
        m1=params.d / lnp.d,
        ce=(params.ell + 1) * params.d / lnp.d + 1,
        k_bin=0,
        bounds_to_prove=[
            sqrt(sigma**2 * params.d + params.sigma_**2 * params.d * params.ell)
        ],
        do_labrador=labrador_params,
        approximate_norm_proof=False,
    )

    return sizekb


def oprf_online_sizekb(params):
    """

    :param params: OPRF parameters
    :param amortise:

    """
    r = oprf_ct0_sizekb(params)
    r += oprf_ct0_proof_sizekb(params)
    r += oprf_ct1_sizekb(params)
    r += oprf_ct1_proof_sizekb(params)
    return round(r, 1)


def oprf_offline_sizekb(params):
    r = oprf_com_sizekb(params)
    r += oprf_com_proof_sizekb(params)
    return round(r, 1)


def oprf(params, suffix="", queue=None):
    """

    :param params: OPRF parameters
    :param suffix: Suffix for printing

    """

    ret = f"""% OPRF{suffix} SIZES
  /oprf{suffix}/logq/.initial={params.ell},
  /oprf{suffix}/d/.initial={params.d},
  /oprf{suffix}/epsilon/.initial={params.epsilon},
  /oprf{suffix}/logQ/.initial={ceil(log(params.Q,2.))},
  /oprf{suffix}/logsigmaprime/.initial={ceil(log(params.sigma_,2.))},
  /oprf{suffix}/secpar/.initial={params()},
  /oprf{suffix}/com/sizekb/.initial={oprf_com_sizekb(params)},
  /oprf{suffix}/com/proof/sizekb/.initial={oprf_com_proof_sizekb(params)},
  /oprf{suffix}/ct0/sizekb/.initial={oprf_ct0_sizekb(params)},
  /oprf{suffix}/ct0/proof/sizekb/.initial={oprf_ct0_proof_sizekb(params)},
  /oprf{suffix}/ct1/sizekb/.initial={oprf_ct1_sizekb(params)},
  /oprf{suffix}/ct1/proof/sizekb/.initial={oprf_ct1_proof_sizekb(params)},
  /oprf{suffix}/online/sizekb/.initial={oprf_online_sizekb(params)},
  /oprf{suffix}/offline/sizekb/.initial={oprf_offline_sizekb(params)},"""

    if queue is not None:
        queue.put(ret)
    return ret


def print_all(parallel=True):
    from multiprocessing import Process, Queue

    resv = []

    if not parallel:
        for args in argsv:
            resv.append(oprf(args))
        for res in resv:
            print(res)
    else:
        for args in argsv:
            q = Queue()
            p = Process(target=oprf, args=args + (q,))
            p.start()
            resv.append((p, q))
        for p, q in resv:
            p.join()
            print(q.get())
```

B.3 Size Estimates for [LNP22]

The script is also attached.

```
from sage.all import (
    log,
    ceil,
    sqrt,
    is_prime,
    divisors,
    is_even,
    exp,
    get_verbose,
)

from utils import find_mlwe_level, sis_delta, _kb
from labrador import LaBRADOR, LABRADOR_SLACK


class LNP:
    def __init__(
        self,
        secpar=128,
        logq1=66,
        logq=None,
        nbofdiv=1,
        d=128,
        l=2,
        kappa=2,
        eta=59,
    ):
        """
        :param secpar: Security parameter

        Defining the log of the proof system modulus,
        finding true values will come later:

        :param logq1: log of the smallest prime divisor of q
        :param logq: log of the proof system modulus q
        :param nbofdiv: Number of prime divisors of q, usually 1 or 2

        :param d: Dimension of ‘R = Z[X]/(X^d + 1)‘

        :param l: Number of irreducible factors of ‘X^d + 1‘ modulo each ‘q_i‘,
                  we assume each ‘q_i = 2l+1 (mod 4l)‘
        :param kappa: Maximum coefficient of a challenge. We want
                     ‘|\\chal| = (2κ+1)^(d/2) >= 2^secpar‘
        :param eta: Heur. bound on ‘\\sqrt[2k](‖ σ_{-1}(c^k)·c^k ‖_1)‘ for ‘k = 32‘
        """
        self.secpar = secpar
        self.target_rhf = 1.00436  # TODO compute from secpar
        self.nbofdiv = nbofdiv
        self.logq1 = logq1
        self.logq = self.logq1 if logq is None else logq
        # number of repetitions for boosting soundness, we assume lambda is even
        self.repetitions = 2 * ceil(self.secpar / (2 * self.logq1))

        self.d = d
        self.l = l
        self.kappa = kappa
        self.eta = eta

    def __call__(
        self,
        alpha,
        alpha_e,
        alpha_d,
        ell,
        m1,
        ce,
        k_bin,
        bounds_to_prove,
        gamma_1=41,
        gamma_2=1.1,
        gamma_e=16,
        gamma_d=1,
        approximate_norm_proof=True,
        do_labrador=(4, 5),
    ):
        """
        TODO describe function

        :param alpha:
        :param alpha_e:
        :param alpha_d:
        :param ell:
        :param m1:
        :param ce:
        :param k_bin:
        :param bounds_to_prove:
        :param gamma_1: Rejection sampling for s1
        :param gamma_2: Rejection sampling for s2
        :param gamma_e: Rejection sampling for Rs^(e)
        :param gamma_d: Rejection sampling for R's^(d),
               ignored when approximate_norm_proof=0
        :param approximate_norm_proof: Boolean
        :param do_labrador: Run LaBRADOR with this base, length or not if False

        """

        approximate_norm_proof = int(approximate_norm_proof)
        ve = len(bounds_to_prove)

        # Setting the standard deviations, apart from stddev_2
        stddev_1 = gamma_1 * self.eta * sqrt(alpha**2 + ve * self.d)
        stddev_e = gamma_e * sqrt(337) * alpha_e
        stddev_d = gamma_d * sqrt(337) * alpha_d

        nu = 1  # randomness vector s2 with coefficients between -nu and nu

        hardness, dim_mlwe = find_mlwe_level(
            nu,
            self.d,
            self.logq,
            secpar=self.secpar,
            verbose=get_verbose() >= 2,
        )
        if get_verbose():
            print(f"Security level for MLWE: {hardness}")

        # Finding an appropriate Module-SIS dimension dim_sis
        dim_sis = 0  # dimension of the Module-SIS problem
        D = 0  # dropping low-order bits of t_A
        gamma = 0  # dropping low-order bits of w

        # bound on bar{z}_1
        bound_1 = 2 * stddev_1 * sqrt(2 * (m1 + ve) * self.d) * LABRADOR_SLACK

        def sis_okay(m2, gamma=0, D=0):
            # set stddev_2 with the current candidate for dim_sis
            stddev_2 = gamma_2 * self.eta * nu * sqrt(m2 * self.d)

            # bound on bar{z}_2 = (bar{z}_{2,1},bar{z}_{2,2})
            bound_2 = (
                2 * stddev_2 * sqrt(2 * m2 * self.d)
                + 2**D * self.eta * sqrt(dim_sis * self.d)
                + gamma * sqrt(dim_sis * self.d)
            )
            # bound on the extracted MSIS solution
            bound = 4 * self.eta * sqrt(bound_1**2 + bound_2**2)
            return (
                bound < 2**self.logq
                and sis_delta(dim_sis * self.d, 2**self.logq, bound)
                < self.target_rhf
            )

        # 1/ Search for dim_sis
        while True:
            dim_sis += 1
            # we use the packing optimisation from Section 5.3
            m2 = (
                dim_mlwe
                + dim_sis
                + ell
                + self.repetitions / 2
                + 256 / self.d
                + 1
                + approximate_norm_proof * 256 / self.d
                + 1
            )
            if sis_okay(m2):
                break

        # 2/ Given dim_sis, find the largest possible γ
        gamma = 2**self.logq  # initialisation
        while True:  # searching for right gamma
            gamma /= 2  # decrease the value of gamma
            if sis_okay(m2, gamma):
                break

        q, q1 = self.qf(gamma)

        # 3/ Given dim_sis and γ, find the largest possible D
        D = self.logq  # initialisation
        while True:  # searching for right D
            D -= 1  # decrease the value of D
            if (
                sis_okay(m2, gamma, D)
                and 2 ** (D - 1) * self.kappa * self.d < gamma
            ):
                break

        # Checking knowledge soundness conditions from Theorem 5.3
        t = 1.64  # TODO magic constants!
        b_e = 2 * sqrt(256 / 26) * t * stddev_e * LABRADOR_SLACK

        if q < 41 * ce * self.d * b_e:
            raise ValueError("Cannot use Lemma 2.9.")

        if q <= b_e**2 + b_e * sqrt(k_bin * self.d):
            raise ValueError(
                "Cannot prove E_bin*s + v_bin has binary coefficients."
            )

        if q <= b_e**2 + b_e * sqrt(ve * self.d):
            raise ValueError("Cannot prove all x_i have binary coefficients.")

        for i, bound in enumerate(bounds_to_prove):
            if q <= 3 * bound**2 + b_e**2:
                raise ValueError(f"Cannot prove ‖E_i*s - v_i‖ ≤ β_{i}")

        rep_rate = (
            2
            * exp(14 / gamma_1 + 1 / (2 * gamma_1**2))
            * exp(1 / (2 * gamma_2**2))
            * exp(1 / (2 * gamma_e**2))
            * (
                (1 - approximate_norm_proof)
                + approximate_norm_proof * exp(1 / (2 * gamma_d**2))
            )
        )

        b_d = 2 * 14 * stddev_d  # TODO: magic constants 2 and 14

        # Knowledge soundness error from Theorem 5.3
        soundness_error = (
            2 * 1 / (2 * self.kappa + 1) ** (self.d / 2)
            + q1 ** (-self.d / self.l)
            + q1 ** (-self.repetitions)
            + 2 ** (-128)
            + approximate_norm_proof * 2 ** (-256)
        )

        full_size = (
            dim_sis * self.d * (self.logq - D)
            + (
                ell
                + 256 / self.d
                + 1
                + approximate_norm_proof * 256 / self.d
                + 2 * self.repetitions
                + 2
            )
            * self.d
            * self.logq
        )

        stddev_2 = gamma_2 * self.eta * nu * sqrt(m2 * self.d)
        challenge = ceil(log(2 * self.kappa + 1, 2)) * self.d
        short_size1 = (m1 + ve) * self.d * (ceil(log(stddev_1, 2) + 2.57)) + (
            m2 - dim_sis
        ) * self.d * (ceil(log(stddev_2, 2) + 2.57))
        short_size2 = 256 * (
            ceil(log(stddev_e, 2) + 2.57)
        ) + approximate_norm_proof * 256 * (ceil(log(stddev_d, 2) + 2.57))
        hint = 2.25 * dim_sis * self.d

        sizekb = _kb(full_size + challenge + short_size1 + short_size2 + hint)

        if do_labrador:
            labrador = LaBRADOR(logq=self.logq, d=1)
            greedy, base, length = do_labrador
            if greedy:
                f = labrador.greedy
            else:
                f = labrador
            labrador_size, recursion = f(
                n=m1 + ve,
                r=self.d / labrador.d,
                beta=bound_1 / (2 * LABRADOR_SLACK),
                base=base,
                length=length,
                verbose=get_verbose() >= 2,
            )
            labrador_saving = (
                _kb((m1 + ve) * self.d * (ceil(log(stddev_1, 2) + 2.57)))
                - labrador_size
            )
            sizekb = (
                _kb(full_size + challenge + short_size1 + short_size2 + hint)
                - labrador_saving
            )
        if get_verbose() >= 1:
            print(f"Proof system modulus q: {q}")
            print(f"Smallest prime divisor q_1 of q: {q1}")
            print(f"Parameter γ for dropping low-order bits of w: {gamma}")
            print(f"Parameter D for dropping low-order bits of t_A : {D}")
            print(f"Module-SIS dimension: {dim_sis}")
            print(f"Module-LWE dimension: {dim_mlwe}")
            print(f"Length of the randomness vector s2: {m2}")
            print(
                f"Standard deviation stddev_1: 2^{float(log(stddev_1, 2)):.2f}"
            )
            print(
                f"Standard deviation stddev_2: 2^{float(log(stddev_2, 2)):.2f}"
            )
            print(
                f"Standard deviation stddev_e: 2^{float(log(stddev_e, 2)):.2f}"
            )
            print(
                f"Standard deviation stddev_d: 2^{float(log(stddev_d, 2)):.2f}"
            )

            print(f"Repetition rate: {rep_rate:2}")
            print(
                f"Knowledge soundness error: 2^{ceil(log(soundness_error, 2))}"
            )

            print(f"Full-sized polynomials {_kb(full_size)}kB.")
            print(f"Challenge c in {_kb(challenge)}kB")
            print(
                f"Short polynomials: {_kb((short_size1 + short_size2 + hint))}kB"
            )

        return sizekb, q, b_d

    def qf(self, gamma):
        # we need q1 to be congruent to 2l+1 modulo 4l
        q1 = 4 * self.l * int(2**self.logq1 / (4 * self.l)) + (2 * self.l + 1)
        while True:
            q1 = q1 - 4 * self.l
            while not is_prime(q1):  # we need q1 to be prime
                q1 -= 4 * self.l
            if (
                self.nbofdiv == 1
            ):  # if number of divisors of q is 1, then q = q1
                q = q1
            else:
                # we need q2 to be congruent to 2l+1 modulo 4l
                q2 = (
                    4 * self.l * int(2 ** (self.logq) / (4 * self.l * q1))
                    + 2 * self.l
                    + 1
                )
                while not is_prime(q2):  # we need q2 to be prime
                    q2 -= 4 * self.l
                q = q1 * q2  # if number of divisors of q is 2, then q = q1*q2
            Div_q = divisors(q - 1)  # consider divisors of q-1
            for i in Div_q:
                # find a divisor which is close to gamma
                if gamma * 4 / 5 < i and i <= gamma and is_even(i):
                    gamma = i  # we found a good candidate for gamma
                    return q, q1
```

B.4 Size Estimates for LaBRADOR

The script is also attached.

```
"""
LaBRADOR Pari/GP Code in Sage.
"""

from sage.all import (
    log,
    ceil,
    sqrt,
    vector,
    round,
    floor,
    exp,
    ZZ,
    RR,
    pi,
    cached_function,
    cached_method,
    Infinity,
    get_verbose,
)

LABRADOR_SLACK = float(sqrt(128 / 30))


def gaussian_entropy(sigma):
    if sigma >= 4:
        a = floor(sigma / 2)
        sigma /= a
    else:
        a = 1

    d = 1 / (2 * sigma**2)
    n = sum(exp(-(i**2) * d) for i in range(-ceil(15 * sigma), 0))
    n = 2 * n + 1
    logn = log(n)
    e = 0
    for i in range(-ceil(15 * sigma), 0):
        f = exp(-(i**2) * d)
        e += f * (log(f) - logn)
    e = (-2 * e + logn) / (n * log(2))

    return float(e + log(a, 2))


def deltaf(b):
    """
    Compute root Hermite factor for block size ``b``.
    """
    small = (
        (2, 1.02190),
        (5, 1.01862),
        (10, 1.01616),
        (15, 1.01485),
        (20, 1.01420),
        (25, 1.01342),
        (28, 1.01331),
        (40, 1.01295),
    )

    if b <= 2:
        return 1.0219
    elif b < 40:
        for i in range(1, len(small)):
            if small[i][0] > b:
                return small[i - 1][1]
    elif b == 40:
        return small[-1][1]
    else:
        return float(b / (2 * pi * exp(1)) * (pi * b) ** (1.0 / b)) ** (
            1.0 / (2 * b - 2.0)
        )


def block_sizef(delta):
    b = 40
    while deltaf(2 * b) > delta:
        b *= 2
    while deltaf(b + 10) > delta:
        b += 10
    while deltaf(b) >= delta:
        b += 1

    return b


def adps16(block_size):
    return block_size * log(sqrt(3.0 / 2.0), 2.0)


default_costf = adps16


@cached_function
def sis_hard_enough(kappa, eta, b, q):
    """
    Return ‘i‘ such that for ‘n = i · η‘ and a sufficiently big ‘m‘ SIS_β on
    ‘ZZ_q^{n × m}‘ requires block size ‘κ‘.
    """
    if b > q:
        raise ValueError(f"Size bound {b} > modulus {q}.")

    i = 1
    while True:
        n = i * eta
        delta = deltaf(kappa - 1)
        d = sqrt(n * log(q) / log(delta))
        if delta ** (d - 1) * q ** (n / d) > b:
            return i
        i += 1


class LaBRADOR:
    def __init__(
        self,
        d: int = 64,
        logq: int = 32,
        tau: int = 71,
        T: int = 15,
        slack: float = LABRADOR_SLACK,
        max_beta: int = 0,
        secpar: int = 100,
        costf=default_costf,
    ):
        self.d = d
        self.logq = logq
        self.tau = tau
        self.T = T
        self.slack = slack
        self.max_beta = max_beta
        self.secpar = secpar
        self.costf = default_costf
        block_size = None
        for block_size in range(self.secpar, 2048, 32):
            if self.costf(block_size) >= self.secpar:
                break

        for block_size in range(block_size - 32, block_size + 1):
            if self.costf(block_size) >= self.secpar:
                self.block_size = block_size
                break

    def sis_rank(self, beta):
        self.max_beta = max(self.max_beta, beta)

        # we round to a nearby value to allow for caching which improves
        # performance
        # beta = 1.2 ** ceil(log(beta, 1.2))

        try:
            return sis_hard_enough(
                self.block_size, self.d, ceil(beta), 2**self.logq
            )
        except ValueError:
            return Infinity

    def main(self, n, r, beta, nu, decompose):
        old_beta = vector(beta).norm(2).n()
        # NOTE: this hardcodes secpar=128
        size = 256 * gaussian_entropy(
            float(old_beta / sqrt(2.0))
        )  # JL projection
        size += ceil(128 / self.logq) * self.d * self.logq  # JL proof

        sigs = [float(beta[i] / sqrt(r[i] * n * self.d)) for i in range(len(r))]
        sigz = sqrt(
            sigs[0] ** 2 * (1 + (r[0] - 1) * self.tau)
            + sum([sigs[i] ** 2 * r[i] * self.tau for i in range(1, len(r))])
        )
        sigh = float(sqrt(2 * n * self.d) * max(sigs) ** 2)

        if decompose:
            t = 2
            b = round(sqrt(sqrt(12) * sigz))
        else:
            t = 1
            b = 1

        t1 = round(self.logq / log(sqrt(12) * sigz / b, 2))
        t1 = max(2, t1)
        t1 = min(14, t1)

        b1 = ceil(2 ** (self.logq / t1))
        t2 = round(log(sqrt(12) * sigh) / log(sqrt(12) * sigz / b))
        t2 = max(1, t2)
        b2 = ceil((sqrt(12) * sigh) ** (1 / t2))

        r = sum(r)
        beta = [0, 0]
        beta[0] = float(sigz / float(b) * sqrt(t * n * self.d))
        for i in range(16):
            kappa = i + 1
            beta[1] = float(
                sqrt(
                    b1**2 / 12.0 * t1 * r * kappa * self.d
                    + (b1**2 * t1 + b2**2 * t2)
                    / 12.0
                    * (r**2 + r)
                    / 2.0
                    * self.d
                )
            )
            new_beta = vector(beta).norm(2).n()
            if (
                self.sis_rank(
                    max(
                        6 * self.T * b * self.slack * new_beta,
                        2 * b * self.slack * new_beta
                        + 4 * self.T * self.slack * old_beta,
                    )
                )
                <= kappa
            ):
                break

        kappa1 = self.sis_rank(2 * self.slack * new_beta)
        size += 2 * kappa1 * self.d * self.logq
        # outer commitments
        m = t1 * r * kappa + (t1 + t2) * (r**2 + r) / 2
        mu = round(m / ceil(n / nu))
        mu = max(1, mu)
        n = ceil(n / nu)
        m = ceil(m / mu)
        n = max(n, m)
        r = [t * nu, mu]

        if get_verbose() >= 3:
            print("Main:")
            print(
                "Commitments: kappa = %d; kappa1 = kappa2 = %.2f"
                % (kappa, kappa1)
            )
            print("Decomposition bases: b = %d; b1 = %d; b2 = %d" % (b, b1, b2))
            print("Expansion factors: t = %d; t1 = %d; t2 = %d" % (t, t1, t2))
            print("Target relation: n = %d; r = %s; b = %s" % (n, r, b))
            print(
                "Norm balance: %.2f%%"
                % ((beta[1] - beta[0]) / max(beta[0], beta[1]) * 100)
            )

        return size, n, r, beta

    def tail(self, n, r, beta):
        old_beta = vector(beta).norm(2).n()
        size = 256 * gaussian_entropy(
            float(old_beta / sqrt(2.0))
        )  # JL projection
        size += ceil(128 / self.logq) * self.d * self.logq  # JL proof
        size += 128  # challenges

        sigs = [float(beta[i] / sqrt(r[i] * n * self.d)) for i in range(len(r))]
        sigh = float(sqrt(2 * n * self.d) * max(sigs) ** 2)

        t1 = round(self.logq / log(sqrt(12) * sum(sigs) / len(sigs), 2))
        t1 = max(2, t1)
        t1 = min(14, t1)
        b1 = ceil(2 ** (self.logq / t1))
        t2 = round(log(sqrt(12) * sigh) / log(sqrt(12) * sum(sigs) / len(sigs)))
        t2 = max(1, t2)
        b2 = ceil((sqrt(12) * sigh) ** (1 / t2))

        for i in range(16):
            kappa = i + 1
            x = sum(r)
            n2 = x * kappa * t1 + (x**2 + x) / 2 * t2
            r2 = round(n2 / n)
            r2 = max(1, r2)
            sigz = sqrt(
                sigs[0] ** 2 * (1 + (r[0] - 1) * self.tau)
                + sum([sigs[i] ** 2 * r[i] for i in range(1, len(r))])
                * self.tau
                + r2 * max(b1, b2) ** 2 / 12.0 * self.tau
            )

            beta = sigz * sqrt(max(n, ceil(n2 / r2)) * self.d)
            if self.sis_rank(6 * self.T * beta) <= kappa:
                break

        r = sum(r)
        n = max(n, ceil(n2 / r2))

        size += r2 * kappa * self.d * self.logq  # outer commitments
        size += (
            2 * r2 * self.d * gaussian_entropy(sigh)
        )  # quadratic garbage polys
        size += (
            (2 * (r - 1) + 2 * r2) * self.d * self.logq
        )  # linear garbage polys
        size += n * self.d * float(gaussian_entropy(sigz))  # masked opening

        if get_verbose() >= 3:
            print("Tail:")
            print("Outer Commitments: kappa = %d" % kappa)
            print("Additional multiplicity: r2 = %d" % r2)
            print("Decomposition bases: b1 = %d b2 = %d" % (b1, b2))
            print("Expansion factors: t1 = %d t2 = %d" % (t1, t2))
            print("Final relation: n = %d β = %s" % (n, beta))
        return size

    def size(self, n, r, beta, nuvec):
        """
        Size in kilobytes
        """
        s = 0
        r, beta = [r], [RR(beta)]
        for i in range(len(nuvec)):
            size, n, r, beta = self.main(
                n, r, beta, nuvec[i], i < len(nuvec) - 1
            )
            s += size

        s += self.tail(n, r, beta)

        return round(s / 2**13, 2)

    @cached_method
    def __call__(self, n, r, beta, base, length, verbose=None):
        if verbose is None:
            verbose = get_verbose() > 1

        def i2v(i):
            return vector(ZZ(i).digits(base, padto=length)) + vector(
                ZZ, length, [1] * length
            )

        best = self.size(n, r, beta, i2v(0)), 0

        for i in range(base**length):
            current = self.size(n, r, beta, i2v(i)), i
            if current[0] < best[0]:
                best = current
                if verbose:
                    print(f"{best[0]:.2f}kB, {i2v(best[1])}")

        return best[0], i2v(best[1])
```

```
@cached_method
def greedy(self, n, r, beta, base, length, verbose=None):
    if verbose is None:
        verbose = get_verbose() > 1
    best = self.size(n, r, beta, [base] * length), length

    while True:
        length += 1
        current = self.size(n, r, beta, [base] * length), length
        if current[0] < best[0]:
            best = current
            if verbose:
                print(f"{best[0]:.2f}kB, {length}")
        else:
            break

    return best[0], tuple([base] * best[1])
```