Crypto Dark Matter on the Torus
Oblivious PRFs from shallow PRFs and TFHE

Martin R. Albrecht[1], Alex Davidson[2], Amit Deo[3], and Daniel Gardham[4]

[1] King's College London and SandboxAQ
[2] NOVA LINCS & DI, FCT, Universidade NOVA de Lisboa
[3] Crypto Quantique and Zama
[4] University of Surrey

Abstract. Partially Oblivious Pseudorandom Functions (POPRFs) are 2-party protocols that allow a client to learn pseudorandom function (PRF) evaluations on inputs of its choice from a server. The client submits two inputs, one public and one private. The security properties ensure that the server cannot learn the private input and the client cannot learn more than one evaluation per POPRF query. POPRFs have many applications including password-based key exchange and privacy-preserving authentication mechanisms. However, most constructions are based on classical assumptions, and those with post-quantum security suffer from large efficiency drawbacks.
In this work, we construct a novel POPRF from lattice assumptions and the "Crypto Dark Matter" PRF candidate (TCC'18) in the random oracle model. At a conceptual level, our scheme exploits the alignment of this family of PRF candidates, relying on mixed modulus computations, and programmable bootstrapping in the torus fully homomorphic encryption scheme (TFHE). We show that our construction achieves malicious client security based on circuit-private FHE, and client privacy from the semantic security of the FHE scheme. We further explore a heuristic approach to extend our scheme to support verifiability based on the difficulty of computing cheating circuits in low depth. This would yield a verifiable (P)OPRF. We provide a proof-of-concept implementation and benchmarks of our construction using the `tfhe-rs` software library. For the core online OPRF functionality, we require amortised 5.0kB communication per evaluation and a one-time per-client setup communication of 16.8MB.

Keywords: oblivious PRF, lattices, FHE

1 Introduction

Oblivious pseudorandom functions allow two parties to compute a pseudorandom function (PRF) $z := F_k(x)$ together: a server supplying a key k and a user supplying a private input x. The server does not learn x or z and the user does not learn k. If the user can be convinced that z is correct (i.e. that evaluation is performed under the correct key), then the function is "verifiable oblivious" (VOPRF), otherwise it is only "oblivious" (OPRF). Both can be efficiently realised from Diffie-Hellman (DH) and may be used in many cryptographic applications. Example applications include anonymous credentials (e.g. Cloudflare's PrivacyPass [DGS+18]) and Private Set Intersection (PSI) enabling e.g. privacy-preserving contact look-up on chat platforms [CHLR18].

The obliviousness property can be too strong in many applications where it is sufficient or even necessary to only hide part of the client's input. In this case, the public and private inputs are separated by requiring an additional public input t, called the *tag*. Then we say that we have a *Partially* Oblivious PRF (POPRF). POPRFs are typically used in protocols where a server may wish to rate-limit OPRF evaluations made by a client. Such example protocols include Password-Authenicated Key Exchanges (e.g. OPAQUE [JKX18], which is in the process of Internet Engineering Task Force (IETF) standardisation) and the Pythia PRF service [ECS+15]. This latter work also proposed a bilinear pairing-based construction of a *Verifiable* POPRF (VPOPRF), which is the natural inclusion of both properties: some of the input is revealed to the server and the client is able to check the correct evaluation of its full input.

Despite the wide use of (VP)OPRFs, most constructions are based on classical assumptions, such as DH, RSA or even pairing-based assumptions. The latest in this line of research is a recent VPOPRF construction based on a novel DH-like assumption [TCR+22] and DH-based OPRFs are currently being standardised by the IETF. Their vulnerability to quantum adversaries makes it desirable to find post-quantum solutions. However, known candidates are much less efficient.

Given fully homomorphic encryption (FHE), there is a natural (P)OPRF candidate. The client FHE encrypts input x and sends it with tag t. The server then evaluates the PRF homomorphically or "blindly" using a key derived from t and its own secret key. Finally, the client decrypts the resulting ciphertext to obtain the PRF output. The first challenge with this approach is performance, PRFs tend to have sufficiently deep circuits that FHE schemes struggle to evaluate them efficiently. Even special purpose PRFs such as the LowMC construction [ARS+15] require depth ten or more, making them somewhat impractical. More generally, in a binary circuit model we expect to require depth $\Theta(\log \lambda)$ to obtain a PRF resisting attacks with complexity $2^{\Theta(\lambda)}$.

Yet, if we expand our circuit model to arithmetic circuits with both mod p and mod q gates for $p \neq q$ both primes, shallow proposals exist [BIP+18,DGH+21]. In particular, the (weak) PRF candidate in [BIP+18] is

$$z \coloneqq \sum (\boldsymbol{A} \cdot \boldsymbol{x} \bmod 2) \bmod 3$$

where arithmetic operations are over the Integers and $\boldsymbol{A}$ is the secret key. The same work also contains a proposal to "upgrade" this weak PRF, defined for uniformly random inputs $\boldsymbol{x}$, to a full PRF, taking any $\boldsymbol{x}$. Furthermore, the works [BIP+18,DGH+21] already provide oblivious PRF candidates based on this PRF and MPC, but with non-optimal round complexity. Thus, a natural question to ask is if we can construct a round-optimal (or, 2 message) POPRF based on this PRF candidate using the FHE-based paradigm mentioned above.

1.1 Contributions

Our starting point is the observation that the computational model in [BIP+18] aligns well with that of the TFHE encryption scheme [CGGI20] and its "programmable bootstrapping" technique [MP20,Joy21]. Programmable bootstrapping allows us to realise arbitrary, not necessarily low degree, small look-up tables and thus function evaluations on (natively) single inputs. Thus, it is well positioned to realise the required gates.[5] Indeed, FHE schemes natively compute plaintexts modulo some $P \in \mathbb{Z}$ and programmable bootstrapping allows us to switch between these plaintext moduli, e.g. from mod P_1 to mod P_2. This implies a weak PRF with two levels of bootstrapping only.[6] We believe this simple observation and conceptual contribution will have applications beyond this work. We further hope that by giving another application domain for the PRF candidate from [BIP+18] – it is not just MPC-friendly but also FHE-friendly – we encourage further cryptanalysis on it.

After some preliminaries in Section 2 we specify our POPRF candidate in Section 3. As is typical with FHE-based schemes, we require the involved parties – here the client – to prove that its inputs are well-formed. We also make use of the protected encoded-input PRF (PEI-PRF) paradigm from [BIP+18] where the client performs some computations not dependent on secret key material and then submits the output together with a NIZK proof of well-formedness to the server for processing.

We prove our construction secure in the random oracle model in Section 4. We show that our construction meets the security definitions from [TCR+22]: pseudorandomness even in the presence of malicious clients (POPRF security) and privacy for clients. This latter property has two flavours based on the capabilities of the adversary, POPRIV1 (which we achieve) captures security against an honest-but-curious server whereas POPRIV2 ensures security even when the server is malicious. Here, the client maintains privacy by detecting malicious behaviour of the server. POPRF security for the server essentially rests on a variant of circuit-privacy obtained from TFHE bootstrapping and client NIZK. The NIZK is made online extractable in the POPRF proof using a trapdoor and thus avoids any rewinding issues outlined in e.g. [SG98], and similarly mitigates the problem of rewinding for post-quantum security, cf. [Unr12]. POPRIV1 security for the client against a semi-honest server essentially relies on the IND-CPA security of TFHE.

Initially, we focus on oblivious rather than verifiable oblivious PRFs. This is motivated by the presumed high cost associated with zero-knowledge proofs

[5] The security of the PRF candidate in [BIP+18] rests on the absence of any low-degree polynomial interpolating it, ruling out efficient implementations using FHE schemes that only provide additions and multiplications.

[6] We require two, not one, levels because we appeal to the circuit-privacy properties of a final bootstrapping operation.

for performing FHE computations. In Section 5, we explore a different approach to adding verifiability to our OPRF, inspired by and based on a discussion in [ADDS21]. The idea here is that the server commits to a set of evaluation "check" points and that the client can use the oblivious nature of the PRF to request PRF evaluations of these points to catch a cheating server. However, achieving security of this "cut-and-choose" approach in this setting is non-trivial as the server may still obliviously run a cheating circuit that agrees on those check points but diverges elsewhere.

We explore the feasibility of such a cheating circuit using direct cryptanalysis. In more detail, inspired by the heuristic approach in [CHLR18] for achieving malicious security – forcing the server to compute a deep circuit in FHE parameters supporting only shallow circuits – we explore cheating circuits in bootstrapping depth two. While we were unable to find such a cheating circuit, and conjecture that one does not exist, we stress that this part of our work is highly speculative. Under the heuristic assumption that our construction is verifiable, in Appendix C we then show that it also satisfies POPRIV2. We hope that our work encourages further exploration of such strategies, as these will have applications elsewhere to upgrade FHE-based schemes to malicious security.[7]

We present our proof-of-concept Rust implementation in Section 6, and provide benchmarks that illustrate how quickly the core OPRF functionality (without (verifying the) NIZK proofs) can be run. Our implementation is open-source[8] and makes use of Zama's `tfhe-rs` library for implementing TFHE. While the public key material sent by the client to the server is large (16.8MB) this cost can be amortised by reusing the same material for several evaluations. Individual PRF evaluations can then cost about 30.4kB or as little as 5.0kB when amortising client NIZK proofs across several OPRF queries. In terms of runtimes, client online functions run in 36ms on one core and server online functions are expected to run in 123ms on 64 cores. We discuss the (significantly worse) performance of our VOPRF in Section 5.

In Appendix B we then estimate costs of the required non-interactive zero-knowledge proofs. First, we use [BS22] (with a zero-knowledge "shim") to proof well-formedness of the bootstrapping key material. Second, as previously mentioned, our OPRF construction makes use of the PEI-PRF paradigm [BIP$^+$18] which must be made compatible with our zero-knowledge proofs. Here we rely on and specialise the proofs in [LNP22]. We also show this is extendable to the stronger property of *non-unique encoding* required in our verifiable OPRF.

1.2 Related Work

Oblivious PRFs and variants thereof are an active area of research. A survey of constructions, variants and applications was given in [CHL22]. In this work we are interested in plausibly post-quantum and round-optimal constructions.

[7] We note that a hybrid approach of ours and proving correct evaluation would be to prove that only a limited-depth circuit was evaluated.

[8] https://anonymous.4open.science/r/oprf-fhe-C726

The first candidate construction was given in [ADDS21], which built a verifiable oblivious PRF from lattice assumptions following the blueprint of Diffie-Hellman constructions with additive blinding (a construction for multiplicative blinding is given in an appendix of the full version of [ADDS21]). The work provides both semi-honest and malicious secure candidates with the latter being significantly more expensive. We stress that in the former, both parties are semi-honest.

In [BKW20] two candidate constructions from isogenies were proposed. One, a VOPRF related to SIDH, was unfortunately shown to not be secure [BKM+21]. The other, an OPRF related to CSIDH, achieves sub megabyte communication in a malicious setting assuming the security of group-action decisional Diffie-Hellman. In [Bas23] a fixed-and-improved SIDH-based candidate was proposed and in [HMR23] an improved CSIDH-based candidate is presented. Either of these rely on trusted setups. In [SHB21] an OPRF based on the Legendre PRF is proposed based on solving sparse multi-variate quadratic systems of equations. In [DGH+21], which also builds on [BIP+18], an MPC-based OPRF is proposed that is secure against semi-honest adversaries. It achieves much smaller communication complexity compared to all other post-quantum candidates, but in a preprocessing model where correlated randomness is available to the parties. A protocol computing this correlated randomness, e.g. [BCG+22], would add two rounds (or more) and thus make the overall protocol not round-optimal. The question of upgrading security to full malicious security is left as an open problem in [DGH+21].[9]

We give a summary comparison of our construction with prior work in Table 1. The only 2-round construction without preprocessing or trusted-setup in Table 1 is that from [ADDS21], where our construction compares favourably by offering stronger security at smaller size. In particular, even in a semi-honest setting, our construction outperforms that from [ADDS21] in terms of bandwidth for $L = 16$ queries.[10]

1.3 Open Problems

The most immediate open problem posed by this work is to provide a full, end-to-end, implementation of our OPRF candidate. As we will discuss in Section 6, the `tfhe-rs` API does not enable us to obtain full performance in terms of server running time. Furthermore, we did not implement the NIZK proofs attesting well-formedness of inputs.

Another pressing open problem is to refine our understanding of the security of the PRF candidate from [BIP+18]. In particular, our parameter choices may prove to be too aggressive, and we hope that our work inspires cryptanalysis.

Our verifiability approach throws up a range of interesting avenues to explore for VOPRFs but also for verifiable homomorphic computation, more generally.

[9] We note that [DGH+21] may also serve as an indication for the best-case performance of OPRFs built from generic MPC techniques.

[10] We note that while the large sizes for achieving malicious security in [ADDS21] can be avoided using improved NIZKs, the semi-honest base size of 2MB per query stems from requiring $q \approx 2^{256}$ for statistical correctness and security arguments.

Table 1. Post-quantum (P)OPRF candidates in the literature

work	assumption	r	communication cost	flavour	model
[ADDS21]	R(LWE) & SIS	2	$\approx$ 2MB	plain	semi-honest, QROM
[SHB21]	Legendre PRF	3	$\approx \lambda \cdot$ 13K	plain	semi-honest, *pp*, ROM
[BKW20]	CSIDH	3	424KB	plain	malicious client
[Bas23]	SIDH	2	3.0MB	plain	malicious, *ts*, ROM
[HMR23]	CSIDH	2	21KB	plain	semi-honest, *ts*
[HMR23]	CSIDH	4	35KB	plain	malicious client, *ts*
[HMR23]	CSIDH	258	25KB	plain	semi-honest
[DGH+21]	[BIP+18]	2	80B	plain	semi-honest, *pp*
Section 3	lattices, [BIP+18]	2	16.8MB + 41.42KB + 0.5kB + 26.7kB + 3.2kB	plain	malicious client, ROM
Section 3	lattices, [BIP+18]	2	16.8MB + 41.42KB + 0.5kB + 1.3kB + 3.2kB	plain	malicious client, ROM $L = 64$, per query

The column "r" gives the number of rounds. ROM is the random oracle model, QROM the quantum random oracle model, "pp" stands for "preprocessing", and "ts" for "trusted setup". When reporting on our work, the summands are: pk size, pk proof size, client message size, client message proof size, server message size. Our client message proofs can be amortised to e.g. 83.4kB/64 = 1.3kB per query, when amortising over $L = 64$ queries.

First, our OPRF construction relies on programmable bootstrapping. This restricts the choice of FHE scheme we might instantiate our protocol with, but also gives the server the choice which function to evaluate, something our application does not require. That is, we may not need to rely on *evaluator programmable bootstrapping* if it is possible for the client to define the non-linear functions available to a server (*encrypter programmable bootstrapping*). This would enable reasoning about malicious server security more easily.

Related works, e.g. [CHLR18], have also used similar assumptions as our work over the hardness of computing deep circuits in low FHE depth. There is growing evidence that such assumptions allow for new, interesting or more efficient constructions of cryptographic primitives. However, the hardness of these computational problems needs to be better understood.

Finally, we estimate that our VOPRF is orders of magnitude less efficient than our OPRF candidate and we discuss plausible avenues for alleviating that difference in Section 5. A more direct approach would be to construct a NIZK for correct bootstrapping evaluation, which would have applications beyond this work.

2 Preliminaries

We use $\lfloor\cdot\rfloor$, $\lceil\cdot\rceil$ and $\lfloor\cdot\rceil$ to denote the standard floor, ceiling and rounding to the nearest integer functions (rounding down in the case of a tie). We denote the Integers by $\mathbb{Z}$ and for any positive $p \in \mathbb{Z}$, the integers modulo p are denoted by $\mathbb{Z}_p$. We typically use representatives of $\mathbb{Z}_p$ in $\{-p/2, \ldots, (p/2) - 1\}$ if p is even and $\{-\lfloor p/2\rfloor, \ldots, \lfloor p/2\rfloor\}$ if p is odd, but we will also consider $\mathbb{Z}_p$ as $\{0, 1, \ldots, p - 1\}$.

Since it will always be clear from context or stated explicitly which representation we use, this does not create ambiguity. The p-adic decomposition of an integer $x \geq 0$ is a tuple $(x_i)_{0 \leq i < \lceil \log_p(x) \rceil}$ with $0 \leq x_i < p$ such that $x = \sum p^i \cdot x_i$. We denote the set S_m to be the permutation group of m elements.

Let $\mathbb{Z}[X]$ denote the polynomial ring in the variable X whose coefficients belong to $\mathbb{Z}$. We also denote power-of-two cyclotomic rings $\mathcal{R} \coloneqq \mathbb{Z}[X]/(X^d + 1)$ where d is a power-of-two, and $\mathcal{R}_q \coloneqq \mathcal{R}/(q\mathcal{R})$ for any integer "modulus" q. Bold letters denote vectors and upper case letters denote matrices. Abusing notation we write $(\boldsymbol{x}, \boldsymbol{y})$ for the concatenation of the vectors $\boldsymbol{x}$ and $\boldsymbol{y}$. We extend this notation to scalars, too. Additionally, $\| \cdot \|$and $\| \cdot \|_\infty$ denote standard Euclidean and infinity norms respectively.

For a distribution D, we write $x \leftarrow\!\!\$\, D$ to denote that x is sampled according to the distribution D. An example of a distribution is the discrete Gaussian distribution over $\mathbb{Z}$ with parameter $\sigma > 0$ denoted as $D_{\mathbb{Z},\sigma}$. This distribution has its probability mass function proportional to the Gaussian function $\rho_\sigma(x) \coloneqq \exp(-\pi x^2/\sigma^2)$. We use λ to denote the security parameter. We use the standard asymptotic notation $(\Omega, \mathcal{O}, \omega$ etc.) and use $\mathsf{negl}(\lambda)$ to denote a negligible function, i.e. a function that is $\lambda^{\omega(1)}$. Further, we write $\mathsf{poly}(\lambda)$ to denote a polynomial function i.e. a function that is $\mathcal{O}(n^c)$ for some constant c. An algorithm is said to be polynomially bounded if it terminates after $\mathsf{poly}(\lambda)$ steps and uses $\mathsf{poly}(\lambda)$-sized memory. Two distribution ensembles $D_1(1^\lambda)$ and $D_2(1^\lambda)$ are said to be *computationally* indistinguishable if for any probabilistic polynomially bounded algorithm $\mathcal{A}$, $\mathrm{Adv}(\mathcal{A}) \coloneqq \| \Pr[1 \leftarrow\!\!\$\, \mathcal{A}_X(1^\lambda)] - \Pr[1 \leftarrow\!\!\$\, \mathcal{A}_Y(1^\lambda)] \| \leq \mathsf{negl}(\lambda)$. In such a case we write $D_1(1^\lambda) \approx_c D_2(1^\lambda)$. The distribution ensembles are said to be *statistically* indistinguishable if the same holds for all unbounded algorithms, in which case we write $D_1(1^\lambda) \approx_s D_2(1^\lambda)$.

For a keyspace $\mathcal{K}$, input space $\mathcal{X}$ and output space $\mathcal{Z}$, a PRF is a function $F : \mathcal{K} \times \mathcal{X} \longrightarrow \mathcal{Z}$ with a pseudorandomness property. Rather than writing $F(k, x)$ for $k \in \mathcal{K}$ and $x \in \mathcal{X}$, we write $F_k(x)$. The pseudorandomness property of a PRF requires that over a secret and random choice of $k \leftarrow\!\!\$\, \mathcal{K}$, the single input function $F_k(\cdot)$ is computationally indistinguishable from a uniformly random function. Note here that the dependence of the parameters on λ is present, but is not explicitly written for simplicity. We also use the standard cryptographic notion of a (non-interactive) zero-knowledge proof/argument. For more details on these standard cryptographic notions, see e.g. [Gol04].

2.1 Random Oracle Model

We will prove security by modelling hash functions as random oracles. Since our schemes will make use of more than one hash function, it will be useful to have a general abstraction for the use of ideal primitives, following the treatment in [TCR+22]. A random oracle RO specifies algorithms $\mathsf{RO.Init}$ and $\mathsf{RO.Eval}$. The initialisation algorithm has syntax $st_{\mathsf{RO}} \leftarrow\!\!\$\, \mathsf{RO.Init}(1^\lambda)$. The stateful evaluation algorithm has syntax $y \leftarrow\!\!\$\, \mathsf{RO.Eval}(x, st_{\mathsf{RO}})$. We sometimes use A^{RO} as shorthand for giving algorithm A oracle access to $\mathsf{RO.Eval}(\cdot, st_{\mathsf{RO}})$. We combine access to multiple random oracles $\mathsf{RO} = \mathsf{RO}_0 \times \ldots \times \mathsf{RO}_{m-1}$ in the obvious way. We may arbitrarily label our random oracles to aid readability e.g. $\mathsf{RO}_{\mathsf{key}}$ to denote a random oracle applied to some "key".

2.2 (Verifiable) (Partial) Oblivious Pseudorandom Functions

We adopt the notation and definitions for oblivious pseudorandom functions from [TCR+22]. An OPRF is a protocol between two parties: a server S who holds a private key and a client who wants to obtain evaluations of F_k on inputs of its choice. We write $z \coloneqq F_k(x)$. We say that an OPRF is a partial OPRF (OPRF) if part of the client's input is given to the server. In this case, we write $z \coloneqq F_k(t, x)$ where t is in the clear and x is hidden from S. When C can verify that the PRF was evaluated correctly we speak of a verifiable OPRF (VOPRF) or VPOPRF when the protocol also supports partially known inputs t.

Definition 1 (Partial Oblivious PRF [TCR+22]). *A partial oblivious PRF (POPRF) $\mathcal{F}$ is a tuple of PPT algorithms*

$$(\mathcal{F}.\mathsf{Setup}, \mathcal{F}.\mathsf{KeyGen}, \mathcal{F}.\mathsf{Request}, \mathcal{F}.\mathsf{BlindEval}, \mathcal{F}.\mathsf{Finalise}, \mathcal{F}.\mathsf{Eval})$$

The setup and key generation algorithm generate public parameter pp and a public/secret key pair $(\mathsf{pk}, \mathsf{sk})$. Oblivious evaluation is carried out as an interactive protocol between C and S, here presented as algorithms $\mathcal{F}.\mathsf{Request}$, $\mathcal{F}.\mathsf{BlindEval}$, $\mathcal{F}.\mathsf{Finalise}$ working as follows:

1. *First, C runs the algorithm $\mathcal{F}.\mathsf{Request}^{\mathsf{RO}}_{\mathsf{pp}}(\mathsf{pk}, t, x)$ taking a public key pk, a tag or public input t and a private input x. It outputs a local state st and a request message req, which is sent to the server.*
2. *S runs $\mathcal{F}.\mathsf{BlindEval}^{\mathsf{RO}}_{\mathsf{pp}}(\mathsf{sk}, t, req)$ taking as input a secret key sk, a tag t and the request message req. It produces a response message rep sent back to C.*
3. *Finally, C runs $\mathcal{F}.\mathsf{Finalise}(rep, st)$ which takes the response message and its previously constructed state st and either outputs a PRF evaluation or $\perp$ if rep is rejected.*

The unblinded evaluation algorithm $\mathcal{F}.\mathsf{Eval}$ is deterministic and takes as input a secret key sk, an input pair (t, x) and outputs a PRF evaluation z.

We also define sets $\mathcal{F}.\mathsf{SK}$, $\mathcal{F}.\mathsf{PK}$, $\mathcal{F}.\mathsf{T}$, $\mathcal{F}.\mathsf{X}$ and $\mathcal{F}.\mathsf{Out}$ representing the secret key, public key, tag, private input, and output space, respectively. We define the input space $\mathcal{F}.\mathsf{In} = \mathcal{F}.\mathsf{T} \times \mathcal{F}.\mathsf{X}$. We assume efficient algorithms for sampling and membership queries on these sets.

Remark 1. Fixing t, e.g. $t = \perp$, recovers the definition of an OPRF.

Remark 2. In Figure 1, the oracle $\mathrm{Prim}(x)$ captures access to the random oracle used in the POPRF construction. For $b = 0$ (the case where the adversary interacts with a simulator and truly random function) the simulator may only use a limited number of random function queries to simulate the random oracle accessed via $\mathrm{Prim}(x)$.

```
Game $\mathrm{POPRF}^{\mathcal{A},b}_{\mathcal{F},\mathsf{S},\mathsf{RO}}(\lambda)$
$q_{s,t}, q_t \leftarrow 0, 0$
$st_{\mathsf{Fn}} \leftarrow\!\!\$\ \mathsf{RO}_{\mathsf{Fn}}.\mathsf{Init}(1^\lambda)$   // $\mathcal{F}.\mathsf{In} \rightarrow \mathcal{F}.\mathsf{Out}$
$st_{\mathsf{RO}} \leftarrow\!\!\$\ \mathsf{RO}.\mathsf{Init}(1^\lambda)$
$\mathsf{pp}_1 \leftarrow\!\!\$\ \mathcal{F}.\mathsf{Setup}(1^\lambda)$
$(st_{\mathsf{S}}, \mathsf{pk}_0, \mathsf{pp}_0) \leftarrow\!\!\$\ \mathsf{S}.\mathsf{Init}(\mathsf{pp}_1)$
$(sk, \mathsf{pk}_1) \leftarrow\!\!\$\ \mathcal{F}.\mathsf{KeyGen}^{\mathsf{RO}}_{\mathsf{pp}_1}(1^\lambda)$
$b' \leftarrow\!\!\$\ \mathcal{A}^{\mathrm{Eval},\mathrm{BlindEval},\mathrm{Prim}}(\mathsf{pp}_b, \mathsf{pk}_b)$
return $b'$
```

```
Oracle $\mathrm{Eval}(t, x)$
$z_0 \leftarrow \mathsf{RO}_{\mathsf{Fn}}.\mathsf{Eval}((t, x), st_{\mathsf{Fn}})$
$z_1 \leftarrow \mathcal{F}.\mathsf{Eval}^{\mathsf{RO}}_{\mathsf{pp}_1}(sk, t, x)$
return $z_b$

Oracle $\mathrm{LimitEval}(t, x)$
$q_{t,s} \leftarrow q_{t,s} + 1$
if $q_{t,s} \leq q_t$ then
  return $\mathrm{Eval}(t, x)$
return $\perp$
```

```
Oracle $\mathrm{BlindEval}(t, req)$
$q_t \leftarrow q_t + 1$
$(rep_0, st_{\mathsf{S}}) \leftarrow\!\!\$\ \mathsf{S}.\mathsf{BlindEval}^{\mathrm{LimitEval}}(t, req, st_S)$
$rep_1 \leftarrow\!\!\$\ \mathcal{F}.\mathsf{BlindEval}^{\mathsf{RO}}_{\mathsf{pp}_1}(sk, t, req)$
return $rep_b$

Oracle $\mathrm{Prim}(x)$
$(h_0, st_{\mathsf{S}}) \leftarrow\!\!\$\ \mathsf{S}.\mathsf{Eval}^{\mathrm{LimitEval}}(x, st_S)$
$h_1 \leftarrow\!\!\$\ \mathsf{RO}.\mathsf{Eval}(x, st_{\mathsf{RO}})$
return $h_b$
```

Fig. 1. Pseudorandomness against malicious clients.

We adapt the correctness notion from [TCR+22], permitting a small failure probability.

Definition 2 (POPRF Correctness (adapted from [TCR+22])). *A partial oblivious PRF (POPRF)*

$$(\mathcal{F}.\mathsf{Setup}, \mathcal{F}.\mathsf{KeyGen}, \mathcal{F}.\mathsf{Request}, \mathcal{F}.\mathsf{BlindEval}, \mathcal{F}.\mathsf{Finalise}, \mathcal{F}.\mathsf{Eval})$$

is correct if

$$\Pr\left[z = \mathcal{F}.\mathsf{Eval}^{\mathsf{RO}}_{\mathsf{pp}}(\mathsf{sk}, t, x) \,\middle|\, \begin{array}{l} \mathsf{pp} \leftarrow\!\!\$\ \mathcal{F}.\mathsf{Setup}(1^\lambda) \\ (\mathsf{pk}, \mathsf{sk}) \leftarrow\!\!\$\ \mathcal{F}.\mathsf{KeyGen}^{\mathsf{RO}}_{\mathsf{pp}}(1^\lambda) \\ (st, req) \leftarrow\!\!\$\ \mathcal{F}.\mathsf{Request}^{\mathsf{RO}}_{\mathsf{pp}}(\mathsf{pk}, t, x) \\ rep \leftarrow\!\!\$\ \mathcal{F}.\mathsf{BlindEval}^{\mathsf{RO}}_{\mathsf{pp}}(\mathsf{sk}, t, req) \\ z \leftarrow\!\!\$\ \mathcal{F}.\mathsf{Finalise}^{\mathsf{RO}}_{\mathsf{pp}}(rep, st) \end{array} \right] = 1 - \mathsf{negl}(\lambda).$$

We target the same pseudorandomness guarantees against malicious clients as [TCR+22].

Definition 3 (Pseudorandomness (POPRF) [TCR+22]). *We say a partial oblivious PRF $\mathcal{F}$ is pseudorandom if for all* PPT *adversaries $\mathcal{A}$, there exists a PPT simulator* S *such that the following advantage is* $\mathsf{negl}(\lambda)$*:*

$$\mathsf{Adv}^{\mathrm{po-prf}}_{\mathcal{F},\mathsf{S},\mathsf{RO},\mathcal{A}}(\lambda) = \left| \Pr\left[\mathrm{POPRF}^{\mathcal{A},1}_{\mathcal{F},\mathsf{S},\mathsf{RO}}(\lambda) \Rightarrow 1\right] - \Pr\left[\mathrm{POPRF}^{\mathcal{A},0}_{\mathcal{F},\mathsf{S},\mathsf{RO}}(\lambda) \Rightarrow 1\right] \right|.$$

Remark 3. The intuition of this definition is that it requires the simulator to explain a random output (defined via $\mathsf{RO}_{\mathsf{Fn}}$) as an evaluation point of the PRF. The simulator provides its own public key and public parameters, but it gets at most one query to $\mathsf{RO}_{\mathsf{Fn}}()$ per $\mathsf{BlindEval}$ query that it has to simulate. The simulator queries $\mathsf{RO}_{\mathsf{Fn}}$ through calls to LimitEval, where the check $q_{t,s} \leq q_t$ enforces the number of queries per $\mathsf{BlindEval}$ query and tag t. This implies that $\mathsf{BlindEval}$ and Eval queries essentially leak nothing beyond the evaluation at this exact point to the client.

```
Game $\text{POPRIV1}^{\mathcal{A},b}_{\mathcal{F},\mathsf{RO}}(\lambda)$
$\mathsf{pp} \leftarrow\!\!\$\ \mathcal{F}.\mathsf{Setup}(1^\lambda)$
$(\mathsf{pk},\mathsf{sk}) \leftarrow\!\!\$\ \mathcal{F}.\mathsf{KeyGen}^{\mathsf{RO}}_{\mathsf{pp}}(1^\lambda)$
$b' \leftarrow\!\!\$\ \mathcal{A}^{\mathrm{Run},\mathsf{RO}}(\mathsf{pp},\mathsf{pk},\mathsf{sk})$
return $b'$

Oracle $\mathrm{Run}(t, x_0, x_1)$
for $j \in \{0,1\}$ do
    $(st_j, req_j) \leftarrow\!\!\$\ \mathcal{F}.\mathsf{Request}^{\mathsf{RO}}_{\mathsf{pp}}(\mathsf{pk}, t, x_j)$
    $rep_j \leftarrow\!\!\$\ \mathcal{F}.\mathsf{BlindEval}^{\mathsf{RO}}_{\mathsf{pp}}(sk, t, req_j)$
    $z_j \leftarrow\!\!\$\ \mathcal{F}.\mathsf{Finalise}^{\mathsf{RO}}_{\mathsf{pp}}(rep_j, st_j)$
$\tau_0 \leftarrow (req_b, rep_b, z_0)$
$\tau_1 \leftarrow (req_{1-b}, rep_{1-b}, z_1)$
return $(\tau_0, \tau_1)$
```

```
Game $\text{POPRIV2}^{\mathcal{A},b}_{\mathcal{F},\mathsf{RO}}(\lambda)$
$\mathsf{pp} \leftarrow\!\!\$\ \mathcal{F}.\mathsf{Setup}(1^\lambda)$
$i \leftarrow 0$
$b' \leftarrow\!\!\$\ \mathcal{A}^{\mathrm{Request},\mathrm{Finalise},\mathsf{RO}}(\mathsf{pp})$
return $b'$

Oracle $\mathrm{Request}(\mathsf{pk}, t, x_0, x_1)$
$i \leftarrow i + 1$
$(st_{i,0}, req_0) \leftarrow\!\!\$\ \mathcal{F}.\mathsf{Request}^{\mathsf{RO}}_{\mathsf{pp}}(\mathsf{pk}, t, x_0)$
$(st_{i,1}, req_1) \leftarrow\!\!\$\ \mathcal{F}.\mathsf{Request}^{\mathsf{RO}}_{\mathsf{pp}}(\mathsf{pk}, t, x_1)$
return $(req_b, req_{1-b})$

Oracle $\mathrm{Finalise}(j, rep_0, rep_1)$
if $j > i$ then return $\perp$
$z_b \leftarrow\!\!\$\ \mathcal{F}.\mathsf{Finalise}^{\mathsf{RO}}_{\mathsf{pp}}(rep_0, st_{j,b})$
$z_{1-b} \leftarrow\!\!\$\ \mathcal{F}.\mathsf{Finalise}^{\mathsf{RO}}_{\mathsf{pp}}(rep_1, st_{j,1-b})$
if $z_0 = \perp$ or $z_1 = \perp$ then return $\perp$
return $(z_0, z_1)$
```

Fig. 2. Request privacy against honest-but-curious servers (left) and against malicious servers (right).

Moreover, the simulator is restricted in that LimitEval oracle will error if more queries are made to it than the number of BlindEval queries (on t) at any point in the game. Meaningful relaxations of this definition are discussed in [TCR+22] but for completeness we opt for the full definition.

Definition 4 (Request Privacy (POPRIV) [TCR+22]). *We say a partial oblivious PRF $\mathcal{F}$ has request privacy against honest-but-curious and malicious servers respectively if for all* PPT *adversary $\mathcal{A}$ the following advantage is* $\mathsf{negl}(\lambda)$ *for $k = 1$ and $k = 2$ respectively:*

$$\mathsf{Adv}^{\mathrm{po-priv}k}_{\mathcal{F},\mathsf{S},\mathsf{RO},\mathcal{A}}(\lambda) = \left|\Pr\left[\mathrm{POPRIV}\,k^{\mathcal{A},1}_{\mathcal{F},\mathsf{RO}}(\lambda) \Rightarrow 1\right] - \Pr\left[\mathrm{POPRIV}\,k^{\mathcal{A},0}_{\mathcal{F},\mathsf{RO}}(\lambda) \Rightarrow 1\right]\right|.$$

2.3 Hard Lattice Problems

We will rely on both the M-SIS and the M-LWE problems. Instantiating these over $\mathcal{R} = \mathbb{Z}$ recovers the SIS and LWE problems respectively. Further, instantiating these over some ring of integer of some number field and with $n = 1$, recovers the Ring-SIS and Ring-LWE problems respectively.

Definition 5 (M-SIS, adapted from [LS15]). *Let $\mathcal{R}, q, n, \ell, \beta$ depend on λ. The Module-SIS (or M-SIS) problem, denoted M-$\mathsf{SIS}_{\mathcal{R}_q,n,\ell,\beta^*}$, is: Given a uniform $\boldsymbol{A} \leftarrow\!\!\$\ \mathcal{R}_q^{n\times\ell}$ find some $\boldsymbol{u} \neq \boldsymbol{0} \in \mathcal{R}^\ell$ such that $\|\boldsymbol{u}\| \leq \beta^*$ and $\boldsymbol{A}\cdot\boldsymbol{u} \equiv \boldsymbol{0} \bmod q$.*

Definition 6 (M-LWE, adapted from [LS15]). *Let $\mathcal{R}, q, n, m$ depend on λ and let χ_s, χ_e be distributions over $\mathcal{R}_q$. Denote by M-$\mathsf{LWE}_{\mathcal{R}_q,n,m,\chi_s,\chi_e}$ the* *probability distribution on $\mathcal{R}_q^{m\times n} \times \mathcal{R}_q^m$ obtained by sampling the coordinates of the matrix $\boldsymbol{A} \in \mathcal{R}_q^{m\times n}$ independently and uniformly over $\mathcal{R}_q$, sampling the coordinates of $\boldsymbol{s} \in \mathcal{R}_q^n$, $\boldsymbol{e} \in \mathcal{R}^m$ independently from χ_s and χ_e respectively, setting $\boldsymbol{b} \coloneqq \boldsymbol{A}\cdot\boldsymbol{s}+\boldsymbol{e} \bmod q$ and outputting $(\boldsymbol{A}, \boldsymbol{b})$. The M-LWE problem is to distinguish the uniform distribution over $\mathcal{R}_q^{m\times n} \times \mathcal{R}_q^m$ from $M\text{-}\mathsf{LWE}_{\mathcal{R}_q,n,m,\chi_s,\chi_e}$.*

2.4 Matrix NTRU Trapdoors

The original formulation [HPS96] of the NTRU problem considers rings of integers of number fields or polynomial rings, but a matrix version is implicit and considered for cryptanalysis in the literature.

Definition 7. *Given integers n, p, q, β where p and q are coprime, the matrix-NTRU assumption (denoted $\mathsf{mat\text{-}NTRU}_{n,p,q,\beta}$) states that no PPT algorithm can distinguish between $\boldsymbol{A}$ and $\boldsymbol{B}$ where*

- $\boldsymbol{A} \leftarrow\$ \mathbb{Z}_q^{n\times n}$
- $\boldsymbol{B} = p^{-1}\cdot\boldsymbol{G}^{-1}\cdot\boldsymbol{F} \mod q$ *with*

$$\boldsymbol{F} \leftarrow\$ \{0, \pm1, \ldots, \pm\beta\}^{n\times n}, \boldsymbol{G} \leftarrow\$ \{0, \pm1, \ldots, \pm\beta\}^{n\times n} \cap \left(\mathbb{Z}_q^{n\times n}\right)^*$$

where $\left(\mathbb{Z}_q^{n\times n}\right)^*$ denotes the set of invertible $(n\times n)$ matrices over $\mathbb{Z}_q$.

We will use the matrix-NTRU assumption to define a trapdoor. In what follows, we assume an odd q and an even p that is coprime to q. In particular, we define the following algorithms:

$\mathsf{NTRUTrapGen}(n, q, p, \beta)$**:** Sample

$$\boldsymbol{F} \leftarrow\$ \{0, \pm1, \ldots, \pm\beta\}^{n\times n}, \boldsymbol{G} \leftarrow\$ \{0, \pm1, \ldots, \pm\beta\}^{n\times n} \cap \left(\mathbb{Z}_q^{n\times n}\right)^*$$

and output public information $\mathsf{pp} \coloneqq (p^{-1}\cdot\boldsymbol{G}^{-1}\cdot\boldsymbol{F} \bmod q, q)$ and a trapdoor $\tau \coloneqq (\boldsymbol{F}, \boldsymbol{G}, p)$.

$\mathsf{NTRUDec}(\boldsymbol{c}, \tau)$**:** For $\boldsymbol{c} \in \mathbb{Z}_q^n$, $\tau \coloneqq (\boldsymbol{F}, \boldsymbol{G}, p)$, compute $\boldsymbol{c}_1 = p\cdot\boldsymbol{G}\cdot\boldsymbol{c} \bmod q$, $\boldsymbol{c}_2 = \boldsymbol{c}_1 \bmod (p/2)$, $\boldsymbol{c}_3 = \boldsymbol{c} - p^{-1}\cdot\boldsymbol{G}^{-1}\cdot\boldsymbol{c}_2 \bmod q$. Finally, compute and output $\boldsymbol{m}' \coloneqq \left\lfloor \frac{2}{q-1}\cdot\boldsymbol{c}_3 \right\rceil$ where the multiplication and rounding is done over the rationals.

The trapdoor functionality is summarised in the lemma below.

Lemma 1. *Suppose that p, q are coprime where p is even and q is odd. Suppose also that $\beta\cdot\beta_s'\cdot n < p/4$, and that $\beta_s', \beta_e' \in \mathbb{R}$ satisfies $\beta\cdot n\cdot(\beta_s'+p\cdot(2\,\beta_e'+1)/2) < q/2$. Sample $(\mathsf{pp} \coloneqq (\boldsymbol{B}, q),\ \tau) \leftarrow\$ \mathsf{NTRUTrapGen}(n, q, p, \beta)$. Then:*

1. *$\boldsymbol{B}$ is indistinguishable from uniform over $\mathbb{Z}_q^{n\times n}$ if the $\mathsf{mat\text{-}NTRU}_{n,p,q,\beta}$ assumption holds.*
2. *If $\boldsymbol{c} = \boldsymbol{B}\cdot\boldsymbol{s}+\boldsymbol{e}+\lfloor q/2\rceil\cdot\boldsymbol{m} \bmod q$ where $\boldsymbol{m} \in \mathbb{Z}_2^n$, $(\|\boldsymbol{s}\|_\infty \le \beta_s' \vee \|\boldsymbol{s}\|_2 \le \beta_s'\cdot\sqrt{n})$ and $(\|\boldsymbol{e}\|_\infty \le \beta_e' \vee \|\boldsymbol{e}\|_e \le \beta_e'\cdot\sqrt{n})$, then $\mathsf{NTRUDec}(\boldsymbol{c}, \tau) = \boldsymbol{m}$.*

Proof. For the first part, simply note that distinguishing $\boldsymbol{B}$ from uniform is exactly the matrix-NTRU problem for (n, p, q, β). For the second part, reusing the same notation from the description of $\mathsf{NTRUDec}(\boldsymbol{c}, \tau := (\boldsymbol{F}, \boldsymbol{G}, p))$ gives

$$\begin{aligned}\boldsymbol{c}_1 &= \boldsymbol{F} \cdot \boldsymbol{s} + p \cdot \boldsymbol{G} \cdot (\boldsymbol{e} + ((q-1)/2) \cdot \boldsymbol{m}) \bmod q \\ &= \boldsymbol{F} \cdot \boldsymbol{s} + (p/2) \cdot \boldsymbol{G} \cdot (2\,\boldsymbol{e} - \boldsymbol{m}) \bmod q \\ &= \boldsymbol{F} \cdot \boldsymbol{s} + (p/2) \cdot \boldsymbol{G} \cdot (2\,\boldsymbol{e} - \boldsymbol{m})\end{aligned}$$

over $\mathbb{Z}$ because $\|\boldsymbol{F} \cdot \boldsymbol{s} + (p/2) \cdot \boldsymbol{G} \cdot (2\,\boldsymbol{e} - \boldsymbol{m})\|_\infty < q/2$. We then have $\boldsymbol{c}_2 = \boldsymbol{F} \cdot \boldsymbol{s} \bmod p/2 = \boldsymbol{F} \cdot \boldsymbol{s}$ over $\mathbb{Z}$ because $\|\boldsymbol{F} \cdot \boldsymbol{s}\|_\infty < p/4$. Next, $\boldsymbol{c}_3 = \boldsymbol{e} + ((q-1)/2) \cdot \boldsymbol{m}$. Note that the conditions in the lemma statement imply that $2\,\beta'_e \cdot \sqrt{n} < (q-1)/2$. This gives the final output $\lfloor \boldsymbol{m} + \frac{2}{q-1} \cdot \boldsymbol{e} \rceil = \boldsymbol{m}$ because $\|\frac{2\boldsymbol{e}}{q-1}\|_\infty < 1/2$. $\square$

Choosing Parameters. Looking ahead, we will instantiate this trapdoor for $q \approx 2^{32}$ and $n = 2^{11}$. So, we require $\beta \cdot \beta'_s < p/(4\,n)$ and, say, $\beta \cdot \beta'_e < q/(4\,n \cdot p)$. Picking $p = 2^{16}$, we get $\log(\beta) + \log(\beta'_s) < 16 - 2 - 11 = 3$ and $\log(\beta) + \log(\beta'_e) < 32 - 2 - 11 - 16 = 3$. Picking $\beta \approx 2^2$ and $\beta'_s = \beta'_e \approx 2$ we obtain an NTRU instance requiring BKZ block size 333 to solve (using the (overstretched) NTRU estimator [DvW21]) and an LWE instance requiring BKZ block size 594 to solve (using the lattice estimator [APS15]). According to the cost model from [MAT22] this costs about 2^{132} classical operations.[11]

2.5 Homomorphic Encryption and TFHE

Fully homomorphic encryption (FHE) allows to perform computations on plaintexts by performing operations on ciphertexts. In slightly more detail, an FHE scheme consists of four algorithms: $\mathsf{FHE.KeyGen}, \mathsf{FHE.Enc}, \mathsf{FHE.Eval}, \mathsf{FHE.Dec}$. The key generation, encryption and decryption algorithms all work similarly to normal public key encryption. Together, they provide privacy (i.e. IND-CPA security) and decryption correctness. The interesting part of FHE is its homomorphic property. Assume that $\mathcal{M}$ is the message space, e.g. $\mathcal{M} := \mathbb{Z}_P$. The homomorphic property is enabled by the $\mathsf{FHE.Eval}$ function which takes as input a public key pk, an arbitrary function $f : \mathcal{M}^k \longrightarrow \mathcal{M}$, a sequence of ciphertexts $(c_i)_{i \in [\mathbb{Z}_k]}$ encrypting plaintexts $(m_i)_{i \in \mathbb{Z}_k}$, and outputs a ciphertext $c' \leftarrow \mathsf{FHE.Eval}(\mathsf{pk}, f, (c_0, \ldots, c_{k-1}))$. The homomorphic property ensures that c' is an encryption of $f(m_0, \ldots, m_{k-1})$. Intuitively, FHE allows to perform arbitrary computation on encrypted data without having to decrypt. Importantly, the privacy of the plaintext is maintained. In addition, an FHE scheme may also maintain the privacy of the evaluated computation (see below).

FHE was first realised by Gentry [Gen09]. A considerable amount of influential follow-up research provides the basis of most practically feasible schemes [FV12,BGV11,GSW13,CKKS17]. We will be focusing on an extension of the

[11] We note that quantum algorithms offer only marginal, i.e. less than square-root, speedups here [AGPS20].

third of these works known as TFHE [CGGI20] because its programmable bootstrapping technique lends itself well to our construction. For a good summary of TFHE, see the guide [Joy21].

Programmable bootstrapping. A crucial ingredient of any FHE scheme is a bootstrapping procedure. Essentially, homomorphic evaluation increases ciphertext noise, meaning that after a prescribed number of evaluations, a ciphertext becomes so noisy that it cannot be decrypted correctly. Bootstrapping provides a method of resetting the size of the noise in a ciphertext to allow for correct decryption using some bootstrapping key material. Note that the bootstrapping operation can either produce a ciphertext encrypted under the original key or a new one depending on the bootstrapping key material used.[12]

In TFHE we have access to look-up tables from $\mathbb{Z}_d$ to $\mathbb{Z}_P$ which we will denote by $\mathsf{LUT}_{f(x)}()$ when realising $f(\cdot) : \mathbb{Z}_d \to \mathcal{M}$. For example we may write $\mathsf{LUT}_{x \bmod 3}()$ for the table interpreting entries in $\mathbb{Z}_d$ as integer $\in \{0, \ldots, d-1\}$ and returning the result modulo 3 and treating $\{0, 1, 2\}$ as elements of the plaintext space. TFHE generalises bootstrapping by applying the look-up table to the plaintext at the same time as resetting the size of the noise. Here, d is the degree of a cyclotomic ring $\mathcal{R} = \mathbb{Z}[X]/(X^d + 1)$ and $\mathbb{Z}_P$ is the plaintext space. There is a slight problem here in that the look-up table does not take plaintext-space inputs. However, this is overcome by using appropriate approximations between $\mathbb{Z}_n$ and $\mathbb{Z}_Q$ where n is a plain LWE dimension and Q the ciphertext modulus [CGGI20,Joy21]. We note that this functionality also implies the ability to map from a plaintext space $\mathbb{Z}_{P_0}$ (which is interpreted as an element in $\mathbb{Z}_d$) to a plaintext space $\mathbb{Z}_{P_1}$ with $P_0 \neq P_1$.[13]

Ax-hiding. In order to prevent an attacker from discovering details of the circuit, circuit privacy ensures that the resulting evaluated ciphertext does not leak anything about the circuit (beyond the evaluation of the circuit on a particular point). We relax this notion to allow the client to learn some structure of the circuit, but not the secret input of the server. In our setting we will not be interested in hiding the circuit but only part of the input (the secret key $\boldsymbol{A}$). In particular, we do not aim to hide the lookup tables which are public (essentially: mod p and mod q operations). We call this property **Ax**-hiding, and give the formal definition below. We note, however, that all techniques we are aware of for achieving **Ax**-hiding (see below), including heuristic approaches, seem to also achieve circuit privacy. We still believe formalising the security goal more precisely is a useful approach.

Definition 8. *Let* LUT *be an arbitrary lookup table. Let* $(\boldsymbol{A}, \boldsymbol{x}_1), (\boldsymbol{B}, \boldsymbol{x}_2)$ *have matching dimensions, matching base rings and satisfy* $\mathsf{LUT}(\boldsymbol{A} \cdot \boldsymbol{x}_1) = \mathsf{LUT}(\boldsymbol{B} \cdot \boldsymbol{x}_2)$*. We say an FHE scheme is* $\boldsymbol{Ax}$*-hiding if for any such* $(\boldsymbol{A}, \boldsymbol{x}_1), (\boldsymbol{B}, \boldsymbol{x}_2)$*,* $\underline{\textit{any}}$

[12] This allows to restrict the number of sequential bootstrappings that can be performed, a fact we will rely on below.

[13] This is accomplished by picking the "test polynomial" [Joy21, Sec. 5.3] appropriately.

$(\mathit{FHE}.\mathsf{pk}, \mathit{FHE}.\mathsf{sk}) \leftarrow\!\!\$\ \mathit{FHE}.\mathsf{KeyGen}()$, *any* $\mathsf{ct}_1, \mathsf{ct}_2$ *such that* $\mathit{FHE}.\mathsf{Dec}(\mathsf{sk}, \mathsf{ct}_1) = \boldsymbol{x}_1$ *and* $\mathit{FHE}.\mathsf{Dec}(\mathsf{sk}, \mathsf{ct}_2) = \boldsymbol{x}_2$, *given* $\mathit{FHE}.\mathsf{sk}$ *we have*

$$\mathit{FHE}.\mathsf{Eval}(\mathit{FHE}.\mathsf{pk}, \mathsf{LUT}(\boldsymbol{A} \cdot \star), \mathsf{ct}_1) \approx_s \mathit{FHE}.\mathsf{Eval}(\mathit{FHE}.\mathsf{pk}, \mathsf{LUT}(\boldsymbol{B} \cdot \star), \mathsf{ct}_2).$$

A first approach to achieving circuit privacy and thus **Ax**-hiding is to rely on noise drowning [Gen09]. An improvement to this technique was proposed by Ducas and Stehlé [DS16]. Instead of applying super-polynomial noise flooding in one step, Ducas and Stehlé proceed by iteratively applying the following κ times: perform bootstrapping and add some modest noise flooding (close to the size of the ciphertext noise). For FHEW [DM15] and TFHE, it is estimated that κ may be chosen between 8 and 16. Note that all of the extra bootstrapping is done once at the end of the homomorphic evaluation to "sanitise" the ciphertext and remove all remnants of the circuit that was applied.[14]

An alternative, also achieving full circuit privacy, is to use the work of Kluczniak [Klu22] that applies a randomised bootstrapping procedure. Heuristically, plugging the bounds of [Klu22, Theorem 1] into [Klu22, Lemma 7] suggests the noise grows modestly by a rough factor of $\sqrt{\lambda} \cdot \sqrt{B_{br}}$ where B_{br} is an infinity norm bound on the bootstrapping key noise. We consider this the most promising approach. We give a third, heuristic approach in Appendix A.

Malicious security. Finally, note that the definition above is required to hold for "any" $\mathsf{FHE.pk}, \mathsf{FHE.sk}, \mathsf{ct}_1$ and ct_2 generated honestly. More precisely, this means for *any* possible (i.e. valid) outputs of the appropriate FHE algorithms. For example, the word "any" for an error term distributed as a discrete Gaussian would mean any value satisfying some appropriate bound. Intuitively, malicious circuit privacy (or in our case **Ax**-hiding) requires that an adversary cannot learn anything about the circuit (or $(\boldsymbol{A}, \boldsymbol{x})$) even in the presence of maliciously generated keys and ciphertexts. Thus if we can ensure the well-formedness of keys and ciphertexts, then a semi-honest circuit private FHE scheme is also secure against malicious adversaries. In the random oracle model, we can achieve this with NIZKs that show well-formedness of the keys and that the ciphertext is a valid ciphertext under that public key. Then, $\mathsf{FHE.Eval}$ could explicitly check that the proof verifies and abort otherwise. As noted in [OPP14], the circuit privacy of FHE needs to hold even if the error distributions of the ciphertext are not generated according some canonical distribution, and are simply in the support of valid key/ciphertext pairs. For simplicity, we effectively ignore this fact and note that this caveat has been discussed in prior works, such as in [OPP14]. We note that the techniques from [BdMW16] achieve this enhanced version of circuit privacy.

[14] This adds about 3 seconds of running time to $F_{\mathsf{poprf}}.\mathsf{BlindEval}$ using our implementation from Section 6. We note, however, that this approach is incompatible with our heuristic VOPRF construction.

Table 2. PRF Parameters

	$\lambda = 128$	Explanation		$\lambda = 128$	Explanation
p	2	modulus of $\boldsymbol{x}, \boldsymbol{A}$	n_p, m_p	256	dimensions of $\boldsymbol{A}$
q	3	modulus of $\boldsymbol{z}, \boldsymbol{G}_{\mathsf{inp}}, \boldsymbol{G}_{\mathsf{out}}$	n	128	dim. of $\boldsymbol{x}$ (mod p)
n_q	192	rows of $\boldsymbol{G}_{\mathsf{inp}}$	m	128	dim. of $\boldsymbol{z}$ (mod q)

2.6 Crypto Dark Matter PRF

Let p, q be two primes where $p < q$. We now describe the "Crypto Dark Matter" PRF candidate [BIP+18,DGH+21]. It is built from the following *weak* PRF proposal $F_{\mathsf{weak}} : \mathbb{Z}_p^{m_p \times n_p} \times \mathbb{Z}_p^{n_p} \to \mathbb{Z}_q$ where

$$F_{\mathsf{weak}}(\boldsymbol{A}, \boldsymbol{x}) = \sum_{j=0}^{m_p-1} (\boldsymbol{A} \cdot \boldsymbol{x} \bmod p)_j \bmod q.$$

Here $\boldsymbol{A}$ is the secret key, $\boldsymbol{x}$ is the input and $(\boldsymbol{A} \cdot \boldsymbol{x} \bmod p)_j$ denotes the j-th component of $\boldsymbol{A} \cdot \boldsymbol{x} \bmod p$. In order to describe the strong PRF construction, we introduce a fixed public matrix $\boldsymbol{G}_{\mathsf{inp}} \in \mathbb{Z}_q^{n_q \times n}$ and a p-adic decomposition operation $\mathsf{decomp} : \mathbb{Z}_q^{n_q} \to \mathbb{Z}_p^{\lceil \log_p(q) \rceil \cdot n_q}$ where $\lceil \log_p(q) \rceil \cdot n_q = n_p$. The *strong* PRF candidate is $F_{\mathsf{one}} : \mathbb{Z}_p^{m_p \times n_p} \times \mathbb{Z}_p^{n} \to \mathbb{Z}_q$ where

$$F_{\mathsf{one}}(\boldsymbol{A}, \boldsymbol{x}) \coloneqq F_{\mathsf{weak}}(\boldsymbol{A}, \mathsf{decomp}(\boldsymbol{G}_{\mathsf{inp}} \cdot \boldsymbol{x} \bmod q)).$$

In order to extend the small output of the above PRF constructions, the authors of [BIP+18] introduce another matrix $\boldsymbol{G}_{\mathsf{out}} \in \mathbb{Z}_q^{m \times m_p}$ (with $m < m_p$) which is the generating matrix of some linear code. Then full PRF is $F_{\mathsf{strong}} : \mathbb{Z}_p^{m_p \times n_p} \times \mathbb{Z}_p^{n} \to \mathbb{Z}_q^{m}$ where

$$F_{\mathsf{strong}}(\boldsymbol{A}, \boldsymbol{x}) \coloneqq \boldsymbol{G}_{\mathsf{out}} \cdot (\boldsymbol{A} \cdot \mathsf{decomp}(\boldsymbol{G}_{\mathsf{inp}} \cdot \boldsymbol{x} \bmod q) \bmod p) \bmod q.$$

Given access to gates implementing mod p and mod q this PRF candidate can be implemented in a depth 3 arithmetic circuit. We give an example implementation in Appendix D and in the attachment.[15]

Note that $\mathsf{decomp}(\boldsymbol{G}_{\mathsf{inp}} \cdot \boldsymbol{x} \bmod q) \in \mathbb{Z}_p^{n_p}$ does not depend on the PRF key. Thus, in an OPRF construction it could be precomputed and submitted by the client knowing $\boldsymbol{x}$. However, in this case, we must enforce that the client is doing this honestly via a zero-knowledge proof π that $\boldsymbol{y} \coloneqq \mathsf{decomp}(\boldsymbol{G}_{\mathsf{inp}} \cdot \boldsymbol{x} \bmod q)$ is well formed. Specifically, following [BIP+18], if $\boldsymbol{H}_{\mathsf{inp}} \in \mathbb{Z}_q^{(n_q - n) \times n_q}$ is the parity check matrix of $\boldsymbol{G}_{\mathsf{inp}}$ and $\boldsymbol{G}_{\mathsf{gadget}} \coloneqq (p^{\lceil \log_p(q) \rceil - 1}, \ldots, 1) \otimes \boldsymbol{I}_{n_p}$ we may check

$$\boldsymbol{H}_{\mathsf{inp}} \cdot \boldsymbol{G}_{\mathsf{gadget}} \cdot \boldsymbol{y} \equiv \boldsymbol{0} \bmod q.$$

[15] If the reader's PDF viewer does not support PDF attachments (e.g. Preview on MacOS does not), then e.g. pdfdetach can be used to extract these files.

Note that as stated this does not enforce $\boldsymbol{x} \in \mathbb{Z}_p^n$ but $\boldsymbol{x} \in \mathbb{Z}_q^n$. Since it is unclear if this has a security implication, we may avoid this issue relying on a comment made in [BIP+18] that we may, wlog, replace $\boldsymbol{G}_{\mathsf{inp}}$ with a matrix in systematic (or row echelon) form. That is, writing $\boldsymbol{G}_{\mathsf{inp}} = [\boldsymbol{I} \mid \boldsymbol{A}]^T \in \mathbb{Z}_q^{n_p \times n}$, $\boldsymbol{y} = (\boldsymbol{y}_0, \boldsymbol{y}_1) \in \mathbb{Z}_q^{n_p}$, the *protected encoded-input* PRF is defined as

$$F_{\mathsf{pei}}(\boldsymbol{A}, \boldsymbol{y}) \coloneqq \begin{cases} \boldsymbol{G}_{\mathsf{out}} \cdot (\boldsymbol{A} \cdot \boldsymbol{y} \bmod p) \bmod q & \text{if } \boldsymbol{H}_{\mathsf{inp}} \cdot \boldsymbol{G}_{\mathsf{gadget}} \cdot \boldsymbol{y} \equiv \boldsymbol{0} \bmod q \\ & \text{and } \boldsymbol{y}_0 \in \{0,1\}^n \\ \perp & \text{otherwise.} \end{cases}$$

Note that with this definition of $\boldsymbol{G}_{\mathsf{inp}}$ the "most significant bits" of $\boldsymbol{y}_0$ will always be zero, so there is no point in extracting those when running decomp. Thus, we adapt decomp to simply return the first n output values in $\{0, 1\}$ and to perform the full decomposition on the remaining $(n_q - n)$ entries. We thus obtain $n_p \coloneqq n + \lceil \log_p(q) \rceil \cdot (n_q - n)$. A similar strategy is discussed in Remark 7.13 of the full version of [BIP+18].

Security Analysis. The initial work [BIP+18] provided some initial cryptanalysis and relations to known hard problems to substantiate the security claims made therein. When $\boldsymbol{A}$ is chosen to be a circulant rather than a random matrix, the scheme has been shown to have degraded security [CCKK21] contrary to the expectation stated in [BIP+18]. The same work [CCKK21] also proposes a fix. Further cryptanalysis was preformed in [DGH+21], supporting the initial claims of concrete security. Our choices for $\lambda = 128$ (classically) are aggressive, especially for a post-quantum construction. This is, on the one hand, to encourage cryptanalysis. On the other hand, known cryptanalytic algorithms against the proposals in [BIP+18,DGH+21] require exponential memory in addition to exponential time, a setting where Grover-like square-root speed-ups are less plausible, cf. [AGPS20] (which, however, treats the Euclidean distance rather than Hamming distance).

3 OPRF Candidate

We wish to design an (P)OPRF where the server homomorphically evaluates the PRF using its secret key and uses some form of circuit private FHE to protect its key. We use the same notation as in Section 2.6.

3.1 Extending the PEI PRF

Here, we first observe that the PRFs defined in Section 2.6 trivially fail to achieve pseudorandomness as they map $\boldsymbol{0} \in \mathbb{Z}_p^n \to \boldsymbol{0} \in \mathbb{Z}_q^m$ which holds with $\mathsf{negl}(\lambda)$ probability for a random function. We thus define

$$F_{\mathsf{strong}}(\boldsymbol{A}', \boldsymbol{x}) \coloneqq \boldsymbol{G}_{\mathsf{out}} \cdot (\boldsymbol{A} \cdot (\mathsf{decomp}(\boldsymbol{G}_{\mathsf{inp}} \cdot \boldsymbol{x} \bmod q), 1) \bmod p) \bmod q.$$

and

$$F_{\mathsf{pei}}(\boldsymbol{A}', \boldsymbol{y}) := \begin{cases} \boldsymbol{G}_{\mathsf{out}} \cdot (\boldsymbol{A}' \cdot (\boldsymbol{y}, 1) \bmod p) \bmod q & \text{if } \boldsymbol{H}_{\mathsf{inp}} \cdot \boldsymbol{G}_{\mathsf{gadget}} \cdot \boldsymbol{y} \equiv \boldsymbol{0} \bmod q \\ \perp & \text{otherwise.} \end{cases}$$

for $\boldsymbol{A}' \in \mathbb{Z}_p^{m_p \times (n_p+1)}$, i.e. extended by one column. Furthermore, we wish to support an additional input $\boldsymbol{t} \in \mathbb{Z}_p^n$ to be submitted in the clear. For this, we deploy the standard technique of using a key derivation function to derive a fresh key per tag $\boldsymbol{t}$ [CHL22,JKR18]. In particular, let $\mathsf{RO}_{\mathsf{key}} : \mathbb{Z}_p^{m_p \times n_p} \times \mathbb{Z}_p^n \to \mathbb{Z}_p^{m_p \times (n_p+1)}$ be a random oracle, we then define our PRF candidate $F_{\boldsymbol{A}}^{\mathsf{RO}_{\mathsf{key}}}(\boldsymbol{t}, \boldsymbol{x})$ in Algorithm 1. Clearly, if $F_{\mathsf{pei}}(\boldsymbol{A}', \boldsymbol{y})$ is a PRF then $F_{\boldsymbol{A}}^{\mathsf{RO}_{\mathsf{key}}}(\cdot, \cdot)$ is a PRF with input $(\boldsymbol{t}, \boldsymbol{x})$ as $\boldsymbol{A}_{\boldsymbol{t}}$ in Algorithm 1 is simply a fresh $F_{\mathsf{pei}}()$ key for each distinct value of $\boldsymbol{t}$.

Algorithm 1 $F_{\boldsymbol{A}}^{\mathsf{RO}_{\mathsf{key}}}(\boldsymbol{t}, \boldsymbol{x})$

```
Input: $\boldsymbol{A} \in \mathbb{Z}_p^{m_p \times n_p}$,   $\boldsymbol{x} \in \mathbb{Z}_p^n$,   $\boldsymbol{t} \in \mathbb{Z}_p^n$
Output: $F_{\boldsymbol{A}}(\boldsymbol{t}, \boldsymbol{x})$
    $\boldsymbol{A}_{\boldsymbol{t}} \leftarrow \mathsf{RO}_{\mathsf{key}}(\boldsymbol{A}, \boldsymbol{t})$
    $\boldsymbol{y} \leftarrow \mathsf{decomp}(\boldsymbol{G}_{\mathsf{inp}} \cdot \boldsymbol{x} \bmod q)$
    $\boldsymbol{z} \leftarrow \boldsymbol{G}_{\mathsf{out}} \cdot (\boldsymbol{A}_{\boldsymbol{t}} \cdot (\boldsymbol{y}, 1) \bmod p) \bmod q$
    return $\boldsymbol{z}$
```

3.2 TFHE-based Instantiation

Our main TFHE-based instantiation is given in Figure 3. An important alteration to the FHE key generation is that $\mathsf{FHE.KeyGen}^{(\mathsf{pp})}$ will output a commitment to secret key $\mathsf{FHE.sk} = \boldsymbol{s} \in \mathbb{Z}_2^{\mathfrak{e}}$ in addition to the standard public key $\mathsf{FHE.pk}$. In particular, $\mathsf{FHE.KeyGen}^{(\mathsf{pp})}()$ begins by running $(\mathsf{FHE.pk}, \mathsf{FHE.sk}) \leftarrow\$ \mathsf{FHE.KeyGen}()$ and then adds the commitment $\boldsymbol{b}_{\mathsf{pk}}$ to $\mathsf{FHE.pk}$. This commitment takes the form $\boldsymbol{b}_{\mathsf{pk}} = \boldsymbol{A}_{\mathsf{pp}} \cdot \boldsymbol{r} + \boldsymbol{e} + \lfloor Q/2 \rfloor \cdot (\boldsymbol{s}, \boldsymbol{0}) \in \mathbb{Z}_Q^N$ where Q is the ciphertext modulus, $\boldsymbol{r}, \boldsymbol{e} \leftarrow\$ (\chi')^N$ where χ' is a discrete Gaussian of standard deviation $\beta' \approx 4$ and $N = 2048$ as in Section 2.4. One can view $\boldsymbol{b}_{\mathsf{pk}}$ as a partial symmetric LWE encryption of the secret key from the (T)LWE encryption scheme within TFHE, so χ' is simply an error distribution. Therefore, using the same LWE assumption from Section 2.4, $\boldsymbol{b}_{\mathsf{pk}}$ is indistinguishable from random and it is easy to check that its presence does not affect the IND-CPA property of TFHE. Furthermore, since $\boldsymbol{b}_{\mathsf{pk}}$ is simply a randomised function of $(\mathsf{FHE.pk}, \mathsf{FHE.sk})$, it can be constructed by an adversarial client. Thus, its advantage against the **Ax**-hiding property (or semi-honest circuit privacy) of FHE in which the key also contains $\boldsymbol{b}_{\mathsf{pk}}$ remains unchanged. To summarise, the public key material output by $\mathsf{FHE.KeyGen}^{(\mathsf{pp})}$ is $\mathsf{FHE.pk}^{(\mathsf{pp})} := (\boldsymbol{b}_{\mathsf{pk}}, \mathsf{FHE.pk})$.

Note that although the server does not need to create encryptions itself, we still use the public key encryption version of TFHE rather than a symmetric key

```
$F_{\mathsf{poprf}}.\mathsf{Setup}(1^\lambda)$
    $\boldsymbol{A}_{\mathsf{pp}} \leftarrow\!\!\$\ \mathbb{Z}_Q^{N \times N}$
    $\mathsf{pp} \leftarrow \boldsymbol{A}_{\mathsf{pp}}$
    return $\mathsf{pp}$

$F_{\mathsf{poprf}}.\mathsf{KeyGen}(1^\lambda)$
    $\mathsf{pk} \leftarrow \perp$
    $\mathsf{sk} \leftarrow\!\!\$\ \mathbb{Z}_p^{m_p \times n_p}$
    return $(\mathsf{pk}, sk)$

$F_{\mathsf{poprf}}.\mathsf{Eval}(\mathsf{sk} = \boldsymbol{A}, t = \boldsymbol{t}, x = \boldsymbol{x})$
    $\boldsymbol{z}' := F_{\boldsymbol{A}}^{\mathsf{RO}_{\mathsf{key}}}(\boldsymbol{t}, \boldsymbol{x})$
    $\boldsymbol{z} := \mathsf{RO}_{\mathsf{fin}}(\boldsymbol{t}, \boldsymbol{x}, \boldsymbol{z}')$
    return $\boldsymbol{z}$

$F_{\mathsf{poprf}}.\mathsf{Finalise}(rep = \mathsf{ct})$
    if $\mathsf{ct}'$ not a ctxt then return $\perp$
    $\boldsymbol{z}' \leftarrow \mathsf{FHE.Dec}(\mathsf{FHE.sk}, \mathsf{ct})$
    $\boldsymbol{z} \leftarrow \mathsf{RO}_{\mathsf{fin}}(\boldsymbol{t}, \boldsymbol{x}, \boldsymbol{z}')$
    return $\boldsymbol{z}$

$F_{\mathsf{poprf}}.\mathsf{Request}(\mathsf{pk}, t = \boldsymbol{t}, x = \boldsymbol{x}; pp)$
    $\mathsf{FHE.pk}^{(\mathsf{pp})}, \mathsf{FHE.sk} \leftarrow\!\!\$\ \mathsf{FHE.KeyGen}^{(\mathsf{pp})}()$
    $\boldsymbol{y} \leftarrow \mathsf{decomp}(\boldsymbol{G}_{\mathsf{inp}} \cdot \boldsymbol{x} \bmod p)$
    $\mathsf{ct} \leftarrow\!\!\$\ \mathsf{FHE.Enc}(\mathsf{FHE.sk}, \boldsymbol{y})$
    $\pi \leftarrow\!\!\$\ \mathsf{NIZKAoK_C}(\mathsf{FHE.pk}^{(\mathsf{pp})}, \mathsf{ct}; \mathsf{FHE.sk}, x)$
    $req \leftarrow (\mathsf{FHE.pk}^{(\mathsf{pp})}, \mathsf{ct}, \pi, t)$
    return $req$

$F_{\mathsf{poprf}}.\mathsf{BlindEval}(sk = \boldsymbol{A}, t = \boldsymbol{t}, req; \mathsf{pp})$
    $(\mathsf{FHE.pk}^{(\mathsf{pp})}, \mathsf{ct}, \pi) \leftarrow req$
    $\boldsymbol{A}_{\boldsymbol{t}} \leftarrow \mathsf{RO}_{\mathsf{key}}(\boldsymbol{A}, \boldsymbol{t})$
    $\mathsf{ct}' \leftarrow F_{\mathsf{poprf}}.\mathsf{HEEval}(\mathsf{FHE.pk}^{(\mathsf{pp})}, \boldsymbol{A}_{\boldsymbol{t}}, \boldsymbol{t}, \mathsf{ct})$
    if $\pi$ does not verify then $\mathsf{ct}' = \perp$
    $rep \leftarrow \mathsf{ct}'$
    return $rep$
```

Client | **Server**

$F_{\mathsf{poprf}}.\mathsf{Request}(\mathsf{pk}, t = \boldsymbol{t}, x = \boldsymbol{x}; \mathsf{pp})$

$\xrightarrow{req = (\mathsf{FHE.pk}^{(\mathsf{pp})}, \mathsf{ct}, \pi, t)}$ $F_{\mathsf{poprf}}.\mathsf{BlindEval}(sk = \boldsymbol{A}, t = \boldsymbol{t}, req)$

$F_{\mathsf{poprf}}.\mathsf{Finalise}(rep)$ $\xleftarrow{rep = \mathsf{ct}'}$

Output: $\mathsf{RO}_{\mathsf{fin}}(\boldsymbol{t}, \boldsymbol{x}, F_{\boldsymbol{A}}^{\mathsf{RO}_{\mathsf{key}}}(\boldsymbol{t}, \boldsymbol{x}))$

Fig. 3. Main construction.

version. The extra key material in the public key version does not affect efficiency in any noticeable way as the bootstrapping key sizes are the main bottleneck. This can be seen in Appendix B where details of $\mathsf{NIZKAoK_C}$ are elaborated and where we fully describe the generation of the bootstrapping keys.

Remark 4. The ciphertext component encrypting 1 from the input of Algorithm 2 may be produced by the server (without knowledge of $\boldsymbol{y}$).

Choosing Parameters & Size Estimates. We require that binary-secret LWE is hard for (e, Q, σ) where e is the dimension of the LWE secret, Q the modulus and σ the standard deviation of the noise. We also require M-LWE to be hard for $(\ell, d, Q, \bar{\sigma})$ where d is the ring dimension, ℓ is the module dimension and $\bar{\sigma}$ is the standard deviation of the noise. To pick parameters, we simply follow the `tfhe-rs` TFHE library [CGGI20] and its (`PARAM_MESSAGE_2_CARRY_0`) parameter set, which permits us to set $Q = 2^{32}$ and then fixes $e = 656, \ell = 2, d = 512, \sigma = 2^{17.1}, \bar{\sigma} = 2^{7.5}$. We note that this is likely not optimal as these parameters are meant to support a wide range of computations. We estimate the size of the bootstrapping key (which may be considered an amortisable offline communication cost) in Appendix B as 16.8MB. We also estimate the size of the zero-knowledge proofs accompanying this key as 41.42KB.

Algorithm 2 $F_{\mathsf{poprf}}.\mathsf{HEEval}$

```
Input: pk HE public key    ▷ with ciphertext modulus $Q$, plaintext modulus $P$
Input: $\boldsymbol{A} \in \mathbb{Z}_p^{m_p \times n_p}$,  $\boldsymbol{t} \in \mathbb{Z}_p^n$,  $\mathsf{ct} \in \mathcal{C}^{n_p}$ encrypting $\boldsymbol{y} \in \mathbb{Z}_p^{n_p}$
Output: $\mathsf{ct}' \in \mathcal{C}^m$ encrypting $F_{\boldsymbol{A}}^{\mathsf{RO}_{\mathsf{key}}}(\boldsymbol{t}, \boldsymbol{x})$ or $\perp$
 1: $\boldsymbol{A_t} \leftarrow \mathsf{RO}_{\mathsf{key}}(\boldsymbol{A}, \boldsymbol{t})$ in the clear.
 2: $\mathsf{ct}' \leftarrow\!\!\$\ \mathsf{FHE.Enc}(\mathsf{pk}, 1)$    ▷ can be e.g. $1 \cdot \lfloor Q/P \rceil$
 3: $\mathsf{ct}^{(1)} \leftarrow \mathsf{FHE.Eval}(\mathsf{pk}, \boldsymbol{A_t} \cdot (\mathsf{ct}, \mathsf{ct}'))$
 4: $\mathsf{ct}_i^{(2)} \leftarrow \mathsf{FHE.Eval}(\mathsf{pk}, \mathsf{LUT}_{x \bmod p}(\mathsf{ct}_i^{(1)}))$  $\forall i \in \mathbb{Z}_{m_p}$    ▷ mod $p$, depth $\geq 1$
 5: $\mathsf{ct}^{(3)} \leftarrow \mathsf{FHE.Eval}(\mathsf{pk}, \boldsymbol{G}_{\mathsf{out}} \cdot \mathsf{ct}^{(2)})$
 6: $\mathsf{ct}_i' \leftarrow \mathsf{FHE.Eval}(\mathsf{pk}, \mathsf{LUT}_{x \bmod q}(\mathsf{ct}_i^{(3)}))$  $\forall i \in \mathbb{Z}_m$    ▷ mod $q$, depth $\geq 1$
 7: return $\mathsf{ct}'$
```

Each request then requires to send LWE encryptions of $n_p = 256$ bits $\boldsymbol{m}$ using protected encoded inputs, i.e. $(\boldsymbol{A}', \boldsymbol{b} \coloneqq \boldsymbol{A}' \cdot \boldsymbol{s} + \boldsymbol{e} + \lfloor Q/P \rfloor \cdot \boldsymbol{m})$ where $\boldsymbol{A}' \in \mathbb{Z}_Q^{n_p \times e}$ and $\boldsymbol{b} \in \mathbb{Z}_Q^{n_p}$. Here, $\boldsymbol{A}'$ can be computed from a small seed of 256 bits. For $\boldsymbol{b}$ we need to transmit $n_p \cdot \log Q$ bits. However, as noted in e.g. [LDK+22], given the large size of the noise ($\sigma = 2^{17.1}$) we may drop the least significant, say, 16 bits, reducing the cost of $\boldsymbol{b}$ to $16 \cdot 256$. In total we have a ciphertext size of 0.5kB. The accompanying zero-knowledge proofs take up 26.7kB but can be amortised to cost about 1.3kB per query when sending 64 queries in one shot (Appendix B).

The message back from the server is $m = 128$ encryptions of the output elements $\in \mathbb{Z}_q$. We can also drop the least significant bits of $\boldsymbol{A}'$, since $\boldsymbol{s}$ is binary. In particular, we may drop, say, the eight least significant bits and we arrive at $e \cdot m \cdot 24$ for $\boldsymbol{A}'$. We need $m \log Q$ bits for $\boldsymbol{b}$. We can use the same trick of dropping lower order bits for $\boldsymbol{b}$ again, so we obtain $128 \cdot 16$. In total we get 246.2kB. We note that it is more efficient to pack all return values into a single RLWE sample using techniques from [CDKS21], since the cost of transmitting $\boldsymbol{A}'$ dominates here. This increases the public key size (since we would include another key switching key) but reduce the size of response to about 3.2kB. For more details on these values, see Appendix B.1. We give our code for estimating parameters in Appendices E and F.

Remark 5. Looking ahead, we will see that our implementation in Section 6 does not achieve these sizes. The reason for this is that (a) $\boldsymbol{A}'$ is not compressed as above and (b) the API we are using defaults to using M-LWE ciphertexts with dimension $\ell \cdot d = 1024$ per bit.

4 Security Proofs

We first prove the pseudorandomness property against malicious clients in Theorem 1 and then privacy (POPRIV1 only) against servers in Theorem 2.

Theorem 1. *Let FHE denote the TFHE scheme. The construction* F_{poprf} *from Figure 3 satisfies the POPRF property from Definition 3, with random oracles* $\mathsf{RO}_{\mathsf{fin}}$ *and* $\mathsf{RO}_{\mathsf{key}}$*, if:*

– *The client zero-knowledge proof is sound*
– *FHE is $\boldsymbol{Ax}$-hiding*
– *The* $\mathsf{mat\text{-}NTRU}_{N,P',Q,B}$ *assumption holds where Q is the FHE modulus.*
– *P' is even and coprime to Q such that $Q > BN(\beta' + P' \cdot (2\beta' + 1)/2)$ where β' is the standard deviation used in FHE.*$\mathsf{KeyGen}^{(\mathsf{pp})}$
– *$F.(\cdot, \cdot)$, defined in Section 3, is a PRF with output range super-polynomial in the security parameter*[16].

Proof. The following simulator S will be used to prove the result:

$\mathsf{S.Init}$**:** Sample $\boldsymbol{A}^{(\mathsf{S})} \leftarrow\!\!\$\ \mathbb{Z}_p^{m_p \times n_p}$ and $\boldsymbol{A}_{\mathsf{pp}}^{(\mathsf{S})} \leftarrow\!\!\$\ \mathbb{Z}_Q^{N \times N}$. Set $st_{\mathsf{S}} = \boldsymbol{A}^{(\mathsf{S})}$, $\mathsf{pp}_0 \coloneqq \boldsymbol{A}_{\mathsf{pp}}^{(\mathsf{S})}$ and $\mathsf{pk}_0 = \perp$.

$\mathsf{S.BlindEval}(t, req, st_{\mathsf{S}})$**:** Return $\mathcal{F}.\mathsf{BlindEval}_{\mathsf{pp}_0}^{\mathsf{RO}_{\mathsf{key}}}(\boldsymbol{A}^{(\mathsf{S})}, t, req)$. Note that this algorithm does not need to make any calls to LimitEval.

$\mathsf{S.Eval}^{\mathrm{LimitEval}}(\boldsymbol{x}_{\mathsf{in}}, st_{\mathsf{S}})$**:** If query $\boldsymbol{x}_{\mathsf{in}} \coloneqq (\boldsymbol{t}, \boldsymbol{x}, \boldsymbol{z})$ appears in st_{S} as a previous query, return the same answer. If $\boldsymbol{z} \neq F_{\boldsymbol{A}^{(\mathsf{S})}}(\boldsymbol{t}, \boldsymbol{x})$ return a uniformly random h and store $(\boldsymbol{x}_{\mathsf{in}}, h)$ in st_{S}. If $\boldsymbol{z} = F_{\boldsymbol{A}^{(\mathsf{S})}}(\boldsymbol{t}, \boldsymbol{x})$, query LimitEval$(\boldsymbol{x}_{\mathsf{in}})$ and return its answer $\boldsymbol{h}'$, storing $(\boldsymbol{x}_{\mathsf{in}}, \boldsymbol{h}')$ in st_{S}.

Define G_0 to be the $\mathrm{POPRF}_{\mathcal{F},\mathsf{Sim},H}^{\mathcal{A},b=0}$ game and G_1 to be the $\mathrm{POPRF}_{\mathcal{F},\mathsf{Sim},H}^{\mathcal{A},b=1}$ game. Note that $\mathcal{A}$ has oracle access to Eval, BlindEval and Prim. These three oracles behave (jointly) identically in G_0 and G_1 as long as S does not get $\perp$ in a LimitEval query when answering $\mathsf{S.Eval}$. Therefore, the distinguishing advantage between G_0 and G_1 is at most the probability that after making q BlindEval queries, $\mathcal{A}$ has managed to submit $q + 1$ distinct tuples of the form $(\boldsymbol{t}_j, \boldsymbol{x}_j, F_{\boldsymbol{A}^{(\mathsf{S})}}(\boldsymbol{t}_j, \boldsymbol{x}_j))$ to Prim in G_0. Denote this event E. To complete the proof, we bound $\Pr[\mathsf{E}]$ using Lemma 2. □

Lemma 2. *Assume that all of the conditions in Theorem 1 hold. Then* $\Pr[\mathsf{E}]$ *is negligible.*

Proof (Of Lemma 2). Note that every adversarial input to the BlindEv oracle is either well-formed, or answered with $rep = \perp$ due to the soundness of the client zero-knowledge proof. Therefore, we know that with overwhelming probability, the ciphertext and $(\mathsf{FHE.pk}^{(\mathsf{pp})}, \mathsf{FHE.sk})$ that the malicious client uses is of the correct form in the event that the response is not $\perp$. We now describe hybrid games $\mathcal{H}_i$ and events E_i for $i \in \{1, 2, 3\}$:

$\mathcal{H}_1$: Here S is changed to S_1, where S_1 is identical to S apart from that pp_0 is replaced by $\mathsf{pp}'_0 = \boldsymbol{A}'$ where $(\boldsymbol{A}', \tau) \leftarrow\!\!\$\ \mathsf{NTRUTrapGen}(N, Q, P', B)$ and τ is added to the initial st_{S} (but is unused throughout). Here, $P > 4N$ is a power of 2 such that $Q > BN(\beta' + P' \cdot (2\beta' + 1)/2)$ with β' the standard deviation of $\boldsymbol{r}$ and $\boldsymbol{e}$ used to produce the commitment part $\boldsymbol{b}_{\mathsf{pk}}$ of $\mathsf{FHE.pk}^{(\mathsf{pp})}$ (see Section 3.2). E_1 is defined as the event that after q BlindEval queries, $\mathcal{A}$ has managed to submit $q+1$ distinct pairs of the form $(\boldsymbol{t}_j, \boldsymbol{x}_j, F_{\boldsymbol{A}^{(\mathsf{S})}}(\boldsymbol{t}_j, \boldsymbol{x}_j))$ to Prim.

[16] e.g. 2^λ works.

$\mathcal{H}_2$: S_1 is modified to S_2, where S_2 is identical to S_1 apart from that S_2.BlindEval is modified in the following way. On input

$$req = (\mathsf{FHE.pk}^{(\mathsf{pp})} := (\boldsymbol{b}_{\mathsf{pk}}, \mathsf{FHE.pk}), \mathsf{ct}, \pi, t),$$

S_2 checks π. If the proof verifies then it sets $\boldsymbol{s} := \mathsf{NTRUDec}(\boldsymbol{b}_{\mathsf{pk}}, \tau)$, $\boldsymbol{y} := \mathsf{FHE.Dec}(\boldsymbol{s}, \mathsf{ct})$, computes $\boldsymbol{x}$ from $\boldsymbol{y}$ using $\boldsymbol{G}_{\mathsf{inp}}$ in systematic form, and computes $\boldsymbol{z} = F_{\boldsymbol{A}^{(\mathsf{S})}}(\boldsymbol{t}, \boldsymbol{x})$ as in the original construction. It then samples a matrix $\boldsymbol{B} \in \mathbb{Z}_2^{m_p \times n_p}$ and further computes $\boldsymbol{w} = \boldsymbol{B} \cdot \boldsymbol{y} \bmod p$. Then, it is easy to compute some $\boldsymbol{G}'_{\mathsf{out}} \in \mathbb{Z}_q^{m \times m_p}$ such that $\boldsymbol{z} = \boldsymbol{G}'_{\mathsf{out}} \cdot \boldsymbol{w} \bmod q$. Finally, define F' to be the function F but using $\boldsymbol{G}'_{\mathsf{out}}$ instead of $\boldsymbol{G}_{\mathsf{out}}$. It returns $\mathsf{FHE.Eval}(\mathsf{FHE.pk}, F'_{\boldsymbol{B}^{(\mathsf{S})}}(\boldsymbol{t}, \cdot), \mathsf{ct})$ in response to req. E_2 is defined as the event that after q BlindEval queries, $\mathcal{A}$ has managed to submit $q+1$ distinct pairs of the form $(\boldsymbol{t}_j, \boldsymbol{x}_j, F_{\boldsymbol{A}^{(\mathsf{S})}}(\boldsymbol{t}_j, \boldsymbol{x}_j))$ to Prim.

$\mathcal{H}_3$: S_3 is S_2 apart from S_3.BlindEval and S_3.Eval. In these two functions, S_3 replaces all computations of $F_{\boldsymbol{A}^{(\mathsf{S})}}(\cdot, \cdot)$ by a truly random function G. Additionally, S_3 does not sample $\boldsymbol{A}^{(\mathsf{S})}$ during $\mathsf{S}_3.\mathsf{Init}$. E_3 is defined as the event that after q BlindEval queries, $\mathcal{A}$ has managed to submit $q+1$ distinct pairs of the form $(\boldsymbol{t}_j, \boldsymbol{x}_j, \mathsf{G}(\boldsymbol{t}_j, \boldsymbol{x}_j))$ to Prim.

We first show that $|\Pr[\mathsf{E}] - \Pr[\mathsf{E}_1]| = \mathsf{negl}(\lambda)$ by the $\mathsf{mat}\text{-}\mathsf{NTRU}_{N,P',Q,B}$ assumption. In order to do so, we build a matrix NTRU distinguisher $\mathcal{B}_{\mathsf{mat}\text{-}\mathsf{NTRU}}$ that implements S for the POPRF adversary $\mathcal{A}$ using its $\mathsf{mat}\text{-}\mathsf{NTRU}_{N,P',Q,B}$ challenge as $\boldsymbol{A}_{\mathsf{pp}}$. Clearly, if the challenge is a uniform matrix, $\mathcal{B}_{\mathsf{mat}\text{-}\mathsf{NTRU}}$ perfectly recreates S whereas if the challenge is non-uniform, it perfectly simulates S_1 from $\mathcal{A}$'s perspective. Therefore, if $|\Pr[\mathsf{E}] - \Pr[\mathsf{E}_i]|$ was not negligible, then $\mathcal{B}_{\mathsf{mat}\text{-}\mathsf{NTRU}}$ would be able to distinguish and break the $\mathsf{mat}\text{-}\mathsf{NTRU}_{N,P,Q}$ assumption by testing whether $\mathcal{A}$ manages to submit $q+1$ distinct tuples of the form $(\boldsymbol{t}_j, \boldsymbol{x}_j, F_{\boldsymbol{A}^{(\mathsf{S})}}(\boldsymbol{t}_j, \boldsymbol{x}_j))$ to Prim given just q BlindEval queries.

Next, we show that $|\Pr[\mathsf{E}_1] - \Pr[\mathsf{E}_2]| = \mathsf{negl}(\lambda)$ by the $\mathbf{Ax}$-hiding property of TFHE. To do so, we consider a sequence of hybrid events $\mathsf{E}_{1,i}$ where simulator $\mathsf{S}_{1,i}$ answers the first i calls to BlindEval as in S_1 and the remainder as in S_2. Clearly, $\mathsf{E}_1 = \mathsf{E}_{1,0}$ and $\mathsf{E}_{1,q_t} = \mathsf{E}_2$ if q_t is a polynomial upper bound on the number of BlindEval queries. Suppose there is an index $i^* \in [q_t]$ such that $|\Pr[\mathsf{E}_{1,i^*}] - \Pr[\mathsf{E}_{1,i^*+1}]|$ is non-negligible. Note that if the (i^*+1)-th request did not have a correctly verifying proof, then $\Pr[\mathsf{E}_{1,i^*}] - \Pr[\mathsf{E}_{1,i^*+1}] = 0$. Therefore, we assume that the proof in the (i^*+1)-th query verifies and all ciphertexts/keys in the request are well-formed. This tell us that the $\mathsf{NTRUDec}$ correctly recovers the FHE secret key which in turn implies that the $\boldsymbol{y}$ and $\boldsymbol{x}$ recovered are the correct ones. Therefore, if $|\Pr[\mathsf{E}_{1,i^*}] - \Pr[\mathsf{E}_{1,i^*+1}]|$ is non-negligible, there would exist a $\mathsf{FHE.pk}, \mathsf{FHE.sk}, \mathsf{ct} = \mathsf{FHE.Enc}(\mathsf{FHE.pk}, \boldsymbol{x})$ (in particular, the key-pair and ciphertext associated to $\mathcal{A}$'s (i^*+1)-th BlindEval query) that allows an efficient distinguisher $\mathcal{B}$ between

$$\mathsf{FHE.Eval}(\mathsf{FHE.pk}, F_{\boldsymbol{A}^{(\mathsf{S})}}(\boldsymbol{t}, \cdot), \mathsf{ct}) \tag{1}$$

and

$$\mathsf{FHE.Eval}(\mathsf{FHE.pk}, F'_{\boldsymbol{B}^{(\mathsf{S})}}(\boldsymbol{t}, \cdot), \mathsf{ct}) \tag{2}$$

for the value of $\boldsymbol{t}$ from the (i^*+1)-th query. Note that by construction $F'_{\boldsymbol{B}^{(\mathsf{S})}}(\boldsymbol{t},\boldsymbol{x}) = F_{\boldsymbol{A}^{(\mathsf{S})}}(\boldsymbol{t},\boldsymbol{x})$ yet $F'_{\boldsymbol{B}^{(\mathsf{S})}}$ contains no secret information.

Essentially, $\mathcal{B}$ implements S_{1,i^*} for $\mathcal{A}$ apart from the (i^*+1)-th BlindEval query. Suppose $\mathcal{B}$ is given $\mathsf{FHE.pk}, \boldsymbol{t}$ and ct in the (i^*+1)-th query. Suppose also that it is given a challenge ciphertext ct' that is either of the form (1) or (2), and that it uses ct' to respond to the (i^*+1)-th BlindEval query. If ct' is of the first form, then $\mathcal{B}$ implements S_{1,i^*} and if it is of the latter form, $\mathcal{B}$ implements S_{1,i^*+1}. Note that, the forms (1) and (2) can be rewritten as the $\mathsf{FHE.Eval}(\mathsf{FHE.pk}, \mathsf{LUT}(\boldsymbol{G}_{\mathsf{out}}\cdot\star), \tilde{\mathsf{ct}}_1)$ and $\mathsf{FHE.Eval}(\mathsf{FHE.pk}, \mathsf{LUT}(\boldsymbol{G}'_{\mathsf{out}}\cdot\star), \tilde{\mathsf{ct}}_2)$ respectively, where $\tilde{\mathsf{ct}}_i$ are the intermediate ciphertexts that encrypt the vectors $\tilde{\boldsymbol{w}}_i$ of intermediate evaluations in the two cases. Therefore, distinguishing between ct' of form (1) and (2) is precisely breaking the $\mathbf{Ax}$-hiding property. Assume $\mathcal{B}$ decides its output by checking whether $\mathcal{A}$ manages to submit $q+1$ distinct pairs of the form $(\boldsymbol{t}_j, \boldsymbol{x}_j, F_{\boldsymbol{A}^{(\mathsf{S})}}(\boldsymbol{t}_j,\boldsymbol{x}_j))$ to Prim given just q BlindEval queries. If $|\Pr[\mathsf{E}_{1,i^*}] - \Pr[\mathsf{E}_{1,i^*+1}]|$ is non-negligible, then $\mathcal{B}$ can distinguish the two cases for ct' and thus break $\mathbf{Ax}$-hiding.

Next, we show $|\Pr[\mathsf{E}_2]-\Pr[\mathsf{E}_3]| = \mathsf{negl}(\lambda)$. Suppose that this was not the case. Then we construct an algorithm $\mathcal{B}$ that distinguishes $F_{\boldsymbol{A}^{(\mathsf{S})}}(\cdot,\cdot)$ from uniform random. $\mathcal{B}$ interacts with a PRF challenger, and uses it to implement S_2 by querying the PRF challenger on input $(\boldsymbol{t},\boldsymbol{x})$ instead of computing $F_{\boldsymbol{A}^{(\mathsf{S})}}(\boldsymbol{t},\boldsymbol{x})$. In doing this, if the PRF challenger is returning uniform values, $\mathcal{B}$ simulates S_3 for $\mathcal{A}$. Otherwise, $\mathcal{B}$ perfectly simulates S_2 for $\mathcal{A}$. Therefore, if $\Pr[\mathsf{E}_2] - \Pr[\mathsf{E}_3]|$ was not negligible, then $\mathcal{B}$ could distinguish F from a PRF by checking whether $\mathcal{A}$ manages to submit $q+1$ distinct pairs of the form $(\boldsymbol{t}_j, \boldsymbol{x}_j, \boldsymbol{z}_j))$ to Prim such that $\boldsymbol{z}_j$ agrees with the PRF oracle on input $(\boldsymbol{t}_j, \boldsymbol{x}_j)$ given just q BlindEval queries. To complete the proof, we use the next lemma. □

Lemma 3. $\Pr[\mathsf{E}_3] = \mathsf{negl}(\lambda)$ *assuming the PRF output space is super-polynomial in the security parameter.*

Proof. Recall that when considering event E_3, the adversary $\mathcal{A}$ has oracle access to Eval, BlindEval and Prim. Further, these oracles are implemented by S_3 using a truly random function G instead of $F_{\boldsymbol{A}^{(\mathsf{S})}}$. $\mathcal{A}$ has access to G outputs through BlindEval queries. In particular, one G output is computed for every distinct BlindEval query. The oracles Eval and Prim can be used by $\mathcal{A}$ to see whether a query *is not of the form* $(\boldsymbol{t}_j, \boldsymbol{x}_j, \mathsf{G}(\boldsymbol{t}_j, \boldsymbol{x}_j))$ through consistency. This is the full extent to which $\mathcal{A}$ has access to G. Therefore, from the perspective of $\mathcal{A}$, G is a random oracle with an additional oracle that tells if a guessed output is *incorrect.* Given this setup, an adversary has a negligible probability of producing $q+1$ outputs of G if it only knows q evaluations of G if the output space of G is super-polynomial. Therefore, $\Pr[\mathsf{E}_3] = \mathsf{negl}(\lambda)$. □

Theorem 2. *Let* FHE *denote the TFHE scheme. The construction $\mathcal{F}$ from Figure 3 satisfies the POPRIV1 property if the following hold:*

- FHE *is IND-CPA.*
- *The client proof is zero-knowledge.*

– *The* $\mathsf{LWE}_{Q,N,N,\chi',\chi'}$ *assumption holds where* $\chi' = D_{\mathbb{Z},\beta'}$ *is the error distribution used in* $\mathit{FHE}.\mathsf{KeyGen}^{(\mathsf{pp})}$.

Proof. Let G_0 and G_1 denote the $\mathit{POPRIV1}^{\mathcal{A},b}_{\mathcal{F},H}$ game for $b = 0$ and $b = 1$ respectively. Furthermore, let $\bar{\mathrm{G}}_0$ and $\bar{\mathrm{G}}_1$ be the G_0 and G_1 modified so that all Trans oracle queries have their zero-knowledge proofs in the requests replaced by *simulated* zero-knowledge proofs. Clearly, $\mathrm{G}_0 \approx_c \bar{\mathrm{G}}_0$ and $\mathrm{G}_1 \approx_c \bar{\mathrm{G}}_1$ by the zero-knowledge property of the client proofs. Next, we let G'_0 and G'_1 be the same as $\bar{\mathrm{G}}_0$ and $\bar{\mathrm{G}}_1$ apart from the way all public keys of the form $\mathsf{FHE.pk}^{(pp)} := (\boldsymbol{b}_{\mathsf{pk}}, \mathsf{FHE.pk})$ are sampled. In particular, $\mathsf{FHE.pk}$ will remain the same, however, $\boldsymbol{b}_{\mathsf{pk}}$ will be replaced by uniform random values $\boldsymbol{u}_{\mathsf{pk}}$. Note that $\boldsymbol{b}_{\mathsf{pk}} = \boldsymbol{A}_{pp} \cdot \boldsymbol{r} + \boldsymbol{e} + \lfloor Q/2 \rceil \cdot \boldsymbol{s}$ is an LWE encryption. As we argue next, $\bar{\mathrm{G}}_0 \approx_c \mathrm{G}'_0$ and $\bar{\mathrm{G}}_1 \approx_c \mathrm{G}'_1$ assuming the $\mathsf{LWE}_{Q,N,N,\chi',\chi'}$ assumption holds. Suppose we want to prove $\bar{\mathrm{G}}_0 \approx_c \mathrm{G}'_0$. This can be formally argued by building a distinguisher $\mathcal{B}$ between an M multi-secret LWE challenge of the form $(\boldsymbol{A}, \boldsymbol{B}) \in \mathbb{Z}_Q^{N\times N} \times \mathbb{Z}_Q^{N\times M}$ where $\boldsymbol{B} \leftarrow\$\ \mathbb{Z}_Q^{N\times M}$ or $\boldsymbol{B} = \boldsymbol{A}\cdot\boldsymbol{R} + \boldsymbol{E})$ where $\boldsymbol{R} \leftarrow\$\ (\chi')^{N\times M}, \boldsymbol{E} \leftarrow\$\ (\chi')^{N\times M}$. Denoting $\boldsymbol{b}_i$ as the i-th column of $\boldsymbol{B}$, $\boldsymbol{b}_i$ is either uniform or $\boldsymbol{b}_i = \boldsymbol{A}\cdot\boldsymbol{r}_i + \boldsymbol{e}_i$. Therefore, the distinguisher $\mathcal{B}$ can simulate $\bar{\mathrm{G}}_0$ or G'_0 for an adversary $\mathcal{A}$ by setting $\boldsymbol{A}_{pp} = \boldsymbol{A}$ and running all algorithms as specified in $\bar{\mathrm{G}}_0$ apart from $\mathsf{FHE.KeyGen}^{(pp)}$. To run the i-th instance of $\mathsf{FHE.KeyGen}^{(pp)}$, $\mathcal{B}$ samples $\mathsf{FHE.pk}, \mathsf{FHE.sk} \leftarrow\$\ \mathsf{FHE.KeyGen}$ and then sets $\mathsf{FHE.pk}^{(pp)} := (\boldsymbol{b}_i + \lfloor Q/2 \rceil \mathsf{FHE.sk})$. Clearly, if $\boldsymbol{b}_i$ is not uniform, it perfectly simulates $\bar{\mathrm{G}}_0$ for $\mathcal{A}$. Otherwise, it perfectly simulates G'_0 as shifting a uniform value does not affect uniformity. Therefore, $\mathcal{B}$ would have the same advantage in distinguishing its multi-secret LWE problem as $\mathcal{A}$ has in distinguishing $\bar{\mathrm{G}}_0$ and G'_0. By a standard hybrid, the multi-secret LWE assumption holds (with a polynomial number of secrets) if the plain single secret LWE assumption holds. Therefore, $\bar{\mathrm{G}}_0 \approx_c \mathrm{G}'_0$ and similarly, $\bar{\mathrm{G}}_1 \approx_c \mathrm{G}'_1$.

Now we show that $\mathrm{G}'_0 \approx_c \mathrm{G}'_1$ using the IND-CPA property of FHE. We will use a sequence of hybrids $\mathrm{G}'_{0,i}$ where $i \in [q_R]$ for polynomial q_R that bounds the number of Run queries. In $\mathrm{G}_{0,i}$, all Run queries after the i-th one are answered as in G'_0. All prior Run queries are answered according to the following description:

– Set $req' = (\mathsf{FHE.pk}'^{(pp)} := (\boldsymbol{b}'_{\mathsf{pk}}, \mathsf{FHE.pk}'), \mathsf{ct}', t, \pi')$ where $\mathsf{FHE.pk}'$ is a freshly sampled public key with a uniform $\boldsymbol{b}'_{\mathsf{pk}}$, $\mathsf{ct}' = \mathsf{FHE.Enc}(\mathsf{FHE.pk}', x_1)$ and π' is a simulated zero-knowledge proof.
– Compute $rep' \leftarrow\$\ \mathcal{F}.\mathrm{BlindEval}(\mathsf{sk}, req')$ and set $y_0 = \mathcal{F}.\mathrm{Eval}(\mathsf{sk}, t, x_0)$
– Set $\tau'_0 = (req', rep', y_0)$ and τ_1 as in G'_0.
– Return (τ'_0, τ_1)

We now show that for every i, $\mathrm{G}'_{0,i} \approx_c \mathrm{G}'_{0,i+1}$. In order to show this, we build an IND-CPA adversary $\mathcal{B}$ that distinguishes the IND-CPA game for FHE with the same advantage that $\mathcal{A}$ has in distinguishing $\mathrm{G}'_{0,i}$ from $\mathrm{G}'_{0,i+1}$. $\mathcal{B}$ acts as the POPRIV1 challenger for $\mathcal{A}$, sampling $\mathsf{pp} \leftarrow\$\ \mathcal{F}.\mathsf{Setup}, (\mathsf{pk}, \mathsf{sk}) \leftarrow\$\ \mathcal{F}.\mathsf{KeyGen}$ and initialising the random oracles RO and $\mathsf{RO}_{\mathsf{key}}$ honestly. This allows $\mathcal{B}$ to answer the $(i+1)$-th Run query onwards as in G_0. For the first $(i-1)$ queries, the Run queries are answered by $\mathcal{B}$ as in $\mathrm{G}_{0,i}$ i.e. as described above. However, for the i-th

Run query $\mathcal{A}$ makes, denoted $(t^{(i)}, x_0^{(i)}, x_1^{(i)})$, $\mathcal{B}$ asks its IND-CPA challenger for a public key $\mathsf{FHE.pk}^*$, queries the IND-CPA challenge oracle to encrypt either $x_0^{(i)}$ or $x_1^{(i)}$ receiving ct^* in response. $\mathcal{B}$ then sets $req^* = (\mathsf{FHE.pk}^*, \mathsf{ct}^*, t^{(i)}, \pi^*)$ where π^* is a simulated proof. Finally, $\mathcal{B}$ responds to the i-th Run query by setting $\tau_0' = ((req^*, \mathcal{F}.\mathsf{BlindEval}(sk, req^*)), \mathcal{F}.\text{Eval}(\mathsf{sk}, t, x_0))$ and returning (τ_0', τ_1) to $\mathcal{A}$ where τ_1 is computed in the same way as G_0'. If $\mathcal{B}$ received an encryption of $x_0^{(i)}$, it perfectly simulates the game $\mathrm{G}_{0,i}'$ for $\mathcal{A}$. Otherwise it perfectly simulates $\mathrm{G}_{0,i+1}'$. Therefore, if $\mathcal{A}$ distinguishes $\mathrm{G}_{0,i}'$ from $\mathrm{G}_{0,i+1}'$, then $\mathcal{B}$ also distinguishes the IND-CPA game with the same advantage. This allows us to conclude that $\mathrm{G}_0' \approx_c \mathrm{G}_{0,q}'$. We can run a symmetric argument, changing the way τ_1 is computed in G_1' (i.e. by encrypting x_0 instead of x_1 in the request message) to show $\mathrm{G}_{0,q}' \approx_c \mathrm{G}_1'$ by the IND-CPA property of FHE. Putting everything together and noting that there are a polynomial number of hybrid experiments, we have that $\mathrm{G}_0 \approx_c \mathrm{G}_1$ as required. $\square$

5 Verifiability

In this section we aim to leverage the oblivious nature of the OPRF to extend our POPRF construction to achieve verifiability. We base our technique on the heuristic trick informally discussed in [ADDS21, Sec. 3.2] but with some modifications. In particular, we identify a blind-evaluation attack on this verifiability strategy in our context, the mitigation for which requires enlarging certain parameters. We then use cryptanalysis to study the security of our protocol, i.e. our construction does not reduce to a known (or even new but clean) hard problem. We view our analysis as an exploration into achieving secure protocols from bounded depth circuits which we hope has applications beyond this work.

We now describe our verifiability procedure, based on our OPRF presented in Figure 3 using the cut-and-choose method from [ADDS21, Sec. 3.2]. Intuitively, the client, C, sends the server, S, a set of known answer checkpoints amongst genuine OPRF queries. The client checks if the "check points" match the known answer values, if evaluations on the same points produce the same outputs and if evaluations on different points produce different outputs. C may then assume that S computed the (P)OPRF correctly provided the size of the checkpoint set is large enough. In more detail, let $\gamma = \alpha + \beta$ be the number of points C submits to S.

1. For some fixed $\boldsymbol{t}$, S commits to κ points $\boldsymbol{z}_k^\star \coloneqq F_{\boldsymbol{A}}(\boldsymbol{t}, \boldsymbol{x}_k^\star)$ for $k \in \mathbb{Z}_\kappa$ and publishes them (or sends them to C); S attaches a NIZK proof that these are well-formed, i.e. satisfy the relation

$$\mathfrak{R}_{\boldsymbol{t}} \coloneqq \{\boldsymbol{A}, \boldsymbol{t}, \boldsymbol{z}_k^\star; \boldsymbol{x}_k^\star \text{ for } k \in \mathbb{Z}_\kappa \mid \boldsymbol{z}_k^\star = F_{\boldsymbol{A}}(\boldsymbol{t}, \boldsymbol{x}_k^\star)\}$$

2. C wishes to evaluate α distinct points $\boldsymbol{x}_i^{(\alpha)}$ for $i \in \mathbb{Z}_\alpha$. It samples $\boldsymbol{x}_j^{(\beta)} \leftarrow\!\$\ \{\boldsymbol{x}_k^\star\}_{k \in \mathbb{Z}_\kappa}$ for $j \in \mathbb{Z}_\beta$. C and shuffles the indices and submits these queries.

3. C receives (claimed) $\boldsymbol{z}_i^{(\alpha)} \overset{?}{=} F_{\boldsymbol{A}}(\boldsymbol{t}, \boldsymbol{x}_i^{(\alpha)})$ and $\boldsymbol{z}_j^{(\beta)} \overset{?}{=} F_{\boldsymbol{A}}(\boldsymbol{t}, \boldsymbol{x}_j^{(\beta)})$. C rejects if check points do not match, i.e $\boldsymbol{z}_j^{(\beta)} \neq \boldsymbol{z}_k^{\star}$. Otherwise C accepts.

We formalise the security definition with a game in Figure 4. In the first phase, the adversary provides n check points. Since the tag $\boldsymbol{t}$ remains constant throughout we suppress it here. The experiment samples $m - n$ additional points. It executes the request algorithms and permutes the output according to $\rho \leftarrow\$ S_m$. In the second phase, the adversary receives the m requests and it must also output m responses. The adversary wins if all points have been correctly evaluated and there exists at least one additional point which does not correspond to a point evaluated under $F_{\boldsymbol{A}}$. We say a (P)OPRF is verifiable if the following advantage is negligible in the security parameter λ:

$$\mathsf{Adv}^{\text{verif}}_{\mathcal{F},\mathsf{S},\mathcal{A}}(\lambda) = \Pr\left[\text{VERIF}^{\mathcal{A}}_{\mathcal{F},m,n}(\lambda) = 1\right]$$

```
$\text{VERIF}^{\mathcal{A},m,n}_{\mathcal{F},\mathsf{RO}}$
1 : $\mathsf{pp} \leftarrow \mathcal{F}.\mathsf{Setup}(\lambda)$
2 : $(\{\boldsymbol{x}_i\}_{i=0}^{n-1}, (\mathsf{sk}, \mathsf{pk})) \leftarrow \mathcal{A}; \quad \{\boldsymbol{x}_i\}_{i=n}^{m-1} \leftarrow\$ \mathbb{Z}_p^n$
3 : $req_i \leftarrow \mathcal{F}.\mathsf{Request}(\mathsf{pk}, \boldsymbol{t}, \boldsymbol{x}_i),\ i \in \mathbb{Z}_m$
4 : $\rho \leftarrow\$ S_m$
5 : $\{rep'_i\}_{i=0}^{m-1} \leftarrow \mathcal{A}^{\text{Request},\text{Finalise},\mathsf{RO}}(req_{\rho(0)}, ..., req_{\rho(m-1)})$
6 : $\boldsymbol{z}_i \leftarrow \mathcal{F}.\mathsf{Eval}(\mathsf{sk}, \boldsymbol{t}, \boldsymbol{x}_i),\ i \in \mathbb{Z}_m$
7 : if $\forall i \in [0, n-1]\ \mathcal{F}.\mathsf{Finalise}(rep'_i) = \boldsymbol{z}_{\rho(i)} \wedge\ \exists j \in [n, m-1]$ s.t. $\mathcal{F}.\mathsf{Finalise}(rep'_i) \neq \boldsymbol{z}_{\rho(i)}$
8 :   return 1
9 : else return 0
```

Fig. 4. Verifiability Experiment for (P)OPRF

5.1 Verifiability from Levelled FHE

The heuristic we use to claim security is inspired by [CHLR18] which argues that evaluating a deep circuit in an FHE scheme supporting only shallow circuits is a hard problem. We pursue the same line of reasoning, albeit with significantly tighter security margins. That is, our assumption is significantly stronger than that in [CHLR18].

First, we will assume that the bootstrapping keys for the FHE scheme provided by C to S do not provide fully homomorphic encryption but restrict to a limited number of levels. More precisely, while our main construction Figure 3 and its implementation in Section 6 make use of *fully* homomorphic encryption (by relying on circular security), to target verifiability we pick parameters such that only Lines 4 and 6 of Algorithm 2 cost a bootstrapping operation, i.e. all linear operations are realised without bootstrapping. We note that this significantly increases the parameters required to evaluate our VOPRF candidate.

Specifically, we observe that our POPRF in Algorithm 2 can then be evaluated in depth two and S only receives two bootstrapping keys, presumably preventing it from computing higher depth circuits.

Secondly, we assume that C provides non-unique encodings of its inputs. For example if its input is $\boldsymbol{x} \in \mathbb{Z}_2^n$ it may provide them as a vector in e.g. $\{-1, 0, 1\}^n$ equivalent to $\boldsymbol{x}$ modulo 2. This forces a cheating server to perform normalisation to prevent a cheating POPRF circuit from being computed in depth two.

Third, here we also assume that the native TFHE plaintext modulus P is coprime with $p = 2$. This is because when $p \mid P$, the non-unique encoding approach we utilise here is futile as a unique encoding can be recovered by considering $P/2 \cdot m$ where m is the non-unique encoding of a value $\in \{0, 1\}$.[17]

Parameters. We estimate parameters of our VOPRF in Table 3. While the online cost (134.6kB) is perhaps acceptable in some settings, the public key size (4077.8MB) is clearly beyond the realm of practicality. These parameters are dictated by requiring $P \approx n_p$ so that $\boldsymbol{A} \cdot (\boldsymbol{y}, 1)$ can be computed over the Integers. We thus base our parameters on `PARAM_MESSAGE_8_CARRY_0` in `tfhe-rs` (but replace $P = 2^8$ with a prime close to it).

Table 3. Post-quantum VOPRF candidates in the literature

work	assumption	r	communication cost	flavour	model
[ADDS21]	R(LWE) & SIS	2	> 128GB	verifiable	malicious, QROM
[Bas23]	SIDH	2	8.7MB	verifiable	malicious, *ts*, ROM
Section 5	heuristic	2	0.5MB + 4077.8MB + 57.86KB + 0.8kB + 37.6kB + 96.2kB	verifiable	malicious, ROM
Section 5	heuristic	2	0.5MB + 4077.8MB + 57.86KB + 0.8kB + 3.4kB + 96.2kB	verifiable	malicious, ROM $L = 64$, per query

The column "r" gives the number of rounds. ROM is the random oracle model, QROM the quantum random oracle model and "pp" stands for "preprocessing" and "ts" for "trusted setup". When reporting on our work, the summands are: check point size, pk size, pk proof size, client message size, client message proof size, server message size. Our client message proofs can be amortised to e.g. 217.1kB/64 = 3.4kB per query when amortising over $L = 64$ queries.

5.2 Cryptanalysis

We explore cheating strategies of a malicious server.

Guessing. Assume the adversary guesses the positions of the check points in order to make C accept an incorrect output. Recall that $\gamma = \alpha + \beta$ is the number of points C submits to S, where there are β such check point positions

[17] We would also expect that P should be somewhat far away from a power of two, but we leave the detailed analysis for future work.

of a possible κ choices. Thus, if we assume semantic security of the underlying homomorphic encryption scheme then the probability of an adversarial server guessing correctly is bounded by the probability it selects the positions of a particular check point, for each of the β check points. We obtain a probability $(1/\kappa)^{\beta} \cdot 1/\binom{\gamma}{\beta}$ of guessing correctly.

Interpolation. We consider an adversarial S that uses a circuit F'_{pei}, of depth at most 2, to win the verifiability experiment. At a high level, it solves a quadratic system of equations and then uses the second level to do modular reduction. More precisely, it chooses F'_{pei} such that it agrees on the κ points with of the POPRF circuit $F_{\mathsf{pei}}(\boldsymbol{A}, \boldsymbol{t}, \cdot)$, but that can differ elsewhere. Since the server knows the checkpoints, this can be trivially done by the server. To prevent such an attack, one would need to publish $\kappa = n + \frac{n}{2}(n+1) + \mathcal{O}(1)$ check points where n is the input/output dimension. This, implies no quadratic polynomial interpolation exists. If we let $\kappa = 128^3$, then the communication cost of checkpoints is approximately an additional 64MB, which can be reduced to 32MB by generating the input values from a seed. If the assumption is weakened such that we only require $\kappa = 128^2$ then this becomes only 0.5MB.

Check and cheat. Finally, we consider that the adversary is able to construct a shallow cheating circuit, which is described in Algorithm 3, with $[\cdot]$ denoting a homomorphic encryption of a value. Intuitively, it homomorphically checks the clients inputs against known answers and then homomorphically selects which output to return. This circuit has depth three. Here we use the non-unique encoding of the input to force a malicious server to perform the initial mod p operation, costing a level.

Algorithm 3 Cheating Circuit

```
Input: non-unique encoding $[\boldsymbol{y}] \in \mathbb{Z}_p^{n_p}$; checkpoints $\{\boldsymbol{y}_j^\star\}$; $H : \mathbb{Z}_p^{n_p} \to \mathbb{Z}_q^m$ any function
  $[\boldsymbol{r}], [\mathsf{found}] \leftarrow 0, 0$
  for all $\boldsymbol{y}_j^\star$ do
      $[\boldsymbol{d}] \leftarrow [\boldsymbol{y}] - \boldsymbol{y}_j^\star$
      $[d_i^{(0)}] \leftarrow \mathsf{LUT}_{x \bmod p}(d_i)\ \forall i \in \mathbb{Z}_{n_p}$        ▷ mod $p$, depth 1
      $[h] \leftarrow \sum_{i \in \mathbb{Z}_{n_p}} d_i^{(0)}$
      if $[h] = 0$ then                                                        ▷ CMUX, depth 1
          $[\boldsymbol{r}] \leftarrow [\boldsymbol{r}] + \boldsymbol{y}_j^*$;   $[\mathsf{found}] \leftarrow [\mathsf{found}] + 1$
      else
          $[\boldsymbol{r}] \leftarrow [\boldsymbol{r}] + \boldsymbol{0}$;   $[\mathsf{found}] \leftarrow [\mathsf{found}] + 0$
      end if
  end for
  return $\mathsf{CMUX}_{[\mathsf{found}]}([\boldsymbol{r}], H([\boldsymbol{y}]))$                  ▷ depth 1
```

5.3 Compression Potential

The central reason for the large parameters reported above is that we must force a cheating server to compute a circuit in depth three whereas a honest server only requires depth two. We achieve this by making the client submit non-unique encodings and thus need to pick parameters that $\boldsymbol{A} \cdot (\boldsymbol{y}, 1)$ can be computed over $\mathbb{Z}$, i.e. a plaintext space $\approx \mathbb{Z}_{n_p}$. A first approach to circumvent this issue is by making an even stronger assumption: our cheating circuits all need to homomorphically sum over κ values related to our checkpoints. In our analysis, we assume additions are "free", i.e. hardly increase the noise, but as we increase the number of check points this assumption breaks down. Indeed, $\kappa \approx n^2$ already suggests a noise growth of n, plausibly too big for our OPRF parameters. This, of course, assumes this noise growth cannot be managed more efficiently by a cheating server and is thus a stronger assumption, as mentioned.

A second approach is to reduce the bootstrapping depth of an honest server. Recall, that we rely on our final bootstrapping to argue **Ax**-*hiding* of the evaluation. If, instead, we could rely solely on the circuit privacy provided by bootstrapping after computing $\boldsymbol{A} \cdot (\boldsymbol{y}, 1)$, we would obtain a depth one circuit for the honest server. Thus, a cheating server in depth two would not break security and we could drop the non-unique encoding and evaluation over $\mathbb{Z}$ blowing up our parameters. We leave exploring these approaches for future work.

6 Proof-of-Concept Implementation

We demonstrate that we can build a simple and efficient implementation of the basic oblivious pseudorandom function given in Figure 3. Our implementation is written in Rust and provided open-source.[18] It makes use of Zama's `tfhe-rs` FHE library [CGGI20] for writing FHE functionalities using TFHE. The implementation itself requires less than 700 lines of code for instantiating the core functionality (not including tests and examples).[19]

Implementation choices. The matrix $\boldsymbol{G}_{\mathsf{out}} \in \mathbb{Z}_q^{m \times m_p}$ is constructed (as in Section 2.6) by sampling each entry uniformly in $\mathbb{Z}_3$. The matrix $\boldsymbol{G}_{\mathsf{inp}} \in \mathbb{Z}_q^{n_q \times n}$ is sampled with an identity matrix in the top $n \times n$ component, and the remaining $(n_q - n)$ rows as random vectors in $\mathbb{Z}_3$ (i.e. in systematic form, see Section 2.6). The matrix $\boldsymbol{A}$ is sampled randomly in $\mathbb{Z}_2^{m_p \times n_p}$. The dimensions of these matrices are set using the values given in Table 2. The matrix $\boldsymbol{A}_t$ is constructed as the output of a random oracle $\mathsf{RO}_{\mathsf{key}}$, where for each column $\boldsymbol{A}[i]$, the random oracle outputs column $\boldsymbol{A}_t[i] \in \mathbb{Z}_2^{m_p}$ by evaluating $\mathsf{RO}_{\mathsf{key}}(t, \boldsymbol{A}[i],$ `"voprf_fhe_rom_key"`$)$. The random oracle $\mathsf{RO}_{\mathsf{key}}$ is implemented using SHA-384, while the random oracle $\mathsf{RO}_{\mathsf{fin}}$, which is used for finalising the client input (as in [TCR+22]), is implemented using SHA-256. Note that in our implementation, all matrices are transposed and therefore all multiplication operations are therefore reversed.

[18] `https://anonymous.4open.science/r/oprf-fhe-C726`

[19] We do not implement the zero-knowledge arguments that are used as part of the main protocol construction, but provide bandwidth estimates for these in Appendix B.

Table 4. Computational runtimes for running OPRF functionality.

(P)OPRF function	Time
$\mathsf{C} : F_{\mathsf{poprf}}.\mathsf{KeyGen}$	220ms
$\mathsf{C} : F_{\mathsf{poprf}}.\mathsf{Request}$	36ms
$\mathsf{S} : F_{\mathsf{poprf}}.\mathsf{BlindEval}$ ("slow")	14.4s
$\mathsf{S} : F_{\mathsf{poprf}}.\mathsf{BlindEval}$ ("full")	123ms
$\mathsf{C} : F_{\mathsf{poprf}}.\mathsf{Finalise}$	0.1ms

Table 5. Sizes of cryptographic material, RAM usage, and bootstrapping depth per ciphertext.

Data point	Size
C: Secret key	0.02MB
C: Public key	66.8MB
C: Request	2.02MB
S: Response	1.01MB

The client public key includes the size of both the bootstrapping and key switching key used in the implementation of Concrete.

`tfhe-rs` implementation and parameters. The Rust implementation of Concrete that we use corresponds to the `tfhe-rs` library (version "0.2.4").[20] with parameter set `PARAM_MESSAGE_2_CARRY_0`. Ideally, we would implement our PRF candidate by first computing with a native TFHE plaintext modulus 2, followed by a bootstrapped key switch to native TFHE plaintext modulus 3. However, as it stands, the `tfhe-rs` API does not support this. We, thus, implement a "slow" variant, where we perform all computations modulo 4 (implied by the parameter set `PARAM_MESSAGE_2_CARRY_0`) where we carefully bootstrap and reduce modulo 3 to avoid overflows. That is, $\mathbb{Z}_3$ is implemented using the set $\{0, 1, 2\} \in \mathbb{Z}_4$.[21] We then also give estimates (rather than benchmarks) for a "full" variant, where we avoid these additional bootstrapping operations to work around the API limitation.

Benchmarks. The results that we discuss below were acquired by using a server with 96 Intel Xeon Gold 6252 CPU @ 2.10GHz cores and 768 GB of RAM. Server evaluation is run with parallelisation enabled, meaning that each multiplication of the client input encrypted vector with a matrix column is run in its own separate thread, but we only use 64 threads/cores. Client evaluations use only a single core. Each of the benchmarks was established after running it ten times and taking the average runtime.

Runtimes. As shown in Table 4, key generation (KeyGen) is the heaviest client-side: only taking 220ms. Even so, this operation can be regarded as a one-time cost in many applications. For example, considering OPAQUE [JKX18], clients and servers already register persistent identifiers for each other (such as the client-specific OPRF key). Therefore, the client keypair can be registered as part of this process. Similarly for Privacy Pass [DGS+18], the issuance phase of the protocol does not discount clients from registering persistent information that they use whenever they make VOPRF evaluations (which could include this key information). As a result, in many applications, clients will generate a single FHE keypair and use that over multiple interactions with the server.

[20] https://docs.rs/tfhe/0.2.4/tfhe/index.html

[21] We note that this implementation is incompatible with our VOPRF candidate.

For the online client-side algorithms (Request and Finalise) that must be run on every invocation of the OPRF, it is clear that the overheads are minimal, and are suitable even for relatively constrained devices. For the online server-side algorithm (BlindEval), the runtime is *estimated* as just 123ms, which may be quick enough for certain applications that have a hard requirement to ensure post-quantum security. Previous classical constructions of (P)OPRFs, such as [TCR+22] take only a few milliseconds to run the server evaluation algorithm, and so the efficiency gap between our "full" FHE-based approach and previous work is evident, but fairly minimal.

Bandwidth and storage costs. As shown in Table 5, the client public key dominates the bandwidth and storage costs for any given run-through of the protocol, at around 70MB. However, since the client key pair can be used across multiple invocations of the protocol, the bandwidth cost does not occur during subsequent evaluations (the "online phase"). The online phase of the protocol incurs a total communication cost of approximately 3MB, since ciphertexts are stored in M-LWE format in the `tfhe-rs` library. However, as remarked in Section 3.2, much smaller communication costs are possible by carefully selecting the format (LWE instead of M-LWE), compressing and amortising.

Acknowledgements

We thank Christian Weinert for discussing MPC approaches with us; we thank Ilaria Chillotti, Ben Curtis and Jean-Baptiste Orfila for answering questions about TFHE and Concrete; we thank Nicolas Gama for answering questions about TFHE. We thank Ward Beullens and Gregor Seiler for answering questions about LaBRADOR and sharing their size estimation Pari/GP script with us.

The research of Martin Albrecht was supported by EPSRC grants EP/S020330/1, EP/S02087X/1 and by the European Union Horizon 2020 Research and Innovation Program Grant 780701. Part of this work was done while Martin Albrecht was at Royal Holloway, University of London, and while Alex Davidson was at Brave Software. The majority of this research was done while Amit Deo was at Crypto Quantique.

References

ADDS21. Martin R. Albrecht, Alex Davidson, Amit Deo, and Nigel P. Smart. Round-optimal verifiable oblivious pseudorandom functions from ideal lattices. In Garay [Gar21], pages 261–289. Full version available at https://eprint.iacr.org/2019/1271. 1.1, 1.2, 10, 1, 5, 3

AGPS20. Martin R. Albrecht, Vlad Gheorghiu, Eamonn W. Postlethwaite, and John M. Schanck. Estimating quantum speedups for lattice sieves. In Moriai and Wang [MW20], pages 583–613. 11, 2.6

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. 2.4

ARS$^+$15. Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In Oswald and Fischlin [OF15], pages 430–454. 1

Bas23. Andrea Basso. A post-quantum round-optimal oblivious PRF from isogenies. Cryptology ePrint Archive, Report 2023/225, 2023. https://eprint.iacr.org/2023/225. 1.2, 1, 3

BCG$^+$22. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Nicolas Resch, and Peter Scholl. Correlated pseudorandomness from expand-accumulate codes. In Dodis and Shrimpton [DS22], pages 603–633. 1.2

BdMW16. Florian Bourse, Rafaël del Pino, Michele Minelli, and Hoeteck Wee. FHE circuit privacy almost for free. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part II*, volume 9815 of *LNCS*, pages 62–89. Springer, Heidelberg, August 2016. 2.5

BGV11. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. Cryptology ePrint Archive, Report 2011/277, 2011. https://eprint.iacr.org/2011/277. 2.5

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, Heidelberg, November 2018. Full version available at https://eprint.iacr.org/2018/1218. 1, 1.1, 5, 1.2, 1.3, 1, 2.6, 2.6, 2.6

BKM$^+$21. Andrea Basso, Péter Kutas, Simon-Philipp Merz, Christophe Petit, and Antonio Sanso. Cryptanalysis of an oblivious PRF from supersingular isogenies. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part I*, volume 13090 of *LNCS*, pages 160–184. Springer, Heidelberg, December 2021. 1.2

BKW20. Dan Boneh, Dmitry Kogan, and Katharine Woo. Oblivious pseudorandom functions from isogenies. In Moriai and Wang [MW20], pages 520–550. 1.2, 1

BS22. Ward Beullens and Gregor Seiler. LaBRADOR: Compact proofs for R1CS from module-SIS. Cryptology ePrint Archive, Report 2022/1341, 2022. https://eprint.iacr.org/2022/1341. 1.1, B, B.1, B.1

CCKK21. Jung Hee Cheon, Wonhee Cho, Jeong Han Kim, and Jiseung Kim. Adventures in crypto dark matter: Attacks and fixes for weak pseudorandom functions. In Garay [Gar21], pages 739–760. 2.6

CDKS21. Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient homomorphic conversion between (ring) LWE ciphertexts. In Kazue Sako and Nils Ole Tippenhauer, editors, *ACNS 21, Part I*, volume 12726 of *LNCS*, pages 460–479. Springer, Heidelberg, June 2021. 3.2

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. 1.1, 2.5, 2.5, 3.2, 6, B.1

CHL22. Sílvia Casacuberta, Julia Hesse, and Anja Lehmann. Sok: Oblivious pseudorandom functions. In *7th IEEE European Symposium on Security and Privacy, EuroS&P 2022*, pages 625–646. IEEE, 2022. 1.2, 3.1

CHLR18. Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from fully homomorphic encryption with malicious security. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1223–1237. ACM Press, October 2018. 1, 1.1, 1.3, 5.1

CKKS17. Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 409–437. Springer, Heidelberg, December 2017. 2.5

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, Heidelberg. 1, 1.2, 9, 1, 2.6, 2.6

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, 6

DM15. Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In Oswald and Fischlin [OF15], pages 617–640. 2.5

DS16. Léo Ducas and Damien Stehlé. Sanitization of FHE ciphertexts. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 294–310. Springer, Heidelberg, May 2016. 2.5

DS22. Yevgeniy Dodis and Thomas Shrimpton, editors. *CRYPTO 2022, Part II*, volume 13508 of *LNCS*. Springer, Heidelberg, August 2022. 6

DvW21. Léo Ducas and Wessel P. J. van Woerden. NTRU fatigue: How stretched is overstretched? In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part IV*, volume 13093 of *LNCS*, pages 3–32. Springer, Heidelberg, December 2021. 2.4

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. 1

FV12. Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. `https://eprint.iacr.org/2012/144`. 2.5

Gar21. Juan Garay, editor. *PKC 2021, Part II*, volume 12711 of *LNCS*. Springer, Heidelberg, May 2021. 6

Gen09. Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `crypto.stanford.edu/craig`. 2.5, 2.5

Gol04. Oded Goldreich. *Foundations of Cryptography*, volume 2. Cambridge University Press, 2004. 2

GSW13. Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 75–92. Springer, Heidelberg, August 2013. 2.5, B.1

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`. 1.2, 1

HPS96. Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A new high speed public key cryptosystem, 1996. Draft Distributed at Crypto'96, available at `http://web.securityinnovation.com/hubfs/files/ntru-orig.pdf`. 2.4

JKR18. Stanislaw Jarecki, Hugo Krawczyk, and Jason Resch. Threshold partially-oblivious PRFs with applications to key management. Cryptology ePrint Archive, Report 2018/733, 2018. https://eprint.iacr.org/2018/733. 3.1

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, Heidelberg, April / May 2018. 1, 6

Joy21. Marc Joye. Guide to fully homomorphic encryption over the [discretized] torus. Cryptology ePrint Archive, Report 2021/1402, 2021. https://eprint.iacr.org/2021/1402. 1.1, 2.5, 2.5, 13

Klu22. Kamil Kluczniak. Circuit privacy for fhew/tfhe-style fully homomorphic encryption in practice. Cryptology ePrint Archive, Paper 2022/1459, 2022. https://eprint.iacr.org/2022/1459. 2.5, B.1

LDK$^+$22. Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. 3.2, B.1, B.1, B.2

LNP22. Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Maxime Plançon. Lattice-based zero-knowledge proofs and applications: Shorter, simpler, and more general. In Dodis and Shrimpton [DS22], pages 71–101. 1.1, B.1, B.1, B.2, B.2, B.3

LS15. Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, June 2015. 5, 6

MAT22. MATZOV. Report on the Security of LWE: Improved Dual Lattice Attack. Available at https://doi.org/10.5281/zenodo.6412487, April 2022. 2.4

MP20. Daniele Micciancio and Yuriy Polyakov. Bootstrapping in FHEW-like cryptosystems. Cryptology ePrint Archive, Report 2020/086, 2020. https://eprint.iacr.org/2020/086. 1.1

MW20. Shiho Moriai and Huaxiong Wang, editors. *ASIACRYPT 2020, Part II*, volume 12492 of *LNCS*. Springer, Heidelberg, December 2020. 6

OF15. Elisabeth Oswald and Marc Fischlin, editors. *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*. Springer, Heidelberg, April 2015. 6

OPP14. Rafail Ostrovsky, Anat Paskin-Cherniavsky, and Beni Paskin-Cherniavsky. Maliciously circuit-private FHE. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 536–553. Springer, Heidelberg, August 2014. 2.5

SG98. Victor Shoup and Rosario Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. In Kaisa Nyberg, editor, *EUROCRYPT'98*, volume 1403 of *LNCS*, pages 1–16. Springer, Heidelberg, May / June 1998. 1.1

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. https://eprint.iacr.org/2021/182. 1.2, 1

TCR+22. Nirvan Tyagi, Sofía Celi, Thomas Ristenpart, Nick Sullivan, Stefano Tessaro, and Christopher A. Wood. A fast and simple partially oblivious PRF, with applications. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 674–705. Springer, Heidelberg, May / June 2022. 1, 1.1, 2.1, 2.2, 1, 2.2, 2, 2.2, 3, 3, 4, 6, 6

Unr12. Dominique Unruh. Quantum proofs of knowledge. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 135–152. Springer, Heidelberg, April 2012. 1.1

A Alternative heuristic approach for Ax-hiding

We conjecture that the following approach, relying on the form of programmable bootstrapping when implementing an LUT, heuristically achieves **Ax**-hiding. Programmable bootstrapping of an LWE ciphertext with look-up table LUT_f consists of three steps: blind rotation, sample extraction and key-switching. In particular, blind rotation begins with an LWE ciphertext $\boldsymbol{c} = (\boldsymbol{a}, b = \boldsymbol{a} \cdot \boldsymbol{s} + e + \mu \cdot (Q/P))$ encrypting $\mu \in \mathbb{Z}_P$ and a programmed evaluation polynomial $v(X)$ for function f. Note that we do not wish to hide f and thus $v(X)$. Furthermore, let $(\boldsymbol{a}_z, b_z)$ be some encryption of zero under the same LWE secret key.

We first compute $(\boldsymbol{a}, b) \leftarrow (\boldsymbol{a}, b) + (v \cdot \boldsymbol{a}_z + \boldsymbol{e}', v \cdot b_z + e'')$ for some short $v, \boldsymbol{e}', e''$ to rerandomise $(\boldsymbol{a}, b)$, especially the $\boldsymbol{a}$ part (using the LWE assumption). Then using the bootstrapping key, blind rotation (homomorphically) multiplies $v(X)$ by X^{k_i} where k_i is either 0 or some value depending on $\boldsymbol{a}$ or b for $i = 0, \ldots, n-1$. Whether k_i is 0 or not depends on the bootstrapping key. In more detail, these blind rotations are realised using CMux gates acting on some accumulator α which is update to '$\alpha \leftarrow G^{-1}\left((X^{a_i} - 1) \cdot \alpha\right) \cdot (\boldsymbol{E} + \mu \cdot \boldsymbol{G}^T) + \alpha$ where $\boldsymbol{G}$ is a known gadget matrix, $G^{-1}(\cdot)$ is a matching (possibly randomised) decomposition function and $\boldsymbol{E}$ is some small error. At the end of the blind rotation phase, we have an encryption of $X^{\lfloor (b - \boldsymbol{a} \cdot \boldsymbol{s}) \cdot 2n/Q \rceil} \cdot v(X) = X^{\lfloor (e + \mu \cdot (Q/P)) \cdot 2n/Q \rceil} \cdot v(X)$. Here $\lfloor (b - \boldsymbol{a} \cdot \boldsymbol{s}) \cdot 2n/Q \rceil$ indicates that $(b - \boldsymbol{a} \cdot \boldsymbol{s})$ is mapped from values in $\{0, \ldots, Q-1\}$ to values in $\{0, \ldots, 2n-1\}$ via rounding. We then perform sample extraction to get an LWE encryption of $f(\lfloor (P/Q) \cdot (b - \boldsymbol{a} \cdot \boldsymbol{s}) \rceil)$.

Now, in our variant, we initialise the accumulator to $v(x)$ (and not $X^b \cdot v(x)$). We then run the CMUXes realising the blind rotation in some random order. Finally, we rotate the result in the accumulator α by X^b. Here, the dependency on $(\boldsymbol{a}, b)$ is almost removed, using the LWE assumption on $v \cdot \boldsymbol{a}_z + \boldsymbol{e}'$. In the final rotation step we do get a dependency based on the original computation, but a key difference is that we do not need a CMUX to perform this rotation. This is then followed by a sample extraction that essentially picks out a single coefficient of the noisy RLWE ciphertext, partially obscuring the rotation. We stress, however, that this argument is heuristic, as b is not independent of $\boldsymbol{a}$ if the secret key is known.

B Client Non-Interactive Zero Knowledge Proofs

A key part of security against malicious clients is the non-interactive zero knowledge proof that a POPRF request is well-formed. This requires a proof system that can (a) show that the public key material of the FHE scheme is well-formed and (b) show that the accompanying ciphertext is an encryption of a valid input. To make things modular, we discuss how to prove (a) and (b) separately. The overall system can be trivially obtained by combining the two proofs in a straight-forward manner.

For the VOPRF construction, there are two main complications as far as the client-side zero-knowledge proofs are concerned. The first is that the public key material sent by the client must only permit two bootstrapping operations. The second is that PRF inputs must have non-unique encodings. To avoid repetition, we describe the client proofs for (a) in the more complicated context of VOPRFs and derive cost estimates for the OPRF construction as a simpler case.
We provide our scripts computing our size estimates in Appendices E and F. The former is based off a script provided by the authors of [BS22].

B.1 Public Keys

There are two components to the public key material sent by the client: a normal TLWE public key/commitment and a TFHE bootstrapping key. A TFHE bootstrapping key consists of three components: A standard GSW key [GSW13], a standard bootstrapping key and a key-switching key. The latter two are used in TFHE's (programmable) bootstrapping procedure whereas the first is used to create the bootstrapping key. One could remove the GSW key, but then the zero-knowledge proofs become far more expensive because there are large elements that form part of the witness (namely the uniform elements used in the creation of a private GSW encryption). Note that the bootstrapping key takes the form of GSW ciphertexts over *rings*, i.e. over $\mathcal{R}_Q = \mathbb{Z}_Q[X]/(X^d+1)$ whereas the key-switching key is a collection of *plain* LWE ciphertexts, i.e. over $\mathbb{Z}_Q^e$.

To perform normal homomorphic operations, all that is required is the ciphertexts. In particular, we do not really need to use a (T)LWE public key (denoted $\boldsymbol{c}_{\mathsf{pk}}$ below) for this. However, the circuit privacy techniques of Kluczniak [Klu22] does use the public key to re-randomise ciphertexts. Therefore, it is important that we include $\boldsymbol{c}_{\mathsf{pk}}$ in the proofs.

In the VOPRF construction, we will only be providing TFHE allowing two bootstrapping operations. After each bootstrapping, the TFHE encryption key changes. Let $\bar{s}^{(\tau)} = (\bar{s}_0^{(\tau)}, \ldots, \bar{s}_{e-1}^{(\tau)}) \hookleftarrow\$ \mathbb{Z}_2^e$ denote the TFHE key after the τ-th bootstrapping for $\tau = 0, 1, 2$. Also, take the gadget matrix permitting *approximate* decompositions to be

$$\boldsymbol{G} = (\lfloor Q/B \rceil, \ldots, \lfloor Q/B^\ell \rceil) \otimes \boldsymbol{I}_2$$

for decomposition parameters B and ℓ. The encryption key used to produce the τ-th bootstrapping key for $\tau \in \{1, 2\}$ will be $\tilde{s}^{(\tau)} = \sum_{i=0}^{d-1} \tilde{s}_i^{(\tau)} \cdot X^i \hookleftarrow\$ \mathcal{R}_2$.

We will denote error distributions for LWE/MLWE assumptions by χ and $\bar{\chi}$ respectively.

Key Material. Overall, the key material sent from the client to the server is:

– **Root TLWE Public Key/Root Secret Key Commitment:**

$$\boldsymbol{b}_{\mathsf{pk}} = \boldsymbol{A}_{pp} \cdot \boldsymbol{r}_{\mathsf{com}} + \boldsymbol{e}_{\mathsf{com}} + \lfloor Q/2 \rceil \cdot \begin{bmatrix} \bar{s}^{(0)} \\ \mathbf{0}^{e_{\mathsf{com}}-e} \end{bmatrix} \in \mathbb{Z}_Q^{e_{\mathsf{com}}}$$

$$\boldsymbol{c}_{\mathsf{pk}} = \boldsymbol{A}_{\mathsf{pk}} \cdot \bar{s}_0 + \boldsymbol{e}_{\mathsf{pk}} \in \mathbb{Z}_Q^{e \log Q}$$

where $\boldsymbol{A}_{pp} \in \mathbb{Z}_Q^{e_{\mathsf{com}} \times e_{\mathsf{com}}}$ is from the public parameters, $\boldsymbol{A}_{\mathsf{pk}} \leftarrow\!\!\$\ \mathbb{Z}_Q^{e \log Q \times e}$, $\boldsymbol{e}_{\mathsf{com}}, \boldsymbol{r}_{\mathsf{com}} \leftarrow\!\!\$\ \chi_{\mathsf{com}}^{e_{\mathsf{com}}}$ and $\boldsymbol{e}_{\mathsf{pk}} \leftarrow\!\!\$\ \chi^{e \log Q}$.

– **GSW Public Keys for bootstrapping:** For $\tau \in \{1, 2\}$

$$\boldsymbol{A}_{GSW}^{(\tau)} := \begin{bmatrix} \boldsymbol{a}^{(\tau)} \\ \tilde{s}^{(\tau)} \cdot \boldsymbol{a}^{(\tau)} + \boldsymbol{e}^{(\tau)} \end{bmatrix} \in \mathcal{R}_Q^{2 \times 2}$$

where $(\boldsymbol{a}^{\tau})^{\top} \leftarrow\!\!\$\ \mathcal{R}_Q^2, \boldsymbol{e}^{(\tau)} \leftarrow\!\!\$\ \bar{\chi}^2$.

– **Bootstrapping Keys:** For $i = 0, \ldots, e-1$ and $\tau \in \{1, 2\}$:

$$\mathsf{bsk}[\tau, i] := \boldsymbol{A}_{GSW}^{(\tau)} \cdot \boldsymbol{R}_i^{(\tau)} + \boldsymbol{E}_i^{(\tau)} + \bar{s}_i^{(\tau-1)} \cdot \boldsymbol{G}$$

where $\boldsymbol{R}_i^{(\tau)} \leftarrow\!\!\$\ \mathcal{R}_2^{2 \times 2\ell}, \boldsymbol{E}_i^{(\tau)} \leftarrow\!\!\$\ \bar{\chi}^{2 \times 2\ell}$.

– **Key-Switching Keys:** For $i = 0, \ldots, d-1, j = 0, \ldots, \tilde{\ell} - 1, \tau \in \{1, 2\}$,

$$\mathsf{ksk}[\tau, i, j] := \left(\boldsymbol{a}_{i,j}^{(\tau)}, \sum_{k=0}^{e-1} (\boldsymbol{a}_{i,j}^{(\tau)})_k \cdot \bar{s}_k^{(\tau)} + e_{i,j}^{(\tau)} + \tilde{s}_i^{(\tau)} \cdot \lfloor Q/\tilde{B}^j \rceil \right),$$

where $\boldsymbol{a}_{i,j}^{(\tau)} \leftarrow\!\!\$\ \mathbb{Z}_Q^e, e_{i,j}^{(\tau)} \leftarrow\!\!\$\ \chi$ and $\tilde{B}, \tilde{\ell}$ are decomposition parameters.

In order to set some example parameters, we rely on the parameters from our implementation and only consider the simpler non-verifiable (P)OPRF case. In particular these parameters are (`PARAM_MESSAGE_2_CARRY_0`) defined in the `tfhe-rs` library [CGGI20] with $e_{\mathsf{com}} = 2^{11}$ and χ_{com} a discrete Gaussian with standard deviation $\sigma_{\mathsf{com}} = 2^2$ (see Section 2.4 for setting the latter). The TFHE scheme uses discretisation over a torus where the resolution of the discretisation allows us to pick modulus $Q \approx 2^{32}$ and the other parameters are $e = 656, d = 1,024, \ell = 2, \tilde{\ell} = 4$. Note that our description of the bootstrapping key is written in terms of R-LWE for simplicity. However, our implementation parameters use a ring dimension of 512 with rank 2, explaining why $d = 1,024$ in our analysis.

If a random seed is expanded to generate all of the random elements and considering the OPRF case, only the root public key, $\boldsymbol{A}_{GSW}^1$, $\mathsf{bsk}[1, i]$ and $\mathsf{ksk}[1, i, j]$ need to be sent (when calculating $\mathsf{ksk}[1, i, j]$, we use $\bar{s}^{(1)} = \bar{s}^{(0)}$). Therefore, the size of the key material is

$$(e_{\mathsf{com}} + e \log Q) \cdot \log Q + 2\, d \log Q + (2 \cdot 2\, \ell) \cdot e \cdot d \log Q + d\, \tilde{\ell} \cdot \log Q.$$

which is dominated by $2\,e \cdot 2\ell \cdot d \log Q$.

Finally, to pack multiple LWE response ciphertexts into a *single* RLWE ciphertext, we need to add some key-switching material to the public keys. These key-switching keys are associated to a total of $\log(d)$ automorphisms and therefore consist in a total of $\log(d) \cdot d \cdot \tilde{\ell}$ noisy LWE products. Using the parameters above, we end up adding 72kB to the public key material. Overall, we get around 16.8MB of public key material sent by the client for the OPRF, which we obtain by also dropping some lower-order bits [LDK+22]. The VOPRF parameters `PARAM_MESSAGE_8_CARRY_0` use larger parameters $Q \approx 2^{64}, d = 32768, \ell = 2, \tilde{\ell} = 5$ to ensure that two bootstrappings on the server side suffices for PRF evaluation. Unfortunately, this results in around 4091MB of public key material.

Proofs. Essentially, the well-formedness of the public key material is a large "noisy" linear system where some of the equations are over $\mathcal{R}_Q$ and others are over $\mathbb{Z}_Q$. Additionally, some of the solution is binary i.e. $\bar{s}^{(\tau)}, \tilde{s}^{(\tau)}$ and $\boldsymbol{R}_i^{\tau}$, while the rest of the solution i.e. $\boldsymbol{r}_{\mathsf{com}}, \boldsymbol{e}_{\mathsf{com}}, \boldsymbol{e}_{\mathsf{pk}}, \boldsymbol{E}_i^{(\tau)}$ and $e_{i,j}^{(\tau)}$ is small.

We give proof sizes both in the zero-knowledge proof system from [LNP22] and for a zero-knowledge variant of [BS22]. We rely on the former for proving ciphertexts, so we chose to also give the costs under this proof system for the FHE public keys.

[LNP22]. In order to estimate the cost of the zero-knowledge proof, we rewrite the statement being proved in a form compatible with [LNP22]. We will have the unknowns $\bar{s}^{(0)}, \bar{s}^{(1)}, \bar{s}^{(2)} \in \mathbb{Z}_2^e, \|\boldsymbol{r}_{\mathsf{com}}\| \leq \beta_{\mathsf{com}}, \tilde{s}^{(0)}, \tilde{s}^{(1)} \in \mathcal{R}_2$,

$$\{\boldsymbol{r}_{i,j}^{(\tau)} \in \mathcal{R}_2^2 : i \in \{0, \dots, e-1\}, j \in \{0, \dots, 2\ell - 1\}, \tau \in \{1, 2\}\}$$

where $\boldsymbol{r}_{i,j}^{(\tau)}$ is the j-th column of $\boldsymbol{R}_i^{(\tau)}$ and also

$$\{e_{i,j}^{(\tau)} : i \in \{0, \dots, d-1\}, j \in \{0, \dots, \tilde{\ell} - 1\}, \tau \in \{1, 2\}\}.$$

We rewrite the statement as follows:

– (Binary $\bar{s}^{(0)}, \bar{s}^{(1)}, \bar{s}^{(2)}, \tilde{s}^{(1)}, \tilde{s}^{(2)}, \{\boldsymbol{r}_{i,j}^{(\tau)}\}$) Take $\boldsymbol{x}$ to be the concatenation of coefficients of $\bar{s}^{(0)}, \bar{s}^{(1)}, \bar{s}^{(2)}, \tilde{s}^{(1)}, \tilde{s}^{(2)}$ and all of the $\boldsymbol{r}_{i,j}$. Then $\langle \boldsymbol{x}, \boldsymbol{x} - \boldsymbol{1} \rangle \equiv 0 \bmod Q$ and $\|\boldsymbol{x}\|$ short enough so that the former holds over $\mathbb{Z}$ too. This implies that these components are binary.

– (**Root public key/Root Secret Key Commitment**)

$$\|\boldsymbol{r}_{\mathsf{com}}\| \leq \beta_{\mathsf{com}},$$
$$\left\| \boldsymbol{b}_{\mathsf{pk}} - \boldsymbol{A}_{pp} \cdot \boldsymbol{r}_{\mathsf{com}} - \lfloor Q/2 \rceil \bar{s}^{(0)} \right\| \leq \beta_{\mathsf{com}},$$
$$\left\| \boldsymbol{c}_{\mathsf{pk}} - \boldsymbol{A}_{\mathsf{pk}} \cdot \bar{s}^{(0)} \right\| \leq \beta_{\mathsf{pk}}$$

where β_{com} and β_{pk} are length-bounds on $\boldsymbol{e}_{\mathsf{com}} \in \mathbb{Z}_Q^{e_{\mathsf{com}}}$ and $\boldsymbol{e}_{\mathsf{pk}} \in \mathbb{Z}_Q^e$ respectively.

– (**GSW Public Keys**) For $\tau \in \{1, 2\}$

$$\left\| \left(\boldsymbol{A}_{GSW}^{(\tau)}\right)^{\top} \cdot \begin{bmatrix} \tilde{s}^{(\tau)} \\ -1 \end{bmatrix} \right\| \leq \beta$$

where β is a length bound on error vector $\boldsymbol{e}^{(\tau)} \in \mathcal{R}_Q^2$.

– (**Bootstrapping**) For $\tau \in \{1, 2\}, i = 0, \ldots, e-1, j = 0, \ldots, 2\ell - 1$,

$$\|\boldsymbol{A}_{GSW}^{(\tau)} \cdot \boldsymbol{r}_{i,j}^{(\tau)} + \bar{s}_i^{(\tau-1)} \cdot \boldsymbol{G}_j - \mathsf{bsk}[\tau, i]_j\| \leq \beta'$$

where the index j in $\boldsymbol{G}_j, \mathsf{bsk}[\tau, i]_j$ denotes the j-th column and β' is a bound on the columns of $\boldsymbol{E}'_i$.

– (**Key-Switching**) For $\tau \in \{1, 2\}, j = 0, \ldots, \tilde{\ell} - 1$,

$$\|(\boldsymbol{A}_j^{(\tau)}) \cdot \bar{s}^{(\tau)} + \tilde{s}^{(\tau)} \cdot \lfloor Q/\tilde{B}^j \rceil - \mathsf{ksk}[\tau, j]\| \leq \beta''$$

where $\boldsymbol{A}_j^{(\tau)}$ is the matrix whose i-th row is $\boldsymbol{a}_{i,j}^{(\tau)}$, $\mathsf{ksk}[\tau, j] \in \mathbb{Z}_Q^d$ has i-th entry $\mathsf{ksk}[\tau, i, j]$ and β'' is a bound on the vector whose i-th component is $e_{i,j}^{(\tau)}$.

Following [LNP22], the size of the non-interactive proof is around

$$(n + \bar{\ell} + 2 \cdot \lceil 256/d \rceil + \lambda + 2) \cdot d \log(Q) + \log(2\kappa + 1) \cdot d + (m_1 + v_e)d \cdot \log(2^{2.57}\,\mathfrak{s}_1) \\ + m_2 \cdot d \cdot \log(2^{2.57}\mathfrak{s}_2) + 256 \cdot \log(2^{2.57}\mathfrak{s}^{(e)})$$

where

– n is the height of a M-SIS matrix
– d is a ring dimension
– $\bar{\ell} = 0$ because the entire witness is small
– Q is a modulus
– λ is a repetition rate so that $Q^{-\lambda}$ is negligible
– κ is a bound on the challenge coefficients e.g. $\kappa = 2$
– m_1 is the module dimension of $\left(\boldsymbol{r}_{\mathsf{com}}, \bar{s}^{(0)}, \bar{s}^{(1)}, \bar{s}^{(2)}, \tilde{s}^{(0)}, \tilde{s}^{(1)}, \{\boldsymbol{r}_{i,j}^{(\tau)}\}\right)$
– v_e is the module dimension of $\boldsymbol{x}'$ where $\boldsymbol{x}'$ consists of binary representation of norms associated to the exact bound statements. Specifically, $\boldsymbol{x}'$ consists of one ring element per exact bound amounting to $v_e = (3 + 2 + 4e\ell + 2\tilde{\ell}$ ring elements.
– M-$\mathsf{SIS}_{\mathcal{R}_q, n, m_1+v_e+m_2, \beta_{\mathsf{MSIS}}}$ is hard for

$$\beta_{\mathsf{MSIS}} = 8 \cdot 59 \cdot \sqrt{2d(m_1 + v_e)\mathfrak{s}_1^2 + 2dm_2\mathfrak{s}_2^2}.$$

– $\mathfrak{s}_1$ is a small multiple[22] of $\left\|\left(\boldsymbol{r}_{\mathsf{com}}, \bar{s}^{(0)}, \bar{s}^{(1)}, \bar{s}^{(2)}\tilde{s}^{(1)}, \tilde{s}^{(2)}, \{\boldsymbol{r}_{i,j}^{(\tau)}\}, \boldsymbol{x}'\right)\right\|$, i.e.

$$\mathfrak{s}_1 = \gamma_1 \cdot 59\sqrt{\beta_{\mathsf{com}}^2 d\lceil e_{\mathsf{com}}/d \rceil + d(m_1 - 1 + v_e)}$$

[22] The constant γ_1, as well as the constants γ_2 and $\gamma^{(e)}$ for $\mathfrak{s}_2$ and $\mathfrak{s}^{(e)}$ appear as part of the repetition rate of the proof.

- $\mathfrak{s}_2$ is a part of the M-SIS bound (e.g. $\mathfrak{s}_2 = \gamma_2 \cdot 59 \cdot \sqrt{dm_2}$)
- $\mathfrak{s}^{(e)}$ is a small multiple of

$$\sqrt{\begin{aligned}(m_1-1)\cdot d + 2\,\beta_{\mathsf{com}}^2 + \beta_{\mathsf{pk}}^2 + 2\,\beta^2 + 2\,e\cdot\ell\cdot(\beta')^2 + 2\,\tilde{\ell}(\beta'')^2 + 2\,\log(\beta_{\mathsf{com}})\\ + \log(\beta_{\mathsf{pk}}) + 2\,\log(\beta) + 2\,e\ell\log(\beta') + 2\,\tilde{\ell}\log(\beta'')\end{aligned}}$$

  i.e. $\gamma^{(e)}\sqrt{337}$ times the above.
- The M-$\mathsf{LWE}_{\mathcal{R}_q,m_2-n-\bar{\ell}-\lambda-4,n+\bar{\ell}+\lambda+4,\mathbb{Z}_3,\mathbb{Z}_3}$ problem is hard.

To offer some concrete estimates, we use our OPRF implementation parameters so that $d = 512, e = 656, Q \approx 2^{32}, \lambda = 4$. In actual fact, the structured LWE assumption used by the parameters is a M-LWE assumption in rank 2. As mentioned before, we effectively set $d = 1,024$ in our analysis. We also have a standard deviation of $\sigma \approx 2^{17}$ for the plain LWE error distributions, $\bar{\sigma} \approx 174$ for the M-LWE (or R-LWE as we are viewing it) error distributions and $\sigma_{\mathsf{com}} = 2$. This implies that we should take $\beta_{\mathsf{com}} = 2 \cdot \sqrt{e_{\mathsf{com}}} = 2 \cdot \sqrt{2048}$, $\beta_{\mathsf{pk}} = 2^{17} \cdot \sqrt{e \log Q} = 2^{44} \cdot \sqrt{656 \cdot 32}$, $\beta = \beta' = 174 \cdot \sqrt{2d} = 174 \cdot \sqrt{2048}$, $\beta'' = 2^{17} \cdot \sqrt{d} = 2^{17} \cdot \sqrt{1024}$. We also have $\ell = 2, \bar{\ell} = 4$. Recall that for our OPRF, the variables $\bar{s}_1, \bar{s}_2, \tilde{s}^{(1)}$ and $\{\boldsymbol{r}_{i,j}^{(\tau)} : \tau = 2\}$ are not present in the OPRF setting so we update the proof parameters accordingly.

For the M-SIS assumption, we aim for $\delta = 1.0045$ as in [LNP22]. We may then take $\gamma_1 = 40, \gamma_2 = 10, n = 3$, for the M-SIS problem noting that the parameter m_2 dimension has essentially no effect on hardness. Next, the lattice estimator tells us that a module rank of 2 is all that is required to make the M-LWE problem hard. Therefore, we require that $m_2 \geq 2 + n + \bar{\ell} + \lambda + 4 = 11$. With the constants $\gamma_1 = 40, \gamma_2 = \gamma^{(e)} = 10$, we get an approximate proof size of 25.2MB. Additionally, the expected repetition rate of the prover (that is, the expected number of times the prover runs the proof before the rejection sampling algorithms output a success) with these parameters is less than 3. Note that there is a trade-off between the prover's expected running time and the proof size. That is, we can reduce the "γ" parameters to get better proof sizes. However, the price to pay is a larger expected repetition rate. For example, taking $\gamma_1 = 10, \gamma_2 = \gamma^{(e)} = 5$ gives an expected repetition rate of less than 9 and a proof size of around 23.2MB.

Dilithium Compression. We can also use the compression techniques from the Dilithium signature scheme [LDK+22]. By dropping D bits for certain ring elements, we save $(D - 2.25) \cdot n \cdot d \cdot \log Q$ bits in the proof size. However, even with the overly optimistic $D = 30$, this only saves a few hundred kilobytes so the optimisation does not change the overall proof size in any meaningful way.

Verifiable OPRF. In the verifiable case, our parameters are somewhat speculative and we do not recommend implementing our scheme. However, to give an illustration of performance, we may choose parameters so that the bootstrapping depth of the server evaluation is exactly 2. For this, we may use the

`PARAM_MESSAGE_8_CARRY_0` from the `tfhe-rs` library, modifying the plaintext space to $P = 251$. We also re-introduce the variables $\bar{s}_1, \bar{s}_2, \tilde{s}^{(1)}$ and $\{\boldsymbol{r}_{i,j}^{(\tau)} : \tau = 2\}$ and the associated bounds. The VOPRF parameters are $Q \approx 2^{64}, d = 32768, e = 1017, \sigma \approx 2^{40}, \bar{\sigma} = 4$. Additionally, the decomposition parameters are $\ell = 2$ and $\tilde{\ell} = 5$. The reason for this increase in parameters is to allow all linear operations in the homomorphic PRF evaluation to be performed without any bootstrapping. Using $(m_2, n) = (8, 1)$ and the same repetition rate parameters, the overall proof size becomes around 2763MB or 2572MB for repetition rates less than 3 and 9 respectively. Note that these proof sizes may be compressed by working over a smaller ring size as is done in using the alternative proof system below. For brevity, we choose not to explore this here.

LaBRADOR [BS22]. The recursive LaBRADOR proof system [BS22] can also handle the type of statements we require zero-knowledge proofs for here. LaBRADOR is a recursive proof system achieving a poly-logarithmic size and a very small slackness parameter of approximately 2. This slackness factor should be taken into account when choosing TFHE parameters because the size of the errors in the bootstrapping key may lead to incorrect evaluation and the leaking of secret information. That is, the extracted witness is at most twice as long as a genuine witness. We note that LaBRADOR is not zero-knowledge but as pointed out in [BS22] this can be addressed by adding a thin "zero-knowledge shim". In what follows, we will assume that this does not change the order of the proof sizes.

In order to use LaBRADOR proofs, we will consider the error terms

$$\boldsymbol{e}_{\mathsf{com}}, \boldsymbol{e}_{\mathsf{pk}}, \boldsymbol{e}^{(\tau)}, \boldsymbol{E}_i^{(\tau)}, e_{i,j}^{(\tau)}$$

as part of the witness where the super-script τ either ranges over $\{1, 2\}$ or is simply dropped depending on whether we are in the verifiable or non-verifiable case. In particular, whether these terms are over $\mathbb{Z}_Q$ or not, they will be considered as $\mathcal{R}_Q$ elements where at most d coefficients are placed in a single $\mathcal{R}_Q$ element. The error terms add up to

$$\lceil (e_{\mathsf{com}} + e \cdot \log Q)/d \rceil + T_v \cdot (2 + 4\, e \cdot \ell + \tilde{\ell})$$

ring elements where $T_v = 2$ in the verifiable case and 1 in the non-verifiable case. Adding in the terms coming from the $\bar{s}^{(\cdot)}, \tilde{s}^{(\cdot)}, \boldsymbol{r}_{\mathsf{com}}$ and $\boldsymbol{r}_{i,j}^{(\tau)}$, we end up with a witness consisting of

$$\lceil ((2T_v - 1) \cdot e + 2\, e_{\mathsf{com}} + e \cdot \log Q)/d \rceil + T_v \cdot (3 + 8\, e \cdot \ell + \tilde{\ell})$$

ring elements. In order to cost LaBRADOR proofs, we also require a Euclidean norm bound on the witness. In the case of our statement, this equates to

$$\sqrt{(2T_v - 1)e + T_v d + 4T_v e\ell d + \sigma_{\mathsf{com}}^2 \cdot (2e_{\mathsf{com}}) \\ + \sigma^2 \cdot (e \log Q + T_v \tilde{\ell} d) + \bar{\sigma}^2 T_v \cdot (2d + 4e\ell d)}$$

if all Gaussians with the same standard deviation are grouped together. Plugging in the aforementioned OPRF parameters, the witness consists of 21069 ring elements and the norm bound on the witness is around 2^{11}. Using the LaBRADOR proof system then yields a proof size of 41.42kB. In order to justify this number, we write our constraints/witness over $\mathbb{Z}_q[X]/(X^d + 1)$ for power-of-two $d > 64$ as constraints/witness over $\mathbb{Z}_q[X]/(X^{64} + 1)$. Next, we divide our 168464 witness ring elements ($\in \mathbb{Z}_q[X]/(X^{64} + 1)$) into 24 module elements, each of rank 7019 (padding the witness to make this splitting exact). Finally, we exhaustively search over five recursion levels to find recursion parameters (3, 3, 2, 1, 3). That is the first recursion uses $\nu = 3$, the second uses $\nu = 3$, the third $\nu = 2$ etc.

For the verifiable OPRF parameters, splitting the witness into 256 module elements yields a LaBRADOR proof size is around 69.85kB at five recursion levels. Note that all of these costs may be optimised further by tweaking the proof parameters.

Remark 6. We note that in our final constructions the preparation of public keys and accompanying zero-knowledge proofs can be a one-time cost, as they can be re-used.

Ciphertext Packing. Finally, we will estimate the cost of the LaBRADOR proofs in the case where we pack multiple LWE ciphertexts into a single RLWE ciphertext to save bandwidth. Essentially, we add $\log(d) \cdot d \cdot \tilde{\ell}$ error terms to the witness, each of which a discrete Gaussian with parameter $\sigma \approx 2^{17}$. There are no extra secret terms as the key-switching keys are associated to automorphisms that essentially shuffle the secret coefficients. With these updated values, the LaBRADOR proofs remains roughly the same size as reported above. Therefore, the additional key material does not contribute significantly to the communication cost of our constructions.

B.2 Protected Encoded Input Proofs (OPRF)

Computation of the function $F_{\mathsf{poprf}}.\mathsf{HEEval}$ (see Algorithm 2) requires a ciphertext encrypting $\boldsymbol{y}$ that $\boldsymbol{H}_{\mathsf{inp}} \cdot \boldsymbol{G}_{\mathsf{gadget}} \cdot \boldsymbol{y} \equiv \boldsymbol{0} \bmod q$. Therefore, if we want the server to compute this function, the client must prove that its ciphertext is well-formed, i.e. that it encrypts a vector $\boldsymbol{y} \in \mathbb{Z}_p^{n_p}$ satisfying the requirement. In the following section we consider the case where $\boldsymbol{y}$ is a non-unique ternary representation of a valid encoded binary input which is useful for verifiability. Recall that our suggested parameters are $p = 2, q = 3$ and $n_p = 256$. An honest server never has to multiply ciphertexts, so a full GSW ciphertext is not necessary and we can assume a TLWE ciphertext of $\boldsymbol{y} \in \mathbb{Z}_p^{n_p}$ is sent. As before, we let Q denote the ciphertext modulus of TFHE, e the TLWE dimension and e_{com} the commitment dimension. The plaintext modulus will be denoted P. We utilise a public key, or more specifically in our case, the commitment $\boldsymbol{b}_{\mathsf{pk}}$ to the root TFHE secret key $\boldsymbol{s} \in \mathbb{Z}_2^e$ (this is the key denoted $\boldsymbol{s}^{(0)}$ in the previous section). Then we prove that the ciphertext is a *secret key* encryption of $\boldsymbol{y}$ where the key used is consistent with $\boldsymbol{b}_{\mathsf{pk}}$. This is more efficient than proving knowledge of the comparatively large amount of encryption randomness required to produce encryptions using the public key method. In what follows, we assume the matrix $\boldsymbol{A}$ is sampled uniformly from $\mathbb{Z}_Q^{n_p \times e}$. We want a zero knowledge proof for $\boldsymbol{s} \in \mathbb{Z}_2^e, \boldsymbol{e} \in \mathbb{Z}_Q^{n_p}, \boldsymbol{r}_{\mathsf{com}}, \boldsymbol{e}_{\mathsf{com}} \in \mathbb{Z}_Q^{e_{\mathsf{com}}}$ as well as some $\boldsymbol{y} \in \mathbb{Z}_p^{n_p}$ such that:

- $\boldsymbol{b}_{\mathsf{pk}} = \boldsymbol{A}_{pp} \cdot \boldsymbol{r}_{\mathsf{com}} + \boldsymbol{e}_{\mathsf{com}} + \lfloor Q/2 \rceil \cdot \boldsymbol{s} \bmod Q$
- $\boldsymbol{C} = \begin{bmatrix} \boldsymbol{A}^\top \\ \boldsymbol{s}^\top \boldsymbol{A}^\top + \boldsymbol{e}^\top \end{bmatrix} + \begin{bmatrix} \boldsymbol{0}^{e \times n_p} \\ \lfloor Q/P \rceil (\boldsymbol{y})^\top \end{bmatrix} \bmod Q$
- The entries of $\boldsymbol{y}$ are all in $\mathbb{Z}_p$
- $\boldsymbol{H}_{\mathsf{inp}} \cdot \boldsymbol{G}_{\mathsf{gadget}} \cdot \boldsymbol{y} \equiv 0 \bmod q$
- $\boldsymbol{s} \in \mathbb{Z}_2^e$
- $\|\boldsymbol{r}_{\mathsf{com}}\|, \|\boldsymbol{e}_{\mathsf{com}}\| \leq \beta_{\mathsf{com}}$
- $\|\boldsymbol{e}\| \leq \beta_{\mathsf{ct}}$

Note that $\boldsymbol{H}_{\mathsf{inp}} \cdot \boldsymbol{G}_{\mathsf{gadget}} \in \mathbb{Z}_q^{(n_q - n) \times n_p}$. If we assume that Q is big enough so that $\boldsymbol{H}_{\mathsf{inp}} \cdot \boldsymbol{G}_{\mathsf{gadget}} \cdot \boldsymbol{y}$ does not wrap-around modulo Q, then we may replace the final equation by $\boldsymbol{H}_{\mathsf{inp}} \cdot \boldsymbol{G}_{\mathsf{gadget}} \cdot \boldsymbol{y} = q\boldsymbol{v}$ over the Integers where $\|q\boldsymbol{v}\|_\infty \leq \|\boldsymbol{H}_{\mathsf{inp}} \cdot \boldsymbol{G}_{\mathsf{gadget}}\|_1 \ll Q$. This implies that the same equation holds modulo Q i.e. $\boldsymbol{H}_{\mathsf{inp}} \cdot \boldsymbol{G}_{\mathsf{gadget}} \cdot \boldsymbol{y} = q\boldsymbol{v} \bmod Q$ where $\|\boldsymbol{v}\|_\infty \leq \frac{\|\boldsymbol{H}_{\mathsf{inp}} \cdot \boldsymbol{G}_{\mathsf{gadget}}\|_1}{q} =: B_v \ll Q$.

Recall that $(p, q) = (2, 3)$. Denoting $\boldsymbol{c}$ as the last row of $\boldsymbol{C}$, we want a zero-knowledge proof of $\boldsymbol{s} \in \mathbb{Z}_Q^e, \boldsymbol{y} \in \mathbb{Z}_Q^{n_p}$ and $\boldsymbol{r}_{\mathsf{com}} \in \mathbb{Z}_Q^{e_{\mathsf{com}}}$ such that

- $\boldsymbol{y}$ and $\boldsymbol{s}$ have binary entries (these statements are split into a quadratic relation and a bound check)
- $\|\boldsymbol{r}_{\mathsf{com}}\| \leq \beta_{\mathsf{com}}$
- $\|\boldsymbol{b}_{\mathsf{pk}} - \boldsymbol{A}_{pp} \cdot \boldsymbol{r}_{\mathsf{com}} - \lfloor Q/2 \rceil \cdot \boldsymbol{s}\| \leq \beta_{\mathsf{com}}$
- $\|\boldsymbol{c} - \boldsymbol{A} \cdot \boldsymbol{s} - \lfloor Q/P \rceil \boldsymbol{y}\| \leq \beta_{\mathsf{ct}}$
- $\|\boldsymbol{v}\|_\infty := \left\|q^{-1} \cdot \boldsymbol{H}_{\mathsf{inp}} \cdot \boldsymbol{G}_{\mathsf{gadget}} \cdot \boldsymbol{y}\right\|_\infty \leq B_v$ (approximately)

Note that only $\boldsymbol{r}_{\mathsf{pk}}, \boldsymbol{s}$ and $\boldsymbol{y}$ are part of the witness in the above. In particular, $\boldsymbol{v}$ is not part of the witness. When expressed as a proof of the form captured by [LNP22], the cost of running a proof is around

$$(n + \bar{\ell} + 2 \cdot \lceil 256/d \rceil + \lambda + 2) \cdot d \cdot \log(Q) + \log(2\kappa + 1) \cdot d + (m_1 + v_e) \cdot d \cdot \log(2^{2.57}\mathfrak{s}_1) \\ + m_2 \cdot d \cdot \log(2^{2.57}\mathfrak{s}_2) + 256 \cdot \left(\log(2^{2.57}\,\mathfrak{s}^{(e)}) + \log(2^{2.57}\,\mathfrak{s}^{(d)})\right)$$

where

- n is the height of a M-SIS matrix
- $\bar{\ell} = 0$ as the entire witness is short
- $d = 128$ is a ring dimension
- Q is a modulus
- λ is a repetition rate so that $Q^{-\lambda}$ is negligible
- κ is a bound on the challenge coefficients e.g. $\kappa = 2$
- m_1 is the module dimension required to store $(\boldsymbol{r}_{\mathsf{com}}, \boldsymbol{s}, \boldsymbol{y})$ i.e.

$$m_1 = \lceil e_{\mathsf{com}}/128 \rceil + \lceil e/128 \rceil + \lceil n_p/128 \rceil$$

– v_e is the module dimension of $\boldsymbol{x}'$ where $\boldsymbol{x}'$ is the concatenation of binary representation of the witness norms in the exact norm proofs i.e.

$$v_e = 2\lceil \log \beta_{\mathsf{com}}/128\rceil + \lceil \beta_{\mathsf{ct}}/128\rceil = 3$$

– $M\text{-}\mathsf{SIS}_{\mathcal{R}_Q,n,m_1+v_e+m_2,\beta}$ is hard for $\beta = 8\cdot 59\cdot \sqrt{2d(m_1+v_e)\mathfrak{s}_1^2 + 2dm_2\mathfrak{s}_2^2}$.
– $\mathfrak{s}_1$ is a small multiple[23] of $\|(\boldsymbol{r}_{\mathsf{pk}}, \boldsymbol{s}, \boldsymbol{y}, \boldsymbol{x}')\|$ where $\boldsymbol{x}'$ is a binary module element of rank v_e i.e.

$$\mathfrak{s}_1 = \gamma_1 \cdot 59\sqrt{\beta_{\mathsf{com}}^2 e_{\mathsf{com}} + d(m_1 - \lceil e_{\mathsf{com}}/128\rceil + v_e)}$$

– $\mathfrak{s}_2$ is a part of the $M\text{-}\mathsf{SIS}$ bound, e.g. $\mathfrak{s}_2 = \gamma_2\cdot 59\cdot\sqrt{dm_2}$
– $\mathfrak{s}^{(e)} = \gamma^{(e)}\cdot\sqrt{337}\cdot\sqrt{e+n_p+2\beta_{\mathsf{com}}^2+\beta_{\mathsf{ct}}^2+2\log(\beta_{\mathsf{com}})+\log(\beta_{\mathsf{ct}})}$.
– $\mathfrak{s}^{(d)}$ is a small multiple of B_v i.e. $\gamma^{(d)}\cdot\sqrt{337}\cdot B_v$
– The $M\text{-}\mathsf{LWE}_{\mathcal{R}_Q,m_2-n-\bar{\ell}-\lambda-4,n+\bar{\ell}+\lambda+4,\mathbb{Z}_3,\mathbb{Z}_3}$ problem is hard.

For the implementation parameter set, we take $Q = 2^{32}, \beta_{\mathsf{com}} = 2\cdot\sqrt{2048}, \beta_{\mathsf{ct}} \approx 2^{17}\cdot\sqrt{n_p} = 2^{17}\cdot\sqrt{256}$. We also take $B_v = 128$. To choose parameters so that the $M\text{-}\mathsf{SIS}$ problem is hard, we may take $n = 11$ for the moment. Note that the $M\text{-}\mathsf{SIS}$ problem does not depend heavily on the number of columns so there is no resulting restriction on m_2 aside from its appearance in the bound term. However, when analysing the $M\text{-}\mathsf{LWE}$ problem using the lattice estimator, we find that we must take $m_2 \geq 11 + n + \bar{\ell} + \lambda + 4 = 30$ where $\lambda = 4$. Taking the most efficient choice of $m_2 = 30$, it can be verified that the $M\text{-}\mathsf{SIS}$ problem is indeed hard when aiming for a Hermite root factor of $\delta = 1.0045$ as in [LNP22] with $\gamma_1 = 40, \gamma_2 = \gamma^{(e)} = \gamma^{(d)} = 10$. This gives an overall proof size of 23.8kB and an expected repetition rate of less than 3. We stress that there is a trade-off between the number of repetitions by the prover and proof-size, so these estimates only give a rough idea of the communication cost. For example, taking $\gamma_1 = 10, \gamma_2 = \gamma^{(e)} = \gamma^{(d)} = 5$ allows us to pick $n = 10, m_2 = 29$ leading to an expected repetition rate of less than 9 and a 22.1kB proof.

Dilithium Compression. We can compress the proof further by cutting $D > 0$ lower bits of certain full-sized module elements and $\log(\gamma)$ bits of smaller module elements at the expense of introducing some small hint data. This idea was introduced by the Dilithium signature scheme [LDK+22]. When applying Dilithium compression, the $M\text{-}\mathsf{SIS}$ assumption changes to $M\text{-}\mathsf{SIS}_{\mathcal{R}_Q,n,m_1+v_e+m_2,\beta'}$ where $\beta' = 4\cdot 59\sqrt{\beta_1^2+\beta_2^2}$ for $\beta_1 := 2\mathfrak{s}_1\sqrt{2(m_1+v_e)d}$ and $\beta_2 := 2\mathfrak{s}_2\sqrt{2m_2d} + 2^D\cdot 59\sqrt{nd} + \gamma\sqrt{nd}$. In order to pick new parameters, we stick to the same value of n, choose γ to be the largest such that the $M\text{-}\mathsf{SIS}$ problem remains hard for $D = 0$ *and* $(Q-1)/\gamma \in \mathbb{Z}$, then pick the largest D such that the $M\text{-}\mathsf{SIS}$ problem is hard and $2^{D-1}\kappa d < \gamma$ [LNP22]. For this, we set $Q = 2^{32} - 315 \approx 2^{32}$, $\gamma = 947\cdot 281\cdot 5$ and $D = 12$. The proof size decreases by $(D - 2.25)\cdot nd$ bits which equates to 1.7kB. The compression parameters for repetition rate 9 can be chosen as $D = 11, \gamma = 957 * 281$ leading to a reduction of 1.4kB. Taking all of this into account, we have a final proof size of around 22.1kB at an expected repetition rate of less than 3 and 26.7kB at a rate of less than 9.

[23] Again, the constants are related to the repetition rate of the proof.

Amortisation Note that the above is concerned only with the case where the client submits a single encrypted PRF input. However, in the interest of amortising the cost of zero-knowledge proofs, we can consider the case where the client wishes to send L ciphertexts at once. The main changes to the proof parameters are that $m_1 = \lceil e_{\mathsf{com}}/128 \rceil + \lceil e/128 \rceil + L \cdot \lceil n_p/128 \rceil, v_e = 2 + L, \mathfrak{s}^{(e)} = \gamma^{(e)} \cdot \sqrt{337} \cdot \sqrt{e + L \cdot n_p + 2\beta_{\mathsf{com}}^2 + L \cdot \beta_{\mathsf{ct}}^2 + 2\log(\beta_{\mathsf{com}}) + L \cdot \log(\beta_{\mathsf{ct}})}, s^{(d)} = \gamma^{(d)} \cdot \sqrt{337} \cdot \sqrt{L \cdot B_v^2}$. Other than that, the description of the proof is the same. As an example we can take $L = 64$. Before Dilithium compression, the proof size is around 93.1kB or 85.1kB for the repetition rate 3 and 9 parameters respectively. Note that these estimates are computed using securely chosen $(m_2, n) = (31, 12)$ and $(30, 11)$ for the repetition rate 3 and 9 parameters respectively. Furthermore, Dilithium compression parameters can be chosen as $(D, \gamma) = (13, 947 \cdot 281 \cdot 6)$ and $(12, 947 \cdot 281 \cdot 2)$ to result in a decrease of 2.0kB and 1.7kB respectively. This results in an amortised client online proof size of 1.42kB or 1.3kB per query.

LaBRADOR Proofs. We also tried to use the LaBRADOR proof system for the statements in this section. However, when attempting this, it turns out that the size of the statements for e.g. $L = 64$ render recursion ineffective and the proof sizes are larger than those reported above.

B.3 Non-Unique Encoding Proofs (VOPRF)

We now discuss the costs of the protected encoded input proofs when the encoding is not unique. This is used in our VOPRF candidate. In other words, we now discuss the case where a client ciphertext encrypts an input $\bar{\boldsymbol{y}}$ that is equivalent modulo p to an $\boldsymbol{y} \in \mathbb{Z}_p^{n_p}$ such that $\boldsymbol{H}_{\mathsf{inp}} \cdot \boldsymbol{G}_{\mathsf{gadget}} \cdot \boldsymbol{y} \equiv \boldsymbol{0} \bmod q$. Reusing the notation from Section B.2, we want a zero knowledge proof for $\boldsymbol{s} \in \mathbb{Z}_2^e, \boldsymbol{e} \in \mathbb{Z}_Q^{n_p}, \boldsymbol{r}_{\mathsf{com}}, \boldsymbol{e}_{\mathsf{com}} \in \mathbb{Z}_Q^{e_{\mathsf{com}}}$ as well as some $\boldsymbol{y} \in \mathbb{Z}_p^{n_p}$ and $\bar{\boldsymbol{y}} \in \mathbb{Z}_q^{n_p}$ such that:

- $\boldsymbol{b}_{\mathsf{pk}} = \boldsymbol{A}_{pp} \cdot \boldsymbol{r}_{\mathsf{com}} + \boldsymbol{e}_{\mathsf{com}} + \lfloor Q/2 \rceil \boldsymbol{s} \bmod Q$
- $\boldsymbol{C} = \begin{bmatrix} \boldsymbol{A}^\top \\ \boldsymbol{s}^\top \boldsymbol{A}^\top + \boldsymbol{e}^\top \end{bmatrix} + \begin{bmatrix} \boldsymbol{0}^{e \times n_p} \\ \lfloor Q/P \rceil \left(\bar{\boldsymbol{y}}^{(2)}\right)^\top \end{bmatrix} \bmod Q$
- $\boldsymbol{H}_{\mathsf{inp}} \cdot \boldsymbol{G}_{\mathsf{gadget}} \cdot \boldsymbol{y} \equiv 0 \bmod q$
- The entries of $\boldsymbol{y}$ are all in $\mathbb{Z}_p$ and the entries of $\bar{\boldsymbol{y}}$ are in $\mathbb{Z}_q$
- $\bar{\boldsymbol{y}}$ and $\boldsymbol{y}$ encode the same value modulo p
- $\boldsymbol{s} \in \mathbb{Z}_2^e$
- $\|\boldsymbol{r}_{\mathsf{com}}\|, \|\boldsymbol{e}_{\mathsf{com}}\| \leq \beta_{\mathsf{com}}$
- $\|\boldsymbol{e}\| \leq \beta_{\mathsf{ct}}$

Once again, we consider $(p, q) = (2, 3)$ concretely. Suppose we restrict $\boldsymbol{y}$ to be a binary, i.e. $\mathbb{Z}_p$, vector. Suppose also that we restrict $\bar{\boldsymbol{y}}$ to be ternary i.e. a $\mathbb{Z}_q$ vector (assuming integers modulo $q = 3$ are represented in $\{-1, 0, 1\}$). Then one way to provide a non-unique encoding is to require the existence of a short vector $\boldsymbol{w}$, such that $\boldsymbol{y} = \bar{\boldsymbol{y}} + 2\boldsymbol{w}$ over the Integers. Note that on the assumption that $\boldsymbol{y}$ and $\bar{\boldsymbol{y}}$ are binary and ternary respectively, we can prove this by showing that

$\boldsymbol{y} = \bar{\boldsymbol{y}} + 2\boldsymbol{w} \bmod Q$ and providing a very loose range proof on $\boldsymbol{w}$. Furthermore, in order to show that $\bar{\boldsymbol{y}}$ is ternary, we may write it as the difference of two binary vectors e.g. $\bar{\boldsymbol{y}} = \bar{\boldsymbol{y}}_\ell - \bar{\boldsymbol{y}}_r$ and show that $\bar{\boldsymbol{y}}_\ell$ and $\bar{\boldsymbol{y}}_r$ are both binary. Taking all of this into account and using the strategy of the previous section, we may prove knowledge of $\boldsymbol{r}_{\mathsf{com}} \in \mathbb{Z}_Q^{e_{\mathsf{com}}}, \boldsymbol{s} \in \mathbb{Z}_Q^e, \boldsymbol{y}, \bar{\boldsymbol{y}}_\ell, \bar{\boldsymbol{y}}_r, \boldsymbol{w} \in \mathbb{Z}_Q^{n_p}$ such that:

- $\boldsymbol{y}, \bar{\boldsymbol{y}}_\ell, \bar{\boldsymbol{y}}_r$ and $\boldsymbol{s}$ have binary entries (which are treated as a quadratic relation plus a bound check)
- $\|\boldsymbol{r}_{\mathsf{com}}\| \le \beta_{\mathsf{com}}$
- $\|\boldsymbol{b}_{\mathsf{pk}} - \boldsymbol{A}_{pp} \cdot \boldsymbol{r}_{\mathsf{com}} - \lfloor Q/2 \rceil \boldsymbol{s}\| \le \beta_{\mathsf{com}}$
- $\|\boldsymbol{c} - \boldsymbol{A} \cdot \boldsymbol{s} - \lfloor Q/P \rceil (\bar{\boldsymbol{y}}_\ell - \bar{\boldsymbol{y}}_r)\| \le \beta_{\mathsf{ct}}$
- $\|\boldsymbol{v}\|_\infty := \left\|3^{-1} \cdot \boldsymbol{H}_{\mathsf{inp}} \cdot \boldsymbol{G}_{\mathsf{gadget}} \cdot \boldsymbol{y}\right\|_\infty \le B_v$ (approximately)
- $\boldsymbol{y} = (\bar{\boldsymbol{y}}_\ell - \bar{\boldsymbol{y}}_r) + 2\boldsymbol{w} \bmod Q$
- $\|\boldsymbol{w}\|_\infty \le 1$ (approximately)

Once again, the cost of running a proof using [LNP22] is around

$$(n + \bar{\ell} + 2 \cdot \lceil 256/d \rceil + \lambda + 2) \cdot d \cdot \log(Q) + \log(2\kappa + 1) \cdot d + (m_1 + v_e) \cdot d \cdot \log(2^{2.57}\mathfrak{s}_1) \\ + m_2 \cdot d \cdot \log(2^{2.57}\mathfrak{s}_2) + 256 \cdot \left(\log(2^{2.57}\,\mathfrak{s}^{(e)}) + \log(2^{2.57}\,\mathfrak{s}^{(d)})\right)$$

where

- n is the height of a M-SIS matrix
- $\bar{\ell} = 0$ as the entire witness is short
- $d = 128$ is a ring dimension
- Q is a modulus
- λ is a repetition rate so that $Q^{-\lambda}$ is negligible
- κ is a bound on the challenge coefficients e.g. $\kappa = 2$
- m_1 is the module dimension required to store $(\boldsymbol{r}_{\mathsf{com}}, \boldsymbol{s}, \boldsymbol{y}, \bar{\boldsymbol{y}}_\ell, \bar{\boldsymbol{y}}_r, \boldsymbol{w})$ i.e. $m_1 = \lceil e_{\mathsf{com}}/d \rceil + \lceil e/d \rceil + 4 \cdot \lceil n_p/d \rceil$
- v_e is the module dimension of $\boldsymbol{x}'$ where $\boldsymbol{x}'$ contains the binary representation of the vectors in the exact norm proofs (i.e. $v_e = 2\lceil \log(\beta_{\mathsf{com}})/128 \rceil + \lceil \log(\beta_{\mathsf{ct}})/128 \rceil = 3$)
- M-$\mathsf{SIS}_{\mathcal{R}_Q, n, m_1+v_e+m_2, \beta}$ is hard for $\beta = 8 \cdot 59 \cdot \sqrt{2d(m_1 + v_e)\mathfrak{s}_1^2 + 2dm_2\mathfrak{s}_2^2}$.
- $\mathfrak{s}_1 = \gamma_1 \cdot 59\sqrt{\beta_{\mathsf{com}}^2 + d(m_1 - \lceil e_{\mathsf{com}}/128 \rceil + v_e)}$
- $\mathfrak{s}_2 = \gamma_2 \cdot 59 \cdot \sqrt{dm_2}$
- $\mathfrak{s}^{(d)} = \gamma^{(d)} \cdot \sqrt{337} \cdot \sqrt{B_v^2 + n_p}$
- $\mathfrak{s}^{(e)} = \gamma^{(e)} \cdot \sqrt{337} \cdot \sqrt{e + 3n_p + 2\beta_{\mathsf{com}}^2 + \beta_{\mathsf{ct}}^2 + 2\log(\beta_{\mathsf{com}}) + \log(\beta_{\mathsf{ct}})}$
- The M-$\mathsf{LWE}_{\mathcal{R}_Q, m_2-n-\bar{\ell}-\lambda-4, n+\bar{\ell}+\lambda+4, \mathbb{Z}_3, \mathbb{Z}_3}$ problem is hard.

Pending further cryptanalysis, we do not recommend implementing our VOPRF. However, we may still provide an account of performance by using the `PARAM_MESSAGE_8_CARRY_0` parameters assuming a plaintext space of $P = 251$. The main parameters of interest are $Q \approx 2^{64}, e = 1017, \sigma \approx 2^{40}$. The commitment parameters σ_{com} and e_{com} are unchanged. For the M-LWE problem to be hard, we now require a module rank of 20. Before Dilithium compression, the proofs for repetition rate 3 and 9 use $(m_2, n) = (32, 6)$ and $(31, 5)$ respectively resulting in proof sizes of 40.8kB and 37.4kB. We will use $Q = 2^{64} - 179$ along with

Dilithium compression parameters $(D, \gamma) = (12, 4051 \cdot 41 \cdot 4)$ and $(11, 4051 \cdot 41 \cdot 2)$ respectively, we obtain proof sizes of 40.8kB or 37.6kB.

For the amortised case where we send $L = 64$ ciphertexts at once. The pre-compressed proofs have sizes of 238.2kB and 217.1kB using $(m_2, n) = (33, 7)$ and $(32, 6)$ respectively. It can be shown that we may use Dilithium compression parameters $(D, \gamma) = (17, 6199 \cdot 101 \cdot 41)$ and $(14, 6199 \cdot 101 \cdot 4)$. Overall this leads to an amortised proof size of 3.7kB or 3.4kB per query.

C Verifiability Construction

In this section, we present our verifiable construction in Figure 5. Note that for simplicity, we have presented a VOPRF, i.e. it is not partial, by setting the tag $\boldsymbol{t} = \perp$.

C.1 POPRIV2 from Verifiability

We show that our verifiability property implies POPRIV2 under some additional assumptions.

Theorem 3. *Our VPOPRF construction satisfies POPRIV2 if*

- F_{vpoprf} *satisfies POPRIV1*
- F_{vpoprf} *is verifiable*
- *NIZKAoK is sound.*

Proof. Let G_0 and G_1 denote the $\mathrm{POPRIV2}^{\mathcal{A},b}_{\mathcal{F},H}$ game for $b = 0$ and $b = 1$ respectively. We consider the possibility of two events, E_0 and E_1, that occur in a query to the Finalise oracle:

1. ct_0 is not the output of $F_{\mathsf{vpoprf}}.\mathsf{BlindEval}(\boldsymbol{A}, \boldsymbol{t}, req_0)$ but $F_{\mathsf{vpoprf}}.\mathsf{Verify}(\boldsymbol{z}_0, \boldsymbol{z}_0^{\star}, \rho_0) = 1$.
2. ct_1 is not the output of $F_{\mathsf{vpoprf}}.\mathsf{BlindEval}(\boldsymbol{A}, \boldsymbol{t}, req_1)$ but $F_{\mathsf{vpoprf}}.\mathsf{Verify}(\boldsymbol{z}_1, \boldsymbol{z}_1^{\star}, \rho_1) = 1$.

We claim that the probability of these events occurring is bound by an adversary against the verifiability property of $\mathcal{F}$ and soundness of NIZKAoK.

To see this, we consider the game G_0' that is defined as G_0 but with the change that the Request oracle requires the adversary to submit its secret key $\boldsymbol{A}$. We further require an additional check when the oracle executes $\mathcal{F}.\mathsf{Request}^{\mathsf{RO}}$, on line 2 we verify that $\hat{\boldsymbol{z}} = \mathcal{F}.\mathsf{Eval}(\boldsymbol{A}, \boldsymbol{t}, \hat{\boldsymbol{x}})$. The success probability of an adversary between these games is bound by the soundness of NIZKAoK. This follows from the fact that the extra check is ensuring that the relation of the argument of knowledge holds using the witness. Any adversary that wins against G_0 but not against G_0' has created a proof $\hat{\pi}$ such that $\hat{\boldsymbol{z}} \neq F_{\mathsf{vpoprf}}.\mathsf{Eval}(\boldsymbol{A}, \boldsymbol{t}, \hat{\boldsymbol{x}})$. A similar argument also bounds the difference in winning probability between G_1 and an analogous game G_1'. Since we assume that $\mathrm{NIZKAoK}_{Rt}$ is sound, we have that $\mathrm{G}_0 \approx_c \mathrm{G}_0'$ and similarly, $\mathrm{G}_1 \approx_c \mathrm{G}_1'$.

We then next observe that if the event E_0 or E_1 has occurred in G_0' or G_1', then we break the verifiability property of $\mathcal{F}$. Once more we initially consider the experiment G_0'. We construct an adversary $\mathcal{B}_{\text{verif}}$ against G_0' when ct_0 is not the output of $\mathcal{F}.\mathsf{BlindEval}(\boldsymbol{A}, \boldsymbol{t}, req_0)$ but it is accepted by the Finalise oracle. To do so, it invokes its own verifiability experiment and receives the public parameters pp, which it uses as pp in G_0'. $\mathcal{A}$ can make queries of two types, Request and Finalise, which $\mathcal{B}_{\text{verif}}$ handles by simply forwarding to its own oracles in the verifiability experiment. Then, $\mathcal{B}_{\text{verif}}$ must guess which of the queries contains the 'winning' response. It does so with probability $1/q_R$ for polynomial q_R which bounds the number of Request queries. It receives the checkpoints $\boldsymbol{x}_i, i \in \mathbb{Z}_n$ and forwards these to its verifiability experiment. It receives back a permuted set of req_i which it sends to $\mathcal{A}$ in response to its Request query. It waits for the corresponding call to the Finalise oracle, at which point it forwards the set rep_i to the experiment G_0'. It wins if $\mathcal{A}$ was able to provide rep that passed verification but was not the honest output of $\mathsf{BlindEval}$. By assumption, this probability is negligible, and therefore we have $\Pr\left[\mathrm{G}_0' \Rightarrow 1\right] \leq \mathsf{negl}(\lambda)$. An analogous argument holds for when the bit $b = 1$ and thus we have shown that probability of event E_0 or E_1 in G_0 or G_1 is negligible.

Then, since we have shown G_i always correctly computes responses rep (with all but negligible probability), we can apply the same argument as we have for POPRIV1. Thus we conclude that the transcript observed by an adversary for POPRIV2 is independent of the challenge bit b, and hence the advantage of the adversary against POPRIV2 is negligible.

By considering the sequence of games, we have shown that $\mathrm{G}_0 \approx_c \mathrm{G}_1$ and thus we obtain the theorem statement. □

D PRF Sage Example

```
# -*- coding: utf-8 -*-
from sage.all import set_random_seed, GF, vector, matrix, random_matrix, codes


class WeakPRF:
    """
    Based on Construction 3.1 of:

    - Boneh, D., Ishai, Y., Passelègue, A., Sahai, A., & Wu, D. J. (2018).
      Exploring crypto dark matter: new simple PRF candidates and their
      applications. Cryptology ePrint Archive, Report 2018/1218.
      https://eprint.iacr.org/2018/1218
    """

    def __init__(self, m_p=256, n_p=256, m_bound=128, p=2, q=3, t=None, seed=None):
        # p.40, optimistic, λ=128
        self.m_p, self.n_p = m_p, n_p
        if seed is not None or t is not None:
            set_random_seed(hash((t, seed)))
        self.A = random_matrix(GF(p), m_p, n_p + 1)
        self.q = q

        if m_bound == 1:
            self.Gout = matrix(GF(q), 1, m_p, [1] * m_p)
        else:
            i = 2
            for i in range(2, m_p // 2):
                C = codes.BCHCode(GF(q), m_p, i)
                if C.dimension() <= m_bound:
                    self.Gout = C.generator_matrix()
                    break
            else:
                raise RuntimeError

    def __call__(self, x):
        y = self.A * vector(list(x) + [1])
        y = y.lift_centered()
        z = self.Gout * y
        return z


class PRF:
    """
    Based on Construction 7.9 of:

    - Boneh, D., Ishai, Y., Passelègue, A., Sahai, A., & Wu, D. J. (2018).
      Exploring crypto dark matter: new simple PRF candidates and their
      applications. Cryptology ePrint Archive, Report 2018/1218.
      https://eprint.iacr.org/2018/1218
    """

    def __init__(self, m_p=256, n_p=256, n=128, m_bound=128, p=2, q=3, t=None, seed=None):
        if p != 2 or q != 3:
            raise NotImplementedError
        self.F = WeakPRF(m_p=m_p, n_p=n_p, m_bound=m_bound, p=p, q=q, t=t, seed=seed)
        self.n = n
        self.Ginp = identity_matrix(GF(q), n).stack(random_matrix(GF(q), (n_p-n)//2, n))

    def __call__(self, x):
        x = x.lift().change_ring(GF(self.F.q))
        y = self.Ginp * x
        z = []
        for y_ in y[:self.n].lift():
            z.append(y_)
        for y_ in y[self.n:].lift():
            z.append(y_ % 2)
            z.append(y_ // 2)
        z = vector(GF(2), self.F.n_p, z)
        return self.F(z)
```

E Size Estimates for LaBRADOR

The script is also attached.

```
# -*- coding: utf-8 -*-
"""
LaBRADOR Pari/GP Code in Sage.
"""

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


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 delta(b) >= delta:
        b += 1

    return b


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

```
default_costf = adps16


def sis_required_block_size(
    n,
    q,
    b,
    m=None,
    step_size=10,
    start=40,
):
    """
    Hardness of SIS in the ℓ_2 norm on A ▫ ZZ_q^{n × m}

    :param n: SIS rank
    :param q: Integer modulus
    :param b: Target ℓ_2 norm of the solution.
    :param m: SIS dimension (default: n ▫ log q)
    :param step_size: Search in steps of this size first (default: 10)
    :param start: Start search at this block size.

    """
    if b > q:
        raise ValueError(f"Size bound {b} > modulus {q}.")

    if m is None:
        m = ceil(2 * n * log(q, 2))

    for block_size in range(start, m + step_size, step_size):
        delta = deltaf(block_size)
        d = min(sqrt(n * log(q) / log(delta)), m)
        if delta ** (d - 1) * q ** (n / d) < b:
            break

    for block_size in range(block_size - step_size, block_size + 1):
        delta = deltaf(block_size)
        d = min(sqrt(n * log(q) / log(delta)), m)
        if delta ** (d - 1) * q ** (n / d) < b:
            return block_size

    return Infinity


@cached_function
def sis_hard_enough(kappa, eta, b, q):
    """
    Return `i` such that for `n = i ▫η ` and a sufficiently big `m` βSIS_ 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 = sqrt(128 / 30.0),
        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()
        # TODO: this does not reflect secpar yet
        size = 256 * gaussian_entropy(old_beta / sqrt(2))  # 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() >= 1:
            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%%" % ((b[1] - b[0]) / max(b[0], b[1]) * 100))

        return size, n, r, beta

    def tail(self, n, r, beta):
        old_beta = vector(beta).norm(2).n()
        size = 256 * gaussian_entropy(old_beta / sqrt(2))  # 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() >= 1:
            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=True):
        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])
```

F Parameter Estimates

The script is also attached.

```
"""
Estimating sizes.
```

```python
"""

from dataclasses import dataclass
from functools import partial
from sage.all import sqrt, ceil, log, cached_function
from labrador import LaBRADOR

# Parameters


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


@dataclass
class Parameters:
    e_com: int
    sigma_com: float
    log_q: int
    n_p: int
    m: int
    tfhe: HashableDict

    def __hash__(self):
        return hash((self.e_com, self.sigma_com, self.log_q, self.n_p, self.m, self.tfhe))


# https://github.com/zama-ai/tfhe-rs/blob/0.2.4/tfhe/src/shortint/parameters/mod.rs#L200
PARAM_MESSAGE_2_CARRY_0 = HashableDict(
    {
        "lwe_dimension": 656,
        "glwe_dimension": 2,
        "polynomial_size": 512,
        "lwe_modular_std_dev": 0.000034119201269311964,
        "glwe_modular_std_dev": 0.00000004053919869756513,
        "pbs_base_log": 8,
        "pbs_level": 2,
        "ks_level": 4,
        "ks_base_log": 3,
        "pfks_level": 1,
        "pfks_base_log": 15,
        "pfks_modular_std_dev": 0.00000000037411618952047216,
        "cbs_level": 0,
        "cbs_base_log": 0,
        "message_modulus": 4,
        "carry_modulus": 1,
    }
)

OPRF = Parameters(
    e_com=2048, sigma_com=2, log_q=32, n_p=256, m=128, tfhe=PARAM_MESSAGE_2_CARRY_0
)

# https://github.com/zama-ai/tfhe-rs/blob/0.2.4/tfhe/src/shortint/parameters/mod.rs#L827
PARAM_MESSAGE_8_CARRY_0 = HashableDict(
    {
        "lwe_dimension": 1017,
        "glwe_dimension": 1,
        "polynomial_size": 32768,
        "lwe_modular_std_dev": 0.0000000460803851108693,
        "glwe_modular_std_dev": 0.0000000000000000002168404344971009,
        "pbs_base_log": 15,
        "pbs_level": 2,
        "ks_level": 5,
        "ks_base_log": 4,
        "pfks_level": 2,
        "pfks_base_log": 15,
        "pfks_modular_std_dev": 0.0000000000000000002168404344971009,
        "cbs_level": 0,
        "cbs_base_log": 0,
        "message_modulus": 251,  # changed
        "carry_modulus": 1,
    }
)

VOPRF = Parameters(
    e_com=4096, sigma_com=2, log_q=64, n_p=256, m=128, tfhe=PARAM_MESSAGE_8_CARRY_0
)


# Utilities


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 consistency_check(params):
    """
    Enforce that we do not pick q too small
    """
    if -params.tfhe["lwe_modular_std_dev"] >= params.log_q:
        raise ValueError(f"log(q) = {params.log_q} is too small.")
    if -params.tfhe["glwe_modular_std_dev"] >= params.log_q:
        raise ValueError(f"log(q) = {params.log_q} is too small.")


def oprf_pk_sizemb(params, compress_pk=True, compress_ct1=True, voprf=False):
    """
    The size of the OPRF public key in MB.

    :param params:
    :param compress_pk: drop lower-order bits where possible
    :param compress_ct1: compress ct1 into one ring element
    :param voprf: estimate the VOPRF

    """
    e = params.tfhe["lwe_dimension"]
    d = params.tfhe["polynomial_size"] * params.tfhe["glwe_dimension"]
    ell = params.tfhe["pbs_level"]
    ell_ = params.tfhe["ks_level"]

    if voprf:
        tau = 2
    else:
        tau = 1

    if compress_pk:
        lwe_keep_bits = ceil(-log(params.tfhe["lwe_modular_std_dev"], 2) + 1)
        glwe_keep_bits = ceil(-log(params.tfhe["glwe_modular_std_dev"], 2) + 1)
    else:
        lwe_keep_bits = params.log_q
        glwe_keep_bits = params.log_q
    if not compress_ct1:
        zeta = tau
    else:
        zeta = tau + log(d, 2)

    return _mb(
        (params.e_com + e * params.log_q) * lwe_keep_bits
        + tau * 2 * d * glwe_keep_bits
        + tau * 2 * e * 2 * ell * d * glwe_keep_bits
        + zeta * d * ell_ * lwe_keep_bits
    )


voprf_pk_sizemb = partial(oprf_pk_sizemb, voprf=True)


def oprf_pk_proof_old_sizemb(params, gamma1=10, gamma2=5, gammae=5):
    """
    Well-formedness proof of the OPRF public key using [LNP22].

    :param params:
    :param gamma1:
    :param gamma2:
    :param gammae:

    """

    ell = params.tfhe["pbs_level"]
    tl = params.tfhe["ks_level"]
    eta = 1  # we absorbed the module dimension in the ring dimension
    d = params.tfhe["polynomial_size"] * params.tfhe["glwe_dimension"]
    e = params.tfhe["lwe_dimension"]
    Q = 2**params.log_q
    lamb = 2 * 128 // params.log_q
    kap = 2
    m1 = (params.e_com + e) // d + eta + 4 * e * ell
    delta = 1.0045
    ve = 4 + 2 * e * ell + tl
    bm1 = m1 + ve
    bcom = params.sigma_com * sqrt(params.e_com)
    s1 = gamma1 * 59 * sqrt(bcom**2 * d * ceil(params.e_com / d) + d * (bm1 - 1))

    bpk = params.tfhe["lwe_modular_std_dev"] * Q * sqrt(e * params.log_q)
    b = params.tfhe["glwe_modular_std_dev"] * Q * sqrt(2 * d)
    bp = b
    bpp = params.tfhe["lwe_modular_std_dev"] * Q * sqrt(eta * d)
    inside_se = (
        (m1 - 1) * d
        + 2 * bcom**2
        + bpk**2
        + b**2
        + e * ell * bp**2
        + tl * bpp**2
        + 2 * log(bcom, 2)
        + log(bpk, 2)
        + log(b, 2)
        + e * ell * log(bp, 2)
        + tl * log(bpp, 2)
    )
    se = gammae * sqrt(337) * sqrt(inside_se)

    for n in range(1, 10):
        m2 = 2 + n + lamb + 4
        s2 = gamma2 * 59 * sqrt(d * m2)

        # see if n large enough for the SIS problem
        beta = 8 * 59 * sqrt(2 * d * bm1 * s1**2 + 2 * d * m2 * s2**2)
        if beta > 2 ** (2 * sqrt(n * d * params.log_q * log(delta, 2))):
```

```python
                continue

            lwe_estimator_dim = (m2 - n - lamb - 4) * d
            print(f"Check that ternary LWE in dimension {lwe_estimator_dim} is hard.")

            tot = (
                (n + 2 + lamb + 2) * d * params.log_q
                + log(2 * kap + 1, 2) * d
                + bm1 * d * log(s1 * 2**2.57, 2)
                + m2 * d * log(s2 * 2**2.57, 2)
                + 256 * log(se * 2**2.57, 2)
            )

            return _mb(tot)


def voprf_pk_proof_old_sizemb(params, gamma1=10, gamma2=5, gammae=5):
    """
    Well-formedness proof of the OPRF public key using [LNP22].

    :param params:
    :param gamma1:
    :param gamma2:
    :param gammae:

    """
    ell = params.tfhe["pbs_level"]
    eta = 1
    tl = params.tfhe["ks_level"]
    d = params.tfhe["polynomial_size"] * params.tfhe["glwe_dimension"]
    e = params.tfhe["lwe_dimension"]
    ecom = params.e_com
    Q = 2**params.log_q
    lamb = 128 // params.log_q
    kap = 2
    delta = 1.0045

    for n in range(1, 10):
        m1 = (ecom + 3 * e) // d + 2 * eta + 8 * e * ell
        m2 = 3 + n + tl + lamb + 4
        ve = 5 + 4 * e * ell + 2 * tl
        bm1 = m1 + ve
        bcom = params.sigma_com * sqrt(ecom)
        s1 = gamma1 * 59 * sqrt(bcom**2 * d * ceil(ecom / d) + d * (bm1 - 1))
        s2 = gamma2 * 59 * sqrt(d * m2)

        # see if n large enough for the sis problem
        beta = 8 * 59 * sqrt(2 * d * bm1 * s1**2 + 2 * d * m2 * s2**2)
        if beta > 2 ** (2 * sqrt(n * d * log(Q, 2) * log(delta, 2))):
            continue

        bpk = params.tfhe["lwe_modular_std_dev"] * Q * sqrt(e * params.log_q)
        b = params.tfhe["glwe_modular_std_dev"] * Q * sqrt(2 * d)
        bp = b
        bpp = params.tfhe["lwe_modular_std_dev"] * Q * sqrt(eta * d)
        inside_se = (
            (m1 - 1) * d
            + 2 * bcom**2
            + bpk**2
            + 2 * b**2
            + 2 * e * ell * bp**2
            + 2 * tl * bpp**2
            + 2 * log(bcom, 2)
            + log(bpk, 2)
            + 2 * log(b, 2)
            + 2 * e * ell * log(bp, 2)
            + 2 * tl * log(bpp, 2)
        )
        se = gammae * sqrt(337) * sqrt(inside_se)

        tot = (
            (n + 2 + lamb + 2) * d * params.log_q
            + log(2 * kap + 1, 2) * d
            + bm1 * d * log(s1 * 2**2.57, 2)
            + m2 * d * log(s2 * 2**2.57, 2)
            + 256 * log(se * 2**2.57, 2)
        )

        return _mb(tot)


@cached_function
def oprf_pk_proof_sizekb(params, compress_ct1=True):
    """
    Well-formedness proof of the OPRF public key using LaBRADOR.
    """
    e = params.tfhe["lwe_dimension"]
    eta = 1  # we will absorb the module rank in the ring dimension
    d = params.tfhe["polynomial_size"] * params.tfhe["glwe_dimension"]
    sig = params.tfhe["lwe_modular_std_dev"]
    sigring = params.tfhe["glwe_modular_std_dev"]
    ell = params.tfhe["pbs_level"]
    tl = params.tfhe["ks_level"]

    if not compress_ct1:
        zeta = eta
    else:
        zeta = eta + log(d, 2)

    witness_dim_original = (
        ceil((e + 2 * params.e_com + e * params.log_q) / d) + 3 + 8 * e * ell + zeta * tl
    )
    witness_dim = witness_dim_original * (d // 64)
    norm_bound = sqrt(
        e
        + eta * d
        + 4 * e * ell * d
        + params.sigma_com**2 * (2 * params.e_com)
        + sig**2 * (e * params.log_q + zeta * tl * d)
        + sigring**2 * (2 * d + 4 * e * ell * d)
    )

    print(f"LaBRADOR witness dimension: {witness_dim}")

    r = 24
    n = ceil(witness_dim / 24)

    L = LaBRADOR(d=64, logq=32)
    size, params = L(n, r, norm_bound, 3, 5, verbose=False)
    return size, params


@cached_function
def voprf_pk_proof_sizekb(params, compress_ct1=True):
    """
    Well-formedness proof of the VOPRF public key using LaBRADOR.
    """
    e = params.tfhe["lwe_dimension"]
    eta = 1
    d = params.tfhe["polynomial_size"] * params.tfhe["glwe_dimension"]
    sig = params.tfhe["lwe_modular_std_dev"]
    sigring = params.tfhe["glwe_modular_std_dev"]
    ell = params.tfhe["pbs_level"]
    tl = params.tfhe["ks_level"]

    if not compress_ct1:
        zeta = eta
    else:
        zeta = eta + log(d, 2)

    witness_dim_original = (
        ceil((3 * e + 2 * params.e_com + e * params.log_q) / d)
        + 4
        + 2 * eta
        + 16 * e * ell
        + 2 * zeta * tl
    )
    # need to expand this dimension to reach ring dimension 64
    witness_dim = witness_dim_original * ceil(d // 64)
    norm_bound = sqrt(
        3 * e
        + 2 * eta * d
        + 8 * e * ell * d
        + params.sigma_com**2 * (2 * params.e_com)
        + sig**2 * (e * params.log_q + 2 * tl * zeta * d)
        + sigring**2 * (4 * d + 8 * e * ell * d)
    )

    r = 256
    n = ceil(witness_dim / 256)
    L = LaBRADOR(d=64, logq=params.log_q)

    return L(n, r, norm_bound, 5, 5, verbose=False)


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

    keep_bits = ceil(-log(params.tfhe["lwe_modular_std_dev"], 2.0) + 1)

    return _kb(params.n_p * keep_bits + 256)


voprf_ct0_sizekb = oprf_ct0_sizekb


@cached_function
def oprf_ct0_proof_amortised_all_sizekb(
    params,
    gamma1=10,
    gamma2=5,
    compress=True,
    D=12,
    gamma=947 * 281 * 2,
    L=64,
    voprf=False,
):
    """
    Well-formedness proof of OPRF request using [LNP22].

    :param params:
    :param gamma1:
    :param gamma2:
    :param compress: Use Dilithium-style compression
    :param D:
    :param gamma:
    :param L: Amortise over this many proofs.
    :param voprf: Estimate VOPRF case.
    """
    d = 128
    lamb = ceil(128 / params.log_q)
    kappa = 2
```

```python
e = params.tfhe["lwe_dimension"]
bcom = params.sigma_com * sqrt(params.e_com)
Bv = 128
Bct = 2**params.log_q * params.tfhe["lwe_modular_std_dev"] * sqrt(params.n_p)
if voprf:
    m1 = ceil(params.e_com / d) + ceil(e / d) + 4 * L * ceil(params.n_p / d)
else:
    m1 = ceil(params.e_com / d) + ceil(e / d) + L * ceil(params.n_p / d)
ve = 2 + L
bm1 = m1 + ve - ceil(params.e_com / d)
if voprf:
    mlwe_dim = 20
else:
    mlwe_dim = 11
delta = 1.0045

gammad = gamma2
gammae = gamma2
if voprf:
    sd = gammad * sqrt(337) * sqrt(L * Bv**2 + L * params.n_p)
else:
    sd = gammad * sqrt(337) * sqrt(L * Bv**2)
if voprf:
    voprf_expand = 3
else:
    voprf_expand = 1
se = (
    gammae
    * sqrt(337)
    * sqrt(
        e
        + voprf_expand * L * params.n_p
        + 2 * bcom**2
        + L * Bct**2
        + 2 * log(bcom, 2)
        + L * log(Bct, 2)
    )
)
s1 = gamma1 * 59 * sqrt(bcom**2 + d * bm1)

print(
    f"Check that ternary MLWE in dimension {mlwe_dim} x {d} mod 2^{params.log_q} is hard."
)

for n in range(1, 32):
    m2 = mlwe_dim + n + lamb + 4
    s2 = gamma2 * 59 * sqrt(d * m2)

    beta = 8 * 59 * sqrt(2 * d * (m1 + ve) * s1**2 + 2 * d * m2 * s2**2)

    # Is n picked so that the sis problem is hard?
    if beta > 2 ** (2 * sqrt(n * d * params.log_q * log(delta, 2))):
        continue

    tot = (
        (n + 2 * ceil(256 / d) + lamb + 2) * d * params.log_q
        + log(2 * kappa + 1, 2) * d
        + (m1 + ve) * d * log(s1 * 2**2.57, 2)
        + m2 * d * log(s2 * 2**2.57, 2)
        + 256 * (log(se * 2.57, 2) + log(sd * 2.57, 2))
    )

    if compress:
        return round(
            _kb(tot)
            - dilithium_sistest(
                D, gamma, params, m2, n, gamma1, gamma2, L, voprf=voprf
            ),
                1,
            )
        else:
            return _kb(tot)


oprf_ct0_proof_sizekb = partial(
    oprf_ct0_proof_amortised_all_sizekb,
    gamma1=10,
    gamma2=5,
    compress=False,
    D=11,
    gamma=957 * 281,
    L=1,
)

voprf_ct0_proof_amortised_all_sizekb = partial(
    oprf_ct0_proof_amortised_all_sizekb, voprf=True
)

voprf_ct0_proof_sizekb = partial(
    voprf_ct0_proof_amortised_all_sizekb,
    gamma1=10,
    gamma2=5,
    D=11,
    gamma=4051 * 41 * 2,
    L=1,
)


def dilithium_sistest(
    D, gamma, params, m2=32, n=13, gamma1=10, gamma2=5, L=1, voprf=False
):
    d = 128
    e = params.tfhe["lwe_dimension"]
    bcom = params.sigma_com * sqrt(params.e_com)

    expansion = 1 if not voprf else 4

    m1 = ceil(params.e_com / d) + ceil(e / d) + L * expansion * ceil(params.n_p / d)
    ve = 2 + L
    bm1 = m1 + ve - ceil(params.e_com / d)
    s1 = gamma1 * 59 * sqrt(bcom**2 + d * bm1)
    s2 = gamma2 * 59 * sqrt(d * m2)
    kappa = 2

    delta = 1.0045
    b1 = 2 * s1 * sqrt(2 * (m1 + ve) * d)
    b2 = 2 * s2 * sqrt(2 * m2 * d) + 2**D * 59 * sqrt(n * d) + gamma * sqrt(n * d)
    betap = 4 * 59 * sqrt(b1**2 + b2**2)

    # make sure the sis problem is still hard and the compression params make sense
    if betap >= 2 ** (2 * sqrt(n * d * params.log_q * log(delta, 2))):
        raise ValueError(f"betap {betap} too large.")
    if 2 ** (D - 1) * kappa * d >= gamma:
        raise ValueError(f"gamma {gamma} too small.")

    return _kb((D - 2.25) * n * d)


def oprf_ct0_proof_amortised_sizekb(params, L=64):
    """
    Well-formedness proof of OPRF request per request using [LNP22].
    """
    return round(oprf_ct0_proof_amortised_all_sizekb(params, L=L) / L, 1)


def voprf_ct0_proof_amortised_sizekb(params, L=64):
    """
    Well-formedness proof of VOPRF request per request using [LNP22].
    """
    return round(voprf_ct0_proof_amortised_all_sizekb(params, L=L) / L, 1)


def oprf_ct1_sizekb(params, compress_ct1=True):
    """
    Response size in kilobytes.
    """
    # TODO make number of bits we keep dependent on params
    if compress_ct1 is False:
        e = params.tfhe["lwe_dimension"]
        return _kb(params.m * e * 24 + params.m * 16)
    else:
        ell = params.tfhe["glwe_dimension"]
        d = params.tfhe["polynomial_size"]
        return _kb(d * ell * 24 + params.m * 16)


voprf_ct1_sizekb = oprf_ct1_sizekb


def oprf_online_sizekb(params, compress_ct1=True, amortise=True):
    r = oprf_ct0_sizekb(params)
    if amortise:
        r += oprf_ct0_proof_amortised_sizekb(params)
    else:
        r += oprf_ct0_proof_sizekb(params)
    r += oprf_ct1_sizekb(params, compress_ct1=compress_ct1)
    return round(r, 1)


def voprf_online_sizekb(params, amortise=True, compress_ct1=True):
    r = voprf_ct0_sizekb(params)
    if amortise:
        r += voprf_ct0_proof_amortised_sizekb(params)
    else:
        r += voprf_ct0_proof_sizekb(params)
    r += voprf_ct1_sizekb(params, compress_ct1=compress_ct1)
    return round(r, 1)


def oprf(params):
    consistency_check(params)

    print(
        f"""
  % OPRF SIZES
  /oprf/pk/sizemb/.initial={oprf_pk_sizemb(params)},
  /oprf/pk/proof-old/sizemb/.initial={oprf_pk_proof_old_sizemb(params)},
  /oprf/pk/proof/sizekb/.initial={oprf_pk_proof_sizekb(params)[0]},
  /oprf/pk/proof/params/.initial={{{oprf_pk_proof_sizekb(params)[1]}}},
  /oprf/ct0/sizekb/.initial={oprf_ct0_sizekb(params)},
  /oprf/ct0/proof/sizekb/.initial={oprf_ct0_proof_sizekb(params)},
  /oprf/ct0/proof/amortised-all/sizekb/.initial={oprf_ct0_proof_amortised_all_sizekb(params)},
  /oprf/ct0/proof/amortised/sizekb/.initial={oprf_ct0_proof_amortised_sizekb(params)},
  /oprf/ct1/sizekb/.initial={oprf_ct1_sizekb(params, compress_ct1=False)},
  /oprf/ct1-compressed/sizekb/.initial={oprf_ct1_sizekb(params)},
  /oprf/online/amortised/sizekb/.initial={oprf_online_sizekb(params)},
  /oprf/online/sizekb/.initial={oprf_online_sizekb(params, amortise=False)},
    """
    )


def voprf(params):
    consistency_check(params)
    print(
        f"""
  % VOPRF SIZES
  /voprf/pk/sizemb/.initial={voprf_pk_sizemb(params)},
  /voprf/pk/proof-old/sizemb/.initial={voprf_pk_proof_old_sizemb(params)},
  /voprf/pk/proof/sizekb/.initial={voprf_pk_proof_sizekb(params)[0]},
  /voprf/pk/proof/params/.initial={{{voprf_pk_proof_sizekb(params)[1]}}},
  /voprf/ct0/sizekb/.initial={voprf_ct0_sizekb(params)},
  /voprf/ct0/proof/sizekb/.initial={voprf_ct0_proof_sizekb(params)},
  /voprf/ct0/proof/amortised-all/sizekb/.initial={voprf_ct0_proof_amortised_all_sizekb(params)},
  /voprf/ct0/proof/amortised/sizekb/.initial={voprf_ct0_proof_amortised_sizekb(params)},
  /voprf/ct1/sizekb/.initial={voprf_ct1_sizekb(params, compress_ct1=False)},
  /voprf/ct1-compressed/sizekb/.initial={voprf_ct1_sizekb(params)},
  /voprf/online/amortised/sizekb/.initial={voprf_online_sizekb(params)},
  /voprf/online/sizekb/.initial={voprf_online_sizekb(params, amortise=False)},
    """
    )


def both():
    oprf(OPRF)
    voprf(VOPRF)
```

```
$F_{\mathsf{vpoprf}}.\mathsf{Setup}(1^\lambda)$
    $\boldsymbol{A}_{\mathsf{pp}} \leftarrow\$ \mathbb{Z}_Q^{N\times N}$
    $\mathsf{pp} \leftarrow \boldsymbol{A}_{\mathsf{pp}}$
    **return** $\mathsf{pp}$

$F_{\mathsf{vpoprf}}.\mathsf{KeyGen}(1^\lambda)$
    $\mathsf{pk} \leftarrow \perp$
    $\mathsf{sk} \leftarrow\$ \mathbb{Z}_p^{m_p\times n_p}$
    $\boldsymbol{x}_i \leftarrow X$ for $i \in \mathbb{Z}_\beta$
    $\boldsymbol{z} \leftarrow F_{\mathsf{vpoprf}}.\mathsf{Eval}(\mathsf{sk}, t = \boldsymbol{t}, x = \boldsymbol{x})$
    $\pi \leftarrow \mathsf{NIZKAoK}_{Rt}(\boldsymbol{A}; \boldsymbol{t}, \boldsymbol{z}, \boldsymbol{x})$
    **return** $(\mathsf{pk}, sk)$

$F_{\mathsf{vpoprf}}.\mathsf{Eval}(\mathsf{sk} = \boldsymbol{A}, \boldsymbol{t} = \perp, x = \boldsymbol{x})$
    $\boldsymbol{z}' := F_{\boldsymbol{A}}(\boldsymbol{t}, \boldsymbol{x})$
    $\boldsymbol{z} := \mathsf{RO}_{\mathsf{fin}}(\boldsymbol{t}, \boldsymbol{x}, \boldsymbol{z}')$
    **return** $\boldsymbol{z}$

$F_{\mathsf{vpoprf}}.\mathsf{Request}(\mathsf{pk}, \boldsymbol{t} = \perp, x = \boldsymbol{x}; pp)$
    Parse $\mathsf{pp}$ as $\boldsymbol{A}_{\mathsf{pp}}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{z}}, \hat{\pi}$
    **if** $\hat{\pi}$ does not verify, **abort**
    $\mathsf{FHE.pk}^{(\mathsf{pp})}, \mathsf{FHE.sk} \leftarrow\$ \mathsf{FHE.KeyGen}^{(\mathsf{pp})}()$
    $\rho \leftarrow S_m$
    $\boldsymbol{x}' \leftarrow (\boldsymbol{x}, \hat{\boldsymbol{x}})$
    $\boldsymbol{x}' \leftarrow \boldsymbol{x}'_{\rho(i)}$ for $i \in \mathbb{Z}_\gamma$
    $\boldsymbol{y} \leftarrow \mathsf{decomp}(\boldsymbol{G}_{\mathsf{inp}} \cdot \boldsymbol{x}' \bmod p)$
    $\mathsf{ct} \leftarrow\$ \mathsf{FHE.Enc}(\mathsf{FHE.sk}, \boldsymbol{y})$
    $\pi \leftarrow\$ \mathsf{NIZKAoK}_{\mathsf{C}}(\mathsf{FHE.pk}^{(\mathsf{pp})}, \mathsf{ct}; \mathsf{FHE.sk}, x)$
    $req \leftarrow (\mathsf{FHE.pk}^{(\mathsf{pp})}, \mathsf{ct}, \pi, t)$
    **return** $req, \rho$

$F_{\mathsf{vpoprf}}.\mathsf{BlindEval}(sk = \boldsymbol{A}, \boldsymbol{t} = \perp, req; \mathsf{pp})$
    $(\mathsf{FHE.pk}^{(\mathsf{pp})}, \mathsf{ct}, \pi) \leftarrow req$
    $\mathsf{ct}' \leftarrow F_{\mathsf{vpoprf}}.\mathsf{HEEval}(\mathsf{FHE.pk}^{(\mathsf{pp})}, \boldsymbol{A}, \boldsymbol{t}, \mathsf{ct})$
    **if** $\pi$ does not verify **then** $\mathsf{ct}' = \perp$
    $rep \leftarrow \mathsf{ct}'$
    **return** $rep$

$F_{\mathsf{vpoprf}}.\mathsf{Finalise}(\mathsf{FHE.sk}, rep = \mathsf{ct}, \rho)$
    **if** $\mathsf{ct}'$ not a ctxt **then return** $\perp$
    $\boldsymbol{z}^\star \leftarrow \mathsf{FHE.Dec}(\mathsf{FHE.sk}, \mathsf{ct})$
    **if** $F_{\mathsf{vpoprf}}.\mathsf{Verify}(\boldsymbol{z}, \boldsymbol{z}^\star, \rho) = 0,$ **then return** $\perp$
    $\boldsymbol{z}'_i \leftarrow \boldsymbol{z}^\star_{\rho(i)}$ for $i \in \mathbb{Z}_\alpha$
    $\boldsymbol{z} \leftarrow \mathsf{RO}_{\mathsf{fin}}(\boldsymbol{t}, \boldsymbol{x}, \boldsymbol{z}')$
    **return** $\boldsymbol{z}$

$F_{\mathsf{vpoprf}}.\mathsf{Verify}(\boldsymbol{z}, \boldsymbol{z}^\star, \rho)$
    **if** $\boldsymbol{z}^{(\beta)}_{j,\rho(k+\alpha)} \neq \boldsymbol{z}^\star_{\rho(k+\alpha)}$ for any $k \in \mathbb{Z}_\beta$, **return** 0
    **if** $\boldsymbol{z}^{(\alpha)}_{\rho(i_0+r_\alpha \ell)} \neq \boldsymbol{z}^{(\alpha)}_{\rho(i_1+r_\alpha \ell)}$ for $i_0 \neq i_1 \in \mathbb{Z}_{r_\alpha}$, **return** 0
    **if** $\boldsymbol{z}^{(\alpha)}_{\rho(i_0+r_\alpha \ell_0)} = \boldsymbol{z}^{(\alpha)}_{\rho(i_1+r_\alpha \ell_1)}$ for $i_0, i_1 \in \mathbb{Z}_{r_\alpha}$ and $\ell_0 \neq \ell_1 \in \mathbb{Z}_\alpha$, **return** 0
    **return** 1
```

Client		**Server**
$F_{\mathsf{vpoprf}}.\mathsf{Request}(\mathsf{pk}, t = \perp, x = \boldsymbol{x}; \mathsf{pp})$		
	$\xrightarrow{req = (\mathsf{FHE.pk}^{(\mathsf{pp})}, \mathsf{ct}, \pi, t)}$	$F_{\mathsf{vpoprf}}.\mathsf{BlindEval}(sk = \boldsymbol{A}, t = \perp, req)$
$F_{\mathsf{vpoprf}}.\mathsf{Finalise}(rep, \rho)$	$\xleftarrow{rep = \mathsf{ct}'}$	
Output: $\mathsf{RO}_{\mathsf{fin}}(\boldsymbol{t}, \boldsymbol{x}, F_{\boldsymbol{A}}^{\mathsf{RO}_{\mathsf{key}}}(\boldsymbol{t}, \boldsymbol{x}))$		

Fig. 5. Our construction for VPOPRF, F_{vpoprf}.