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 preliminary benchmarks of our construction. For the core online OPRF functionality, we require amortised 10.0KB communication per evaluation and a one-time per-client setup communication of 2.5MB.

Keywords: oblivious PRF, lattices, FHE

1 Introduction

Oblivious pseudorandom functions allow two parties to compute a pseudorandom function (PRF) $z \coloneqq F_k(x)$ 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 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 Diffie-Hellman (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 we observe that 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 a single level of bootstrapping only. 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 (T)FHE-friendly – we encourage further cryptanalysis of it.

After some preliminaries in Section 2 we specify our POPRF candidate in Section 3 based on programmable bootstrapping for plaintext modulus switching. In particular, we define an operation called $\mathsf{CPPBS}_{(2,3)}$ which uses a programmable bootstrap with a special negacyclic "test polynomial" and a simple linear function to "correct" and realise the desired modulus switch. To our knowledge this functionality has not been explicitly defined and used in prior work. Without the bespoke design of $\mathsf{CPPBS}_{(2,3)}$, we are forced to either use only half of the plaintext space, or use a sequence of two programmable bootstraps [LMP22]. The former drawback prevents simple bootstrapping-less homomorphic addition modulo P_1, whereas the latter does not permit a depth one bootstrapping construction.

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]. In particular, 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 circuit-privacy obtained from TFHE bootstrapping [Klu22] and a 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

[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.

the problem of rewinding for post-quantum security, cf. [Unr12]. POPRIV1 security for the client against a semi-honest server relies on the IND-CPA security of TFHE.

Targeting roughly 100 bits of security, we obtain the following performance. While the public key material sent by the client to the server is large (14.7MB) this cost can be amortised by reusing the same material for several evaluations. Individual PRF evaluations can then cost about 48.9KB or as little as 5.3KB when amortising client NIZK proofs across several OPRF queries. Applying the public-key compression technique of [KLD$^+$23], we obtain 2.5MB of public-key material at the cost of increasing the amortised cost to 10.0KB (see Table 1).

Initially, we focus on oblivious rather than verifiable oblivious PRFs. This is motivated by the presumed high cost associated with zero-knowledge proofs 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 our 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 one. While we were unable to find such a cheating circuit, and conjecture that none exists, we stress that this part of our work is highly speculative. Note that the assumption here depends on the bootstrapping depth of the OPRF, i.e. if depth $d > 1$ was required for OPRF evaluation, the assumption would need to be that there is no cheating circuit in depth d. Therefore, our depth one construction leads to an "optimal" assumption for the cut-and-choose method. Under the heuristic assumption that our construction is verifiable, in Section 5.3 we then establish 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 and OPRFs to VOPRFs.

We present a proof-of-concept SageMath implementation and some indicative Rust benchmarks in Section 6. Our SageMath implementation covers all building blocks except for the zero-knowledge proofs, which we consider out of scope. In particular, we re-implemented TFHE [CGGI20], including circuit privacy [Klu22] and ciphertext and public-key compression [CDKS21,KLD$^+$23]. Our Rust benchmarks make use of Zama's `tfhe-rs` library for implementing TFHE, which – however – does not implement many of the building blocks we make use of[6] and thus mostly serves as an initial, best-case performance eval-

[6] These are: plaintext moduli that are not powers of two, circuit privacy, ciphertext and bootstrapping-key compression.

uation. In particular, we expect circuit privacy and public-key compression to increase the runtime by a factor of, say, ten (we discuss this Section 6). We also did not implement the NIZKs in Rust. With these caveats in mind, the client online functions we could implement run in 28.9ms on one core and server online functions run in 151ms on 64 cores.

In Appendix A we then estimate costs of the required non-interactive zero-knowledge proofs. We use a combination of [LNP22] and [BS22] to show that the cost of the proofs does not add significant overhead to the communication of our protocol.

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. The first 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, both of which 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]. In [FOE23] a generic MPC solution not relying on novel assumptions is proposed, that, while not round-optimal, reportedly provides good performance in various settings.

We give a summary comparison of our construction with prior work in Table 1. The only 2-round constructions without preprocessing or trusted-setup in Table 1 are those from [ADDS21], where our construction compares favourably by offering stronger claimed qualitative security at smaller size, albeit under novel assumptions. In particular, even in a semi-honest setting, our construction outperforms that from [ADDS21] in terms of bandwidth for $L \geq 2$ queries.[7]

Table 1. Post-quantum (P)(V)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*
[FOE23]	AES	?	4746KB	plain	malicious client, *pp*
Section 3	lattices, [BIP+18]	2	14.7MB + 90.7KB + 0.9KB + 44.8KB + 3.2KB	plain	malicious client, ROM
Section 3	lattices, [BIP+18]	2	14.7MB + 90.7KB + 0.9KB + 1.2KB + 3.2KB	plain	malicious client, ROM $L = 64$, per query
Section 3, CBR	lattices, [BIP+18]	2	2.4MB + 137.4KB + 2.0KB + 63.0KB + 6.2KB	plain	malicious client, ROM
Section 3, CBR	lattices, [BIP+18]	2	2.4MB + 137.4KB + 2.0KB + 1.8KB + 6.2KB	plain	malicious client, ROM $L = 64$, per query
[ADDS21]	R(LWE) & SIS	2	$>$ 128GB	verifiable	malicious, QROM
[Bas23]	SIDH	2	8.7MB	verifiable	malicious, *ts*, ROM
Section 5	heuristic	2	256KB + 14.7MB + 90.7KB + 11.1$\cdot$ 0.9KB + 11.1$\cdot$ 44.8KB + 11.1$\cdot$ 3.2KB	verifiable	malicious, ROM
Section 5	heuristic	2	256KB + 14.7MB + 90.7KB + 11.1$\cdot$ 0.9KB + 11.1$\cdot$ 1.2KB + 11.1$\cdot$ 3.2KB	verifiable	malicious, ROM $L = 64$, per query
Section 5, CBR	heuristic	2	256KB + 2.4MB + 137.4KB + 11.1$\cdot$ 2.0KB + 11.1$\cdot$ 63.0KB + 11.1$\cdot$ 6.2KB	verifiable	malicious, ROM
Section 5, CBR	heuristic	2	256KB + 2.4MB + 137.4KB + 11.1$\cdot$ 2.0KB + 11.1$\cdot$ 1.8KB + 11.1$\cdot$ 6.2KB	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, "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. 79.3KB/64 = 1.2KB per query, when amortising over $L = 64$ queries. The factor of 11.1 accounts for the "check point" evaluations, cf. Section 5. The rows marked as "CBR" apply public-key compression [KLD+23]. We picked parameters targeting roughly 100 bits of security for our constructions. See Table 3 for 128 bits of security.

1.3 Open Problems

A 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.

A key bottleneck for implementations will be bootstrapping, an effect that will exacerbated by the need for circuit-private bootstrapping. It is an open problem to establish if this somewhat heavy machinery is required given that we

[7] 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.

are only aiming to hide the secret key $\boldsymbol{A}$ and that we can randomise our circuit by randomly flipping signs in the additions induced by $\boldsymbol{A}$.

Considering Table 1, we note that many candidates forgo round-optimality to achieve acceptable performance. It is an interesting open question how critical this requirement is for various applications, since dropping it seems to enable significantly more efficient post-quantum instantiations of (V)OPRFs.

Our verifiability approach throws up a range of interesting avenues to explore for VOPRFs but also for verifiable homomorphic computation, more generally. 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 to reason 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, our VOPRF is even more speculative than our OPRF candidate. 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 $\{-p/2, \dots, (p/2)-1\}$ if p is even and $\{-\lfloor p/2\rfloor, \dots, \lfloor p/2\rfloor\}$ if p is odd, but we will also consider $\mathbb{Z}_p$ as $\{0, 1, \dots, p-1\}$. 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 key space $\mathcal{K}$, 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 (POPRF) 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.

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].

Game $\mathrm{POPRF}^{\mathcal{A},b}_{\mathcal{F},\mathsf{S},\mathsf{RO}}(\lambda)$	Oracle $\mathrm{Eval}(t,x)$	Oracle $\mathrm{BlindEval}(t, req)$
$q_{s,t}, q_t \leftarrow 0, 0$	$z_0 \leftarrow \mathsf{RO}_{\mathsf{Fn}}.\mathsf{Eval}((t,x), st_{\mathsf{Fn}})$	$q_t \leftarrow q_t + 1$
$st_{\mathsf{Fn}} \leftarrow\!\!\$\ \mathsf{RO}_{\mathsf{Fn}}.\mathsf{Init}(1^\lambda)$ // $\mathcal{F}.\mathsf{In} \to \mathcal{F}.\mathsf{Out}$	$z_1 \leftarrow \mathcal{F}.\mathsf{Eval}^{\mathsf{RO}}_{\mathsf{pp}_1}(sk, t, x)$	$(rep_0, st_{\mathsf{S}}) \leftarrow\!\!\$\ \mathsf{S}.\mathsf{BlindEval}^{\mathrm{LimitEval}}(t, req, st_S)$
$st_{\mathsf{RO}} \leftarrow\!\!\$\ \mathsf{RO}.\mathsf{Init}(1^\lambda)$	**return** z_b	$rep_1 \leftarrow\!\!\$\ \mathcal{F}.\mathsf{BlindEval}^{\mathsf{RO}}_{\mathsf{pp}_1}(sk, t, req)$
$\mathsf{pp}_1 \leftarrow\!\!\$\ \mathcal{F}.\mathsf{Setup}(1^\lambda)$		**return** rep_b
$(st_{\mathsf{S}}, \mathsf{pk}_0, \mathsf{pp}_0) \leftarrow\!\!\$\ \mathsf{S}.\mathsf{Init}(\mathsf{pp}_1)$	Oracle $\mathrm{LimitEval}(t,x)$	
$(sk, \mathsf{pk}_1) \leftarrow\!\!\$\ \mathcal{F}.\mathsf{KeyGen}^{\mathsf{RO}}_{\mathsf{pp}_1}(1^\lambda)$	$q_{t,s} \leftarrow q_{t,s} + 1$	Oracle $\mathrm{Prim}(x)$
$b' \leftarrow\!\!\$\ \mathcal{A}^{\mathrm{Eval},\mathrm{BlindEval},\mathrm{Prim}}(\mathsf{pp}_b, \mathsf{pk}_b)$	**if** $q_{t,s} \leq q_t$ **then**	$(h_0, st_{\mathsf{S}}) \leftarrow\!\!\$\ \mathsf{S}.\mathsf{Eval}^{\mathrm{LimitEval}}(x, st_S)$
return b'	**return** $\mathrm{Eval}(t,x)$	$h_1 \leftarrow\!\!\$\ \mathsf{RO}.\mathsf{Eval}(x, st_{\mathsf{RO}})$
	return $\perp$	**return** h_b

Fig. 1. Pseudorandomness against malicious clients.

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 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 a truly random function) the simulator may only use a limited number of random function queries to simulate the random oracle accessed via $\mathrm{Prim}(x)$.

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 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 BlindEval query and tag t. This implies that BlindEval and Eval queries essentially leak nothing beyond a single evaluation to the client. Moreover, the simulator is restricted in that the 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:*

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

Game $\text{POPRIV1}_{\mathcal{F},\mathsf{RO}}^{\mathcal{A},b}(\lambda)$

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

Oracle $\text{Run}(t, x_0, x_1)$

for $j \in \{0,1\}$ **do**
  $(st_j, req_j) \leftarrow\!\!\$\ \mathcal{F}.\mathsf{Request}_{\mathsf{pp}}^{\mathsf{RO}}(\mathsf{pk}, t, x_j)$
  $rep_j \leftarrow\!\!\$\ \mathcal{F}.\mathsf{BlindEval}_{\mathsf{pp}}^{\mathsf{RO}}(sk, t, req_j)$
  $z_j \leftarrow\!\!\$\ \mathcal{F}.\mathsf{Finalise}_{\mathsf{pp}}^{\mathsf{RO}}(rep_j, st_j)$
$\tau_0 \leftarrow (req_b, rep_b, z_0)$
$\tau_1 \leftarrow (req_{1-b}, rep_{1-b}, z_1)$
return (τ_0, τ_1)

Game $\text{POPRIV2}_{\mathcal{F},\mathsf{RO}}^{\mathcal{A},b}(\lambda)$

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

Oracle $\text{Request}(\mathsf{pk}, t, x_0, x_1)$

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

Oracle $\text{Finalise}(j, rep_0, rep_1)$

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

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 integers 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 $M\text{-}\mathsf{SIS}_{\mathcal{R}_q,n,\ell,\beta^*}$, 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 $M\text{-}\mathsf{LWE}_{\mathcal{R}_q,n,\chi_s,\chi_e}$ 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 $M\text{-}\mathsf{LWE}_{\mathcal{R}_q,n,\chi_s,\chi_e}$.*

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*

$$\boldsymbol{F} \leftarrow\$ \{0, \pm 1, \dots, \pm\beta\}^{n\times n}, \boldsymbol{G} \leftarrow\$ \{0, \pm 1, \dots, \pm\beta\}^{n\times n} \cap \left(\mathbb{Z}_q^{n\times n}\right)^*$$

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

$$\boldsymbol{F} \leftarrow\$ \{0, \pm 1, \dots, \pm\beta\}^{n\times n}, \boldsymbol{G} \leftarrow\$ \{0, \pm 1, \dots, \pm\beta\}^{n\times n} \cap \left(\mathbb{Z}_q^{n\times n}\right)^*$$

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 $\boldsymbol{m} \in \mathbb{Z}_2^n$, $(\|\boldsymbol{s}\|_\infty \leq \beta'_s \vee \|\boldsymbol{s}\|_2 \leq \beta'_s \cdot \sqrt{n})$ and $(\|\boldsymbol{e}\|_\infty \leq \beta'_e \vee \|\boldsymbol{e}\|_2 \leq \beta'_e \cdot \sqrt{n})$, 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 $\mathsf{NTRUDec}(\boldsymbol{c}, \tau \coloneqq (\boldsymbol{F}, \boldsymbol{G}, p))$ 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.[8]

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. $\mathcal{M} \coloneqq \mathbb{Z}_P$. 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 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 third of these works known as TFHE [CGGI20] because its programmable bootstrapping technique lends enables our construction. For a 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.[9] We also note that the FHE schemes considered in this work are additively homomorphic with

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

[9] This allows to restrict the number of sequential bootstrappings that can be performed

very modest noise growth, meaning that many additions of plaintexts can be performed *without* bootstrapping.

In TFHE we have access to *negacyclic* look-up tables from $\mathbb{Z}_{2d}$ to $\mathbb{Z}_P$ which we will denote by $f(\cdot) : \mathbb{Z}_{2d} \to \mathcal{M}$. 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, but this is overcome by approximating $\mathbb{Z}_{2d}$ as $\mathbb{Z}_P$ [CGGI20,Joy21]. TFHE generalises bootstrapping by applying the look-up table to the plaintext at the same time as resetting the size of the noise. The negacyclic property dictates that $f(x+d) = -f(x)$ for $x = 0, 1, \ldots, d-1$, so if the desired look-up table is not negacyclic, one must either restrict to using half the plaintext space or use more complex bootstrapping techniques [LMP22].

2.6 Circuit Private (Programmable) Bootstrapping

Circuit private FHE hides the computation performed on a ciphertext. There are generic methods of achieving circuit privacy [DS16] and more specific ones for GSW [BdMW16] and TFHE [Klu22]. As our OPRF is designed within the specific framework of TFHE, we restrict discussion to the latter work. At a high level, TFHE bootstrapping consists of two steps: blind rotation and key-switching. Blind Rotation is a Generalised GSW [GSW13] based operation that outputs an LWE ciphertext under a different key. The key-switching phase then maps back to the original key to allow for further homomorphic operations. A key contribution of Kluczniak [Klu22] is a generalised Gaussian leftover hash lemma that shows how to randomise the blind rotation phase in order to "clean up" the noise distribution. Ultimately, this entails adding Gaussian preimage sampling to the blind rotation algorithm which affects computation time and correctness parameters. We note that we will not require key-switching (line 1 of Figure 3 in [Klu22]), but this does not affect the statistical distance result below in any way. We avoid giving too many details on the meaning of parameters with respect to the circuit privacy bootstrapping algorithm and refer to [Klu22] for details. In the statement below, $\|\boldsymbol{B}_{L,Q}\|$ denotes the maximum length of a column of

$$\boldsymbol{B}_{L,Q} \coloneqq \begin{bmatrix} L & & & Q_1 \\ -1 & L & & Q_2 \\ & -1 & \ddots & \vdots \\ & & -1 & Q_\ell \end{bmatrix} \in \mathbb{Z}^{\ell \times \ell} \tag{1}$$

where $\ell \coloneqq \lceil \log_L(Q) \rceil$ and $(Q_1, \ldots, Q_\ell)$ is a base-L decomposition of Q. We recall the main circuit privacy result from [Klu22] we will use below.

Theorem 1 ([Klu22]). *Let $\beta_{\mathsf{br}}, \beta_R$ be noise bounds on a blind rotation key and LWE public key respectively and let $\boldsymbol{c}$ be an input LWE ciphertext with secret $\boldsymbol{s} \in \mathbb{Z}_2^n$. Furthermore, assume the use of a test polynomial $v(X)$ such that the* *constant of* $v(X) \cdot X^{\mathsf{Phase}(\boldsymbol{c})}$ *is* $f(m)$*. Then if*

$$\sigma_{\mathsf{rand}} \geq \max\left(4\left((1-\gamma)\cdot(2\epsilon)^2\right)^{-\frac{1}{\ell_R}},\ \sqrt{1+\beta_R}\cdot\|\boldsymbol{B}_{L_R,Q}\|\cdot\sqrt{\frac{\ln(2\ell_R(1+1/\gamma))}{\pi}}\right)$$

and

$$\sigma_{\boldsymbol{x}} \geq \sqrt{1+\beta_{\mathsf{br}}}\cdot\|\boldsymbol{B}_{L_{\mathsf{br}}}\|\cdot\sqrt{\frac{\ln(4\,n\cdot d\cdot\ell_{\mathsf{br}}(1+1/\delta))}{\pi}}$$

then

$$\Delta(\boldsymbol{c}_{\mathsf{out}}, \boldsymbol{c}_{\mathsf{fresh}}) \leq \max(\epsilon + 2\gamma, 2\delta)$$

where $\boldsymbol{c}_{\mathsf{out}}$ *is the output of the algorithm in Figure 3 of [Klu22] and* $\boldsymbol{c}_{\mathsf{fresh}} = (\boldsymbol{a}_{\mathsf{fresh}}, \boldsymbol{a}_{\mathsf{fresh}}\cdot\boldsymbol{s} + f(m) + e_{\mathsf{rand}} + e_{\mathsf{out}})$ *is a well distributed fresh ciphertext with noise distributions* $e_{\mathsf{rand}} \leftarrow\!\$\ \chi_{\mathbb{Z},\sigma_{\mathsf{rand}}\sqrt{1+\ell_R\cdot\sigma_R^2}}$*,* $e_{\mathsf{out}} \leftarrow\!\$\ \chi_{\mathbb{Z},\sigma_{\boldsymbol{x}}\sqrt{1+2\,n\cdot d\cdot\sigma_{\mathsf{br}}^2}}$*.*

Malicious security. Note that the definition above is required to hold for "any" $\mathsf{FHE.pk}, \mathsf{FHE.sk}, \mathsf{ct}_{\mathsf{out}}$ 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 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, $\mathsf{FHE.Eval}$ can explicitly check that the proof verifies and abort otherwise.

2.7 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[10] is $F_{\mathsf{one}} : \mathbb{Z}_p^{m_p \times n_p} \times \mathbb{Z}_p^{n} \to \mathbb{Z}_q$ where

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

[10] As it stands, this strong PRF candidate maps zero to zero and is thus trivially distinguished from a random function, we will address this below.

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	82	dim. of $\boldsymbol{z}$ (mod 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 the 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 G and in the attachment.[11]

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 $\boldsymbol{G}_{\mathsf{gadget}} \coloneqq (p^{\lceil \log_p(q) \rceil - 1}, \dots, 1) \otimes \boldsymbol{I}_{n_p}$ we may check

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

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].

[11] 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.

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 Boostrapping Depth One 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 homomorphic encryption to protect its key. The depth of bootstrapping required is one, which enhances efficiency and is useful in the security of our V(P)OPRF variant in Section 5.

3.1 Extending the PEI PRF

Here, we first note that the PRFs defined in Section 2.7 trivially fail to achieve pseudorandomness as they map $\mathbf{0} \in \mathbb{Z}_p^n \to \mathbf{0} \in \mathbb{Z}_q^m$, 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

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

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

3.2 TFHE-based Instantiation

We ultimately show that the above PRF is highly compatible with TFHE/FHEW, so we describe the OPRF (F_{poprf}) using subroutines from the associated literature. The high-level outline of the construction is given as Figure 3 and a full implementation of TFHE and our OPRF candidate in SageMath is given in Appendix G.

Plaintext Modulus Switching. The main point of interest is in the design of $F_{\mathsf{poprf}}.\mathsf{HEEval}$, which is given in Algorithm 2. The input LWE ciphertexts have plaintext space $\mathbb{Z}_2$ and the output LWE ciphertexts have plaintext space $\mathbb{Z}_3$. In order to perform the plaintext modulus switch, we use a variant of TFHE/FHEW bootstrapping that we denote $\mathsf{FHE.CPPBS}_{(p,q)}$. This algorithm is a variation of the standard TFHE programmable bootstrapping algorithm (see [Joy21] for details) augmented with the circuit-private technique of Kluczniak [Klu22]. The difference is that we apply a simple linear transformation and a special "test polynomial", whilst forgoing the key-switching in [Klu22, Figure 3].

In more detail, note that general TFHE bootstrapping applies a function $f : \mathbb{Z}_{2d} \to \mathbb{Z}_q$ to a plaintext using test polynomial $v(x) = \sum_{i=0}^{d-1} f(i) \cdot x^i$, *assuming* the function has negacyclic form i.e. $f(x) = -f(x+d)$. Recall that TFHE encodes a plaintext $m \in \mathbb{Z}_p$ as $m \cdot \lfloor Q/p \rceil$ during encryption and decodes intervals

$$\left[m \cdot \lfloor Q/p \rceil - \frac{\lfloor Q/p \rceil}{2},\ m \cdot \lfloor Q/p \rceil + \frac{\lfloor Q/p \rceil}{2} \right)$$

to $m \in \mathbb{Z}_p$ during decryption. Consider the simple plaintext switch $f : \lfloor m \cdot (2d/p) \rceil + e \mapsto m \cdot (Q/q)$ for $m \in \mathbb{Z}_p$, where e denotes some LWE error. The corresponding function is not negacyclic, so we would have to restrict plaintext space so that the most significant bit is zero to overcome this. Alternatively, we could apply the techniques of [LMP22] at the cost of two sequential programmable bootstraps. However, in the case $(p, q) = (2, 3)$ (which is the one that we use)[12], one can use the negacyclic function

$$f(x) = \begin{cases} \lfloor Q/3 \rceil \text{ if } x \in [d/2, 3d/2) \\ -\lfloor Q/3 \rceil \text{ otherwise} \end{cases} .$$

[12] Note that we may pick $q \neq 3$ but require $p = 2$.

$F_{\mathsf{poprf}}.\mathsf{Setup}(1^\lambda)$

$\boldsymbol{A}_{\mathsf{pp}} \leftarrow\!\!\$\ \mathbb{Z}_Q^{N \times N}$
$\mathsf{pp} \leftarrow \boldsymbol{A}_{\mathsf{pp}}$
return pp

$F_{\mathsf{poprf}}.\mathsf{KeyGen}(1^\lambda)$

$\mathsf{pk} \leftarrow \perp$
$\mathsf{sk} \leftarrow\!\!\$\ \mathbb{Z}_p^{m_p \times n_p}$
return (pk, sk)

$F_{\mathsf{poprf}}.\mathsf{Eval}(\mathsf{sk} = \boldsymbol{A}, t = \boldsymbol{t}, x = \boldsymbol{x})$

$\boldsymbol{z}' := F_{\boldsymbol{A}}^{\mathsf{RO_{key}}}(\boldsymbol{t}, \boldsymbol{x})$
$\boldsymbol{z} := \mathsf{RO_{fin}}(\boldsymbol{t}, \boldsymbol{x}, \boldsymbol{z}')$
return $\boldsymbol{z}$

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

if ct' not a ctxt **then return** $\perp$
$\boldsymbol{z}' \leftarrow \mathsf{FHE.Dec}(\mathsf{FHE.sk}, \mathsf{ct})$
$\boldsymbol{z} \leftarrow \mathsf{RO_{fin}}(\boldsymbol{t}, \boldsymbol{x}, \boldsymbol{z}')$
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

$F_{\mathsf{poprf}}.\mathsf{BlindEval}(sk = \boldsymbol{A}, t = \boldsymbol{t}, req; \mathsf{pp})$

$(\mathsf{FHE.pk}^{(\mathsf{pp})}, \mathsf{ct}, \pi) \leftarrow req$
$\boldsymbol{A_t} \leftarrow \mathsf{RO_{key}}(\boldsymbol{A}, \boldsymbol{t})$
$\mathsf{ct}' \leftarrow F_{\mathsf{poprf}}.\mathsf{HEEval}(\mathsf{FHE.pk}^{(\mathsf{pp})}, \boldsymbol{A_t}, \boldsymbol{t}, \mathsf{ct})$
if π does not verify **then** $\mathsf{ct}' = \perp$
$rep \leftarrow \mathsf{ct}'$
return rep

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

Fig. 3. Main construction.

Although this is a negacyclic function, it maps regions that decrypt to $m \in \mathbb{Z}_2$ to $-(m+1) \cdot \lfloor Q/3 \rceil \bmod Q$. To correct this, $\mathsf{CPPBS}_{(2,3)}$ completes by negating the ciphertext and subtracting $\lfloor Q/3 \rceil$. To summarise, the complete $\mathsf{CPPBS}_{(2,3)}$ procedure takes as input an LWE encryption of $m \in \mathbb{Z}_2$ and outputs an LWE ciphertext that encrypts m as an element of $\mathbb{Z}_3$. We may then optionally apply the ciphertext compression technique from [CDKS21] to pack multiple answers into a single RLWE ciphertext (we suppress this step in pseudocode for brevity).

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

Input: pk ▷ HE public key with ciphertext modulus Q, plaintext modulus p
Input: $\boldsymbol{A_t} \in \mathbb{Z}_p^{m_p \times n_p}$, $\mathsf{ct} \in \mathcal{C}^{n_p}$ encrypting $\boldsymbol{y} \in \mathbb{Z}_p^{n_p}$
Output: $\mathsf{ct}' \in \mathcal{C}^m$ encrypting $F_{\boldsymbol{A}}^{\mathsf{RO_{key}}}(\boldsymbol{t}, \boldsymbol{x})$ or $\perp$
$\mathsf{ct}' \leftarrow\!\!\$\ \mathsf{FHE.Enc}(\mathsf{pk}, 1)$ ▷ can be e.g. $1 \cdot \lfloor Q/p \rceil$
$\mathsf{ct}^{(1)} \leftarrow \boldsymbol{A_t} \cdot (\mathsf{ct}, \mathsf{ct}') \bmod Q$ ▷ additive hom. plaintext space $\mathbb{Z}_p$
$\mathsf{ct}_i^{(2)} \leftarrow \mathsf{FHE.CPPBS}_{(p,q)}(\mathsf{pk}, \mathsf{ct}_i^{(1)}) \quad \forall i \in \mathbb{Z}_{m_p}$ ▷ plaintext space $\mathbb{Z}_q$
$\mathsf{ct}^{(3)} \leftarrow \boldsymbol{G}_{\mathsf{out}} \cdot \mathsf{ct}^{(2)} \bmod Q$ ▷ m LWE ciphertexts
return $\mathsf{ct}^{(3)}$ ▷ Optionally pack into single RLWE ctxt [CDKS21]

Commitment. A further important alteration is that $\mathsf{FHE.KeyGen}^{(\mathsf{pp})}$ will output a commitment to the secret key $\mathsf{FHE.sk} = \boldsymbol{s} \in \mathbb{Z}_2^e$, 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, and $\boldsymbol{r}, \boldsymbol{e} \leftarrow\$ (\chi')^N$, 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 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 $\mathsf{FHE.pk}^{(\mathsf{pp})} := (\boldsymbol{b}_{\mathsf{pk}}, \mathsf{FHE.pk})$.

Public Key. Note that although the server does not need to create encryptions itself, we still use the public-key version of TFHE rather than a symmetric-key version as this is useful for circuit privacy [Klu22]. This extra key material in the public key does not impact efficiency noticeably, as the bootstrapping key sizes are the main bottleneck. This can be seen in Appendix A where details of $\mathsf{NIZKAoK_C}$ are elaborated, and where we fully describe the generation of the bootstrapping keys.

Noise Analysis Overview. Although key-switching may help reduce the number of loops in the blind rotation phase, we will ignore it as we only have bootstrapping depth one, so it is useless in our setting. Additionally, we do not consider key compression here for simplicity (this can be added by amending β_{br} below). We assume a blind rotation base B_{br} and set $\ell_{\mathsf{br}} = \lceil \log_{B_{\mathsf{br}}}(Q) \rceil$. We also assume the blind rotation key has a noise bound of β_{br}. For the final rerandomisation step of circuit-private bootstrapping, we use B_R and $\lceil \ell_{B_R}(Q) \rceil$, assuming a noise bound of β_R for the LWE instances. $\mathsf{CPPBS}_{(2,3)}$ leads to exactly the same noise term as a circuit-private bootstrapping, so we just analyse the output of a circuit-private bootstrap (without key-switching).

To describe the correctness requirement, we will use $\mathsf{noise}_\star$ to denote an infinity-norm noise bound of ciphertext $\star$ when viewed as an encryption of the "correct" value. Moving through the computation, we can track error terms as:

- $\mathsf{noise}_{\mathsf{ct}^{(1)}} \leq n_p \cdot \mathsf{noise}_{\mathsf{ct}}$
- $\mathsf{noise}_{\mathsf{ct}^{(2)}} \leq \sqrt{\sigma_{\mathsf{rand}}^2 (1 + \ell_R \sigma_R^2) + \sigma_{\boldsymbol{x}}^2 (1 + 2nd\sigma_{\mathsf{br}}^2)} \cdot c$ for some appropriately chosen $c, \sigma_{\mathsf{rand}}, \sigma_{\boldsymbol{x}}$ (see Theorem 1)
- $\mathsf{noise}_{\mathsf{ct}^{(3)}} \leq m_p \cdot \mathsf{noise}_{\mathsf{ct}^{(2)}}$.

Let e denote the TLWE secret key dimension. Then, for correctness, we require that $\mathsf{noise}_{\mathsf{ct}^{(1)}} \cdot \frac{2d}{Q} + \frac{e}{2} \leq d/2$ for the Q-to-$2d$ mod-switching $\mathsf{ct} \mapsto \lfloor \mathsf{ct} \cdot 2d/Q \rceil$

in $\mathsf{CPPBS}_{(2,3)}$, and also that $\mathsf{noise}_{\mathsf{ct}^{(3)}} \leq Q/3$ for decryption correctness of the unpacked output ciphertext. For circuit privacy, the parameters σ_{rand} and $\sigma_{\boldsymbol{x}}$ must be chosen according to Theorem 1. As we see later, for POPRIV1 security, we also need the $M\text{-}\mathsf{LWE}_{\mathbb{Z}_Q,N,\mathbb{Z}_2,\sigma}$, $M\text{-}\mathsf{LWE}_{\mathcal{R}_q,1,\mathbb{Z}_2,\sigma_R}$ and $M\text{-}\mathsf{LWE}_{\mathcal{R}_q,1,\mathbb{Z}_2,\sigma_{\mathsf{br}}}$ assumptions with $\mathcal{R}_q := \mathbb{Z}_q[X]/(X^d+1)$, for the security of the FHE scheme.

Ciphertext compression. The LWE to RLWE ciphertext packing operation from [CDKS21] introduces an additional error whose variance is bounded by $\frac{d^2-1}{3} \cdot V_{\mathsf{ks}}$, where V_{ks} is the variance of a key-switching operation. In particular, $V_{\mathsf{ks}} = \ell_{\mathsf{ks}} \cdot B_{\mathsf{ks}}^2 \cdot \sigma_{\mathsf{ks}}^2$, where $(\ell_{\mathsf{ksk}}, B_{\mathsf{ksk}}, \sigma_{\mathsf{ksk}})$ are analogues to $(\ell_{\mathsf{br}}, B_{\mathsf{br}}, \sigma_{\mathsf{br}})$ in a key-switching key context. Therefore, if ciphertext packing is used, the correctness property $\mathsf{noise}_{\mathsf{ct}^{(3)}} \leq Q/3$ becomes $\mathsf{noise}_{\mathsf{ct}^{(3)}} + \sqrt{\frac{d^2-1}{3} \cdot V_{\mathsf{ks}}} \cdot c' \leq Q/3$, for some appropriately chosen c'. We further assume the hardness of $M\text{-}\mathsf{LWE}_{\mathcal{R}_q,1,\mathbb{Z}_2,\sigma_{\mathsf{ks}}}$.

Choosing Parameters & Size Estimates. To pick parameters, we run the script in Appendix C, which checks the noise/correctness constraints and hardness constraints mentioned above. The script additionally includes correctness constraints for the public key compression techniques in [KLD+23]. Based on this, we estimate the size of the bootstrapping key (which may be considered an amortisable offline communication cost) in Appendices A and D as 14.7MB. The size of the zero-knowledge proofs accompanying this key is 90.7KB. Applying public-key compression, we instead obtain 2.4MB and 137.4KB respectively.

Each request then sends LWE encryptions of $n_p = 256$ bits $\boldsymbol{m}$ using protected encoded inputs, i.e. $(\boldsymbol{A}^{(0)}, \boldsymbol{b} := \boldsymbol{A}^{(0)} \cdot \boldsymbol{s} + \boldsymbol{e} + \lfloor Q/P \rceil \cdot \boldsymbol{m})$ where $\boldsymbol{A}^{(0)} \in \mathbb{Z}_Q^{n_p \times e}$ and $\boldsymbol{b} \in \mathbb{Z}_Q^{n_p}$. Here, $\boldsymbol{A}^{(0)}$ 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], we may drop the least significant bits. In total we have a ciphertext size of 2.0KB. The accompanying zero-knowledge proofs take up 63.0KB but can be amortised to cost about 1.8KB per query when sending 64 queries in one shot (Appendix A).[13]

The message back from the server is $m = 82$ encryptions of the output elements belonging to $\mathbb{Z}_q$. Here, we cannot expand the dominant uniform matrix $\boldsymbol{A}^{(1)}$ from a small seed, but we can drop the least significant bits of $\boldsymbol{A}^{(1)}$, since $\boldsymbol{s}$ is binary. In particular, we may drop, say, the 8 least significant bits and we arrive at $e \cdot m \cdot 24$ bits for $\boldsymbol{A}^{(1)}$. 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 $82 \cdot 16$ bits. In total we get 480.6KB. As mentioned above, it is more efficient to pack all return values into a single RLWE sample using techniques from [CDKS21], since the cost of transmitting $\boldsymbol{A}^{(1)}$ dominates here. This does not require additional key material when using public-key compression and reduces the size of response to about 6.2KB. For more details on these values, see Appendix A.1. We give our code for estimating parameters in Appendices D and F. The communication performance of our scheme without public key compression has smaller online communication cost, as reported in Table 1.

[13] All of these figures assume that we apply public-key compression.

Remark 3. While our parameter selection is largely conservative in applying worst-case bounds and in adopting the noise sizes required for circuit privacy according to [Klu22, Theorem 1], we deviate from the theorem in setting $\ell = 1$ and $L < Q$ in (1) when we do not apply public-key compression. This is because the lower-order bits of the decomposed vectors contain only noise. These random bits are then linearly composed with encryptions of $\boldsymbol{s}$. Thus, the server may simply sample its own random "$\boldsymbol{s}$" to perform this computation outputting noise. Not performing this optimisation would increase the size of the public-key by a factor of three. We use $\ell \coloneqq \lceil \log(Q, L) \rceil$ when applying public-key compression.

4 Security

We first prove the pseudorandomness property against malicious clients in Theorem 2 and then privacy (POPRIV1 only) against servers in Theorem 3.

Theorem 2. *Let FHE denote the TFHE scheme with $q \mid Q$. 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.*
- *For any valid $\mathsf{pk}, \mathsf{ct} = \mathsf{FHE}.\mathsf{Enc}(m)$ for $m \in \mathbb{Z}_p$, $\mathsf{CPPBS}^{(p,q)}(\mathsf{pk}, \mathsf{ct})$ is indistinguishable from a fresh LWE ciphertext encrypting m as a vector in $\mathbb{Z}_q$, with some error distribution χ_{Sim} as in Theorem 1.*
- *The $\mathsf{mat}\text{-}\mathsf{NTRU}_{N,P',Q,B}$ assumption holds where Q is the FHE modulus.*
- *P' is even and coprime to Q, such that $Q > B \cdot N \cdot (\beta' + P' \cdot (2\,\beta' + 1)/2)$ where β' is the standard deviation used in $\mathsf{FHE}.\mathsf{KeyGen}^{(\mathsf{pp})}$.*
- *$F.(\cdot, \cdot)$, defined in Section 3, is a PRF with output range super-polynomial in the security parameter (e.g. 2^{λ}).*

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

$\mathsf{S.Init}$**:** Sample $\boldsymbol{A}^{(\mathsf{S})} \leftarrow\!\!\$\ \mathbb{Z}_p^{m_p \times (n_p+1)}$, $\boldsymbol{A}_{\mathsf{pp}}^{(\mathsf{S})} \leftarrow\!\!\$\ \mathbb{Z}_Q^{N \times N}$, 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 $\mathrm{LimitEval}(\boldsymbol{x}_{\mathsf{in}})$ 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. $\square$

Lemma 2. *Assume that all of the conditions in Theorem 2 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 $\mathcal{H}_i$, simulators S_i and events E_i for $i \in \{1,2,3\}$:

$\mathcal{H}_1$ **(NTRU Trapdoor):** The public parameter 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). Here, $P > 4N$ is a power of 2 such that $Q > B \cdot N \cdot (\beta' + P' \cdot (2\beta' + 1)/2)$ with β' the standard deviation of $\boldsymbol{r}$ and $\boldsymbol{e}$ used to produce the commitment part $\boldsymbol{b}_{\mathsf{pk}}$ of $\mathsf{FHE.pk}^{(\mathsf{pp})}$. The event 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.

$\mathcal{H}_2$ **(Circuit Privacy):** 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 the PRF evaluation $\boldsymbol{z} = F_{\boldsymbol{A}^{(\mathsf{S})}}(\boldsymbol{t}, \boldsymbol{x})$. Next for $i = 1, \ldots, m$, it samples LWE error $\tilde{e}_i \leftarrow\!\!\$\ \chi_{\mathsf{Sim}}$ from Theorem 1 and a matrix $\boldsymbol{A} \leftarrow\!\!\$\ \mathbb{Z}_Q^{m_p \times N}$. Finally it sets

$$\mathsf{ct}_{\mathsf{Sim}}^{(3)} = (\boldsymbol{G}_{\mathsf{out}} \cdot \boldsymbol{A},\ \boldsymbol{G}_{\mathsf{out}} \cdot (\boldsymbol{A} \cdot \boldsymbol{s} + \tilde{\boldsymbol{e}}) + \boldsymbol{z} \cdot (Q/q) \bmod Q)$$

and returns $\mathsf{ct}_{\mathsf{Sim}}^{(3)}$ in response to req. The event 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.

$\mathcal{H}_3$ **(PRF):** In S_3.BlindEval and S_3.Eval, S_3 replaces all calls to $F_{\boldsymbol{A}^{(\mathsf{S})}}(\cdot,\cdot)$ by a truly random function G. Additionally, S_3 does not sample $\boldsymbol{A}^{(\mathsf{S})}$ when it runs $\mathsf{S}_3.\mathsf{Init}$. The event 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 argue that $|\Pr[\mathsf{E}] - \Pr[\mathsf{E}_1]| = \mathsf{negl}(\lambda)$ using the $\mathsf{mat\text{-}NTRU}_{N,Q,P',B}$ assumption: we build a matrix NTRU distinguisher $\mathcal{B}_{\mathsf{mat\text{-}NTRU}}$ that implements S for the POPRF adversary $\mathcal{A}$ using its own $\mathsf{mat\text{-}NTRU}_{N',Q,P',B}$ challenge as $\boldsymbol{A}_{\mathsf{pp}}$. Clearly, if the challenge is a uniform matrix, $\mathcal{B}_{\mathsf{mat\text{-}NTRU}}$ 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 $\mathcal{B}_{\mathsf{mat\text{-}NTRU}}$ would be able to distinguish and break the $\mathsf{mat\text{-}NTRU}_{N',Q,P',B}$ 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 $|\Pr[\mathsf{E}_1] - \Pr[\mathsf{E}_2]| = \mathsf{negl}(\lambda)$ using the assumption on $\mathsf{FHE.CPPBS}$ and $q \mid Q$. 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 $i^* \in \{0, \ldots, q_t^* - 1\}$ 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 well-formed triple

$$\mathsf{FHE.pk},\ \mathsf{FHE.sk} = \boldsymbol{s},\ \mathsf{ct} = \mathsf{FHE.Enc}(\mathsf{FHE.pk}, \boldsymbol{y})$$

(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

$$\boldsymbol{G}_{\mathsf{out}} \cdot \mathsf{FHE.CPPBS}_(p,q)(\mathsf{pk}, \mathsf{ct}^{(1)}) \tag{2}$$

and

$$\boldsymbol{G}_{\mathsf{out}} \cdot (\boldsymbol{A}, \boldsymbol{A} \cdot \boldsymbol{s} + \boldsymbol{e} \bmod Q) + (0, \boldsymbol{z} \cdot Q/q) \tag{3}$$

for $\boldsymbol{z} = F_{\boldsymbol{A}^{(\mathsf{S})}}(\boldsymbol{t}, \boldsymbol{x})$ where $\boldsymbol{t}$ is the value from the (i^*+1)-th query. Note that by construction the latter is sampled without knowledge of any server secret. To show that such a distinguisher could not exist, we rewrite Equations (2) and (3) as

$$\boldsymbol{G}_{\mathsf{out}} \cdot \mathsf{FHE.CPPBS}_{(p,q)}(\mathsf{pk}, \mathsf{FHE.Enc}(\boldsymbol{A}_{\boldsymbol{t}}^{(\mathsf{S})} \cdot \boldsymbol{y})) \tag{4}$$

and

$$\boldsymbol{G}_{\mathsf{out}} \cdot (\boldsymbol{A}, \boldsymbol{A} \cdot \boldsymbol{s} + (\boldsymbol{A}_{\boldsymbol{t}}^{(\mathsf{S})} \cdot \boldsymbol{y} \bmod p) \cdot Q/q + \boldsymbol{e} \bmod Q) \tag{5}$$

respectively. Here we are using that $\boldsymbol{G}_{\mathsf{out}} \cdot (\boldsymbol{A}_{\boldsymbol{t}}^{(\mathsf{S})} \cdot \boldsymbol{y} \bmod p) \cdot Q/q \bmod Q$ and $\boldsymbol{z} \cdot Q/q \bmod Q$ are identically distributed if $q \mid Q$. It should be clear that the existence of $\mathcal{B}$ would lead to an algorithm $\mathcal{B}^*$ that could distinguish between $\mathsf{FHE.CPPBS}_{(p,q)}(\mathsf{pk},\ \mathsf{FHE.Enc}(\boldsymbol{A}_{\boldsymbol{t}}^{(\mathsf{S})} \cdot \boldsymbol{y}))$ and $(\boldsymbol{A},\ \boldsymbol{A} \cdot \boldsymbol{s} + (\boldsymbol{A}_{\boldsymbol{t}}^{(\mathsf{S})} \cdot \boldsymbol{y} \bmod p) \cdot Q/q + \boldsymbol{e} \bmod Q)$. The algorithm $\mathcal{B}^*$ chooses pk, sk as $\mathcal{B}$ does. Then, on input $\boldsymbol{v}$, it simply passes the value of $(\boldsymbol{G}_{\mathsf{out}} \cdot \boldsymbol{v})$ onto $\mathcal{B}$ and returns the same answer as $\mathcal{B}$. Clearly if $\boldsymbol{v}$ is of the first form, $\mathcal{B}^*$ is simulating the distribution described in Equation (4) and otherwise it simulates the distribution from Equation (5). Therefore, $\mathcal{B}^*$'s distinguishing advantage is the same as $\mathcal{B}$'s. Our assumption on the distribution of $\mathsf{FHE.CPPBS}$ then allows us to conclude that $|\Pr[\mathsf{E}_{1,i^*}] - \Pr[\mathsf{E}_{1,i^*+1}]|$ is negligible.

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. $\square$

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, $\Pr[\mathsf{E}_3] = \mathsf{negl}(\lambda)$. $\square$

Remark 4. Note that the security proof asks that $q \mid Q$. However, parts of the OPRF (e.g. efficient zero-knowledge proofs) might require or benefit from Q having a specific form that is not a multiple of $q = 3$. This situation can be remedied by applying an LWE modulus switch to a nearby multiple of Q just after $\mathsf{FHE.CPPBS}$ is applied in Algorithm 2.

Theorem 3. *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 M-$\mathsf{LWE}_{\mathbb{Z}_Q,N,\chi',\chi'}$ assumption holds where $\chi' = D_{\mathbb{Z},\beta'}$ is the error distribution used in $\mathsf{FHE}.\mathsf{KeyGen}^{(\mathsf{pp})}$.*

Proof. Let G_0 and G_1 denote the $POPRIV1^{\mathcal{A},b}_{\mathcal{F},H}$ 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 Run 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. Recall that $\boldsymbol{b}_{\mathsf{pk}} = \boldsymbol{A}_{pp} \cdot \boldsymbol{r} + \boldsymbol{e} + \lfloor Q/2 \rceil \cdot \boldsymbol{s}$ is an LWE encryption. In G_i', we have that $\mathsf{FHE.pk}$ will remain the same, but $\boldsymbol{b}_{\mathsf{pk}}$ is replaced by uniform random values $\boldsymbol{u}_{\mathsf{pk}}$. 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 M-$\mathsf{LWE}_{\mathbb{Z}_Q,N,\chi',\chi'}$ 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 $(\mathsf{FHE.pk}, \mathsf{FHE.sk}) \leftarrow\$ \mathsf{FHE.KeyGen}^{(pp)}()$ 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 $i \in \{0, q_R^* - 1\}$ 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, $\mathrm{G}_{0,i}' \approx_c \mathrm{G}_{0,i+1}'$: 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 $\tau_0' = ((req^*, \mathcal{F}.\mathsf{BlindEval}(sk, req^*)), \mathcal{F}.\mathrm{Eval}(\mathsf{sk}, t, x_0))$ 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 sending more "check" material. 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 "check" points amongst genuine OPRF queries. The client checks if these check points match the known answer values. It also checks if evaluations on the same points produce the same outputs and if evaluations on different points produce different outputs, here we implicitly rely on the PRP-PRF switching lemma [BR06]. If these checks pass, we conjecture that the C may then assume that S computed the (P)OPRF correctly provided parameters are chosen appropriately. In more detail, let $\gamma = \nu \cdot \alpha + \beta$ be the number of points C submits.

1. For some fixed $\boldsymbol{t}$, S commits to κ points $\boldsymbol{z}_k^\star := F_{\boldsymbol{A}}(\boldsymbol{t}, \boldsymbol{x}_k^\star)$ 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. they satisfy the relation
$$\mathfrak{R}_{\boldsymbol{t}} := \left\{\boldsymbol{t},\ \{(\boldsymbol{x}_k^\star, \boldsymbol{z}_k^\star)\}_{k \in \mathbb{Z}_\kappa};\ \boldsymbol{A} \mid \boldsymbol{z}_k^\star = F_{\boldsymbol{A}}(\boldsymbol{t}, \boldsymbol{x}_k^\star)\right\} \tag{6}$$
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 $j \in \mathbb{Z}_\beta$. It constructs the vector
$$\left(\overbrace{\boldsymbol{x}_0^{(\alpha)}, \ldots, \boldsymbol{x}_0^{(\alpha)}}^{\nu \text{ copies}}, \ldots, \overbrace{\boldsymbol{x}_{\alpha-1}^{(\alpha)}, \ldots, \boldsymbol{x}_{\alpha-1}^{(\alpha)}}^{\nu \text{ copies}}, \boldsymbol{x}_0^{(\beta)}, \ldots, \boldsymbol{x}_{\beta-1}^{(\beta)}\right).$$
It then applies a secret permutation ρ, i.e. shuffles the indices, and submits these queries.
3. C applies ρ^{-1} to the responses, i.e. unshuffles the indices, and receives $\boldsymbol{z}_i$. The client C rejects if any of the following conditions is satisfied:
   (a) For $0 \leq k < \alpha$ and for $i, j \in \mathbb{Z}_k$: $\boldsymbol{z}_{\nu \cdot k+i} \neq \boldsymbol{z}_{\nu \cdot k+j}$, i.e. evaluations on the same point disagree.
   (b) For $0 \leq k < \ell < \alpha$ and for $i, j \in \mathbb{Z}_k$: $\boldsymbol{z}_{\nu \cdot k+i} = \boldsymbol{z}_{\nu \cdot \ell+j}$, i.e. evaluations on different points agree.
   (c) For $0 \leq k \leq \beta$: $\boldsymbol{z}_{\nu \cdot \alpha+k} \neq \boldsymbol{z}_k^\star$, i.e. check points do not match.

   Otherwise C accepts.

We formalise the security definition with a game in Figure 4. Since the tag $\boldsymbol{t}$ remains constant throughout we suppress it here. We say a (P)OPRF, $\mathcal{F}$, is verifiable if the following advantage is negligible in the security parameter λ:

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

Remark 5. We phrase our candidate construction above directly in a batch variant that amortises the overhead of verifiability by submitting α points together. To recover the usual definition (also used in our security game) of a single evaluation, we may either simply sample $\alpha - 1$ random points and then call our batch variant or submit more check points. This is necessary since γ is a function of α and β but affects security bounds.

$\underline{\mathrm{VERIF}^{\mathcal{A}}_{\mathcal{F},\mathsf{RO}}}$

$\mathsf{pp} \leftarrow \mathcal{F}.\mathsf{Setup}(\lambda)$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathcal{A};$
$\boldsymbol{x} \leftarrow\$ \mathbb{Z}_p^n$
$req \leftarrow \mathcal{F}.\mathsf{Request}(\mathsf{pk}, \boldsymbol{t}, \boldsymbol{x}; \mathsf{pp})$
$rep \leftarrow \mathcal{A}^{\mathrm{Request},\mathrm{Finalise},\mathsf{RO}}(req)$
$\boldsymbol{z}' \leftarrow \mathcal{F}.\mathsf{Finalise}(rep)$
$\boldsymbol{z} \leftarrow \mathcal{F}.\mathsf{Eval}(\mathsf{sk}, \boldsymbol{t}, \boldsymbol{x})$
if $\boldsymbol{z}' \neq \boldsymbol{z}$ **then return** 1
return 0

Fig. 4. Verifiability Experiment for (P)OPRF

5.1 Verifiability from Levelled HE

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].

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, we pick parameters such that only Line 3 of Algorithm 2 costs a bootstrapping operation, i.e. all linear operations are realised without bootstrapping. This is already how Algorithm 2 is written, but we foreground this as a *security requirement* here. We then stress that our POPRF in Algorithm 2 can be evaluated in depth one, and that the bootstrapping key submitted to S presumably prevents it from computing higher depth circuits. This is enabled by the removal of a key-switching key in our (VP)OPRF. We give our construction in Figure 5.

$\underline{F_{\mathsf{vpoprf}}.\mathsf{Setup}(1^\lambda)}$

$\boldsymbol{A}_{\mathsf{pp}} \leftarrow\!\!\$\ \mathbb{Z}_Q^{N \times N}$

$\mathsf{pp} \leftarrow \boldsymbol{A}_{\mathsf{pp}}$

return pp

$\underline{F_{\mathsf{vpoprf}}.\mathsf{KeyGen}(1^\lambda)}$

$\mathsf{pk} \leftarrow \perp$

$\mathsf{sk} \leftarrow\!\!\$\ \mathbb{Z}_p^{m_p \times n_p}$

$\boldsymbol{x}_i \leftarrow X$ 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 (pk, sk)

$\underline{F_{\mathsf{vpoprf}}.\mathsf{Eval}(\mathsf{sk} = \boldsymbol{A}, \boldsymbol{t} = \perp, x = \boldsymbol{x})}$

$\boldsymbol{z}' := F_{\boldsymbol{A}}(\boldsymbol{t}, \boldsymbol{x})$

$\boldsymbol{z} := \mathsf{RO}_{\mathsf{fin}}(\boldsymbol{t}, \boldsymbol{x}, \boldsymbol{z}')$

return $\boldsymbol{z}$

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

Parse 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, ρ

$\underline{F_{\mathsf{vpoprf}}.\mathsf{BlindEval}(sk = \boldsymbol{A}, \boldsymbol{t} = \perp, req; \mathsf{pp})}$

$(\mathsf{FHE.pk}^{(\mathsf{pp})}, \mathsf{ct}, \pi) \leftarrow req$

$\mathsf{ct}' \leftarrow F_{\mathsf{vpoprf}}.\mathsf{HEEval}(\mathsf{FHE.pk}^{(\mathsf{pp})}, \boldsymbol{A}, \boldsymbol{t}, \mathsf{ct})$

if π does not verify **then** $\mathsf{ct}' = \perp$

$rep \leftarrow \mathsf{ct}'$; **return** rep

$\underline{F_{\mathsf{vpoprf}}.\mathsf{Finalise}(\mathsf{FHE.sk}, rep = \mathsf{ct}, \rho)}$

if 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** $\boldsymbol{z}$

$\underline{F_{\mathsf{vpoprf}}.\mathsf{Verify}(\boldsymbol{z}, \boldsymbol{z}^\star, \rho)}$

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

Fig. 5. Candidate VPOPRF construction, F_{vpoprf}.

5.2 Cryptanalysis

We explore cheating strategies of a malicious server.

Maximal-change guessing. Assume the adversary guesses the positions of the check points in order to make C accept incorrect outputs, i.e. the adversary behaves honestly on the check points but dishonestly yet consistent on all other points. Recall that $\gamma = \nu \cdot \alpha + \beta$ is the number of points C submits to S, where there are β such check point positions. 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 $1/\binom{\gamma}{\beta}$ of guessing correctly.

Minimal-change guessing. Assume the adversary's strategy is to evaluate all points honestly except for one. The consistency check that all ν evaluations of the same point must agree and that all other evaluations must disagree, means this adversary has to guess the ν positions of the target point. Since there are α evaluation points, we obtain a probability $\alpha/\binom{\gamma}{\nu}$ of making a correct guess.

Interpolation. We consider an adversarial S that uses a circuit F'_{pei}, of depth at most one, to win the verifiability experiment. At a high level, it solves a quadratic system of equations, assuming one level of bootstrapping allows to implement one multiplication with arity two. 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 check points, this can be trivially done. 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 $\kappa = 128^2$, then the communication cost of check points is approximately an additional 0.5MB, which can be reduced to 256KB by generating the input values from a seed.

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 two.

Parameters. We may aim for 80-bit security against the statistical guessing attacks above while assuming $\kappa = 128$ and a depth-one check and cheat algorithm does not exist. To do so, we set $\gamma = 1165$, $\beta = 10, \nu = 11$ and thus $\alpha = 105$. This implies a multiplicative size overhead of ≈ 11.1 to heuristically upgrade the OPRF to a VOPRF.

Algorithm 3 Cheating Circuit

Input: $[\boldsymbol{y}] \in \mathbb{Z}_p^{n_p}$; check points $\{\boldsymbol{y}_j^\star\}$; $H : \mathbb{Z}_p^{n_p} \to \mathbb{Z}_q^m$ any function

$[\boldsymbol{r}], [\mathsf{found}] \leftarrow 0, 0$
for all $\boldsymbol{y}_j^\star$ **do**
  $[\boldsymbol{d}] \leftarrow [\boldsymbol{y}] - \boldsymbol{y}_j^\star$
  $[h] \leftarrow \sum_{i \in \mathbb{Z}_{n_p}} d_i$
  if $[h] = 0$ **then** ▷ CMUX, depth one
    $[\boldsymbol{r}] \leftarrow [\boldsymbol{r}] + \boldsymbol{y}_j^*$; $[\mathsf{found}] \leftarrow [\mathsf{found}] + 1$
  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 one

5.3 Security Reduction

Finally, we establish that our verifiability property implies POPRIV2 under some additional assumptions.

Theorem 4. *Our VPOPRF construction satisfies POPRIV2 if*

- F_{vpoprf} *satisfies POPRIV1.*
- F_{vpoprf} *is verifiable.*
- *NIZKAoK*$_{\mathfrak{R}_{\boldsymbol{t}}}$ *is sound for* $\mathfrak{R}_{\boldsymbol{t}}$ *defined in* (6).

Proof. Let G_0 and G_1 denote the $\mathrm{POPRIV2}_{\mathcal{F},H}^{\mathcal{A},b}$ 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 $\mathrm{NIZKAoK}_{\mathfrak{R}_{\boldsymbol{t}}}$. As a reminder, the relation $\mathfrak{R}_{\boldsymbol{t}}$ is defined in (6).

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 $\mathrm{NIZKAoK}_{\mathfrak{R}_{\boldsymbol{t}}}$. 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 $\mathrm{NIZKAoK}_{\mathfrak{R}_t}$ 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 back req 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 rep 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. $\square$

6 Proof-of-Concept Implementations

6.1 SageMath

We give a SageMath [S+23] implementation of TFHE and our OPRF candidate in Appendix G. This implementation is meant to establish and clarify ideas and thus we do not provide benchmarks for it. Our implementation is complete with respect to the core functionality. In particular, we provide a new from-scratch implementation of TFHE [CGGI20] in `tfhe.py`, of circuit-private TFHE bootstrapping [Klu22] in `cpbs.py`, and ciphertext and bootstrapping-key compression [CDKS21,KLD+23] in `compression.py`. Our OPRF in `oprf.py` is then relatively simple and calls the appropriate library functions. We re-implemented the underlying machinery since we are not aware of public implementations that provide all these features yet. We did not implement the zero-knowledge proof systems from [LNP22] and [BS22] since those are somewhat orthogonal to the focus of this work. We did, however, as indicated earlier, adapt or re-implement scripts for estimating their (combined) proof sizes.

6.2 Rust Benchmarks

To give an initial sense of performance, we also implemented the key operations relied upon by our OPRF in Rust. In particular, we use Zama's `tfhe-rs` FHE library [CGGI20]. Unfortunately, several functionalities we rely on are not (yet) implemented in `tfhe-rs`: circuit privacy, ciphertext and public-key compression. Moreover, `tfhe-rs` assumes throughout that plaintext moduli are powers of two, which is incompatible with our OPRF. The two most expensive operations are $F_{\mathsf{poprf}}.\mathsf{KeyGen}$ run by the client and $F_{\mathsf{poprf}}.\mathsf{BlindEval}$ run by the server, which we discuss next.[14]

In our benchmarks $F_{\mathsf{poprf}}.\mathsf{KeyGen}$ took 1s. While this contains neither proving well-formedness nor compressing the public-key, we expect neither to be significantly more expensive. 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.

For the online server-side algorithm ($F_{\mathsf{poprf}}.\mathsf{BlindEval}$), the runtime in our benchmarks was 151ms, which may be quick enough for certain applications that have a hard requirement to ensure post-quantum security. However, as mentioned above, this costs "plain" and not circuit-private boostrapping which would be significantly more expensive.[15] On the other hand, as mentioned under open problems, it is plausible that cheaper alternatives to full circuit privacy might suffice for our setting. More broadly, we note that hardware acceleration of this step is a viable option, cf. [BDV22]. In any case, 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 FHE-based approach and previous work is evident.

These results were acquired by using a server with 96 Intel Xeon Gold 6252 CPU @ 2.10GHz cores and 768 GB of RAM. Server evaluation was 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.[16] Client evaluations use only a single core. Each of the benchmarks was established after running it ten times and taking the average runtime. Our benchmarking code is available at https://anonymous.4open.science/r/oprf-fhe-D2E7.

[14] $F_{\mathsf{poprf}}.\mathsf{Request}$ runs in 28.9ms and$F_{\mathsf{poprf}}.\mathsf{Finalise}$ in 0.2ms in our benchmarks. The former does not include the time to prove well-formedness, but – as below – we do not expect this to radically change this picture.

[15] Table 3 of [Klu22] reports an overhead of 5x to 10x for circuit privacy.

[16] On a single core, server blind evaluation took 7.1s.

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 tfhe-rs; 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/S02-0330/1, 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, while Alex Davidson was at Brave Software, and 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, 1, 7, 5

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. 8, 2.7

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 Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 430–454. Springer, Heidelberg, April 2015. 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

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. 2.6

BDV22. Michiel Van Beirendonck, Jan-Pieter D'Anvers, and Ingrid Verbauwhede. FPT: a fixed-point accelerator for torus fully homomorphic encryption. Cryptology ePrint Archive, Report 2022/1635, 2022. https://eprint.iacr.org/2022/1635. 6.2

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, 1.3, 2.7, 2.7, 2.7, 3

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

BR06. Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, Heidelberg, May / June 2006. 5

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, 6.1, A, A.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. 2.7

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. 1.1, 3.2, 5, 3.2, 3.2, 6.1

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, 6.1, 6.2

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, 1, 2.7, 2.7

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, 6.2

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. 2.6

DS22. Yevgeniy Dodis and Thomas Shrimpton, editors. *CRYPTO 2022, Part II*, volume 13508 of *LNCS*. Springer, Heidelberg, August 2022. 6.2

DvW21. Léo Ducas and Wessel P. J. van Woerden. NTRU fatigue: How stretched is overstretched? In Tibouchi and Wang [TW21], pages 3–32. 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

FOE23. Sebastian Faller, Astrid Ottenhues, and Johannes Ernst. Composable oblivious pseudo-random functions via garbled circuits. Cryptology ePrint Archive, Paper 2023/1176, 2023. https://eprint.iacr.org/2023/1176. 1.2, 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. 6.2

Gen09. Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig. 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, 2.6

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, 6.2

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, 3.2

KLD+23. Andrey Kim, Yongwoo Lee, Maxim Deryabin, Jieun Eom, and Rakyong Choi. Lfhe: Fully homomorphic encryption with bootstrapping key size less than a megabyte. Cryptology ePrint Archive, Paper 2023/767, 2023. https://eprint.iacr.org/2023/767. 1.1, 1, 3.2, 6.1, A.1, A.1, A.1

Klu22. Kamil Kluczniak. Circuit privacy for FHEW/TFHE-style fully homomorphic encryption in practice. Cryptology ePrint Archive, Report 2022/1459, 2022. https://eprint.iacr.org/2022/1459. 1.1, 2.6, 2.6, 1, 3.2, 3.2, 3, 6.1, 15, A.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, A.1

LMP22. Zeyu Liu, Daniele Micciancio, and Yuriy Polyakov. Large-precision homomorphic sign evaluation using FHEW/TFHE bootstrapping. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part II*, volume 13792 of *LNCS*, pages 130–160. Springer, Heidelberg, December 2022. 1.1, 2.5, 3.2

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, 6.1, A.1, A.1, A.1, A.2, E

LNPS21. Vadim Lyubashevsky, Ngoc Khanh Nguyen, Maxime Plançon, and Gregor Seiler. Shorter lattice-based group signatures via "almost free" encryption and other optimizations. In Tibouchi and Wang [TW21], pages 218–248. A.1

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

MP21. Daniele Micciancio and Yuriy Polyakov. Bootstrapping in fhew-like cryptosystems. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC '21, page 17–28, New York, NY, USA, 2021. Association for Computing Machinery. A.1

MW20. Shiho Moriai and Huaxiong Wang, editors. *ASIACRYPT 2020, Part II*, volume 12492 of *LNCS*. Springer, Heidelberg, December 2020. 6.2

S+23. William Stein et al. *Sage Mathematics Software Version 9.8*. The Sage Development Team, 2023. http://www.sagemath.org. 6.1

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, 2.2, 4, 6.2

TW21. Mehdi Tibouchi and Huaxiong Wang, editors. *ASIACRYPT 2021, Part IV*, volume 13093 of *LNCS*. Springer, Heidelberg, December 2021. 6.2

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 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.
We provide our scripts computing our size estimates in Appendices C and F. The former is based off a script provided by the authors of [BS22].

A.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 two components: a blind rotation key consisting of GSW encryptions and a key-switching key. Note that since we only need one level of homomorphic evaluation, the key-switching part of the bootstrapping key is not required. Note that the blind rotation key takes the form of GSW ciphertexts over *rings*, i.e. over $\mathcal{R}_Q = \mathbb{Z}_Q[X]/(X^d + 1)$ whereas the public key/commitment is over $\mathbb{Z}_Q$.

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 technique 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 zero-knowledge proofs.

We take the gadget matrix permitting *approximate* decompositions to be

$$\boldsymbol{g} = (\lfloor Q/B \rceil, \dots, \lfloor Q/B^{\ell} \rceil)$$

for decomposition parameters B and ℓ. The encryption key used to produce the bootstrapping key for will be $\tilde{s} = \sum_{i=0}^{d-1} \tilde{s}_i \cdot X^i \leftarrow\!\!\$\ \mathcal{R}_2$ and the TLWE key will be denoted $\boldsymbol{s} := (s_0, \dots, s_{e-1}) \in \mathbb{Z}_Q^e$. For circuit privacy, we also need LWE samples over $\mathbb{Z}_Q$ with secret $\widetilde{\boldsymbol{s}} := (\tilde{s}_0, \dots, \tilde{s}_{d-1})$. We will denote error distributions for LWE/RLWE assumptions by χ and $\bar{\chi}$ respectively. For LWE samples in dimension d, we will use the RLWE distribution $\bar{\chi}$ for errors. In what follows, we use the GSW formulation from [MP21] in the bootstrapping key.

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

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

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

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

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

– **Bootstrapping Keys:** For $i = 0, \ldots, e-1$:

$$\mathsf{bsk}_i \coloneqq \left[\boldsymbol{a}_i \cdot \tilde{s} + \boldsymbol{e}_i + s_i \cdot \boldsymbol{g},\ \ \boldsymbol{a}_i' \cdot \tilde{s} + \boldsymbol{e}_i' + s_i \cdot \tilde{s} \cdot \boldsymbol{g}\right] \in R_Q^{1 \times 2\ell}$$

where $\boldsymbol{a}_i, \boldsymbol{a}_i' \leftarrow\$ R_Q^{\ell}$ are public and $\boldsymbol{e}_i, \boldsymbol{e}_i' \leftarrow\$ \bar{\chi}^{d\ell}$.

In order to set some example parameters, we rely on our 100-bit secure `PARAM_100` parameters defined in the `estimates.py` 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 in `PARAM_100` dictate $e = 900, d = 1024, \ell = 1$. Note that our description of the bootstrapping key is written in terms of R-LWE for simplicity. However, another parametrisation could use M-LWE. To translate our simplified analysis to M-LWE, one can imagine combining a sequence of ring elements as one ring element in larger dimension e.g. for M-LWE ring dimension of 512 with rank 2, one may use $d = 1024$ in our analysis to get an idea of costs.

If a random seed is expanded to generate all of the random elements, only the root public key/commitment and $\{\mathsf{bsk}_i\}$ need to be sent. Therefore, the size of the key material is

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

which is dominated by the latter term. Note that one can heavily compress this term using the techniques of [KLD+23].

Compression 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 $\log(d) \cdot \tilde{\ell}$ noisy RLWE products where $\tilde{\ell}$ is a decomposition parameter for the key-switching. Using the `PARAM_100` parameters, we have $\tilde{\ell} = 4$ and get around 14.7MB of public key material sent by the client for the OPRF, which we obtain by also dropping some lower-order bits [LDK+22]. As mentioned above, we can compress the bootstrapping key using techniques from [KLD+23]. This replaces the GSW blind rotation key with $\lceil (\ell \cdot e)/d \rceil$ RLWE ciphertexts. An additional key-switching key (or "square key") must be added which adds $\tilde{\ell}$ RLWE ciphertexts to the public key material. Since the compression increases noise, we are forced to use the larger `PARAM_100_C` parameters for OPRF correctness, resulting in 2.4MB of public key material.

Proofs. Essentially, the well-formedness of the public key material is a large "noisy" quadratic 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. $\boldsymbol{s}, \tilde{s}$, while the rest of the solution i.e. $\boldsymbol{r}_{\mathsf{com}}, \boldsymbol{e}_{\mathsf{com}}, \boldsymbol{e}_{\mathsf{pk}}, \boldsymbol{e}_i, \boldsymbol{e}'_i$ is small.

[LNP22]. In order to estimate the cost of the zero-knowledge proof, we rewrite the statement being proved in a form compatible with [LNP22]. In fact, we will need to use a larger prime proof modulus $Q' > Q$ in order to satisfy the soundness requirements of the zero-knowledge system, appealing to the techniques from Section 6.3 of [LNP22]. We begin with unknowns $\boldsymbol{s} \in \mathbb{Z}_2^e, \boldsymbol{e}_{\mathsf{pk}} \in \mathbb{Z}_Q^{\log Q}, \boldsymbol{r}_{\mathsf{com}}, \boldsymbol{e}_{\mathsf{com}} \in \mathbb{Z}_Q^{e_{\mathsf{com}}}, \tilde{s} \in \mathcal{R}_2,\ \boldsymbol{e}_i, \boldsymbol{e}'_i \in R_Q^{\ell}$. We introduce a modulus $Q' > Q$ and vectors $\boldsymbol{v}_{\mathsf{com}}, \boldsymbol{v}_{\mathsf{pk}}, \boldsymbol{v}_{\mathsf{bsk}}$, and rewrite the statement as follows:

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

$$\boldsymbol{b}_{\mathsf{pk}} - \boldsymbol{A}_{pp} \cdot \boldsymbol{r}_{\mathsf{com}} - \boldsymbol{e}_{\mathsf{com}} - \lfloor Q/2 \rceil \cdot \begin{bmatrix} \boldsymbol{s} \\ \boldsymbol{0}^{e_{\mathsf{com}}-e} \end{bmatrix} + Q \cdot \boldsymbol{v}_{\mathsf{com}} = \boldsymbol{0} \bmod Q'$$

$$\boldsymbol{c}_{\mathsf{pk}} - \boldsymbol{A}_{\mathsf{pk}} \cdot \widetilde{\boldsymbol{s}} - \boldsymbol{e}_{\mathsf{pk}} + Q \cdot \boldsymbol{v}_{\mathsf{pk}} = \boldsymbol{0} \bmod Q'$$

  where $\boldsymbol{A}_{pp}, \boldsymbol{A}_{\mathsf{pk}}, \boldsymbol{b}_{\mathsf{pk}}, \boldsymbol{c}_{\mathsf{pk}}$ are public.
- **Bootstrapping Keys:** For $i = 0, \ldots, e-1$:

$$\mathsf{bsk}_i - \left[\boldsymbol{a}_i \cdot \tilde{s} + \boldsymbol{e}_i + s_i \cdot \boldsymbol{g},\ \ \boldsymbol{a}'_i \cdot \tilde{s} + \boldsymbol{e}'_i + s_i \cdot \tilde{s} \cdot \boldsymbol{g}\right] + Q \cdot \boldsymbol{v}_{i,\mathsf{bsk}} = 0 \bmod Q'$$

  where $\mathsf{bsk}_{i,j}, \boldsymbol{a}_i, \boldsymbol{a}'_i, \boldsymbol{g}$ are public.
- **Binary Elements:** $(\boldsymbol{s}, \tilde{s})$ have binary coefficients/entries
- **Euclidean Error Bounds:**
  - $\|(\boldsymbol{r}_{\mathsf{com}}, \boldsymbol{e}_{\mathsf{com}})\| \leq \beta_{\mathsf{com}}$
  - $\|\boldsymbol{e}_{\mathsf{pk}}\| \leq \beta_{\mathsf{pk}}$
  - $\left\|\left((\boldsymbol{e}_i, \boldsymbol{e}'_i)_{i=0}^{e-1}\right)\right\| \leq \beta_{\mathsf{bsk}}$
- **Infinity Norm Bounds:**
  - $\|\boldsymbol{v}_{\mathsf{com}}\|_\infty \leq \sqrt{e_{\mathsf{com}}} \cdot \|\boldsymbol{r}_{\mathsf{com}}\| + 2 =: B_{\infty,\mathsf{com}}$
  - $\|\boldsymbol{v}_{\mathsf{pk}}\|_\infty \leq d + 1 =: B_{\infty,\mathsf{pk}}$
  - $\|(\boldsymbol{v}_{i,\mathsf{bsk}})_{i=0}^{e}\|_\infty \leq d + 2 =: B_{\infty,\mathsf{bsk}}$

Note that the above imagines that all Gaussians of the same width are sampled at once, and the infinity norm bounds on the "$\boldsymbol{v}$" vectors are due to the limited number of possible wrap-arounds modulo Q. In order to permit this rewriting, we want all equations to hold over the integers i.e. we require

$$2 \cdot Q \cdot \max\{B_{\infty,\mathsf{com}}, B_{\infty,\mathsf{pk}}, B_{\infty,\mathsf{bsk}}\} < Q'.$$

Note that the second component of the bootstrapping keys are quadratic in unknowns s_i and $\tilde{s}$. We split bsk and $\boldsymbol{v}_{i,\mathsf{bsk}}$ in half, writing $\mathsf{bsk}_i = (\mathsf{bsk}'_i, \mathsf{bsk}''_i)$ and $\boldsymbol{v}_{i,\mathsf{bsk}} = (\boldsymbol{v}'_{i,\mathsf{bsk}}, \boldsymbol{v}''_{i,\mathsf{bsk}})$. With this, we eliminate all of the $\boldsymbol{v}$ vectors apart from the latter halves of $\boldsymbol{v}_{i,\mathsf{bsk}}$ and write the system as

- **Binary Elements:** $(\boldsymbol{s}, \tilde{s}, \{\boldsymbol{r}_{i,j}\})$ have binary coefficients/entries

– **Euclidean Error Bounds:**
  - $\|(\boldsymbol{r}_{\mathsf{com}}, \boldsymbol{e}_{\mathsf{com}})\| \leq \beta_{\mathsf{com}}$
  - $\|\boldsymbol{e}_{\mathsf{pk}}\| \leq \beta_{\mathsf{pk}}$
  - $\left\|(\boldsymbol{e}_i, \boldsymbol{e}'_i)_{i=0}^{e-1}\right\| \leq \beta_{\mathsf{bsk}}$
– **Infinity Norm Bounds:**
  - $\left\|Q^{-1} \cdot \left(\boldsymbol{A}_{pp} \cdot \boldsymbol{r}_{\mathsf{com}} + \boldsymbol{e}_{\mathsf{com}} + \lfloor Q/2 \rceil \cdot \begin{bmatrix} \boldsymbol{s} \\ \boldsymbol{0}^{e_{\mathsf{com}}-e} \end{bmatrix} - \boldsymbol{b}_{\mathsf{pk}}\right)\right\|_\infty \leq B_{\infty,\mathsf{com}}$
  - $\left\|Q^{-1} \cdot (\boldsymbol{A}_{\mathsf{pk}} \cdot \boldsymbol{s} + \boldsymbol{e}_{\mathsf{pk}} - \boldsymbol{c}_{\mathsf{pk}})\right\|_\infty \leq B_{\infty,\mathsf{pk}}$
  - $\left\|Q^{-1} \cdot \left(\mathsf{bsk}'_i - \boldsymbol{a}_i \cdot \tilde{s} - \boldsymbol{e}_i - s_i \cdot \boldsymbol{g}, \quad \boldsymbol{v}''_{i,\mathsf{bsk}}\right)_{i=0}^{e-1}\right\|_\infty \leq B_{\infty,\mathsf{bsk}}$
– **Quadratic Equation:**

$$\mathsf{bsk}''_i - \boldsymbol{a}'_i \cdot \tilde{s} - \boldsymbol{e}'_i - s_i \cdot \tilde{s} \cdot \boldsymbol{g} + Q \cdot \boldsymbol{v}''_{i,\mathsf{bsk}} = 0 \bmod Q'$$

where Q^{-1} is the inverse of Q modulo Q'. We can almost use [LNP22] to prove this statement. One issue is that our system has ring dimension $d \geq 1024$ whereas the [LNP22] proof uses ring dimension $d = 128$ for efficiency. However, this is no problem as we can rewrite our ring equations in dimension 128 via simple techniques, see e.g. [LNPS21]. Another important caveat is that some of the infinity norm bounded vectors are described by equations over $\mathbb{Z}_{Q'}$ here, rather than over $\mathcal{R}_{Q'} = \mathbb{Z}_{Q'}[X]/(X^{128}+1)$. In order to use their proofs, we will embed vector entries as coefficients of ring element and use a trace operation to pick out individual coefficients. There are two small disadvantages here. The first is that the trace introduces a factor of 128 when picking out each coefficient, and the second is that we require *all* automorphisms which makes the proof slightly more expensive. Further, in the proof system, there is a soundness slack in infinity-norm bounds with factor $\psi\sqrt{\mathsf{dim}}$ where dim is the dimension of the vector/ring element, i.e. the extractable witness for $\boldsymbol{v}_{\mathsf{com}}$ has infinity norm $B_{\infty,\mathsf{com}} \cdot \psi\sqrt{e_{\mathsf{com}}}$ etc. Overall, the parameter requirements associated to the infinity norm formulation become:

– $128 \cdot Q \cdot \left(1 + \psi\sqrt{e_{\mathsf{com}}}\right) \cdot B_{\infty,\mathsf{com}} \leq Q'$
– $128 \cdot Q \cdot \left(1 + \psi\sqrt{\log(Q)}\right) \cdot B_{\infty,\mathsf{pk}} \leq Q'$
– $128 \cdot Q \cdot \left(1 + \psi\sqrt{2\ell e \cdot d}\right) \cdot B_{\infty,\mathsf{bsk}} \leq Q'$

Ciphertext Packing As mentioned before, to perform ciphertext packing, we must also include key-switching keys associated to automorphisms. This culminates in $\log(d) \cdot \tilde{\ell}$ noisy RLWE products, meaning that we must introduce an error term $\boldsymbol{e}_{\mathsf{ksk}} \in \mathcal{R}_Q^{\log(d)\cdot\tilde{\ell}}$ to the witness. These terms are Gaussians with the same parameter as $\boldsymbol{e}_{\mathsf{pk}}$, so we can bundle them together and show $\|\boldsymbol{e}_{\mathsf{pk}}, \boldsymbol{e}_{\mathsf{ksk}}\| \leq \beta'_{\mathsf{pk}}$. In terms of how the infinity norms are affected, we would introduce a vector $\boldsymbol{v}_{\mathsf{ksk}}$ with $\log(d) \cdot d \cdot \tilde{\ell}$ coefficients whose infinity norm is bounded by $B_{\infty,\mathsf{ksk}} = d+2$. The associated requirement becomes $Q \cdot \left(1 + \psi\sqrt{\log(d) \cdot d \cdot \tilde{\ell}}\right) \cdot B_{\infty,\mathsf{ksk}} \leq Q'$. Note that there is no factor 128 here because no trace is necessary in the key-switching keys. Overall, for the PARAM_100 parameters, we must pick $Q' \approx 2^{75}$ resulting in a proof size of 47MB when applying [LNP22]. The repetition rate of this proof is around 7 using proof parameters $(\gamma_1 = 41, \gamma_2 = 1.1, \gamma_e = 16, \gamma_d = 1)$.

Compression with LaBRADOR The size of proof system outlined above is heavily dominated by a vector $\boldsymbol{z}_1$ such that the verifier checks that $\boldsymbol{z}_1$ satisfies some quadratic relation and is shorter than some system-dependent bound B_z. We may therefore use the sublinear proof system LaBRADOR [BS22] instead of including $\boldsymbol{z}_1$ directly to compress sizes. A small caveat is that there is a relaxation factor of $\sqrt{128/30}$ in the proven norm and the extractable norm in LaBRADOR. Fortunately, this factor does not affect the exact Euclidean norm bounds in the original relation as these are proven using binary decompositions/proofs. As long as the factor of $\sqrt{128/30}$ does not violate any of the proof requirements in [LNP22], we still have an exact zero-knowledge proof system overall without any notion of soundness slack. Using 6 recursion levels, the zero-knowledge proof size for the PARAM_100 parameters turns out to be around 90.7kB.

Compressed bootstrapping key If we are to use [KLD+23] to compress bootstrapping keys we need to prove a statement about $\lceil(\ell \cdot e)/d\rceil + \ell$ RLWE encryptions instead of the GSW encryptions above. These take the form of key-switching keys – one of which is a “square key” which means that it is quadratic in unknowns. Note that the automorphism keys are already considered in the above analysis so we ignore them. The system is of the same form as that considered above, so we can introduce a larger modulus Q' and $\boldsymbol{v}$-vectors in order to instantiate the proof system. Ultimately, we may again eliminate all of the $\boldsymbol{v}$-vectors apart from those associated to a quadratic equation (i.e. the square key). For the PARAM_100_C parameters, we end up with proof size 137.4kB.

A.2 Ciphertexts

Computation of the function $F_{\mathsf{poprf}}.\mathsf{HEEval}$ (see Algorithm 2) requires a ciphertext encrypting $\boldsymbol{y}$ satisfying $\boldsymbol{H}_{\mathsf{inp}} \cdot \boldsymbol{G}_{\mathsf{gadget}} \cdot \boldsymbol{y} \equiv \boldsymbol{0} \bmod q$. 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. Recall that our suggested parameters are $p = 2, q = 3$ and $n_p = 256$. As before, we let Q denote the ciphertext modulus of TFHE, e the TLWE dimension and e_{com} the commitment dimension. We utilise a public key, or more specifically in our case, the commitment $\boldsymbol{b}_{\mathsf{pk}}$ to the root TFHE secret key $\boldsymbol{s} \in \mathbb{Z}_2^e$. 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:

– $\boldsymbol{b}_{\mathsf{pk}} = \boldsymbol{A}_{pp} \cdot \boldsymbol{r}_{\mathsf{com}} + \boldsymbol{e}_{\mathsf{com}} + \lfloor Q/2 \rceil \cdot \begin{bmatrix} \boldsymbol{s} \\ \boldsymbol{0} \end{bmatrix} \bmod Q$

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

We can use the proof system and formulation from [LNP22, Section 6.3] as above to instantiate a zero-knowledge proof for this relation (with a few small changes). In particular, we again utilise a prime proof system modulus Q' much larger than Q and introduce vectors $\boldsymbol{v}_{\mathsf{pk}}, \boldsymbol{v}_{\mathsf{ct}}, \boldsymbol{v}$. The system is then rewritten as:

- $\boldsymbol{b}_{\mathsf{pk}} = \boldsymbol{A}_{pp} \cdot \boldsymbol{r}_{\mathsf{com}} + \boldsymbol{e}_{\mathsf{com}} + \lfloor Q/2 \rceil \cdot \begin{bmatrix} \boldsymbol{s} \\ \boldsymbol{0} \end{bmatrix} + Q \cdot \boldsymbol{v}_{\mathsf{pk}} \bmod Q'$
- $\boldsymbol{C} = \begin{bmatrix} \boldsymbol{A}^\top \\ \boldsymbol{s}^\top \boldsymbol{A}^\top + \boldsymbol{e}^\top \end{bmatrix} + \begin{bmatrix} \boldsymbol{0}^{e \times n_p} \\ \lfloor Q/P \rceil (\boldsymbol{y})^\top \end{bmatrix} + \begin{bmatrix} \boldsymbol{0}^{e \times n_p} \\ Q \cdot \boldsymbol{v}_{\mathsf{ct}} \end{bmatrix} \bmod Q'$
- $\boldsymbol{H}_{\mathsf{inp}} \cdot \boldsymbol{G}_{\mathsf{gadget}} \cdot \boldsymbol{y} = q \cdot \boldsymbol{v} \bmod q$
- $\|\boldsymbol{v}_{\mathsf{pk}}\|_\infty \leq \sqrt{e_{\mathsf{com}}} \cdot \|\boldsymbol{r}_{\mathsf{com}}\| + 2 =: B_{\infty,\mathsf{pk}}$
- $\|\boldsymbol{v}_{\mathsf{ct}}\|_\infty \leq e + 2 =: B_{\infty,\mathsf{ct}}$
- $\|\boldsymbol{v}\|_\infty \leq n_p =: B_{\infty,v}$
- The entries of $(\boldsymbol{y}, \boldsymbol{s})$ are binary
- $\|(\boldsymbol{r}_{\mathsf{com}}, \boldsymbol{e}_{\mathsf{com}})\| \leq \beta_{\mathsf{com}}$
- $\|\boldsymbol{e}\| \leq \beta_{\mathsf{ct}}$

To ensure the validity of the above rewriting we require $2 \max\{Q \cdot B_{\infty,\mathsf{pk}}, Q \cdot B_{\infty,\mathsf{ct}}, q \cdot B_{\infty,v}\} \leq Q'$ so that there is no wrap around modulo Q'. The witness remains as $(\boldsymbol{r}_{\mathsf{com}}, \boldsymbol{e}_{\mathsf{com}}, \boldsymbol{e}, \boldsymbol{s}, \boldsymbol{y})$ as we may eliminate the "$\boldsymbol{v}$"-vectors. Doing so yields the following:

- $\left\| Q^{-1} \cdot \left(\boldsymbol{A}_{pp} \cdot \boldsymbol{r}_{\mathsf{com}} + \boldsymbol{e}_{\mathsf{com}} + \lfloor Q/2 \rceil \cdot \begin{bmatrix} \boldsymbol{s} \\ \boldsymbol{0} \end{bmatrix} - \boldsymbol{b}_{\mathsf{pk}} \right) \right\|_\infty \leq B_{\infty,\mathsf{pk}}$
- $\left\| Q^{-1} \cdot \left(\boldsymbol{s}^\top \boldsymbol{A}^\top + \boldsymbol{e}^\top \lfloor Q/2 \rceil (\boldsymbol{y})^\top - \boldsymbol{c}_1 \right) \right\|_\infty \leq B_{\infty,\mathsf{ct}}$
- $\left\| q^{-1} \cdot (\boldsymbol{H}_{\mathsf{inp}} \cdot \boldsymbol{G}_{\mathsf{gadget}} \cdot \boldsymbol{y}) \right\|_\infty \leq B_{\infty,v}$
- The entries of $(\boldsymbol{y}, \boldsymbol{s})$ are binary
- $\|(\boldsymbol{r}_{\mathsf{com}}, \boldsymbol{e}_{\mathsf{com}})\| \leq \beta_{\mathsf{com}}$
- $\|\boldsymbol{e}\| \leq \beta_{\mathsf{ct}}$

As explained for the bootstrapping key proofs, we must use the trace and take care of the soundness slack in infinity norms when applying [LNP22]. Following the same blueprint, we end up with the requirement that the following conditions are satisfied:

- $128 \cdot Q \cdot (1 + \psi \sqrt{e_{\mathsf{com}}}) \cdot B_{\infty,\mathsf{pk}} < Q'$
- $128 \cdot Q \cdot (1 + \psi \sqrt{e}) \cdot B_{\infty,\mathsf{ct}} < Q'$
- $128 \cdot q \cdot (1 + \psi \sqrt{n_q - n}) \cdot B_{\infty,v} < Q'$

Using LaBRADOR compression, the `PARAM_100` and `PARAM_100_C` parameters yield a proof size of 44.8kB and 63.0kB respectively for a single PRF evaluation. These values may be obtained by calling the `oprf_pk_proof_sizekb` function from `estimates.py`.

Amortisation. If a client wishes to evaluate the PRF at multiple values at once, it is more efficient to make a single zero-knowledge proof for the entire batch of ciphertexts. For a batch of L ciphertexts, $\boldsymbol{y}$ and $\boldsymbol{e}$ are replaced by $\boldsymbol{y}_0, \ldots, \boldsymbol{y}_{L-1}$ and $\boldsymbol{e}_0, \ldots, \boldsymbol{e}_{L-1}$. For $L = 64$ we get a proof size of 79.3kB and 116.5kB for `PARAM_100` and `PARAM_100_C` respectively. Once again, the repetition rate of the proof is around 7 using proof parameters $(\gamma_1 = 41, \gamma_2 = 1.1, \gamma_e = 16, \gamma_d = 1)$.

B 128-bit Parameters

Table 3. 128-bit Parameters

work	assumption	r	communication cost	flavour	model
Section 3	lattices, [BIP+18]	2	41.4MB + 92.9KB + 0.9KB + 45.4KB + 6.2KB	plain	malicious client, ROM
Section 3	lattices, [BIP+18]	2	41.4MB + 92.9KB + 0.9KB + 1.3KB + 6.2KB	plain	malicious client, ROM $L = 64$, per query
Section 5	heuristic	2	256KB + 41.4MB + 92.9KB + 11.1· 0.9KB + 11.1· 45.4KB + 11.1· 6.2KB	verifiable	malicious, ROM
Section 5	heuristic	2	256KB + 41.4MB + 92.9KB + 11.1· 0.9KB + 11.1· 1.3KB + 11.1· 6.2KB	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, “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. 80.6KB/64 = 1.3KB per query, when amortising over $L = 64$ queries. The factor of 11.1 accounts for the “check point” evaluations, cf. Section 5. Overall, we require amortised 8.4KB of communication cost per OPRF evaluation.

C Parameter Selection

The script is also attached.

```
"""
OPRF Parameter Selection.

LITERATURE:

[CDKS20] Chen, H., Dai, W., Kim, M., & Song, Y. (2020). Efficient homomorphic conversion
  between (ring) LWE ciphertexts. Cryptology ePrint Archive, Report 2020/015.
  https://eprint.iacr.org/2020/015

[Deo19] Deo, A. (2019). Variants of LWE: Attacks, Reductions, and a Construction. PhD
  thesis at Royal Holloway, University of London.
  https://pure.royalholloway.ac.uk/ws/portalfiles/portal/36929800/2020deoraiphd.pdf

[Kluczniak22] Kluczniak, K. (2022). Circuit privacy for FHEW/TFHE-style fully homomorphic
  encryption in practice. Cryptology ePrint Archive, Report 2022/1459.
  https://eprint.iacr.org/2022/1459
"""

from sage.all import ceil, log, sqrt, ln, pi

from utils import security_level
from estimates import HashableDict


base_sigma_r = 1.0
base_sigma = 50.0


class OPRFParams:
    def __init__(
        self,
        # TFHE
        Q=2**32,
        e=900,
        d=1024,
        sigma=base_sigma,
        pbs_base_log=12,
        pbs_level=1,
        # Circuit Privacy [Kluczniak22]
        sigma_br=base_sigma_r,
        sigma_R=base_sigma,
        R_base_log=1,
        # Compression [CDKS20]
        sigma_auto=base_sigma_r,
        auto_base_log=9,
        # OPRF
        n_p=256,
        m_p=256,
        # Other
        epsilon=2**-100,
        compress_ciphertexts=True,
        compress_br_key=False,
    ):
        """

        PARAMETERS:

        :param Q: Ciphertext modulus
        :param e: LWE dimension
        :param d: RLWE dimension, should be a power-of-two
        :param sigma: Standard deviation of error in LWE samples
        :param pbs_base_log: PBS GSW logarithm base ``B = 2**pbs_base_log``
        :param pbs_level: PBS GSW number of entries
```

```
[Kluczniak22, Thm. 1]:

:param sigma_br: Standard deviation of error in blind rotation key
:param sigma_R:  Standard deviation of error in the masking vector
:param R_base_log: Rerandomisation Gadget matrix logarithm base log

[CDKS20]:

:param sigma_auto: Standard deviation of error in automorphism key-switching key
:param auto_base_log: Automorphism key-switching key base log

:param n_p: Number of columns of OPRF secret key
:param m_p: Number of rows of OPRF secret key

:param epsilon: Target statistical distance of various quantities
:param compress_ciphertexts: Compress LWE ciphertexts into one RLWE ciphertext
:param verbose: Print additional information

[KLD+23]
:param compress_br_key: Compress blind rotation key using [KLD+23]

NOTE:

- TFHE accepts ``2**(pbs_base_log*pbs_level) < Q``, it means we're accepting noise.

"""

if R_base_log != 1:
    raise NotImplementedError("estimates.py does not consider R_base_log != 1.")

self.Q = Q
self.Q_ = 3 * (Q // 3)  # modulus switch for security
self.e = e
self.d = d
self.sigma = sigma
# TODO: add some check for pbs_base_log and pbs_level
self.pbs_base_log = pbs_base_log
self.pbs_level = (
    pbs_level if pbs_level is not None else ceil(log(Q, 2**self.pbs_base_log))
)

self.sigma_R = sigma_R
self.R_base_log = R_base_log
self.ell_R = ceil(log(Q, 2**R_base_log))

self.sigma_auto = sigma_auto
self.auto_base_log = auto_base_log
auto_base = 2**auto_base_log
self.ell_auto = ceil(log(Q, auto_base))

self.compress_br_key = compress_br_key
if compress_br_key:
    delta_auto = ceil(Q / auto_base**self.ell_auto)
    # half of brk has sigma_t σ(~), other half has sigma_h σ(^); p.18 of [EPRINT:KLDEC23]
    sigma_t = sqrt(
        d
        * (
            self.ell_auto * (auto_base**2 / 12) * d * sigma_R**2
            + (delta_auto**2 / 12) * d
        )
    )
    sigma_h = sqrt(
        sigma_t**2 * d
        + self.ell_auto * (auto_base**2 / 12) * sigma_R**2
        + (delta_auto**2 / 12) * d
    )
    # this ensures bound σ_br**2 * d = σ_t**2 * (d/2) + σ_h**2 * (d/2) in pbs_noise()
```

```python
            self.sigma_br = sqrt((sigma_t**2 + sigma_h**2) / 2)
        else:
            self.sigma_br = sigma_br

        self.input_sigma_br = sigma_br

        self.n_p = n_p
        self.m_p = m_p

        self.epsilon = epsilon
        self.delta = epsilon
        self.gamma = epsilon

        self.compress_ciphertexts = compress_ciphertexts

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

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

        Xs = ND.UniformMod(2)

        lwe = LWE.Parameters(
            n=self.e, q=self.Q, Xs=Xs, Xe=ND.DiscreteGaussian(self.sigma)
        )
        level = security_level(
            lwe,
            verbose=verbose,
        )

        if self.sigma_R != self.sigma:
            lwe = LWE.Parameters(
                n=self.e, q=self.Q, Xs=Xs, Xe=ND.DiscreteGaussian(self.sigma_R)
            )
            level = min(
                security_level(
                    lwe,
                    verbose=verbose,
                ),
                level,
            )

        lwe = LWE.Parameters(
            n=self.d, q=self.Q, Xs=Xs, Xe=ND.DiscreteGaussian(self.input_sigma_br)
        )
        level = min(
            security_level(
                lwe,
                verbose=verbose,
            ),
            level,
        )

        if self.compress_ciphertexts and self.input_sigma_br != self.sigma_auto:
            lwe = LWE.Parameters(
                n=self.d, q=self.Q, Xs=Xs, Xe=ND.DiscreteGaussian(self.sigma_auto)
```

```
        )
        level = min(
            security_level(
                lwe,
                verbose=verbose,
            ),
            level,
        )

    return level

def list_assumptions(self):
    s = []
    s.append(
        f"LWE.Parameters(n={self.e}, q=2^{float(log(self.Q,2)):.0f}, Xs=ND.UniformMod(2),"
        + f" Xe=ND.DiscreteGaussian(2^{float(log(self.sigma,2)):.1f}))"
        + " # TFHE LWE"
    )
    if self.sigma_R != self.sigma:
        s.append(
            f"LWE.Parameters(n={self.e}, q=2^{float(log(self.Q,2)):.0f}, Xs=ND.UniformMod(2),"
            + f" Xe=ND.DiscreteGaussian(2^{float(log(self.sigma_R,2)):.1f}))"
            + " # [Kluczniak22] masking vector"
        )
    s.append(
        f"LWE.Parameters(n={self.d}, q=2^{float(log(self.Q,2)):.0f}, Xs=ND.UniformMod(2),"
        + f" Xe=ND.DiscreteGaussian(2^{float(log(self.input_sigma_br,2)):.1f}))"
        + " # [Kluczniak22] BR key"
    )
    if self.compress_ciphertexts and self.input_sigma_br != self.sigma_auto:
        s.append(
            f"LWE.Parameters(n={self.d}, q=2^{float(log(self.Q,2)):.0f}, Xs=ND.UniformMod(2),"
            + f" Xe=ND.DiscreteGaussian(2^{float(log(self.sigma_auto,2)):.1f}))"
            + " # TFHE RLWE"
        )
    return "\n".join(s)

def bound(self, sigma, d=1):
    """
    Convert a standard deviation into ∞-norm bound that holds with prob ≥ 1 - ε .

    [Deo19, Cor. 1]

    :param sigma: Standard deviation per coordinate
    :param d: Number of coordinates

    """
    return float(sigma * sqrt(ln(2 * d / self.epsilon) / pi))

def packed_noise(self, err_in):
    """
    [CDKS20, Appendix A]
    """
    if self.compress_ciphertexts:
        vks = self.ell_auto * ((2**self.auto_base_log) ** 2 * self.sigma_auto**2)
        variance = (self.d**2 - 1) * vks / 3
        return err_in + self.bound(sqrt(variance), self.d)
    else:
        return err_in

@property
def noise_evolution(self):
    """
    Dictionary of values on how noise develops through the computation.
    """
    # NOTE this assumes the input is distributed as a Gaussian with standard deviation
    # σ, but we only have a ZK proof for its Euclidean norm bounds. However, just below we then
    # consider the worst-case for summing up `n_p` of those, so this confusion does
```

```python
        # not cause any issues.
        in_noise = self.bound(self.sigma, self.n_p)

        # NOTE this is worst-case
        At = self.n_p * in_noise

        CPPBS = self.pbs_noise()
        if self.Q == self.Q_:
            CPPBS_MS = CPPBS
        else:
            CPPBS_MS = self.modswitch_err(CPPBS, self.Q, self.Q_, self.d)

        # NOTE this is worst-case
        Gout = self.m_p * CPPBS_MS
        packed = self.packed_noise(Gout)

        r = {
            "in": float(log(in_noise, 2)),
            "At": float(log(At, 2)),
            "CPPBS": float(log(CPPBS, 2)),
            "CPPBS_MS": float(log(CPPBS_MS, 2)),
            "Gout": float(log(Gout, 2)),
            "packed": float(log(packed, 2)),
        }
        return r

    def cp_params(self):
        """
        Return σ_rand and σ_x as in [Kluczniak22, Thm. 1].
        """
        # NOTE this assumes the input is distributed as a Gaussian with standard deviation
        # σ, but we only have a ZK proof for Euclidean norm bounds. This should be fine,
        # given the processing we're done and the worst-case bounds we use e.g. for
        # additions
        beta_br = self.bound(self.sigma_br, self.pbs_level * self.d)

        sigma_rand_1 = 4 * ((1 - self.gamma) * (4 * self.epsilon**2)) ** (
            -1 / self.ell_R
        )
        sigma_rand_2 = (
            sqrt(1 + self.sigma_R)
            * sqrt((2**self.R_base_log) ** 2 + 1)
            * sqrt(ln(2 * self.ell_R * (1 + 1 / self.gamma)) / pi)
        )
        sigma_rand = float(max(sigma_rand_1, sigma_rand_2))

        sigma_x = float(
            sqrt(1 + beta_br)
            * sqrt(2**self.pbs_base_log + 1)
            * sqrt(ln(4 * self.e * self.d * self.pbs_level * (1 + 1 / self.delta)) / pi)
        )

        return sigma_rand, sigma_x

    def modswitch_err(self, err_in, Q, P, d):
        """

        :param err_in: Input noise
        :param Q: Input modulus
        :param P: Output modulus
        :param d: Dimension

        """
        return err_in * P / Q + d / 2

    def check(self, noise_evolution, verbose=False):
        if verbose:
            print("Noise evolution:")
```

```
        for key, val in noise_evolution.items():
            print(f"  {key:8s} : {val:.1f}")

    ms_err = self.modswitch_err(
        2 ** noise_evolution["At"], self.Q, 2 * self.d, self.e
    )
    final_err = 2 ** noise_evolution["packed"]

    check1 = ms_err < self.d / 2
    check2 = final_err < self.Q_ / 3

    if verbose:
        print(f"{str(check1):5s} :: {ms_err:.2f} < {self.d / 2.}")
        print(
            f"{str(check2):5s} :: 2^{log(final_err,2).n():.2f} < 2^{log(self.Q_ / 3.,2).n():.2f}"
        )

    return bool(check1) and bool(check2)

def pbs_noise(self):
    sigma_rand, sigma_x = self.cp_params()

    sigma_1 = sigma_rand * sqrt(1 + self.ell_R * self.sigma_R**2)
    sigma_2 = sigma_x * sqrt(1 + 2 * self.e * self.d * self.sigma_br**2)

    return self.bound(sigma_1) + self.bound(sigma_2)

def tfhe_dict(self, check=True, verbose=False):
    if check or verbose:
        passes = self.check(self.noise_evolution, verbose=verbose)
        if check and not passes:
            raise ValueError("Parameters are not valid.")

    return HashableDict(
        {
            "lwe_dimension": self.e,
            "glwe_dimension": 1,
            "polynomial_size": self.d,
            "lwe_modular_std_dev": float(self.sigma / self.Q),
            "glwe_modular_std_dev": float(self.input_sigma_br / self.Q),
            "pbs_base_log": self.pbs_base_log,
            "pbs_level": self.pbs_level,
            "ks_level": self.ell_auto,
            "ks_base_log": self.auto_base_log,
        }
    )
```

D Size Estimates

The script is also attached.

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

from dataclasses import dataclass
from functools import partial
from sage.all import sqrt, ceil, log, cached_function, round
from lnp import LNP

# Parameters


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


@dataclass
class Parameters:
    e_com: int
    sigma_com: float
    logq: int
    n: int
    n_q: int
    n_p: int
    m: int
    tfhe: HashableDict

    def __hash__(self):
        return hash(
            (
                self.e_com,
                self.sigma_com,
                self.logq,
                self.n,
                self.n_q,
                self.n_p,
                self.m,
                self.tfhe,
            )
        )


PARAM_100 = HashableDict(
    {
        "lwe_dimension": 900,
        "glwe_dimension": 1,
        "polynomial_size": 1024,
        "lwe_modular_std_dev": 1.1641532182693481e-08,
        "glwe_modular_std_dev": 2.3283064365386963e-10,
        "pbs_base_log": 12,
        "pbs_level": 1,
        "ks_level": 4,
        "ks_base_log": 9,
    }
)

OPRF_100 = Parameters(
    e_com=2048, sigma_com=2, logq=32, n=128, n_q=192, n_p=256, m=82, tfhe=PARAM_100
)

PARAM_128 = HashableDict(
    {
```

```python
        "lwe_dimension": 1200,
        "glwe_dimension": 1,
        "polynomial_size": 2048,
        "lwe_modular_std_dev": 7.450580596923828e-09,
        "glwe_modular_std_dev": 1.1641532182693481e-10,
        "pbs_base_log": 12,
        "pbs_level": 1,
        "ks_level": 4,
        "ks_base_log": 9,
    }
)

OPRF_128 = Parameters(
    e_com=2048, sigma_com=2, logq=32, n=128, n_q=192, n_p=256, m=82, tfhe=PARAM_128
)

PARAM_100_C = HashableDict(
    {
        "lwe_dimension": 2000,
        "glwe_dimension": 1,
        "polynomial_size": 2048,
        "lwe_modular_std_dev": 4.336808689942018e-19,
        "glwe_modular_std_dev": 4.336808689942018e-19,
        "pbs_base_log": 5,
        "pbs_level": 12,
        "ks_level": 12,
        "ks_base_log": 5,
    }
)

OPRF_100_C = Parameters(
    e_com=2048, sigma_com=2, logq=60, n=128, n_q=192, n_p=256, m=82, tfhe=PARAM_100_C
)


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.logq:
        raise ValueError(f"log(q) = {params.logq} is too small.")
    if -params.tfhe["glwe_modular_std_dev"] >= params.logq:
        raise ValueError(f"log(q) = {params.logq} is too small.")


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

    :param params: OPRF parameters
    :param compress_pk: drop lower-order bits where possible
    :param compress_ct1: compress ct1 into one ring element

    """
```

```python
    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 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.logq
        glwe_keep_bits = params.logq

    if compress_ct1 or compress_br:
        zeta = log(d, 2)
    else:
        zeta = 0

    if compress_br:
        zeta = zeta + 1  # square key
        brkey = d * ceil(e * ell / d) * glwe_keep_bits
    else:
        brkey = 2 * e * 2 * ell * d * glwe_keep_bits
    return _mb(
        (params.e_com + params.logq) * lwe_keep_bits
        + params.logq * lwe_keep_bits
        + brkey
        + zeta * d * ell_ * glwe_keep_bits
    )


@cached_function
def oprf_pk_proof_sizekb(
    params, compress_br=False, lnp_params=None, labrador_params=(4, 6)
):
    """
    Well-formedness proof of the OPRF public key using [LNP22] and LaBRADOR.

    :param params: OPRF parameters
    :param compress_pk: Drop lower-order bits where possible
    :param compress_ct1: Compress ct1 into one ring element
    :param lnp_params: Parameters passed to LNP22 proof system
    :param labrador_params: Parameters passed to LaBRADOR proof system

    """

    if lnp_params is None:
        lnp_params = {"logq1": round(2.4 * params.logq)}

    lnp = LNP(**lnp_params)

    logq = params.logq
    Q = 2**logq
    e = params.tfhe["lwe_dimension"]
    d_gsw = params.tfhe["polynomial_size"] * params.tfhe["glwe_dimension"]
    log_d_gsw = log(d_gsw, 2)
    # GSW ell decomposition parameter
    ell_gsw = params.tfhe["pbs_level"]
    # KSK decomposition parameter for ctxt packing i.e. tilde{ell}
    ell_ksk = params.tfhe["ks_level"]

    # commitment noise bound (rcom,params.e_com)
    b_com = params.sigma_com * sqrt(2 * params.e_com)
    if compress_br:
        # noise bound on (epk,eksk,esqk,ebsk)
        b_pk_ksk = (
            params.tfhe["glwe_modular_std_dev"]
            * Q
            * sqrt(logq + (log_d_gsw + 1) * d_gsw * ell_ksk + ell_gsw * e)
```

```python
    )
else:
    # noise bound on (epk,eksk)
    b_pk_ksk = (
        params.tfhe["glwe_modular_std_dev"]
        * Q
        * sqrt(logq + log_d_gsw * d_gsw * ell_ksk)
    )
    # bound on GSW noise (e_i,e'_i)
    b_bsk = params.tfhe["glwe_modular_std_dev"] * Q * sqrt(2 * ell_gsw * e * d_gsw)

# infinity norm bound on vcom
b_inf_com = sqrt(params.e_com) * params.sigma_com * sqrt(params.e_com) + 2
b_inf_pk = d_gsw + 1  # infinity norm bound on vpk
b_inf_bsk = d_gsw + 2  # infinity norm bound on vibsk
b_inf_ksk = d_gsw + 2  # infinity norm bound on vksk,vsqk,compressed vbsk

if compress_br:
    # ebsk is the compressed bsk i.e. ceil(ell_gsw * e / d_gsw) ring elements
    numrlwe = ceil(ell_gsw * e / d_gsw)
    # Length and size of the committed messages
    # length of s1 = (rcom,e_com,epk,eksk,ebsk,esqk,s,tilde{s},vsqk)
    m1 = (
        ceil(2 * params.e_com / lnp.d)
        + ceil((logq + (log_d_gsw + 1) * d_gsw * ell_ksk + numrlwe * d_gsw) / lnp.d)
        + ceil((e + d_gsw) / lnp.d)
        + ceil((ell_ksk * d_gsw) / lnp.d)
    )
    ell = 0  # length of m
    # norm of s1
    alpha = sqrt(
        b_com**2 + b_pk_ksk**2 + (e + d_gsw) + b_inf_ksk**2 * (ell_ksk * d_gsw)
    )

    # exact bounds beta_i to prove for i=1,2,...,ve
    bounds_to_prove = (b_com, b_pk_ksk)
    ve = len(bounds_to_prove)
    # length of a vector to prove binary coefficients
    k_bin = ceil((e + d_gsw) / lnp.d)
    # bound alpha^(e) on the vector e^(e)
    # = (rcom,e_com,epk,eksk,ebsk,esqk,s,tilde{s},vsqk)
    # bin. decomp. of b_com^2 - ||r||^2, bin. decomp. of b_pk_ksk^2 - ||e||^2,)
    alpha_e = sqrt(
        b_com**2
        + b_pk_ksk**2
        + b_inf_ksk**2 * (ell_ksk * d_gsw)
        + (k_bin + ve) * lnp.d
    )
    # length of the vector e^(e)
    ce = (
        ceil(2 * params.e_com / lnp.d)
        + ceil((logq + (log_d_gsw + 1) * d_gsw * ell_ksk + numrlwe * d_gsw) / lnp.d)
        + k_bin
        + ve
    )
    # bound alpha^(d) on the vector e^(d) = (vcom,vpk,vbsk,vksk,vsqk) from Equation 74.
    alpha_d = sqrt(
        b_inf_com**2 * (params.e_com)
        + b_inf_pk**2 * (logq)
        + b_inf_bsk**2 * (ell_gsw * e)
        + b_inf_ksk**2 * ((log_d_gsw + 1) * d_gsw * ell_ksk)
    )
else:
    # Length and size of the committed messages
    # length of s1 = (rcom,e_com,epk,eksk,ei,e'i,s,tilde{s},vppibsk)
    m1 = (
        ceil(2 * params.e_com / lnp.d)
        + ceil((logq + log_d_gsw * d_gsw * ell_ksk) / lnp.d)
```

```
        + ceil((2 * ell_gsw * e * d_gsw) / lnp.d)
        + ceil((e + d_gsw) / lnp.d)
        + ceil((ell_gsw * e * d_gsw) / lnp.d)
    )
    ell = 0  # length of m
    # norm of s1
    alpha = sqrt(
        b_com**2
        + b_pk_ksk**2
        + b_bsk**2
        + (e + d_gsw)
        + b_inf_bsk**2 * (ell_gsw * e * d_gsw)
    )

    # exact bounds beta_i to prove for i=1,2,...,ve
    bounds_to_prove = (b_com, b_pk_ksk, b_bsk)
    ve = len(bounds_to_prove)
    # length of a vector to prove binary coefficients
    k_bin = ceil((e + d_gsw) / lnp.d)
    # bound alpha^(e) on the vector e^(e)
    # = (rcom,params.e_com, epk,eksk, ei,e'i, s,tilde{s})
    # bin. decomp. of b_com^2 - ||r||^2, bin. decomp. of b_pk_ksk^2 - ||e||^2, b_bsk^2-||e||^2)
    alpha_e = sqrt(b_com**2 + b_pk_ksk**2 + b_bsk**2 + (k_bin + ve) * lnp.d)
    # length of the vector e^(e)
    ce = (
        ceil(2 * params.e_com / lnp.d)
        + ceil((logq + log_d_gsw * d_gsw * ell_ksk) / lnp.d)
        + ceil((2 * ell_gsw * e * d_gsw) / lnp.d)
        + k_bin
        + ve
    )
    # bound alpha^(d) on the vector e^(d) = (vcom, vpk,vgsw, v) from Equation 74.
    alpha_d = sqrt(
        b_inf_com**2 * (params.e_com)
        + b_inf_pk**2 * (logq)
        + b_inf_bsk**2 * (2 * ell_gsw * e * d_gsw)
        + b_inf_ksk**2 * (log_d_gsw * d_gsw * ell_ksk)
    )

sizekb, q, b_d = lnp(
    alpha,
    alpha_e,
    alpha_d,
    ell,
    m1,
    ce,
    k_bin,
    bounds_to_prove,
    do_labrador=labrador_params,
    approximate_norm_proof=True,
)

if q <= lnp.d * Q * (b_inf_com + b_d):
    print("ERROR: modulo overflow in com eqn")

if q <= lnp.d * Q * (b_inf_pk + b_d):
    print("ERROR: modulo overflow in GSW pk eqn")

if compress_br:
    if q <= lnp.d * Q * (b_inf_bsk + b_d):
        print("ERROR: modulo overflow in packed br key eqn")

    if q <= Q * (b_inf_ksk + b_d):
        print("ERROR: modulo overflow in trace/square key eqn")
else:
    if q <= lnp.d * Q * (b_inf_bsk + b_d):
        print("ERROR: modulo overflow in blind rotation pk eqn")
```

```
        if q <= Q * (b_inf_ksk + b_d):
            print("ERROR: modulo overflow in ctxt packing key eqn")

    return round(sizekb, 1)


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)


@cached_function
def oprf_ct0_proof_amortised_all_sizekb(
    params,
    L=64,
    lnp_params=None,
    labrador_params=(4, 5),
    check_security=None,
):
    """
    Well-formedness proof of OPRF request per request using [LNP22].

    :param params: OPRF parameters
    :param L: Amortise over this many ciphertexts
    :param lnp_params: Parameters passed to LNP22 proof system
    :param labrador_params: Parameters passed to LaBRADOR proof system
    :param check_security: ignored

    """
    if lnp_params is None:
        lnp_params = {"logq1": round(2.1 * params.logq)}
    lnp = LNP(**lnp_params)

    e = params.tfhe["lwe_dimension"]
    Q = 2**params.logq

    # commitment noise bound (rcom,ecom)
    b_com = 2 * sqrt(2 * params.e_com)
    # ciphertext noise bound
    b_ct = params.tfhe["lwe_modular_std_dev"] * Q * sqrt(L * params.n_p)
    # bound on v for commitment part
    b_inf_pk = sqrt(params.e_com) * 2 * sqrt(params.e_com) + 2
    b_inf_ct = e + 2  # bound on v for ciphertext part
    b_inf_v = params.n_p  # bound on v for approx norm of syndrome

    # length and size of the committed messages
    m1 = (
        ceil(2 * params.e_com / lnp.d)
        + ceil(L * e / lnp.d)
        + ceil((e + L * params.n_p) / lnp.d)
    )  # length of s1 = (rcom,ecom,e,s,y)
    ell = 0  # length of m
    alpha = sqrt(b_com**2 + b_ct**2 + (e + L * params.n_p))  # norm of s1

    # Parameters for proving norm bounds
    bounds_to_prove = (b_com, b_ct)  # exact bounds beta_i to prove for i=1,2,...,ve
    ve = len(bounds_to_prove)
    # length of a vector to prove binary coefficients
    k_bin = ceil((e + params.n_p) / lnp.d)
    # bound alpha^(e) on the vector
    # e^(e) = (rcom,ecom,e,s,y, bin. decomp. of b_com^2 - ||r||^2, bin. decomp. of b_ct^2 - ||e||^2)
    alpha_e = sqrt(b_com**2 + b_ct**2 + (k_bin + ve) * lnp.d)
    # length of the vector e^(e)
```

```python
    ce = ceil(2 * params.e_com / lnp.d) + ceil(L * e / lnp.d) + k_bin + ve
    # bound alpha^(d) on the vector e^(d) = vpk, vct, v from Equation 74.
    alpha_d = sqrt(
        b_inf_v**2 * (params.n_q - params.n)
        + b_inf_ct**2 * params.n_p
        + b_inf_pk**2 * params.e_com
    )

    sizekb, q, b_d = lnp(
        alpha,
        alpha_e,
        alpha_d,
        ell,
        m1,
        ce,
        k_bin,
        bounds_to_prove,
        do_labrador=labrador_params,
        approximate_norm_proof=True,
    )

    if q <= lnp.d * Q * (b_inf_pk + b_d):
        raise ValueError("Modulo overflow in PK equation.")

    if q <= lnp.d * Q * (b_inf_ct + b_d):
        raise ValueError("Modulo overflow in CTXT equation.")

    if q <= lnp.d * 3 * (b_inf_v + b_d):
        raise ValueError("Modulo overflow in sydrome equation.")

    return round(sizekb, 1)


oprf_ct0_proof_sizekb = partial(
    oprf_ct0_proof_amortised_all_sizekb,
    L=1,
    labrador_params=False,
)


@cached_function
def oprf_ct0_proof_amortised_sizekb(
    params,
    L=64,
):
    """
    Well-formedness proof of OPRF request per request using [LNP22].

    :param params: OPRF Parameters.
    :param L: Amortise over this many ciphertexts.

    """
    return round(
        oprf_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:
```

```python
        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)


def oprf_online_sizekb(params, compress_ct1=True, amortise=True):
    """

    :param params: OPRF parameters
    :param compress_ct1:
    :param amortise:

    """
    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 oprf_offline_sizemb(params, compress_br=False):
    r = oprf_pk_sizemb(params, compress_br=compress_br)
    r += oprf_pk_proof_sizekb(params, compress_br=compress_br) / 1024.0
    return round(r, 1)


def oprf(params, suffix="", compress_br=False, q=None):
    """

    :param params: OPRF parameters
    :param suffix: Suffix for printing
    :param compress_br: Compress blind rotation key

    """
    consistency_check(params)

    ct0_proof_amortised_all_sizekb = oprf_ct0_proof_amortised_all_sizekb(params)
    ct0_proof_amortised_sizekb = oprf_ct0_proof_amortised_sizekb(params)

    online_amortised_sizekb = oprf_online_sizekb(params)
    online_sizekb = oprf_online_sizekb(params, amortise=False)
    offline_sizemb = oprf_offline_sizemb(params, compress_br=compress_br)

    ret = f"""% OPRF{suffix} SIZES
  /oprf{suffix}/pk/sizemb/.initial={oprf_pk_sizemb(params, compress_br=compress_br)},
  /oprf{suffix}/pk/proof/sizekb/.initial={oprf_pk_proof_sizekb(params, compress_br=True)},
  /oprf{suffix}/ct0/sizekb/.initial={oprf_ct0_sizekb(params)},
  /oprf{suffix}/ct0/proof/sizekb/.initial={oprf_ct0_proof_sizekb(params)},
  /oprf{suffix}/ct0/proof/amortised-all/sizekb/.initial={ct0_proof_amortised_all_sizekb},
  /oprf{suffix}/ct0/proof/amortised/sizekb/.initial={ct0_proof_amortised_sizekb},
  /oprf{suffix}/ct1/sizekb/.initial={oprf_ct1_sizekb(params, compress_ct1=False)},
  /oprf{suffix}/ct1-compressed/sizekb/.initial={oprf_ct1_sizekb(params)},
  /oprf{suffix}/online/amortised/sizekb/.initial={online_amortised_sizekb},
  /oprf{suffix}/online/sizekb/.initial={online_sizekb},
  /oprf{suffix}/offline/sizemb/.initial={offline_sizemb},"""

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


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

```python
argsv = [
    (OPRF_100, "-100", False),
    (OPRF_128, "-128", False),
    (OPRF_100_C, "-100-c", True),
]

resv = []

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

E Size Estimates for [LNP22]

The script is also attached.

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

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


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

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

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

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

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

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

    def __call__(
        self,
        alpha,
        alpha_e,
        alpha_d,
        ell,
        m1,
        ce,
        k_bin,
        bounds_to_prove,
        gamma_1=41,
        gamma_2=1.1,
        gamma_e=16,
        gamma_d=1,
```

```
    approximate_norm_proof=True,
    do_labrador=(4, 5),
):
    """
    TODO describe function

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

    """

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

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

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

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

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

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

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

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

    # 1/ Search for dim_sis
    while True:
        dim_sis += 1
        # we use the packing optimisation from Section 5.3
```

```
    m2 = (
        dim_mlwe
        + dim_sis
        + ell
        + self.repetitions / 2
        + 256 / self.d
        + 1
        + approximate_norm_proof * 256 / self.d
        + 1
    )
    if sis_okay(m2):
        break

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

q, q1 = self.qf(gamma)

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

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

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

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

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

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

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

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

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

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

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

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

    if do_labrador:
        labrador = LaBRADOR(logq=self.logq)
        base, length = do_labrador
        labrador_size, recursion = labrador(
            n=m1 + ve,
            r=self.d / labrador.d,
            beta=bound_1 / (2 * LABRADOR_SLACK),
            base=base,
            length=length,
            verbose=get_verbose() >= 2,
        )
        labrador_saving = (
            _kb((m1 + ve) * self.d * (ceil(log(stddev_1, 2) + 2.57))) - labrador_size
        )
        sizekb = (
            _kb(full_size + challenge + short_size1 + short_size2 + hint)
            - labrador_saving
        )

    if get_verbose() >= 1:
        print(f"Proof system modulus q: {q}")
        print(f"Smallest prime divisor q_1 of q: {q1}")
        print(f"Parameter γ for dropping low-order bits of w: {gamma}")
        print(f"Parameter D for dropping low-order bits of t_A : {D}")
        print(f"Module-SIS dimension: {dim_sis}")
        print(f"Module-LWE dimension: {dim_mlwe}")
        print(f"Length of the randomness vector s2: {m2}")
        print(f"Standard deviation stddev_1: 2^{float(log(stddev_1, 2)):.2f}")
        print(f"Standard deviation stddev_2: 2^{float(log(stddev_2, 2)):.2f}")
        print(f"Standard deviation stddev_e: 2^{float(log(stddev_e, 2)):.2f}")
        print(f"Standard deviation stddev_d: 2^{float(log(stddev_d, 2)):.2f}")

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

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

    return sizekb, q, b_d

def qf(self, gamma):
```

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

F Size Estimates for LaBRADOR

The script is also attached.

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

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

LABRADOR_SLACK = float(sqrt(128 / 30))


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

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

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


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

    if b <= 2:
        return 1.0219
    elif b < 40:
        for i in range(1, len(small)):
```

```
            if small[i][0] > b:
                return small[i - 1][1]
    elif b == 40:
        return small[-1][1]
    else:
        return float(b / (2 * pi * exp(1)) * (pi * b) ** (1.0 / b)) ** (
            1.0 / (2 * b - 2.0)
        )


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

    return b


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


default_costf = adps16


@cached_function
def sis_hard_enough(kappa, eta, b, q):
    """
    Return `i` such that for `n = i ▫η ` and a sufficiently big `m` 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 = LABRADOR_SLACK,
        max_beta: int = 0,
        secpar: int = 100,
        costf=default_costf,
    ):
        self.d = d
        self.logq = logq
        self.tau = tau
        self.T = T
        self.slack = slack
        self.max_beta = max_beta
        self.secpar = secpar
        self.costf = default_costf
```

```
        block_size = None
        for block_size in range(self.secpar, 2048, 32):
            if self.costf(block_size) >= self.secpar:
                break

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

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

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

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

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

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

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

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

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

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

```python
                + 4 * self.T * self.slack * old_beta,
            )
        )
        <= kappa
    ):
        break

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

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

return size, n, r, beta

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

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

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

    for i in range(16):
        kappa = i + 1
        x = sum(r)
        n2 = x * kappa * t1 + (x**2 + x) / 2 * t2
        r2 = round(n2 / n)
        r2 = max(1, r2)

        sigz = sqrt(
            sigs[0] ** 2 * (1 + (r[0] - 1) * self.tau)
            + sum([sigs[i] ** 2 * r[i] for i in range(1, len(r))]) * self.tau
            + r2 * max(b1, b2) ** 2 / 12.0 * self.tau
        )

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

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

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

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

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

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

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

    @cached_method
    def __call__(self, n, r, beta, base, length, verbose=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])
```

G SageMath Implementation

gadget.py

The script is also attached.

```
"""
Gadget Matrices.
"""
from sage.all import ZZ, matrix, identity_matrix, vector, PolynomialRing


def gadget_matrix(n, B, ell, R=ZZ):
    Id = identity_matrix(R, n)
    g = matrix(R, 1, ell, [B**i for i in range(ell)])
    return Id.tensor_product(g)


def decompose_lwe(v, B, ell, R=ZZ):
    """

    EXAMPLE::

        >>> from sage.all import *
        >>> A = random_matrix(GF(127), 3, 4)
        >>> a = decompose_lwe(A, 2, 7)
        >>> G = gadget_matrix(4, 2, 7)
        >>> a*G.T == A
        True

    """
    try:
        _ = v[0, 0]  # is this a matrix?
        is_matrix = True
    except TypeError:
        is_matrix = False

    if is_matrix:
        return matrix(R, [decompose_lwe(v_, B, ell, R) for v_ in v.rows()])

    n = len(v)
    x = vector(ZZ, n * ell)
    for i in range(n):
        v_ = v[i].lift_centered()
        if v_ < 0:
            sgn = -1
            v_ = -v_
        else:
            sgn = 1
        for j in range(ell):
            x[i * ell + j] = sgn * (v_ % B)
            v_ = v_ // B
    return x


def decompose_rlwe(v, B, ell, d, R=PolynomialRing(ZZ, "x")):
    """

    EXAMPLE::

        >>> from sage.all import *
        >>> P = PolynomialRing(GF(127), "x")
        >>> A = matrix(P, 3, 4, [P.random_element(degree=3) for _ in range(3*4)])
        >>> a = decompose_rlwe(A, 2, 7, 4)
        >>> G = gadget_matrix(4, 2, 7)
        >>> a*G.T == A
        True
```

```
    """
    try:
        _ = v[0, 0]  # is this a matrix?
        is_matrix = True
    except TypeError:
        is_matrix = False

    if is_matrix:
        return matrix(R, [decompose_rlwe(v_, B, ell, d=d, R=R) for v_ in v.rows()])

    X = R.gen()
    n = len(v)
    w = vector(R, n * ell)
    for i in range(n):
        for j in range(d):
            v_ = v[i][j].lift_centered()
            if v_ < 0:
                sgn = -1
                v_ = -v_
            else:
                sgn = 1
            for k in range(ell):
                w[i * ell + k] += (sgn * (v_ % B)) * X**j
                v_ = v_ // B
    return w
```

tfhe.py

The script is also attached.

```
"""
Toy Implementation of TFHE.

LITERATURE:

[CGGI20] Chillotti, I., Gama, N., Georgieva, M., & Izabachène, M. (2020). TFHE: fast fully
  homomorphic encryption over the torus. Journal of Cryptology, 33(1), –3491.
  http://dx.doi.org/10.1007/s00145-019-09319-x

[Joye21] Joye, M. (2021). Guide to fully homomorphic encryption over the [discretized]
  torus. Cryptology ePrint Archive, Report 2021/1402. https://eprint.iacr.org/2021/1402

"""

from sage.all import (
    IntegerModRing,
    PolynomialRing,
    ZZ,
    ceil,
    floor,
    log,
    matrix,
    randint,
    round,
    vector,
)
from sage.stats.distributions.discrete_gaussian_integer import (
    DiscreteGaussianDistributionIntegerSampler,
)

from gadget import gadget_matrix, decompose_lwe, decompose_rlwe


def apply_plaintext_matrix(A, other, balance=True, randomize=False):
```

```
    """
    Apply this matrix to vector of elements, considering this matrix over ZZ.

    :param other: some iterable.
    :param balance: consider elements in `(-q//2, …, q//2]`.
    :param randomize: randomize with ± when p=2

    EXAMPLE::

        sage: p = 2
        sage: A = random_matrix(GF(p), 5, 8)
        sage: lwe = LWE(10, 127, "binary", 3.0, p=p)
        sage: x = random_vector(GF(p), 8)
        sage: y = A*x
        sage: c0 = [lwe(x_) for x_ in x]
        sage: c1 = apply_plaintext_matrix(A, c0)
        sage: c2 = apply_plaintext_matrix(A, c0)
        sage: c3 = apply_plaintext_matrix(A, c0, randomize=True)
        sage: z = vector(GF(p), [lwe.decrypt(c_) for c_ in c1])
        sage: y == z
        True
        sage: z = vector(GF(p), [lwe.decrypt(c_) for c_ in c2])
        sage: y == z
        True
        sage: z = vector(GF(p), [lwe.decrypt(c_) for c_ in c3])
        sage: y == z
        True
        sage: c1 == c2
        True
        sage: c2 == c3
        False

    """

    res = [0] * A.nrows()

    if randomize is True and A.base_ring().characteristic() != 2:
        raise ValueError(f"Cannot randomize signs in {A.base_ring()}.")

    for i in range(A.nrows()):
        for j in range(A.ncols()):
            if balance:
                c = A[i, j].lift_centered()
            else:
                c = A[i, j].lift()
            if randomize:
                c = (-1) ** ZZ.random_element(2) * c
            res[i] += c * other[j]
    return tuple(res)


class LWE:
    """
    LWE Distribution.

    EXAMPLE::

        sage: lwe = LWE(10, 127, "binary", 2.0)
        sage: _ = lwe()

    """

    def lift_centered(self, e):
        """
        Lift to the Integers

        :param e: some element
```

```
        EXAMPLE::

            sage: lwe = LWE(10, 127, "binary", 2.0)
            sage: c = lwe()
            sage: cz = lwe.lift_centered(c)
            sage: max(cz) <= 127//2
            True
            sage: min(cz) < 0
            True
            sage: lwe.lift_centered(GF(127)(126))
            -1
        """
        try:
            return e.lift_centered()
        except AttributeError:
            return self.R([e_.lift_centered() for e_ in list(e)])

    def copy(self, **kwds):
        """
        EXAMPLE::

            sage: lwe = LWE(10, 127)
            sage: lwe.copy(n=11, q=128)
            LWE(n=11, q=128, χ_e=2, p=2)
        """
        return LWE(
            n=kwds.get("n", self.n),
            q=kwds.get("q", self.q),
            chi_s=kwds.get("chi_s", self.chi_s),
            chi_e=kwds.get("chi_e", self.chi_e),
            p=kwds.get("p", self.p),
            s=kwds.get("s", self._s[: self.n]),
        )

    def __repr__(self):
        return f"LWE(n={self.n}, q={self.q}, χ_e={self.e.sigma:.1}, p={self.p})"

    @staticmethod
    def normalize_distribution(D):
        """
        Turn user-friendly descriptions of distributions into objects we can use.

        :param D: "binary", "ternary" or a standard deviation

        EXAMPLE::

            sage: D = LWE.normalize_distribution("binary")
            sage: max([D() for _ in range(1000)]) == 1
            True
            sage: min([D() for _ in range(1000)]) == 0
            True

            sage: D = LWE.normalize_distribution("ternary")
            sage: max([D() for _ in range(1000)]) == 1
            True
            sage: min([D() for _ in range(1000)]) == -1
            True

            sage: D = LWE.normalize_distribution(3.0)
            sage: D
            Discrete Gaussian sampler over the Integers with sigma = 3.000000 and c = 0.0...


        """
        if D == "binary":
            return lambda: randint(0, 1)
        if D == "ternary":
            return lambda: randint(-1, 1)
```

```
        try:
            return DiscreteGaussianDistributionIntegerSampler(sigma=D)
        except TypeError:
            return D

    def __init__(self, n, q, chi_s="binary", chi_e=2.0, p=2, s=None):
        """

        :param n: LWE secret dimension
        :param q: Modulus ∈ ZZ and ≥ 2
        :param chi_s: Secret distribution
        :param chi_e: Error distribution
        :param p: Plaintext modulus
        :param s: Explicitly set a secret.

        EXAMPLE::

            sage: lwe = LWE(10, 127)
            sage: lwe = LWE(10, 2^7)
            sage: lwe = LWE(10, 17, s=[0])

        """

        if p > q:
            raise ValueError(
                f"Plaintext space {p} is too big relative to ciphertext modulus {q}."
            )

        self.n = n
        self.q = q
        self.Rq = IntegerModRing(q)
        self.R = ZZ
        chi_s = self.normalize_distribution(chi_s)
        self.chi_s = chi_s
        self.phi = q
        self.e = self.normalize_distribution(chi_e)
        self.chi_e = chi_e
        if s is None:
            self._s = vector([chi_s() for _ in range(n)] + [1])
        else:
            if s == 0:
                s = [0] * self.n
            self._s = vector(list(s) + [1])
        self.p = p
        self.delta = floor(q / float(p))

    def _random_scalar(self):
        """
        Return a random scalar.

        EXAMPLE::

            sage: lwe = LWE(10, 127)
            sage: lwe._random_scalar() in GF(127)
            True

        """
        return self.Rq.random_element()

    def a(self):
        """
        Sample `a` component of an LWE ciphertext.

        EXAMPLE::

            sage: lwe = LWE(10, 127)
            sage: lwe.a() in VectorSpace(GF(127), 10)
            True
```

```
        """
        return vector([self._random_scalar() for _ in range(self.n)])

    def __call__(self, m=0, raw=False):
        """
        Encrypt `m`.

        :param m: m □ ZZ_p
        :param raw: do not encode the value (i.e. m □ ZZ_q)

        """
        a = self.a()
        e = self.e()
        b = (a * self._s[: self.n]) % self.phi + e
        if m:
            if raw:
                b += m
            else:
                m = self.R(self.R(m) % self.p)
                b += self.delta * m

        ab = vector(self.Rq, self.n + 1, list(-a) + [b])
        return ab

    def decrypt(self, c, raw=False):
        """
        Decrypt ciphertext `c`.

        :param c: LWE ciphertext
        :param raw: Do not decode.

        EXAMPLE::

        sage: lwe = LWE(10, 127, "binary", 2.0)
        sage: c = lwe(m=1)
        sage: lwe.decrypt(c)
        1

        sage: lwe = LWE(10, 127, "binary", 2.0, p=7)
        sage: c = lwe(m=6)
        sage: lwe.decrypt(c)
        6
        sage: round(lwe.decrypt(c,raw=True) / lwe.delta) % lwe.p
        6

        """
        z = self.lift_centered(c * self._s)
        if raw:
            return z
        else:
            return round(z / self.delta) % self.p


class GLWE(LWE):
    """
    Generalised LWE Distribution.
    """

    def __init__(self, d, q, k=1, chi_s="binary", chi_e=2.0, p=2, s=None):
        """
        :param d: Ring dimension, not enforced to a power of two, but it should be
        :param q: Modulus □ ZZ and ≥ 2
        :param chi_s: Secret distribution
        :param chi_e: Error distribution
        :param p: Plaintext modulus □ ZZ
        :param s: Explicitly set a secret.
```

```
        EXAMPLE::

            sage: glwe = GLWE(8, 127, 3)
            sage: glwe = GLWE(8, 2^7, 3)
            sage: glwe = GLWE(8, 17, s=[0])

        """
        self.n = k
        self.d = d
        self.q = q
        self.Rq = PolynomialRing(IntegerModRing(q), "x")
        self.R = PolynomialRing(ZZ, "x")
        self.phi = self.R.gen() ** d + 1
        chi_s = self.normalize_distribution(chi_s)
        self.e = self.normalize_distribution(chi_e)

        if s is None:
            self._s = []
            for _ in range(self.n):
                self._s += [self.R([chi_s() for _ in range(self.d)])]
            self._s += [self.R(1)]
            self._s = vector(self._s)
        else:
            self._s = vector([self.R(s_) for s_ in s] + [self.R(1)])

        try:
            self.delta = floor(q / float(p))
            self.p = p
        except:
            self.p = 2
            self.delta = floor(q / 2)

    def __repr__(self):
        """
        EXAMPLE::

            sage: GLWE(16, 127, 2)
            GLWE(d=16, q=127, χ_e=2, p=2)

        """
        return f"GLWE(d={self.d}, q={self.q}, χ_e={self.e.sigma:.1}, p={self.p})"

    def _random_scalar(self):
        """
        Return a random scalar.

        EXAMPLE::

            sage: glwe = GLWE(8, 127, 2)
            sage: glwe._random_scalar() in PolynomialRing(GF(127), "x")
            True

        """
        return self.Rq.random_element(degree=self.d - 1)

    def lift_centered(self, e):
        return self.R([e_.lift_centered() for e_ in list(e)])

    def decrypt(self, c, raw=False):
        """
        Decrypt ciphertext `c`.

        :param c: Ciphertext
        :param raw: Do not decode.

        EXAMPLE::

            sage: glwe = GLWE(4, 127, 3)
```

```
            sage: c = glwe(m=[1,0,0,1])
            sage: glwe.decrypt(c)
            x^3 + 1
            sage: glwe.decrypt(c, raw=True) # random
            63*x^3 + 63

        """
        z = self.lift_centered(c * self._s % self.phi)
        if raw:
            return z
        z = z / self.delta
        return self.R([round(z_) % self.p for z_ in list(z)])


class KeySwitching:
    """
    Switch between LWE keys.
    """

    def __init__(self, lwe_out, lwe_in, B=2):
        """
        :param lwe_out: Output LWE instance
        :param lwe_in: Source LWE instance
        :param B: Decomposition base

        """
        if lwe_in.q != lwe_out.q:
            raise ValueError(f"Modulus mismatch: {lwe_in.q} != {lwe_out.q}")
        self.B = B
        self.ell = ceil(log(lwe_in.q, B))
        self.lwe_i = lwe_in
        self.lwe_o = lwe_out
        self.ksk = self.key_gen()

    def key_gen(self):
        ksk_list = [[[] for j in range(self.ell)] for i in range(self.lwe_i.n)]
        for i in range(self.lwe_i.n):
            for j in range(self.ell):
                ksk_list[i][j] = self.lwe_o(self.lwe_i._s[i] * self.B**j, raw=True)
            ksk_list[i] = tuple(ksk_list[i])
        return tuple(ksk_list)

    def __call__(self, c):
        """
        Switch ciphertext `c` to output LWE instance.

        EXAMPLE::

            sage: lwe0 = LWE(10, 2047, p=7)
            sage: lwe1 = LWE(9, 2047, p=7)
            sage: ks = KeySwitching(lwe_out=lwe1, lwe_in=lwe0)
            sage: c0 = lwe0(5)
            sage: lwe1.decrypt(ks(c0))
            5
            sage: c1 = lwe0(2)
            sage: lwe1.decrypt(ks(c1))
            2

        """
        a, b = c[:-1], c[-1]
        ctxt_out = vector([0] * self.lwe_o.n + [b])

        for i in range(self.lwe_i.n):
            try:
                ai = decompose_lwe([a[i]], self.B, self.ell)
            except:
                ai = decompose_rlwe([a[i]], self.B, self.ell, self.lwe_i.d)
            for j in range(self.ell):
```

```
                ctxt_out += ai[j] * self.ksk[i][j]

        return ctxt_out


class GSW(LWE):
    """
    GSW Distribution.

    """

    def __init__(
        self, n, q, chi_s="binary", chi_e=2.0, B=2, p=2, s=None, force_delta=False
    ):
        """
        :param n: LWE secret dimension
        :param q: Modulus ∈ ZZ and ≥ 2
        :param chi_s: Secret distribution
        :param chi_e: Error distribution
        :param p: Plaintext modulus
        :param s: Explicitly set a secret.
        :param force_delta: throw an error if G[-1, -1] ≠δ

        EXAMPLE::

            sage: gsw = GSW(10, 128)

        """

        super().__init__(n=n, q=q, chi_s=chi_s, chi_e=chi_e, p=p, s=s)
        self.B = B
        self.ell = ceil(log(q, B))
        self.G = gadget_matrix(self.n + 1, self.B, self.ell, self.R)
        if force_delta and self.G[-1, -1] != self.delta:
            raise ValueError(f"G[-1,-1] = {self.G[-1,1]} ≠ {self.delta} = δ.")

    def base_scheme(self):
        """
        Return base LWE instance with matching parameters and secret.

        """
        lwe = LWE(self.n, self.q, p=self.p, s=self._s[:-1])
        lwe.e = self.e
        return lwe

    def __call__(self, m=0):
        """
        Encrypt `m`

        :param m: m ∈ ZZ_p

        EXAMPLE::

            sage: gsw = GSW(9, 256, B=4, p=4)
            sage: C0 = gsw(2)
            sage: C1 = gsw(3)

        """
        Z = []
        for _ in range((self.n + 1) * self.ell):
            Z.append(super().__call__())
        Z = matrix(Z)
        if m:
            m = self.R(self.R(m) % self.p)
            C = Z + m * self.G.T
        else:
            C = Z
        return C
```

```
def decrypt(self, c):
    """
    Decrypt `c`.

    :param c: Ciphertext.

    EXAMPLE::

        sage: gsw = GSW(9, 256, B=4, p=4)
        sage: gsw.decrypt(gsw(2))
        2
        sage: gsw.decrypt(gsw(3))
        3

    """
    return super().decrypt(c[-1])

def mul(self, C0, C1):
    """
    Multiply two ciphertext.

    :param C0: GSW ciphertext.
    :param C1: GSW ciphertext.

    EXAMPLE::

        sage: gsw = GSW(8, 1024, B=4, p=4)
        sage: C0 = gsw(2)
        sage: C1 = gsw(3)
        sage: C = gsw.mul(C0, C1)
        sage: gsw.decrypt(C)
        2
        sage: gsw.decrypt(gsw.mul(C1,C0))
        2
        sage: C = gsw.mul(C1, C0+C1)
        sage: gsw.decrypt(C)
        3

        sage: c1 = C1[-1] # LWE ciphertext
        sage: LWE.decrypt(gsw, gsw.mul(C0, c1))
        2

    """
    C0 = matrix(C0.base_ring(), C0.nrows(), C0.ncols(), C0.list())
    C1 = decompose_lwe(C1, self.B, self.ell, self.R)
    C = C1 * C0
    return C

def cmux(self, Cb, c0, c1):
    """
    Select `c0` or `c1` depending on bit encrypted under `Cb`.

    :param Cb: GSW ciphertext of selector bit.
    :param c0: LWE ciphertext.
    :param c1: LWE ciphertext.

    EXAMPLE::

        sage: gsw = GSW(4, 1024, chi_s="binary", B=4, p=4)
        sage: C0 = gsw(0)
        sage: C1 = gsw(1)
        sage: lwe = gsw.base_scheme()
        sage: c0 = lwe(0)
        sage: c1 = lwe(1)
        sage: lwe.decrypt(gsw.cmux(C0, c0, c1))
        0
        sage: lwe.decrypt(gsw.cmux(C1, c0, c1))
```

```
            1

        """

        return self.mul(Cb, c1 - c0) + c0


class GGSW(GLWE):
    def __init__(
        self, d, q, k, chi_s="binary", chi_e=2.0, B=2, p=2, s=None, force_delta=False
    ):
        """
        :param d: MLWE ring dimension
        :param q: Modulus ∈ ZZ and ≥ 2
        :param k: MLWE module rank
        :param chi_s: Secret distribution
        :param chi_e: Error distribution
        :param B: Decomposition base
        :param p: Plaintext modulus
        :param s: Explicitly set a secret.
        :param force_delta: throw an error if G[-1, -1] ≠δ

        EXAMPLE::

            sage: gsw = GGSW(8, 128, 2)

        """
        super().__init__(d=d, q=q, k=k, chi_s=chi_s, chi_e=chi_e, p=p, s=s)
        self.B = B
        self.ell = ceil(log(q, B))
        self.G = gadget_matrix(self.n + 1, self.B, self.ell, self.R)

        if force_delta and self.G[-1, -1] != self.delta:
            raise ValueError(f"G[-1,-1] = {self.G[-1,1]} ≠ {self.delta} = δ.")

    def base_scheme(self):
        glwe = GLWE(self.d, self.q, k=self.n, p=self.p, s=self._s[: self.n])
        glwe.e = self.e
        return glwe

    def __call__(self, m=None):
        Z = []
        for _ in range((self.n + 1) * self.ell):
            Z.append(super().__call__())
        Z = matrix(self.Rq, Z)
        if m:
            m = self.R(self.R(m) % self.p)
            C = Z + (m * self.G.T) % self.phi
        else:
            C = Z
        return C

    def decrypt(self, c):
        return super().decrypt(c[-1])

    def mul(self, C0, C1):
        """
        EXAMPLE::

            sage: ggsw = GGSW(4, 2**10, 3, B=4, p=4)
            sage: C0 = ggsw([1,1,0,0])
            sage: C1 = ggsw([3,1,0,0])
            sage: ggsw.decrypt(C0)
            x + 1
            sage: ggsw.decrypt(C1)
            x + 3
            sage: C = ggsw.mul(C0, C1)
            sage: ggsw.decrypt(C)
```

```
            x^2 + 3

            sage: c1 = C1[-1] # GLWE ciphertext
            sage: GLWE.decrypt(ggsw, ggsw.mul(C0, c1))
            x^2 + 3

        """
        C0 = matrix(C0.base_ring(), C0.nrows(), C0.ncols(), C0.list())
        C1 = decompose_rlwe(C1, self.B, self.ell, self.d, self.R)
        C = (C1 * C0) % self.phi
        return C

    def cmux(self, Cb, c0, c1):
        """

        :param Cb:
        :param c0:
        :param c1:

        EXAMPLE::

            sage: ggsw = GGSW(6, 1024, k=2, B=4, p=4)
            sage: C0 = ggsw(0)
            sage: C1 = ggsw(1)
            sage: glwe = ggsw.base_scheme()
            sage: c2 = glwe(2)
            sage: c3 = glwe(3)
            sage: glwe.decrypt(ggsw.cmux(C0, c2, c3))
            2
            sage: glwe.decrypt(ggsw.cmux(C1,c2, c3))
            3
        """
        return self.mul(Cb, c1 - c0) + c0


class BlindRotation(GGSW):
    def __init__(self, lwe, d=1024, k=1, chi_s="binary", chi_e=2.0, B=2, p=2, s=None):
        """
        :param lwe: LWE instance
        :param d: MLWE ring dimension
        :param q: Modulus ▫ ZZ and ≥ 2
        :param k: MLWE module rank
        :param chi_s: Secret distribution
        :param chi_e: Error distribution
        :param B: Decomposition base
        :param p: Plaintext modulus
        :param s: Explicitly set a secret.

        """
        super().__init__(d, lwe.q, k, chi_s, chi_e, B, p, s)
        self.lwe = lwe
        self.brk = self.key_gen()

    def key_gen(self):
        brk_list = []
        for j in range(self.lwe.n):
            brk_list.append(super().__call__(self.lwe._s[j]))  # scalar -> constant coeff
        return brk_list

    def modulus_switch(self, ctxt):
        """

        EXAMPLE::

            sage: br = BlindRotation(LWE(n=10, q=2^10, p=2))
            sage: N = 2*br.d
            sage: ctxt = br.modulus_switch(br.lwe(1))
            sage: round((ctxt * br.lwe._s).lift() / (N/br.lwe.p))
```

```
        1
        sage: ctxt = br.modulus_switch(br.lwe(0))
        sage: round((ctxt * br.lwe._s).lift() / (N/br.lwe.p)) % br.lwe.p
        0

    """
    N = ZZ(self.d)
    q = ZZ(self.q)
    ctxt = ctxt.lift()
    ctxt = vector(ZZ, [round(2 * N / q * c_) for c_ in ctxt])
    ctxt = ctxt.change_ring(IntegerModRing(2 * N))
    return ctxt

def test_polynomial(self):
    """
    Return the standard test polynomial

    :param fun:

    EXAMPLE::

        sage: br = BlindRotation(LWE(n=10, q=2^10, p=2), d=8)
        sage: br.test_polynomial()
        512*x^5 + 512*x^4 + 512*x^3 + 512*x^2

    """
    p = ZZ(self.p)
    N = ZZ(self.d)

    v = []
    for j in range(self.d):
        j = (j + self.d // 4) % self.d
        v.append(self.delta * (round((p * j) / (2 * N)) % p))
    return self.Rq(v) % self.phi

def __call__(self, c, v=None, in_clear=False):
    """
    Perform blind rotation.

    :param ctxt:
    :param v: Custom test polynomial

    EXAMPLE::

        sage: br = BlindRotation(LWE(n=2, q=2^30, p=4), d=1024, k=2)
        sage: c1 = br(br.lwe(1))
        sage: GLWE.decrypt(br, c1)[0]
        1
        sage: c0 = br(br.lwe(0))
        sage: GLWE.decrypt(br, c0)[0]
        0

        sage: br = BlindRotation(LWE(n=10, q=2^20, p=4), d=32, k=2)
        sage: all([GLWE.decrypt(br, br(br.lwe(1)))[0] == 1 for _ in range(32)])
        True

        sage: all([GLWE.decrypt(br, br(br.lwe(0)))[0] == 0 for _ in range(32)])
        True


    """

    c = self.modulus_switch(c)
    a, b = c[: self.lwe.n], c[self.lwe.n]
    if v is None:
        v = self.test_polynomial()

    X = self.R.gen()
```

```
        acc = X ** (-b) * vector(self.Rq, [0] * self.n + [v]) % self.phi
        for j in range(self.lwe.n):
            if in_clear:
                if self.lwe._s[j] == 1:
                    acc = (X ** (-a[j]) * acc) % self.phi
            else:
                c0 = acc
                c1 = (X ** (-a[j]) * acc) % self.phi
                acc = self.cmux(self.brk[j], c0, c1)
        return acc


class ModSwitchBootstrap(BlindRotation):
    def __init__(
        self,
        lwe,
        d=1024,
        k=1,
        chi_s="binary",
        chi_e=2.0,
        B=2,
        p_in=2,
        p_out=3,
        s=None,
        ks=False,
    ):
        super().__init__(lwe, d, k, chi_s, chi_e, B, p_in, s=None)
        self.p_out = p_out
        glwesec = []
        for i in range(k):
            rlwesec = self._s[i].coefficients(sparse=False)
            glwesec += rlwesec + [0] * (d - len(rlwesec))
        self.glwesec = glwesec
        self.ksbool = ks
        if ks:
            self.lwe_o = self.lwe.copy(p=p_out)
            self.ks = KeySwitching(self.lwe_o, self.lwe.copy(n=k * d, p=p_out, s=glwesec))
        else:
            self.lwe_o = self.lwe.copy(n=k * d, p=p_out, s=self.glwesec)

    def sample_extract(self, rotated_ct):
        N = self.d
        original_a = []
        for i in range(self.n):
            this_a = rotated_ct[i].coefficients(sparse=False)
            avec = [this_a[0]] + [-1 * this_a[N - i] for i in range(1, N)]
            original_a += avec

        b = rotated_ct[self.n].coefficients(sparse=False)[0]
        return vector(IntegerModRing(self.q), list(original_a) + [b])

    def __call__(self, c):
        """
        Switch from mod 2 to mod 3

        :param c: LWE ciphertext with plaintext modulus 2

        EXAMPLE::

            sage: msbs = ModSwitchBootstrap(LWE(n=20,q=2^10,p=2),d=64, k=3)
            sage: ct0 = msbs(msbs.lwe(0))
            sage: msbs.lwe_o.decrypt(ct0)
            0
            sage: ct1 = msbs(msbs.lwe(1))
            sage: msbs.lwe_o.decrypt(ct1)
            1

        """
```

```
        delta = self.q // self.p_out
        N = self.d
        v_modswitch = [delta] * (N // 2) + [-delta] * (N // 2)
        c = super().__call__(c, v_modswitch)
        c = self.sample_extract(c)
        if self.ksbool:
            c = self.ks(c)
            return c - vector([0] * self.lwe.n + [delta])
        else:
            return c - vector([0] * (len(c) - 1) + [delta])
```

compression.py

The script is also attached.

```
"""
FHE Ciphertext Compression.

LITERATURE:

[ACNS:CDKS21] Chen, H., Dai, W., Kim, M., & Song, Y. (2021). Efficient homomorphic
  conversion between (ring) LWE ciphertexts. In K. Sako, & N. O. Tippenhauer, ACNS 21, Part
  I (pp. -460479). : Springer, Heidelberg.

"""
from sage.all import PolynomialRing, IntegerModRing, ZZ, vector, ceil, log, matrix
from tfhe import GLWE, KeySwitching, BlindRotation, ModSwitchBootstrap
from gadget import gadget_matrix, decompose_rlwe


class Automorphisms(GLWE):
    def __init__(self, d, q):
        """
        :param d: Degree of ring.
        :param q: Modulus.

        """
        self.d = d
        self.q = q
        self.Rq = PolynomialRing(IntegerModRing(q), "x")
        self.R = PolynomialRing(ZZ, "x")
        self.phi = self.R.gen() ** d + 1

    def __call__(self, p, t, centered=True):
        """
        Apply `X -> X^t` on polynomial ``p``.

        :param p: A polynomial or vector of polynomials
        :param t: The power to which X is raised
        :param centered: Output centered representation

        EXAMPLE::

            sage: A = Automorphisms(8,27)
            sage: p = A.Rq([1,2,3,4])
            sage: p
            4*x^3 + 3*x^2 + 2*x + 1
            sage: A(p,3)
            3*x^6 + 2*x^3 - 4*x + 1

            sage: q = A.Rq([0]*4+[1,2,3,4])
            sage: A([p,q],3)
            (3*x^6 + 2*x^3 - 4*x + 1, -2*x^7 + 4*x^5 - x^4 + 3*x^2)
        """
        try:
```

```
            _ = len(p)
            return vector([self.__call__(pi, t, centered) for pi in p])
        except TypeError:
            r = self.Rq((p(self.R.gen() ** t) % self.phi))
            if centered:
                return self.lift_centered(r)
            else:
                return r


class AutoKeySwitching(KeySwitching):
    def __init__(self, rlwe, B=2):
        self.B = B
        self.ell = ceil(log(rlwe.q, B))
        self.rlwe = rlwe
        self.autos = Automorphisms(rlwe.d, rlwe.q)
        self.auto_s = []
        self.auto_ksk = self.key_gen()

    def key_gen(self):
        ak = []
        rlwe = self.rlwe

        self.lwe_o = rlwe
        powers = [rlwe.d / 2**t + 1 for t in range(0, log(rlwe.d, 2))]
        for power in powers:
            self.lwe_i = GLWE(
                d=rlwe.d, q=rlwe.q, k=rlwe.n, p=rlwe.p, s=[self.autos(rlwe._s[0], power)]
            )
            akt = super().key_gen()[0]
            self.auto_s.append(self.autos(rlwe._s[0], power))
            ak.append(akt)
        return ak

    def eval_auto(self, c, t):
        c_auto = self.autos(c, t, False)
        a, b = self.rlwe.Rq(c_auto[0]), self.rlwe.Rq(c_auto[1])

        ctxt_out = vector([0] * self.rlwe.n + [b])
        adec = decompose_rlwe([a], self.B, self.ell, self.rlwe.d)
        t_ind = ZZ(log(self.rlwe.d / (t - 1), 2))
        for j in range(self.ell):
            ctxt_out += (adec[j] * self.auto_ksk[t_ind][j]) % self.rlwe.phi

        return ctxt_out

    def trace_eval(self, c, n=1):
        """
        Evaluate homomorphic trace i.e. zero all non-constant coeffs (adding a factor d)

        :param c: A rlwe ciphertext
        :param n: If n>1 only clear part of non-constant coefficients with smaller factor

        EXAMPLE::

            sage: glwe = GLWE(4, 2**32-1, 1, p=16, s=[[1,1]])
            sage: c0 = glwe([7, 1,1,1])
            sage: AK = AutoKeySwitching(glwe)
            sage: glwe.decrypt(AK.trace_eval(c0))
            12
            sage: (7 * 4) % 16
            12

            sage: glwe.decrypt(AK.trace_eval(c0, 2))
            2*x^2 + 14

        """
        ctxt = c
```

```
        rlwe = self.rlwe
        powers = [rlwe.d / 2**t + 1 for t in range(0, log(rlwe.d / n, 2))]
        for power in powers:
            c_auto = self.eval_auto(ctxt, power)
            ctxt += c_auto

        return vector(self.rlwe.Rq, ctxt)

    def coeff_extract(self, c):
        """
        Extract d rlwe ciphertexts from a single rlwe ciphertext. The d ciphertexts
        encrypt the coefficients of the input rlwe ciphertext with a factor of d.

        :param c: A rlwe ciphertext to extract

        EXAMPLE::

            sage: glwe = GLWE(4, 2**32-1, 1, p=16)
            sage: c0 = glwe([2,1,3,4])
            sage: AK = AutoKeySwitching(glwe)
            sage: ct_list = AK.coeff_extract(c0)
            sage: [glwe.decrypt(ct_list[i]) for i in range(4)]
            [8, 4, 12, 0]


        """
        rlwe = self.rlwe
        res = [vector([]) for _ in range(rlwe.d)]
        res[0] = c
        powers = [rlwe.d / 2**t + 1 for t in range(log(rlwe.d, 2))]
        for power in powers:
            for j in range(rlwe.d / (power - 1)):
                old = res[j]
                tmp = self.eval_auto(old, power)
                res[j] = old + tmp
                poX = -1 * self.rlwe.Rq.gen() ** (rlwe.d - rlwe.d / (power - 1)) % self.rlwe.phi
                res[j + rlwe.d / (power - 1)] = poX * (old - tmp) % self.rlwe.phi

        return res


class PackedBlindRotation(BlindRotation):
    """
    Blind rotation via a compressed blind rotation key. The constructor generates
    a compressed key and then decompresses it.

    EXAMPLE::

        sage: from tfhe import LWE
        sage: pbr = PackedBlindRotation(LWE(n=2, q=2^15-1, p=4), d=32)
        sage: c1 = pbr(pbr.lwe(1))
        sage: GLWE.decrypt(pbr, c1)[0]
        1
        sage: c0 = pbr(pbr.lwe(0))
        sage: GLWE.decrypt(pbr, c0)[0]
        0
        sage: all([GLWE.decrypt(pbr, pbr(pbr.lwe(1)))[0] == 1 for _ in range(32)])
        True
        sage: all([GLWE.decrypt(pbr, pbr(pbr.lwe(0)))[0] == 0 for _ in range(32)])
        True

    """

    def __init__(self, lwe, d=1024, k=1, chi_s="binary", chi_e=2.0, B=2, p=2, s=None):
        """

        :param lwe: LWE instance
        :param d: MLWE ring dimension
```

```
    :param q: Modulus □ ZZ and ≥ 2
    :param k: MLWE module rank
    :param chi_s: Secret distribution
    :param chi_e: Error distribution
    :param B: Decomposition base
    :param p: Plaintext modulus
    :param s: Explicitly set a" secret.

    """
    super(BlindRotation, self).__init__(d, lwe.q, k, chi_s, chi_e, B, p, s)
    self.lwe = lwe
    self.rlwe = self.base_scheme()
    self.auto_ks = AutoKeySwitching(self.rlwe, B)
    self.sqk = self.create_square_key()
    self.compressed_brk = self.key_gen()
    self.brk = self.decompress()

def key_gen(self):
    full_list = []
    lwe_n = self.lwe.n
    invN = self.Rq(1 / (self.d))
    gadget_vec = gadget_matrix(1, self.B, self.ell)
    for i in range(lwe_n):
        full_list += list(invN * self.lwe._s[i] * gadget_vec)[0]

    brk_list = []
    for i in range((self.ell * lwe_n) // self.d + 1):
        this_poly = self.rlwe.Rq(full_list[i * self.d : (i + 1) * self.d])
        brk_list.append(self.rlwe(this_poly, raw=True))

    return brk_list

def create_square_key(self):
    rlwe = self.rlwe
    square = (rlwe._s[0]) ** 2 % self.phi
    square_key = []
    for i in range(self.ell):
        square_key.append(rlwe(square * self.B**i, raw=True))

    return matrix(square_key)

def eval_square_mult(self, ct):
    """
    Send a ciphertext encrypting `m` to a ciphertext encrypting `□sm`

    :param ct: RLWE ciphertext

    EXAMPLE::

        sage: from tfhe import LWE
        sage: lwe = LWE(4, 2**15-3, p=16)
        sage: pbr = PackedBlindRotation(lwe, 4, s=[(1,0,1,1)], p=16)
        sage: pbr.rlwe._s[0]
        x^3 + x^2 + 1
        sage: c0 = pbr.rlwe([0,2,0,3])
        sage: pbr.rlwe.decrypt(pbr.eval_square_mult(c0))
        5*x^3 + 13*x^2 + 15*x + 14
        sage: (pbr.rlwe.decrypt(c0) * pbr.rlwe._s[0]) % pbr.rlwe.phi
        5*x^3 - 3*x^2 - x - 2

    """
    a, b = ct[0], ct[1]
    adec = decompose_rlwe([a], self.B, self.ell, self.d)
    return vector((vector([b, 0]) + adec * self.sqk) % self.rlwe.phi)

def decompress(self):
    cbrk_list = self.compressed_brk
    ct_list = []
```

```
        for ct in cbrk_list:
            ct_list += self.auto_ks.coeff_extract(ct)
        # encryption of si*B**j is i*ell+jth element
        sqct_list = []
        for ct in ct_list:
            sqct_list.append(self.eval_square_mult(ct))

        # now arrange as RGSW ctxts of si
        brk_list = []
        ell = self.ell
        for i in range(self.lwe.n):
            rgswi = sqct_list[i * ell : (i + 1) * ell] + ct_list[i * ell : (i + 1) * ell]
            brk_list.append(matrix(rgswi))

        return tuple(brk_list)


class PackedModSwitchBootstrap(ModSwitchBootstrap, PackedBlindRotation):
    pass


class CiphertextCompression(AutoKeySwitching):
    def __init__(self, lwe, d, B=2):
        self.lwe = lwe
        self.autos = Automorphisms(d, lwe.q)
        self.rlwe = GLWE(d, lwe.q, p=lwe.p, s=[self.convert_secret(d, lwe._s[: lwe.n])])
        super().__init__(self.rlwe, B)

    def convert_secret(self, d, lwe_s):
        """
        Convert an LWE secret to a RLWE secret for the LWE to RLWE conversion.
        This conversion means the same automorphism key is used for key and
        ciphertext compression.

        :param d: The ring dimension
        :param lwe_s: An lwe secret

        EXAMPLE::

            sage: from tfhe import LWE
            sage: lwe = LWE(4, 2**15-3, p=16, s=(1,0,1,0))
            sage: cc = CiphertextCompression(lwe, 4)
            sage: cc.convert_secret(4,lwe._s[:lwe.n])
            x^2 + 1
            sage: lwe._s[:lwe.n]
            (1, 0, 1, 0)

        """
        R = self.autos.R
        ring_s = R(list(lwe_s))
        return ring_s

    def pack_lwes(self, ct_list):
        """
        Pack a list of rlwe ciphertexts (output from to_rlwe function) into a single
        rlwe ciphertext.

        EXAMPLE::

            sage: from tfhe import LWE
            sage: lwe = LWE(4, 2**15-3, p=16)
            sage: cc = CiphertextCompression(lwe, 4)
            sage: ct_list = [cc.to_rlwe(lwe(i)) for i in range(4)]
            sage: cc.rlwe.decrypt(cc.pack_lwes(ct_list))
            12*x^3 + 8*x^2 + 4*x

        """
        ell = log(len(ct_list), 2)
```

```
        N = self.rlwe.d
        if ell == 0:
            return ct_list[0]
        else:
            ct_even = self.pack_lwes([ct_list[2 * j] for j in range(2 ** (ell - 1))])
            ct_odd = self.pack_lwes([ct_list[2 * j + 1] for j in range(2 ** (ell - 1))])
            poX = self.rlwe.Rq.gen() ** (N / (2**ell)) % self.rlwe.phi
            term1 = (ct_even + poX * ct_odd) % self.rlwe.phi
            term2 = self.eval_auto((ct_even - poX * ct_odd) % self.rlwe.phi, 2**ell + 1)
            ct = (term1 + term2) % self.rlwe.phi
            return ct

    def to_rlwe(self, lwe_ct):
        """
        Embed a lwe ciphertext into an rlwe one. The output ciphertext encrypts the
        lwe plaintext in its constant. This uses a LWE to RLWE TFHE transform
        so that the same automorphism key can be used for key compression and
        ciphertext compression.

        EXAMPLE::

            sage: from tfhe import LWE
            sage: lwe = LWE(4, 2**15-3, p=16)
            sage: cc = CiphertextCompression(lwe, 4)
            sage: cc.rlwe.decrypt(cc.to_rlwe(lwe(3)))[0]
            3

        """
        n = self.lwe.n
        d = self.rlwe.d
        a, b = list(lwe_ct[:n]) + [0] * (d - n), lwe_ct[n]
        R = self.autos.R
        phi = R.gen() ** d + 1
        ring_a = R(list(a))
        a = R((ring_a(-R.gen() ** (d - 1)) % phi))
        return vector([self.rlwe.Rq(a), self.rlwe.Rq(b)])

    def lwes_to_rlwe(self, ct_list):
        """
        Use ``to_rlwe`` and ``pack_lwes`` to pack many LWE ciphertexts into a single RLWE
        ciphertext. The output plaintext contains the lwe plaintexts in the non-zero
        coefficients (multiplied by N mod p).

        :param ct_list:

        EXAMPLE::

            sage: from tfhe import LWE
            sage: lwe = LWE(8, 2**15, p=3)
            sage: cc = CiphertextCompression(lwe, 8)
            sage: cc.rlwe.decrypt(cc.lwes_to_rlwe(list(map(lwe,[1,1,1,1, 1,1,1,1]))))
            2*x^7 + 2*x^6 + 2*x^5 + 2*x^4 + 2*x^3 + 2*x^2 + 2*x + 2
            sage: cc.rlwe.decrypt(cc.lwes_to_rlwe(list(map(lwe,[0,1,1,1, 1,1,1,1]))))
            2*x^7 + 2*x^6 + 2*x^5 + 2*x^4 + 2*x^3 + 2*x^2 + 2*x
            sage: cc.rlwe.decrypt(cc.lwes_to_rlwe(list(map(lwe,[0,1,0,1, 0,1,0,1]))))
            2*x^7 + 2*x^5 + 2*x^3 + 2*x
            sage: cc.rlwe.decrypt(cc.lwes_to_rlwe(list(map(lwe,[0,2,0,2, 0,2,0,2]))))
            x^7 + x^5 + x^3 + x

        """
        n = len(ct_list)
        rlwe_ct_list = []
        for ct in ct_list:
            rlwe_ct_list.append(self.to_rlwe(ct))

        ct = self.pack_lwes(rlwe_ct_list)

        return self.trace_eval(ct, n)
```

cpbs.py

The script is also attached.

```python
"""
Circuit Private Bootstrapping.

[Kluczniak22] Kluczniak, K. (2022). Circuit privacy for FHEW/TFHE-style fully homomorphic
  encryption in practice. Cryptology ePrint Archive, Report 2022/1459.
  https://eprint.iacr.org/2022/1459

"""

from sage.all import (
    IntegerModRing,
    PolynomialRing,
    ZZ,
    identity_matrix,
    ceil,
    floor,
    log,
    matrix,
    randint,
    round,
    vector,
)

from sage.stats.distributions.discrete_gaussian_integer import (
    DiscreteGaussianDistributionIntegerSampler,
)
from sage.stats.distributions.discrete_gaussian_lattice import (
    DiscreteGaussianDistributionLatticeSampler,
)

from gadget import gadget_matrix
from tfhe import GSW, GGSW, BlindRotation, ModSwitchBootstrap
from compression import PackedBlindRotation


class GadgetPreimageSampler:
    """
    Samples `x` such that `G □ x = u mod q` for x in `[0,Bℓ-1]^`, u in ZZ
    """

    def __init__(self, B=2, q=2**5, sigma=2.0):
        self.B = B
        self.q = q
        self.ell = ceil(log(q, B))
        self.sigma = sigma
        self.G = gadget_matrix(1, B, self.ell, ZZ)
        self.basis = matrix(self.gadget_basis())

    def gadget_basis(self):
        """
        Gadget basis from MP12
        """
        basis_vectors = []
        for i in range(self.ell - 1):
            basis_vectors += [vector([0] * i + [self.B, -1] + [0] * (self.ell - i - 2))]

        if log(self.q, self.B) in ZZ:
            basis_vectors += [vector([0] * (self.ell - 1) + [self.B])]
        else:
            basis_vectors += [self.q.digits(self.B)]

        return basis_vectors

    def __call__(self, v):
```

```
        """
        sage: gadget = GadgetPreimageSampler(q=3*2**4)
        sage: P.<x> = PolynomialRing(IntegerModRing(gadget.q))
        sage: u = x^2 + x + 1
        sage: v = 34*x^3 + 23*x^2 + 5*x + 10
        sage: G = identity_matrix(ZZ,2).tensor_product(gadget.G)
        sage: G * gadget([u,v]).change_ring(P)
        (x^2 + x + 1, 34*x^3 + 23*x^2 + 5*x + 10)

        """
        try:
            _ = v[0, 0]  # is this a matrix?
            try:
                _ = v[0, 0].coefficients()
                R = PolynomialRing(ZZ, "x")
            except (AttributeError, IndexError):
                R = ZZ
            is_matrix = True
        except TypeError:
            try:
                _ = v[0].coefficients()
                R = PolynomialRing(ZZ, "X")
            except (AttributeError, IndexError):
                R = ZZ
            is_matrix = False

        if is_matrix:
            return matrix(R, [self.__call__(v_) for v_ in v.rows()])

        if R == ZZ:
            preimages = []
            for vi in v:
                coset = vector(ZZ, ZZ(vi).digits(self.B))
                if len(coset) != self.ell:
                    coset = vector(ZZ, list(coset) + [0] * (self.ell - len(coset)))
                D = DiscreteGaussianDistributionLatticeSampler(
                    self.basis, self.sigma, -1 * coset
                )
                preimages += list(D() + coset)
            return vector(ZZ, preimages)

        else:
            # the base ring of u is ZZ[X]
            X = R.gen()
            preimages = []
            for vi in v:
                coeffs = vector(ZZ, vi.coefficients(sparse=False))
                deg = len(coeffs)
                zz_preimage = list(self.__call__(coeffs))
                rearranged = matrix(
                    deg, self.ell, zz_preimage
                ).T  # a row is coeffs of a polynomial
                this_preimage = rearranged * vector([X**i for i in range(deg)])
                preimages += list(this_preimage)
            return vector(R, preimages)


class CPGSW(GSW):
    """
    Circuit-private GSW.
    """

    def __init__(
        self,
        n,
        q,
        chi_s="binary",
        chi_e=2.0,
```

```
        B=2,
        p=2,
        s=None,
        sigma_x=1.0,
        force_delta=False,
    ):
        super().__init__(
            n=n, q=q, chi_s=chi_s, chi_e=chi_e, B=B, p=p, s=s, force_delta=force_delta
        )
        self.gadget = GadgetPreimageSampler(B, q, sigma_x)
        self.sigma_x = sigma_x

    def mul(self, C0, C1):
        """
        Multiply two ciphertexts.

        :param C0: GSW ciphertext.
        :param C1: GSW ciphertext.

        EXAMPLE::

            sage: gsw = CPGSW(8, 2**12, B=4, p=4)
            sage: C0 = gsw(2)
            sage: C1 = gsw(3)
            sage: C = gsw.mul(C0, C1)
            sage: gsw.decrypt(C)
            2
            sage: gsw.decrypt(gsw.mul(C1,C0))
            2
            sage: C = gsw.mul(C1, C0+C1)
            sage: gsw.decrypt(C)
            3

        """
        C0 = matrix(C0.base_ring(), C0.nrows(), C0.ncols(), C0.list())
        C1 = self.gadget(C1)
        C = C1 * C0
        return C


class CPGGSW(GGSW):
    """
    Circuit-private GGSW.
    """

    def __init__(self, d, q, k, chi_s="binary", chi_e=2.0, B=2, p=2, s=None, sigma_x=1.0):
        super().__init__(d, q, k, chi_s, chi_e, B, p, s)
        self.gadget = GadgetPreimageSampler(B, q, sigma_x)
        self.sigma_x = sigma_x

    def mul(self, C0, C1):
        """
        EXAMPLE::

            sage: from tfhe import *
            sage: ggsw = GGSW(4, 2**10, 1, B=4, p=4)
            sage: C0 = ggsw([1,1,0,0])
            sage: C1 = ggsw([3,1,0,0])
            sage: ggsw.decrypt(C0)
            x + 1
            sage: ggsw.decrypt(C1)
            x + 3
            sage: C = ggsw.mul(C0, C1)
            sage: ggsw.decrypt(C)
            x^2 + 3

            sage: c1 = C1[-1] # GLWE ciphertext
            sage: GLWE.decrypt(ggsw, ggsw.mul(C0, c1))
```

```
            x^2 + 3

        """
        C0 = matrix(C0.base_ring(), C0.nrows(), C0.ncols(), C0.list())
        C = (self.gadget(C1).change_ring(C0.base_ring()) * C0) % self.phi
        return C


class CPBlindRotation(BlindRotation, CPGGSW):
    """
    Perform blind rotation with circuit privacy.

    EXAMPLE::

        sage: from tfhe import *
        sage: cpbr = CPBlindRotation(LWE(n=2, q=2^21, p=4), d=32, k=2)
        sage: c1 = cpbr(cpbr.lwe(1))
        sage: GLWE.decrypt(cpbr, c1)[0]
        1
        sage: c0 = cpbr(cpbr.lwe(0))
        sage: GLWE.decrypt(cpbr, c0)[0]
        0

        sage: cpbr = CPBlindRotation(LWE(n=10, q=2^20, p=4), d=32, k=2)
        sage: all([GLWE.decrypt(cpbr, cpbr(cpbr.lwe(1)))[0] == 1 for _ in range(3)])
        True

        sage: all([GLWE.decrypt(cpbr, cpbr(cpbr.lwe(0)))[0] == 0 for _ in range(3)])
        True

    """

    def __init__(
        self, lwe, d=1024, k=1, chi_s="binary", chi_e=2.0, B=2, p=2, s=None, sigma_x=1.0
    ):
        super().__init__(lwe, d, k, chi_s, chi_e, B, p, s)
        self.gadget = GadgetPreimageSampler(B, lwe.q, sigma_x)
        self.sigma_x = sigma_x


class CPModSwitchBootstrap(ModSwitchBootstrap, CPBlindRotation):
    def __init__(
        self,
        lwe,
        d=1024,
        k=1,
        chi_s="binary",
        chi_e=2.0,
        B=2,
        p_in=2,
        p_out=3,
        s=None,
        sigma_x=1.0,
        sigma_r=1.0,
        sigma_rand=1.0,
        LR=2,
    ):
        super().__init__(lwe, d, k, chi_s, chi_e, B, p_in, p_out, s)
        self.gadget = GadgetPreimageSampler(B, lwe.q, sigma_x)
        self.sigma_x = sigma_x

        self.lwe_ext = self.lwe.copy(n=k * d, p=p_out, chi_e=sigma_r, s=self.glwesec)
        self.LR = LR
        V = []
        lR = ceil(log(self.q, LR))
        for _ in range(lR):
            V.append(self.lwe_ext(0))
        self.V = matrix(self.lwe_ext.Rq, V)
```

```python
        self.sigma_rand = sigma_rand
        self.lwe_o = self.lwe_ext

    def __call__(self, c):
        """
        Switch from mod 2 to mod 3 with circuit privacy (no keyswitching necessary)

        :param c: LWE ciphertext with plaintext modulus 2

        EXAMPLE::

            sage: from tfhe import *
            sage: cpmsbs = CPModSwitchBootstrap(LWE(n=20, q=2^10, p=2), d=64, k=1)
            sage: ct0 = cpmsbs(cpmsbs.lwe(0))
            sage: cpmsbs.lwe_o.decrypt(ct0)
            0

            sage: ct1 = cpmsbs(cpmsbs.lwe(1))
            sage: cpmsbs.lwe_o.decrypt(ct1)
            1

        """
        delta = self.q // self.p_out
        N = self.d
        v_modswitch = [delta] * (N // 2) + [-delta] * (N // 2)
        c = CPBlindRotation.__call__(self, c, v_modswitch)
        cext = self.sample_extract(c)

        gadget_rand = GadgetPreimageSampler(self.LR, self.q, self.sigma_rand)
        vecr = gadget_rand([0])
        Dr = DiscreteGaussianDistributionIntegerSampler(self.sigma_rand)
        Dy = DiscreteGaussianDistributionIntegerSampler(self.sigma_x)
        crand = vecr * self.V + vector([0] * (self.n * self.d) + [Dr()])
        cout = cext + crand + vector([0] * (self.n * self.d) + [Dy() - delta])

        return cout


class CPPackedBlindRotation(PackedBlindRotation, CPGGSW):
    """
    Perform blind rotation with circuit privacy with compressed
    keys. Only works for RLWE at the moment.

    EXAMPLE::

        sage: from tfhe import *
        sage: cpbr = CPPackedBlindRotation(LWE(n=2, q=2^20-1, p=4), d=32)
        sage: c1 = cpbr(cpbr.lwe(1))
        sage: GLWE.decrypt(cpbr, c1)[0]
        1
        sage: c0 = cpbr(cpbr.lwe(0))
        sage: GLWE.decrypt(cpbr, c0)[0]
        0

        sage: cpbr = CPBlindRotation(LWE(n=10, q=2^20, p=4), d=32)
        sage: all([GLWE.decrypt(cpbr, cpbr(cpbr.lwe(1)))[0] == 1 for _ in range(3)])
        True

        sage: all([GLWE.decrypt(cpbr, cpbr(cpbr.lwe(0)))[0] == 0 for _ in range(3)])
        True

    """

    def __init__(
        self, lwe, d=1024, k=1, chi_s="binary", chi_e=2.0, B=2, p=2, s=None, sigma_x=1.0
    ):
        super().__init__(lwe, d, k, chi_s, chi_e, B, p, s)
        self.gadget = GadgetPreimageSampler(B, lwe.q, sigma_x)
```

```python
        self.sigma_x = sigma_x


class CPPackedModSwitchBootstrap(CPModSwitchBootstrap, CPPackedBlindRotation):
    pass
```

oprf.py

The script is also attached.

```python
"""
OPRF Candidate from TFHE and Crypto Dark Matter PRF Candidate.

Implements:

- OPRF
- circuit privacy
- ciphertext compression
- bootrstrapping key compression

Does not implement:

- VOPRF/check points
- NIZK proofs

"""

from sage.all import (
    GF,
    codes,
    gcd,
    identity_matrix,
    random_matrix,
    set_random_seed,
    matrix,
    vector,
    ZZ,
)
from tfhe import ModSwitchBootstrap, apply_plaintext_matrix
from cpbs import CPModSwitchBootstrap, CPPackedModSwitchBootstrap
from compression import PackedModSwitchBootstrap, CiphertextCompression


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 gcd(p, q) != 1:
            raise ValueError(f"p={p} and q={q} must be coprime.")

        if m_bound == 1:
            self.G_out = 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.G_out = C.generator_matrix()
                    break
            else:
                raise RuntimeError

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


class PRF:
    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.G_inp = identity_matrix(GF(q), n).stack(random_matrix(GF(q), (n_p - n) // 2, n))

    def __call__(self, x):
        x = vector(x).lift().change_ring(GF(self.F.q))
        y = self.G_inp * 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)


class OPRF(PRF):
    """
    EXAMPLE::

        sage: set_random_seed(1337)
        sage: from tfhe import LWE
        sage: from oprf import OPRF
        sage: oprf = OPRF(LWE(4, 3*127, "binary", 3.0, p=2))
        sage: oprf([0]*8)
        (1, 0, 2, 2, 1, 1, 0, 2, 2, 1, 2, 2)
        sage: c = oprf.blind_eval([0]*8)
        sage: vector([oprf.msbs.lwe_o.decrypt(c_) for c_ in c])
        (1, 0, 2, 2, 1, 1, 0, 2, 2, 1, 2, 2)
    """

    def __init__(
        self,
        lwe,
        d=256,
        k=1,
        m_p=16,
        n_p=16,
        n=8,
        m_bound=16,
        p=2,
        q=3,
        t=None,
        seed=None,
        cp=False,
        key_pack=False,
```

```
        ct_pack=False,
        B=128,
    ):
        """

        :param lwe: LWE instance for blind evaluations
        :param d: MLWE ring dimension.
        :param k: MLWE module rank.
        :param m_p: Number of rows of `A`.
        :param n_p: Number of columns of `A`.
        :param n: Input dimension of vector mod `p`
        :param m_bound: Bound on the output dimension of vector mod `q`
        :param p: Input modulus
        :param q: Output modulus ≠ p
        :param t: Plain input
        :param seed: randomness seed
        :param cp: Enable circuit privacy (extremely slow)
        :param key_pack: Public-key compression
        :param ct_pack: RLWE packing
        :param B: Gadget decomposition base.

        """
        super().__init__(m_p=m_p, n_p=n_p, n=n, m_bound=m_bound, p=p, q=q, t=t, seed=seed)
        self.lwe = lwe

        if not ZZ(3).divides(lwe.q):
            raise ValueError(f"□3{lwe.q}.")

        self.d = d
        if cp and not key_pack:
            self.msbs = CPModSwitchBootstrap(self.lwe, d=d, k=k, B=B)
        elif cp and key_pack:
            self.msbs = CPPackedModSwitchBootstrap(self.lwe, d=d, k=k, B=B)
        elif (not cp) and key_pack:
            self.msbs = PackedModSwitchBootstrap(self.lwe, d=d, k=k, B=B)
        else:
            self.msbs = ModSwitchBootstrap(self.lwe, d=d, k=k, B=B)

        self.ct_pack = ct_pack
        if ct_pack:
            self.packing = CiphertextCompression(self.msbs.lwe_o.copy(p=q), self.msbs.d)

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

        c = [self.lwe(z_) for z_ in z] + [self.lwe(1)]

        # we might as well randomize
        c = apply_plaintext_matrix(self.F.A, c, randomize=True)
        self.prebs = c

        c = [self.msbs(c_) for c_ in c]
        c = apply_plaintext_matrix(self.F.G_out, c)

        if self.ct_pack:
            packed = self.packing.lwes_to_rlwe(c)
            return packed
        else:
            return c
```