LEAP: A Fast, Lattice-based OPRF With Application to Private Set Intersection*

Lena Heimberger[1][0009−0001−9404−7699], Daniel Kales[2][0000−0001−9541−9792], Riccardo Lolato[3**][0009−0000−2356−339X], Omid Mir[4], Sebastian Ramacher[4][0000−0003−1957−3725], and Christian Rechberger[1,2][0000−0003−1280−6020]

[1] Graz University of Technology
[2] Taceo
[3] University of Trento
[4] AIT Austrian Institute of Technology
Contact: lena.heimberger@tugraz.at

Abstract. Oblivious pseudorandom functions (OPRFs) are an important primitive in privacy-preserving cryptographic protocols. The growing interest in OPRFs, both in theory and practice, has led to the development of numerous constructions and variations. However, most of these constructions rely on classical assumptions. Potential future quantum attacks may limit the practicality of those OPRFs for real-world applications.

To close this gap, we introduce LEAP, a novel OPRF based on heuristic lattice assumptions. Fundamentally, LEAP builds upon the SPRING [BBL+15] pseudorandom function (PRF), which relies on the learning with rounding assumption, and integrates techniques from multi-party computation, specifically Oblivious Transfer (OT) and Oblivious Linear Evaluation (OLE). With this combination of oblivious protocols, we construct an OPRF that evaluates in less than a millisecond on a modern computer.

Efficiency-wise, our prototype implementation achieves computation times of just 11 microseconds for the client and 750 microseconds for the server, excluding some base OT preprocessing overhead. Moreover, LEAP requires an online communication cost of 23 kB per evaluation, where the client only has to send around 380 bytes online. To demonstrate the practical applicability of LEAP, we present an efficient private set intersection (PSI) protocol built on top of LEAP. This application highlights LEAP's potential for integration into various privacy-preserving applications: We can compute an unbalanced set intersection with set sizes of 2^{24} and 2^{15} in under a minute of online time and just over two minutes overall.

* This is the full version of a paper which appears in Eurocrypt 2025 – 44th Annual International Conference on the Theory and Applications of Cryptographic Techniques. Please cite the conference version.

** Work done while at AIT Austrian Institute of Technology

1 Introduction

Oblivious Pseudorandom Functions (OPRFs) serve as a fundamental cryptographic building block in privacy-preserving computation. An OPRF involves two parties: a server, which holds a secret key, and a client, who provides the input. These two parties collaboratively compute a pseudorandom function (PRF) $y = \mathcal{F}_k(x)$, where the server supplies the key k and the client supplies the private input x. Importantly, the server learns neither the input x nor the output y, and the client does not learn the key k.

OPRFs emerged as an essential tool for constructing a number of privacy-preserving applications, including secure keyword search [FIPR05], secure data de-duplication [KBR13, CDGS19], password-protected secret sharing [JKK14], secure pattern matching [FHV13], and private set intersection (PSI) [KKRT16]. PSI has already been implemented and utilized by major tech companies such as Google[5] and Facebook.[6] Furthermore, OPRFs are used in password-authenticated key exchange, known as OPAQUE [JKX18]. The OPAQUE protocol, which integrates with TLS 1.3, is currently undergoing standardization.[7] OPRFs also play a pivotal role in ensuring privacy in private browsing with DDoS protection, a technology currently being standardized by the IETF[8] through mechanisms such as the anonymous token known as Privacy Pass [DGS+18].

Post-Quantum OPRFs and Comparison. Real-world deployments of OPRFs commonly use either the 2-Hash Diffie-Hellman (2HashDH) OPRF [JKK14] or the 3-Hash Diffie-Hellman OPRF [TCR+22]. Both protocols have optimal round complexity and are efficiently computable – taking approximately 400 μs and 500 μs to compute. However, both constructions rely on the hardness of the Decisional Diffie-Hellman assumption or variations, such as the One-More-Diffie-Hellman assumption, which may be vulnerable to quantum adversaries. These circumstances prompted the proposal of several post-quantum candidates, which we discuss below.

On the isogeny side, two OPRFs were proposed by Boneh, Kogan and Woo [BKW20]: an SIDH-based construction, which was broken [BKM+21] and later fixed [Bas24b] and updated to fit within a framework of higher-dimensional isogenies [Bas24a]. The update removes the need for a trusted setup and provides implementations in SageMath, with client requests taking approximately 17.7 seconds and server responses 130.7 seconds.

The second OPRF [BKW20] follows the Naor-Reingold approach using CSIDH. Follow-up work [HHM+24a] shows that the construction needs a relational lattice, which is currently available for up to 1024 bits [DFK+23]. CSIDH may need primes up to 5280 bits. The OPRF OPUS [HHM+24a] supports arbitrary bit sizes as it does not require the relation lattice, but has a round complexity of $O(\kappa)$. Finally, Delpech and Pedersen [DP24] introduced another CSIDH-based

[5] https://github.com/google/private-join-and-compute/issues/33
[6] https://engineering.fb.com/2020/07/10/open-source/private-matching/
[7] https://www.ietf.org/archive/id/draft-krawczyk-cfrg-opaque-02.txt
[8] https://datatracker.ietf.org/wg/privacypass/about/

OPRF with a communication complexity ranging from 3072 to 31680 bits, conditional on the CSIDH modulus p. Their scheme again requires a trusted setup knowledge of the class group structure. Their scheme is the only CSIDH-based construction that is secure against malicious adversaries, and they also offer an idea to remove the trusted setup. All CSIDH-based constructions are not proven secure in the Universal Composability (UC) framework.

Albrecht et al. [ADDS21] introduced the first lattice-based OPRF with malicious security. The OPRF requires over 128 GB of communication, which limits its practicality. Another obstacle to using the OPRF in protocols is the use of a non-standard security definition. A more efficient variant of the OPRF [AG24] significantly improves the communication to under half a megabyte of communication in the random oracle model. Follow-up work [ADDG24] evaluates the Dark Matter PRF [BIP+18] using fully homomorphic encryption, yielding a scheme with approximately 70 MB of communication, of which 3 MB are communicated during the online phase, also offering security for semi-honest servers. The 3 MB include a reusable FHE key that can amortize the communication over several rounds. Dinur et al. proposed secret-sharing the Dark Matter PRF [DGH+21], which yields a secure OPRF in the semi-honest model that requires a trusted setup and preprocessing. The OPRF has been improved [APRR24] to less than a thousand bits of communication, or alternatively 4 kB of communication for an estimated evaluation of less than four microseconds. Seres et al. [SHB23] observed the programmability of the Legendre PRF and its potential verifiability using zero-knowledge proofs. However, it lacks composable security guarantees and comes with overheads that only make the OPRFs somewhat efficient. Faller et al. [FOO23] proposed an OPRF based on the secure evaluation of AES using garbled circuits, achieving a security level similar to 2HashDH in the UC framework for semi-honest servers. However, it incurs substantial communication costs. Kolesnikov et al. [KKRT16] demonstrated a construction of OPRFs from a PRF with an input domain of $\{1, \ldots, n\}$ with $\binom{n}{1}$-OT of random messages. While this construction is efficient for a small input domain, our construction requires fewer OT calls due to the large input domain of 2^{128}.

In summary, despite several interesting constructions, no concrete, implementations with low overhead emerged so far. We aim to construct an OPRF where the server bears most of the computational load and minimizes the overall communication complexity to save bandwidth, in particular for the client, based on well-known assumptions.

In Table 1, we provide a succinct comparison of the highlighted schemes discussed above, prioritizing those most relevant to our scheme setting, i.e., lattice-based constructions. Also, we include [Bas24a] and [DP24] as the most recent efficient isogeny-based OPRFs. While our scheme is not round optimal and requires more interaction, this is not a significant issue in many applications, particularly in PSI scenarios where a lot of communication takes place.

Private Set Intersection (PSI). Private Set Intersection protocols enable two parties holding some sets to compute the intersection. In contrast, neither

Table 1: Comparison of our OPRF LEAP with other post-quantum OPRFs. **Impl** denotes the availability of a full implementation. ◐ is security against a semi-honest adversary (Client(C) or Server(S)), ● against a malicious adversary.

Schemes	Assumption	Rounds	Comm. Cost	(C-S)	Impl
[ADDS21]	R(LWE)+SIS	2	2MB	◐-◐	≈♣
[ADDG24]	mod(2,3)+lattices	2	10 kB♠	●-◐	✗♣
[APRR24]	mod(2,3)	2	957 bits	●-◐	✗
[APRR24]	mod(2,3)	2	4 kB	●-◐	✗
[AG24]	R(LWE)+SIS	2	221.5+315.9 kB	●-●	✗
[Bas24a]	Higher-dimensional Isogenies	2	28.9 kB	●-●	✓
[DP24]	Isogenies $\mathbb{F}_p$	2	16.38 kB	●-◐	✗
LEAP	RLWR (heuristic)	6	23 kB♦	◐-◐	✓

♠ Plus 2.5 MB reusable FHE bootstrapping key.
♦ Communication cost for preprocessing is approximately 793 kB, possible amortization when batching multiple OPRF evaluations.
♣ Only a partial implementation is available.

of the parties learns the other party's sets beyond the elements contained in the intersection. Depending on the concrete settings, PSI protocols can be optimized for cases where the sets of both parties are of the same size or where the size of one party is significantly larger. The latter – denoted as unbalanced PSI – is especially interesting for applications such as private contact discovery [DRRT18, KRS+19, HSW23] where servers hold a large database of clients of messaging applications and the clients are interested whether anyone of their contacts is also using the service.

The intuition of OPRF-based PSI, as proposed in [FIPR05, HL08, PSSW09, KLS+17], is that the server holds a PRF key and evaluates all elements of its set with the PRF and stores the resulting outputs. When a client wants to compute the intersection, it requests all PRF evaluations from the server and computes the OPRF for all elements in the client set set with the server. With these evaluations, the client can check which elements are also in the server set. In this scenario, the server does not learn anything beyond the cardinality of the client set.

This application also highlights the need for OPRFs with efficient client-side computation. For the PSI protocol outlined above, the size of the client set determines the number of OPRF evaluations. As a result, a high client computation or communication complexity is prohibitive for OPRF-based PSI. It motivates the design and analysis of OPRF constructions that keep the communication overhead and computation complexity as low as possible.

1.1 Contributions

In this work, we introduce the *LatticE oprf from An efficient Prf* (LEAP), an OPRF that evaluates the Learning-With-Rounding (LWR)-based SPRING PRF obliviously. Our results demonstrate that LEAP achieves OPRF computation in less than a millisecond with less than 400 bytes of client communication and enables us to perform fast (unbalanced) private set intersection. Our contributions are:

1. We present a protocol for the **oblivious evaluation of the lattice-based PRF SPRING**, which results in a very efficient OPRF from generic oblivious transfer (OT) and oblivious linear evaluation (OLE), it handles the deviations of SPRING from the Naor-Reingold PRF construction paradigm by introducing subprotocols for oblivious rounding and bias reduction, resulting in a 6-round protocol. Furthermore, due to the black-box use of these building blocks, our security proof in the Universal Composability (UC) model can be reduced to the security of the underlying building blocks, where only the final application of a BCH code for bias reduction in the SPRING-BCH variant with an uneven modulus requires special attention in the security proof. Our approach of oblivious evaluation may also be useful for other privacy-preserving protocols with operations in the NTT domain.
2. We provide a **reference implementation** of the protocol in C++. In our implementation, we select generic OT and OLE protocols and highlight computation and communication trade-offs during the data-independent preprocessing phase. The performance benchmarks underline that our protocol is computationally efficient and has low communication overhead.
3. Finally, we integrate our protocol into **Private Set Intersection** to highlight one of the key application areas of our OPRF. As shown by Kales et al. [KRS+19], PSI protocols built from OPRFs fare particularly well in the case of unbalanced sets where the server set is significantly larger. We instantiate and implement the protocol of Kales et al. using LEAP in the context of unbalanced PSI, resulting in a highly efficient implementation. Our benchmarks confirm that LEAP is well suited for this application scenario. For example, intersecting two sets with 2^{15} and 2^{24} elements, respectively, takes just over two minutes on commodity hardware.

1.2 Technical Overview

We first provide a short technical overview of our approach to the oblivious evaluation of SPRING in BCH mode. First, recall that PRFs following the Naor-Reingold construction paradigm are expressed as $\mathbf{k}_0 \prod_{i=1}^{\kappa} \mathbf{k}_i^{c_i}$ for key elements $\mathbf{k}_0, \dots \mathbf{k}_\kappa$ and input bits $c_1, \dots, c_\kappa$. These allow the client and the server to perform oblivious evaluation by using a $\binom{2}{1}$-OT protocol per bit to obtain either a random value $\mathbf{r}_i$ if the bit is not set and otherwise the blinded value $\mathbf{r}_i\mathbf{k}_i$ [FIPR05]. After performing this operation for each bit position, the server also provides $\mathbf{k}_0$ masked with the product of all inverse randomizers to the client,

i.e., $\mathbf{k}_0 \prod_{i=1}^{\kappa} \mathbf{r}_i^{-1}$, which allows the client to compute the PRF output by combining all values received during the initial OT phase with the final message.

While SPRING follows the Naor-Reingold paradigm by taking $\mathbf{k}$ from a polynomial ring, the resulting product also needs to be rounded to avoid leaking information about the key to the client. Before applying our rounding subprotocol, we use OLE to transform the multiplicative sharing to an additive sharing, which results in significant performance advantages. Before combining the OT results with the final unblinding message, we call a subprotocol that performs oblivious rounding for each polynomial coefficient. The final step of SPRING is the bias reduction required for odd moduli applies a BCH code, which also deviates from pure Naor-Reingold PRFs. As BCH codes are linear, we integrate the bias reduction into the oblivious rounding protocol.

To further optimize performance, we represent the polynomials in the NTT domain to reduce the computational complexity of multiplications from quadratic to quasilinear, switching the representation back to its normal form before oblivious rounding.

2 Preliminaries

Notation. We use n to denote the dimension of the lattice and κ to denote the security parameter, which also serves as the input length for the clients OPRF input. For a distribution $\mathcal{D}$, we denote the sampling of x according to distribution $\mathcal{D}$ by $x \leftarrow\!\!\$\ \mathcal{D}$. For a finite set X, $x \leftarrow\!\!\$\ X$ indicates sampling uniformly at random from X. We assume all algorithms are polynomial-time (PPT) unless otherwise specified. We will write $\mathsf{BCH}(\mathbf{s}) = \mathbf{s}\mathbf{M}^t$, where $\mathbf{M}$ is the generator matrix of the extended BCH code, to denote the call to the syndrome decoding multiplication. By $\cdot$, we denote polynomial element-wise point-value multiplication. $*$ denotes polynomial multiplication. A *round* is a single message from the client to the server or from the server to the client. An OPRF is *round-optimal* if the server and the client send a message each, totaling two rounds. $\ll$ denotes a bitwise left shift.

We start by recalling the Learning with Rounding assumption in Section 2.1 and continue with the definition of OPRFs in Section 2.2. Then, we give an overview of the strategies for oblivious evaluation in Section 2.5.

2.1 Lattice Assumptions

The security of our scheme relies on well-known computational lattice problems, namely Learning with Errors (LWE) [Ajt96, Reg05] and Learning with Rounding (LWR) [BPR12]. The LWR problem can be seen as a derandomized version of LWE, where the noise term in the inner product of vector multiplication is replaced by a deterministic rounding from a large set q to a smaller subset p. LWR assumes it is hard to distinguish these rounded inner products, using a secret $\mathbf{s} \leftarrow\!\!\$\ \mathbb{Z}_q^n$, from uniformly sampled elements $u \in \mathbb{Z}_p$. More formally:

Definition 1 (LWR). *Let* $f_{\mathbf{s}} : \mathbb{Z}_q^n \to \mathbb{Z}_p$ *where* $f_{\mathbf{s}}(\mathbf{x}) = \lfloor \langle \mathbf{x}, \mathbf{s} \rangle \rceil_p = \lfloor (p/q) \cdot \langle \mathbf{x}, \mathbf{s} \rangle \rceil$. *For any* $\mathbf{s}$ *any polynomially many* $\mathbf{x}_i \leftarrow\!\!\$\, \mathbb{Z}_q^n$, *the two sets*

$$\{(\mathbf{x}_i, f_{\mathbf{s}}(\mathbf{x}_i))\} \text{ and } \{(\mathbf{x}_i, u_i) : u_i \leftarrow\!\!\$\, \mathbb{Z}_p\}$$

are hard to distinguish.

For SPRING we are specifically interested in LWR defined over a ring: the ring version of the LWR problem involves the distinguishability of polynomials sampled from a family of cyclotomic rings of the form $R_p = \mathbb{Z}_p[X]/(X^n + 1)$. Similarly to LWR, the polynomials are multiplied by a secret polynomial $\mathbf{s} \leftarrow\!\!\$\, R_q$ and rounded from a larger ring R_q to a smaller ring R_p. More formally:

Definition 2 (Ring-LWR). *Let* $g_{\mathbf{s}} : R_q \to R_p$ *where* $g_{\mathbf{s}}(\mathbf{x}) = \lfloor \mathbf{x} \cdot \mathbf{s} \rceil_p = \sum_{i=0}^{n-1} \lfloor (p/q) \cdot x_i s_i \rceil X^i$. *For any* $\mathbf{s}$ *any polynomially many* $\mathbf{x}_i \leftarrow\!\!\$\, R_q$, *the two sets*

$$\{(\mathbf{x}_i, g_{\mathbf{s}}(\mathbf{x}_i))\} \text{ and } \{(\mathbf{x}_i, \mathbf{u}_i) : \mathbf{u}_i \leftarrow\!\!\$\, R_p\}$$

are hard to distinguish.

We note that, for suitable choices of parameters, LWR is as hard as LWE and the same holds true for the ring versions of the assumptions [BPR12]. Specifically, this holds also true for small modulus q as shown by Bogdanov et al. [BGM+16].

2.2 Oblivious Pseudorandom Function (OPRF)

As mentioned in the introduction, an OPRF [FIPR05] is an interactive protocol between a client and a server. In this protocol, the client holds a private input x, and the server holds a key k for a PRF $\mathcal{F}$. Together, they engage in a protocol to obliviously evaluate $\mathcal{F}_k$ on x, such that the client learns (and optionally verifies) the evaluation $\mathcal{F}_k(x)$, while the server learns nothing.

Before defining the notion of OPRFs, we recall the definition of a pseudorandom function (PRF) [GGM86] as follows:

Definition 3 (PRF). *Let* $\mathcal{F} : \mathcal{X} \times \mathcal{D} \to \mathsf{R}$ *be a family of functions, and let* Γ *be the set of all functions* $\mathcal{D} \to \mathsf{R}$. *For a PPT distinguisher* $\mathcal{D}$ *we define the advantage function* $\mathsf{Adv}^{\mathsf{PRF}}_{\mathcal{D},\mathcal{F}}(\kappa)$ *as*

$$\left| \Pr_{x \leftarrow\!\!\$ \mathcal{X}} \left[\mathcal{D}^{\mathcal{F}(x,\cdot)}(1^\kappa) = 1 \right] - \Pr_{f \leftarrow\!\!\$ \Gamma} \left[\mathcal{D}^{f(\cdot)}(1^\kappa) = 1 \right] \right|.$$

$\mathcal{F}$ is a pseudorandom function (family) if it is efficiently computable and for all PPT distinguishers $\mathcal{D}$ there exists a negligible function $\varepsilon(\cdot)$ such that

$$\mathsf{Adv}^{\mathsf{PRF}}_{\mathcal{D},\mathcal{F}}(\kappa) \leq \varepsilon(\kappa).$$

Here, we consider only binary input and output spaces and thus set $\mathsf{S} = D = \{0,1\}^\kappa$ and $\mathsf{R} = \{0,1\}^\ell$.

Definition 4 (Oblivious PRF (OPRF) [FIPR05]). *A two-party protocol is an OPRF if there exists some PRF family $\mathcal{F}_k$, such that it privately realizes the following functionality:*

- *Client has input x; Server has key k.*
- *Client outputs $\mathcal{F}_k(x)$; Server outputs nothing.*

Security. Intuitively, an OPRF is secure if the client learns only the PRF output but not the server's key, and the server learns nothing. More formally, we define OPRF security in the UC framework [Can01], following the paradigm of the real and ideal world. We use the strong model with adaptive corruption (see Figure 1) presented in [JKX18]). In Figure 1, $\mathcal{S}'$ represents the server involved in the online evaluation process, distinct from the original server $\mathcal{S}$ that initiated the session. The variable tx acts as a transaction counter, tracking the number of evaluation requests made within a specific session identified by *sid*. This additional variable helps manage state and control access, especially when dealing with compromised servers. The prefix variable prfx captures unique identifiers for evaluation requests, ensuring that each request is distinct and protecting against replay attacks. The model operates under the concept of adaptive compromise, allowing an adversary to dynamically choose which entities to corrupt based on the information gleaned during the protocol's execution, thereby enhancing the realism of the threat model (see [JKX18] for more details). ddone can consider multiple options in defining the concrete security model and additional features (see [CHL22] for a detailed overview and comparison).In this work, we are interested in OPRF security with adaptive compromise and therefore recall the definition of this functionality from [JKX18] in Figure 1.

2.3 Naor-Reingold PRF

The Naor-Reingold PRF (NR-PRF) [NR04] constructs PRFs from Abelian group actions. It requires $\kappa+1$ group elements to compute a PRF for κ input bits. The initial group action starts with the first group element, and for each set bit c_i, a group action is performed using the ith key element. Specifically, the NR-PRF is defined as:

$$\mathcal{F}_{NR}\Big((k_0,\cdots,k_\kappa),(c_1,\cdots,c_\kappa)\Big) = k_0 \cdot k_1^{c_1} \cdot k_2^{c_2} \cdots k_\kappa^{c_\kappa}$$

PRFs following the Naor-Reingold paradigm can be turned into a OPRF as follows (first presented by Freedman et al. [FIPR05]): the two parties engage in a $\binom{2}{1}$-OT protocol, with the sender returning $k_i^{c_i} \cdot r_i$. The receiver aggregates the results to obtain the blinded group element $K \leftarrow \prod_{i=1}^{\kappa} r_i \cdot k_i^{c_i}$. After the OT step, the unblinding element $U \leftarrow k_0 \cdot r_1^{-1} \cdots r_\kappa^{-1}$ is sent. The result of the $\mathcal{F}_{NR}$ function is obtained by applying the group action to the blinded element and the unblinding element. While the key elements k may be reused, the blinding elements must be sampled anew each time to protect the client's input.

Public Parameters:

PRF output-length ℓ, polynomial in security parameter κ.
Note that for every i, x, value $F_{sid,i}(x)$ is initially undefined, and if undefined value $F_{sid,i}(x)$ is referenced then $\mathcal{F}_{\text{OPRF}}$ assigns $F_{sid,i}(x) \leftarrow \{0,1\}^{\ell}$.

Initialization:

On message (INIT, sid) from party $\mathcal{S}$, if this is the first INIT message for sid, set $tx = 0$ and send $(\text{INIT}, sid, \mathcal{S})$ to $\mathcal{A}$. From now on use tag "$\mathcal{S}$" to denote the unique entity which sent the INIT message for the session identifier sid. (Ignore all subsequent INIT messages for sid.)

Server Compromise:

On $(\text{COMPROMISE}, sid, \mathcal{S})$ from $\mathcal{A}$, declare server $\mathcal{S}$ as COMPROMISED. (If $\mathcal{S}$ is corrupted then it is declared COMPROMISED from the beginning.)
Note: Message (COMPROMISE, sid, $\mathcal{S}$) requires permission from the environment.

Offline Evaluation:

On $(\text{OFFLINEEVAL}, sid, i, x)$ from $\mathcal{P} \in \{\mathcal{S}, \mathcal{A}\}$, send $(\text{OFFLINEEVAL}, sid, F_{sid,i}(x))$ to $\mathcal{P}$ if any of the following hold:
(i) $\mathcal{S}$ is corrupted, (ii) $\mathcal{P} = \mathcal{S}$ and $i = \mathcal{S}$, (iii) $\mathcal{P} = \mathcal{A}$ and $i \neq \mathcal{S}$, (iv) $\mathcal{P} = \mathcal{A}$ and $\mathcal{S}$ is compromised.

Online Evaluation:

- On $(\text{EVAL}, sid, ssid, \mathcal{S}', x)$ from $\mathcal{P} \in \{\mathcal{C}, \mathcal{A}\}$, send $(\text{EVAL}, sid, ssid, \mathcal{P}, \mathcal{S}', x)$ to $\mathcal{A}$. On prfx from $\mathcal{A}$, ignore this message if prfx was used before. Else record $\langle ssid, \mathcal{P}, x, \mathsf{prfx}\rangle$ and send $(\text{Prefix}, sid, ssid, \mathsf{prfx})$ to $\mathcal{P}$.
- On $(\text{SNDRCOMPLETE}, sid, ssid')$ from $\mathcal{S}$, send $(\text{SNDRCOMPLETE}, sid, ssid', \mathcal{S})$ to $\mathcal{A}$. On prfx' from $\mathcal{A}$, send $(\text{Prefix}, sid, ssid', \mathsf{prfx}')$ to $\mathcal{S}$. If there is a record $\langle ssid, \mathcal{P}, x, \mathsf{prfx}\rangle$ for $\mathcal{P} \neq \mathcal{A}$ and $\mathsf{prfx} = \mathsf{prfx}'$, change it to $\langle ssid, \mathcal{P}, x, OK\rangle$, else set a counter $tx++$.
- On $(\text{RCVCOMPLETE}, sid, ssid, \mathcal{P}, i)$ from $\mathcal{A}$, ignore this message if there is no record $\langle ssid, \mathcal{P}, x, \mathsf{prfx}\rangle$ or if $(i = \mathcal{S}, tx = 0,$ and $\mathsf{prfx} \neq OK)$. Else send $(\text{EVAL}, sid, ssid, F_{sid,i}(x))$ to $\mathcal{P}$, and if $(i = \mathcal{S}$ and $\mathsf{prfx} \neq OK)$ then set $tx--$.

Fig. 1: $\mathcal{F}_{\text{OPRF}}$ with adaptive compromise [JKX18].

2.4 Spring PRF

Spring [BBL+15] is an efficient PRF based on the Ring-LWR (RLWR) hardness assumption that follows the Naor-Reingold PRF construction paradigm with additional tweaks. The construction is based on ring $R := \mathbb{Z}[X]/(X^n+1)$ whereas the group action is the polynomial multiplication in the ring R. Using $\kappa+1$ ring elements $\mathbf{K} = (\mathbf{k}_0, \mathbf{k}_1, \ldots, \mathbf{k}_\kappa)$, for the evaluation of the PRF the polynomial $\mathbf{k}_i$ is multiplied to a subset-product if and only if the ith input bit c_i is set. After computing the subset product, the result has to be rounded by applying the rounding function S. S rounds each coefficient of the polynomial and outputs 1

if the coefficient is $\geq \frac{q}{4}$ and $\leq \frac{3q}{4}$, and 0 otherwise, resulting in binary outputs of the PRF.

$$\mathcal{F}_{\mathbf{K}}(c_1, \cdots, c_\kappa) = S\left(\mathbf{k}_0 \cdot \prod_{i=1}^{\kappa} \mathbf{k}_i^{c_i}\right)$$

Two modes have been proposed: one using an uneven modulus and another using an even modulus. We focus on the mode with the uneven modulus $q = 257$, also known as BCH mode, which produces 64 output bits. To support longer input and output sizes, including with an even modulus $q = 514$, Spring-CRT and Spring-CTR are available.

Spring BCH Code. Using $q = 257$ introduces a rounding bias of $\frac{1}{257}$. To mitigate this, the rounded coefficients are multiplied by the generator matrix associated with the extended BCH code $[128, 64, 22]$ to obtain their syndrome[9]. This reduces the bias to a negligible $\frac{\sqrt{2^\kappa}}{2}\left(\frac{1}{q}\right)^d = 2^{-145}$, producing a 64-bit output.

The PRF takes $\kappa = 128$ bits of input. The $[128, 64, 22]$ code extends the $[127, 64, 22]$ BCH code with a parity bit to align with a power of two, as BCH codes over $\mathbb{Z}_p$ have a length of $2^t - 1$. The BCH code is computed over $\mathbb{Z}_p$ for implementation efficiency, with matrix rows being cyclic shifts of a single row.

Spring-CRT: Bias Reduction using the Chinese Remainder Theorem (CRT) for even moduli. As an alternative to the uneven modulus is the CRT mode with an even modulus $q = 514 = 2 \cdot 257$. Spring-CRT decomposes the subset product computation over R_{2q}^* into the Chinese Remainder components R_2^* and R_q^*. The latter is computed exactly as in BCH mode from Section 2.4, and the former ring uses sparse generators for cyclic components. The main advantage of this mode is the larger output size and the absence of a rounding bias. The main drawback is the added algebraic structure from the CRT decomposition. Specifically, attacks may cancel out the R_2^* component to recover the R_q^* component. The reference implementation of the PRF is 4.5 times slower than AES. The most efficient attack against Spring-CRT is a subexponential attack [BDFK17], but the attack is currently computationally infeasible.

Spring-CTR: Counter Mode for amortized computation. For longer inputs, Spring can be used with a counter mode. The input blocks are ordered in a Gray code style, so the bits only differ in one position. As a result, only the first block has to be computed fully, as each successive subset product are computed from the previous one with just one more multiplication by either the seed element or its inverse.

[9] The extended BCH code is the $[127, 64, 21]$ code with a parity bit. It has the largest known minimum distance for its rate

2.5 Tools for Oblivious Evaluation

To convert a PRF to an OPRF, we use well-studied primitives from secure multiparty computation (MPC) to keep the client input and server key secret. Specifically, our construction relies on Oblivious Transfer (OT) and Oblivious Linear Equations (OLE) protocols.

Oblivious Transfer OTs enable a receiver holding a choice bit c to obtain the string m_c from a sender holding two strings (m_0, m_1). These protocols ensure the sender gains no information about c, while the receiver does not learn the other string m_{1-c}. This setting with two strings is called $\binom{2}{1}$-OT and can be generalized to k strings, denoted as $\binom{k}{1}$-OT [NP99]. The ideal functionality captures the security properties of OT is given in [ABB+13], outlined in our notation in Figure 2.

The functionality $\mathcal{F}_{\binom{N}{1}\text{-OT}}$ interacts with an adversary $\mathcal{A}$ and a set of parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ via the following queries:

- On message (SEND, $sid, ssid, \mathcal{P}_i, \mathcal{P}_j, (m_1, \ldots, m_N)$) from $\mathcal{P}_i$, with $m_i \in \{0,1\}^\ell$: record the tuple $(sid, ssid, \mathcal{P}_i, \mathcal{P}_j, (m_1, \ldots, m_N))$ and reveal (SEND, $sid, ssid, \mathcal{P}_i, \mathcal{P}_j$) to $\mathcal{A}$. Ignore further SEND messages with the same $ssid$ from $\mathcal{P}_i$.
- On (RECEIVE, $sid, ssid, \mathcal{P}_i, \mathcal{P}_j, s$) from $\mathcal{P}_j$, with $s \in [N]$: record the tuple $(sid, ssid, \mathcal{P}_i, \mathcal{P}_j, s)$ and reveal (RECEIVE, $sid, ssid, \mathcal{P}_i, \mathcal{P}_j$) to $\mathcal{A}$. Ignore further RECEIVE messages with the same $ssid$ from $\mathcal{P}_j$.
- On (SENT, $sid, ssid, \mathcal{P}_i, \mathcal{P}_j$) from $\mathcal{A}$: ignore the message if $(sid, ssid, \mathcal{P}_i, \mathcal{P}_j, (m_1, \ldots, m_N))$ is not recorded; otherwise send (SENT, $sid, ssid, \mathcal{P}_i, \mathcal{P}_j$) to $\mathcal{P}_i$ and ignore further SENT messages with the same $ssid$ from $\mathcal{A}$.
- On (RECEIVED, $sid, ssid, \mathcal{P}_i, \mathcal{P}_j$) from $\mathcal{A}$: ignore the message if $(sid, ssid, \mathcal{P}_i, \mathcal{P}_j, s)$ is not recorded; otherwise send (RECEIVED, $sid, ssid, \mathcal{P}_i, \mathcal{P}_j, m_s$) to $\mathcal{P}_j$ and ignore further RECEIVED messages with the same $ssid$ from $\mathcal{A}$.

Fig. 2: Ideal functionality for $\binom{N}{1}$-OT [ABB+13].

Base OT and OT Extension For efficient protocols, the expensive OT operations are processed before seeing the data in a *base* phase [Bea96]. The client calls the base OT with a random choices bit $c_\$$. In the online phase, the client sends a correction bit b depending on the actual input bit c such that $b = c \oplus c_\$$ to the server, which computes the correct result depending on the functionality of the protocol.

Using κ values from the Base OT, poly(κ) more base OTs can be generated using cheap symmetric operations. These *extensions* yield numerous correlated

OT pairs, wherein instead of producing two independent messages (m_0, m_1), the correlated OT (COT) generates $(m_0, m_0 \oplus \Delta)$ for some Δ. For example, the semi-honest IKNP protocol [IKNP03] performs m correlated k-bit OT using k m-bit OTs, with $m \gg k$, transforming them into m k-bit OTs. The server acts as the receiving party in the correlated OT, obtaining $\Delta_i \in \{0,1\}^\kappa$ and a choice bit c_i. The client, holding a vector of choice bits $C = [c_1, \cdots, c_\kappa]$, chooses $t_i \in \{0,1\}^m, \forall 1 \leq i \leq \kappa$ and inputs with $(t_i, t_i \oplus c_i)$.

Silent OT [BCG+19] enhances OT extensions by significantly reducing preprocessing communication complexity. It combines OT extensions with code-based methods, resulting in a 0 to 3-bit communication complexity to generate a 128-bit OT string. However, these codes introduce substantial overhead and perform optimally when multiple OPRFs are conducted simultaneously, as discussed in Section 5.6.

One-out-of-N Oblivious Transfer $\binom{N}{1}$-OT allows receiver to choose one out of N strings without learning anything about the other $(N-1)$-OT inputs. [NP99] show how to extend $\binom{2}{1}$-OT to $\binom{N}{1}$-OT by using $\log_2 N$ OT call so the client can obtain m_b form the set $\{m_0, m_1, \ldots, m_{N-1}\} \in \{0,1\}^n$, where the client choice $b \in \{0, \ldots, N-1\}$ is used as the $\log_2(N)$-bit index of the message:

- The server generates $L = \log N$ key pairs $(k_0^0, k_0^0), (k_1^0, k_1^1), \ldots, (k_{L-1}^0, k_{L-1}^0)$.
- Each message m_i is encrypted with the key produced by computing the XOR of the index bits $c_i = m_i \oplus \bigoplus_{j=1}^{L} k_j^{i_j}$. i_j is the jth bit of i. For example, when $L = 4$ and $i = 7$, the server encrypts $m_7 \oplus k_0^1 \oplus k_1^1 \oplus k_2^1 \oplus k_3^0$. Each c_i is sent to the receiver.
- To decrypt c_b, the client performs $\binom{2}{1}$-OT for each key k_j, where is j the bitwise decomposition of b.

The KKRT protocol [KKRT16] improves the communication cost of the $\binom{N}{1}$-OT to $\log N$ by using Hadamard codes.

Oblivious Linear Evaluation Oblivious Linear Evaluation (OLE) is a protocol that converts a product of two $\mathbb{Z}_p$ elements into a difference: Given $a, b \in \mathbb{Z}_p$, the protocol returns $y, e \in \mathbb{Z}_p$ such that $y - e \equiv ab$. Such protocols can be used to transform multiplicatively shared elements into additively shared ones. The corresponding functionality can be found in [GNN17] and defined in Figure 3.

3 Leap: Oblivious Evaluation of Spring

With all the ingredients in place, we are now in position to present the details of our OPRF protocol to obliviously evaluate Spring. Overall it is split into three steps: 1) the client and the server engage in $\binom{2}{1}$-OT to evaluate a blinded subset-sum 2) they utilize a OLE to transform the subset sums into subset-products for more efficient computation 3) Rounding is carried out in a blinded way to ensure

Functionality $\mathcal{F}_{\text{OLE}}$

- On message (INPUTS,$sid, ssid, \mathbf{a}, \mathbf{e}$) from $\mathcal{S}$ with $\mathbf{a}, \mathbf{e} \in \mathbb{F}^t$: if there is no stored tuple, store $(\mathbf{a}, \mathbf{e})$ and send (INPUT,$sid, ssid$) to $\mathcal{A}$. Else, ignore that message.
- On (INPUTR,$sid, ssid, \mathbf{b}$) from $\mathcal{R}$, where $\mathbf{b} \in \mathbb{F}^t$: if there is no stored tuple, store $\mathbf{b}$ and send (INPUT,$sid, ssid$) to $\mathcal{A}$. Else, ignore that message.
- On (DELIVER,$sid, ssid, \mathcal{S}$) from $\mathcal{A}$: if both $(\mathbf{a}, \mathbf{e})$ and $\mathbf{b}$ are stored, send (DELIVERED,$sid, ssid$) to $\mathcal{S}$. Else, ignore that message.
- On (DELIVER,$sid, ssid, \mathcal{R}$) from $\mathcal{A}$: if both $(\mathbf{a}, \mathbf{e})$ and $\mathbf{b}$ are stored, set $y_i = a_i \cdot b_i + e_i$ for $i \in [t]$ and send (OUTPUT,$sid, ssid, \mathbf{y}$) to $\mathcal{R}$. Else, ignore that message.

Fig. 3: Ideal functionality for OLE [GNN17]

that only the client gains access to the protocol's output. Finally, the BCH code is applied for bias reduction as in Spring-BCH.

The full protocol dubbed Leap is depicted in Figure 4. As it can be observed from the protocol description, the server and client perform multiple OT and OLE evaluations.
We note that in our protocol, we assume that all polynomials are sampled in subset-sum NTT form as described in Section 5.1.

3.1 Blind Subset Computation

We start with the subprotocol to perform a blinded subset sum computation. The main idea here is to employ a $\binom{2}{1}$-OT protocol (see Section 2.3) to for each input bit a freshly blinded version of the corresponding secret key element if the input bit is set and a random element otherwise. By that, we obtain a blinded sum $\sum_{i=1}^{\kappa} \mathbf{r}_i + c_i \mathbf{k}_i$. The protocol is defined as follows:

- **Input Client:** OPRF input x, bit-decomposed as $x = (c_1, \ldots, c_\kappa)$.
- **Input Server:** Long-term keys $(\mathbf{k}_0, \ldots, \mathbf{k}_\kappa)$.
- **Protocol:** The blinded subset-sum is computed using $\binom{2}{1}$-OT whereas for each $i \in [\kappa]$ the server samples a uniformly random blinding polynomial $\mathbf{r}_i$ and uses it to additively blind $\mathbf{k}_i$. Depending on the input bit, the client obtains $\mathbf{r}_i$ or $\mathbf{r}_i + \mathbf{k}_i$. The client sums all obtained elements to obtain $\sum_{i=1}^{\kappa} \mathbf{r}_i + c_i \mathbf{k}_i$. The server, on the other hand, sums all blinding factors to obtain $\mathbf{k}_0 - \sum_{i}^{\kappa} \mathbf{r}_i$.
- **Result:** The client holds the blinded subset-sum $\sum_{i=1}^{\kappa} \mathbf{r}_i + c_i \mathbf{k}_i$ depending on the choice bits c_i. The server holds $\mathbf{k}_0 - \sum_{i=1}^{\kappa} \mathbf{r}_i$.

After completing the subprotocol, both the client and server lifts the values from subset-sum to subset-product representation. Thus, the client obtains the polynomial $\mathbf{b} = \prod_{i=1}^{\kappa} \mathbf{r}_i \mathbf{k}_i^{c_i}$ and the server gets the polynomial $\mathbf{a} = \mathbf{k}_0 \prod_{i=1}^{\kappa} \mathbf{r}_i^{-1}$. Note that the values represent a multiplicative sharing of $\mathbf{k}_0 \prod_{i=1} \mathbf{k}_i^{c_i}$ which highlights the difference in how Leap performs the unblinding from other Naor-Reingold OPRF protocols.

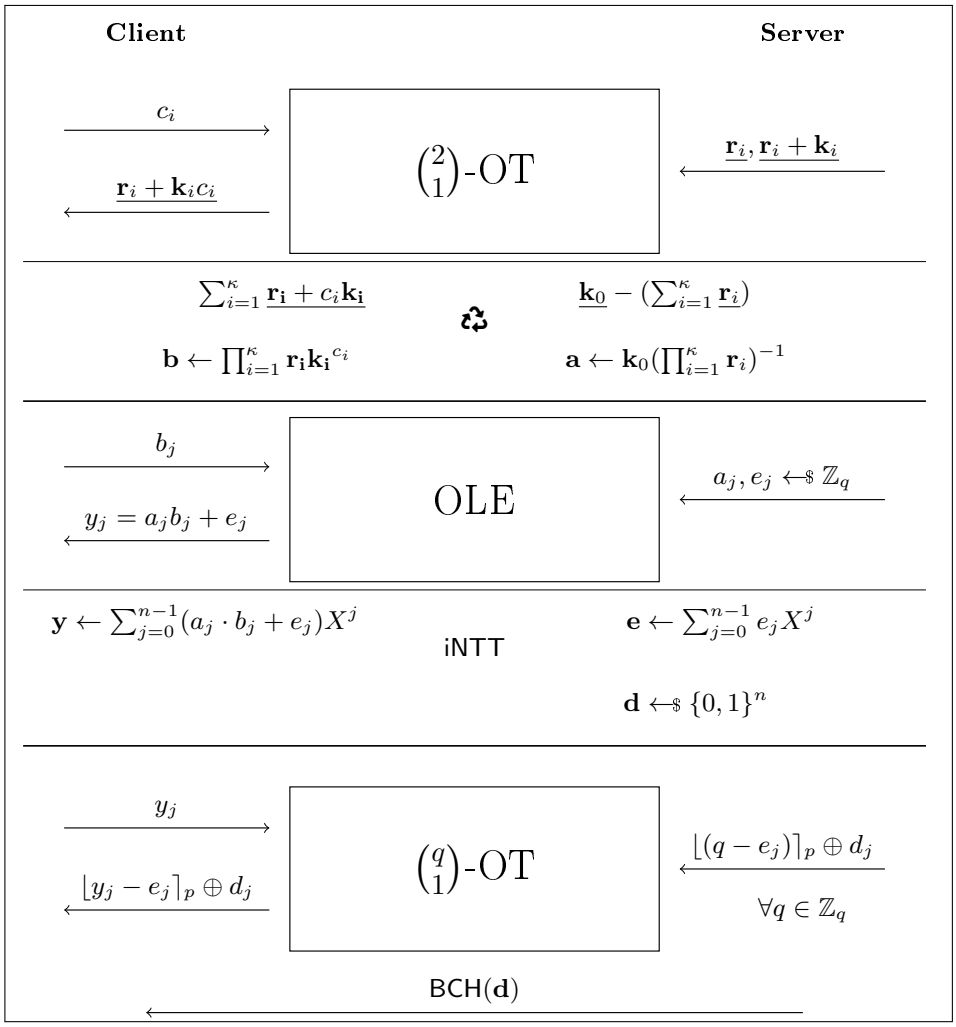

Fig. 4: The full protocol of LEAP. Polynomials in the NTT domain are underlined. ♻ denote the transformation from subset-sum to subset-product and iNTT for the application of the inverse NTT transformation to the client and server polynomials $\mathbf{y}$ and $\mathbf{e}$. The $\binom{2}{1}$-OT step is carried out for each $i \in [\kappa]$, the OLE and $\binom{q}{1}$-OT for each $i \in \{0, \ldots, n-1\}$.

3.2 Oblivious Linear Evaluation

In the next subprotocol, the product of two polynomials is computed to obtain an additive sharing of $\mathbf{k}_0 \prod_{i=1} \mathbf{k}_i^{c_i}$. Note that the polynomials remain in NTT form, and the multiplication of polynomials corresponds to the multiplication of respective coefficients. Therefore, we can employ OLE for each pair of coefficients

to obtain and additive sharing of this result, which allows us to have a protocol with linear instead of superlinear communication complexity.

- **Input Client:** $\mathbf{b} = \prod_{i=1}^{\kappa} \mathbf{r}_i \mathbf{k}_i^{c_i}$ with coefficients $b_0, \ldots, b_{n-1} \in \mathbb{Z}_q$.
- **Input Server:** $\mathbf{a} = \mathbf{k}_0 \prod_{i=1}^{\kappa} \mathbf{r}_i^{-1}$ with coefficients $a_0, \ldots, a_{n-1} \in \mathbb{Z}_q$ and a blinding polynomial $\mathbf{e}$ with coefficients $e_0, \ldots, e_{n-1} \in \mathbb{Z}_q$ sampled uniformly at random.
- **Protocol:** For each pair of coefficients (a_i, b_i), $i = 0, \ldots, n-1$, the client and server run OLE to obtain $y_i = a_i b_i + e_i$. The client collects the coefficients y_i into the polynomial $\mathbf{y}$ and the server does the same for coefficients e_i to obtain the blinding polynomial $\mathbf{e}$.
- **Result:** The client obtains a blinded, additively shared subset-product $\mathbf{y} = \mathbf{ab} + \mathbf{e}$. The server holds the blinding polynomial $\mathbf{e}$.

3.3 Oblivious Rounding

To get the final result, the client polynomial needs to first be unblinded by subtracting $\mathbf{e}$ from $\mathbf{y}$ and then rounded. The intermediate, unblinded result should not be shared between the parties. First, the shares of the server and client are transformed from the point-wise evaluation form to the coefficient space using the inverse NTT (iNTT) transformation. The iNTT is applied only now since the OLE changes the shares from multiplicative to additive, enabling the NTT can be applied linearly.

Both the subtraction and rounding are performed in one step using $\binom{q}{1}$-OT (see Section 2.5). For each coefficient, the server computes all possible rounding results $\lfloor z - e_i \rceil_p$ for $z \in \mathbb{Z}_q$ and submits the result into the $\binom{q}{1}$-OT.

- **Input Client:** Shared subset-product $\mathbf{y} = \mathbf{ab} + \mathbf{e}$.
- **Input Server:** Blinding polynomial $\mathbf{e}$ and a random $\mathbf{d} \leftarrow \{0,1\}^n$.
- **Protocol:** For each $z \in \mathbb{Z}_q$ and each coefficient e_{i-1} of $\mathbf{e}$, $i \in [n]$, the server computes all possible results of the rounding step as $\lfloor z - e_{i-1} \rceil_p \oplus d_i$. Both server and client then perform $\binom{q}{1}$-OT for each coefficient, i.e., for each $i \in [n]$, the client obtains the y_{i-1}-th entry of the server input for the OT which is exactly $\lfloor y_{i-1} - e_{i-1} \rceil_p \oplus d_i$.
- **Result:** The client holds $\mathbf{y}' = (\lfloor y_{i-1} - e_{i-1} \rceil_p \oplus d_i)_{i \in [n]}$.

Note that the size of q is a major factor in the performance of the protocol. Keeping a small q diminishes the computational complexity on the server's side, where it has to iterate over all elements of $\mathbb{Z}_q$. Furthermore, the use of the NTT representation is crucial in the performance of this step as it reduces the complexity of polynomial multiplication from quadratic to quasilinear.

3.4 Applying the BCH Code for Bias Reduction

The SPRING-BCH instantiation uses a BCH code for bias reduction, resulting in a shortened syndrome (see Section 2.4). To apply a BCH code in the two-party computation setting, we leverage the distributive property of the BCH

code. For two binary vectors $\mathbf{f}, \mathbf{g} \in \mathbb{Z}_2^{128}$, the BCH code is commutative such that $\mathsf{BCH}(\mathbf{f} \oplus \mathbf{g}) = \mathsf{BCH}(\mathbf{f}) \oplus \mathsf{BCH}(\mathbf{g})$. Look at this observation, $\mathsf{BCH}(\mathbf{y})$ is computed as $\mathsf{BCH}(\mathbf{y}') \oplus \mathsf{BCH}(\mathbf{d})$. The final step hence requires the server to send $\mathsf{BCH}(\mathbf{d})$ to the client. We present this step as follows:

- **Input Client:** Blinded rounding $\mathbf{y}'$.
- **Input Server:** Blinding factor $\mathbf{d}$.
- **Protocol:** The server computes $\mathsf{BCH}(\mathbf{d})$ and sends it to the server. The client then computes the output of the PRF as $\mathsf{BCH}(\mathbf{y}') \oplus \mathsf{BCH}(\mathbf{d})$.
- **Result:** The client holds $\mathcal{F}_{\mathbf{k}}(x)$.

3.5 OPRF Security

The security of two-party protocols such as OPRFs are commonly proven secure in the UC model. While game-based security definitions exist, they have only been defined for weak notions (cf [CHL22]). Hence, we also consider the security of Leap in the UC model. We present our scheme in a strong model, considering an adversary with the ability to adaptively corrupt parties.

To prove Leap secure, we need to extend the PRF with hash functions (modelled as random oracle) similar to the 2Hash paradigm [JKK14]. We thus consider the modified PRF

$$\mathcal{F}'_{\mathbf{K}}(x) = H_2\left(x, \mathcal{F}_{\mathbf{K}}\left(H_1(x)\right)\right) = H_2\left(x, S\left(\mathbf{k}_0 \cdot \prod_{i=1}^{\kappa} \mathbf{k}_i^{c_i}\right)\right)$$

where $H_1 \colon \{0,1\}^* \to \{0,1\}^\kappa$ with $H_1(x) = c_1, \ldots, c_\kappa$ and $H_2 \colon \{0,1\}^* \times \{0,1\}^k \to \{0,1\}^\ell$ and modeled as a random oracle where ℓ is a polynomial in the security parameter. We note that H_1 does not need to be a random oracle, but map arbitrary message spaces to the input space. Moreover, hash functions are applied on the client side, hence the protocol for oblivious evaluation is unchanged and we do not restate the full protocol.

Theorem 1. *The protocol in Figure 4 for $\mathcal{F}'_{\mathbf{K}}$ realizes the functionality $\mathcal{F}_{OPRF}$ if H_2 modeled as random oracle, the OLE protocol realizes the functionality $\mathcal{F}_{OLE}$, and the two OT protocols realize the functionality $\mathcal{F}_{\binom{N}{1}\text{-}OT}$.*

We sketch the main ideas of the proof here and provide the full proof of the theorem in Section 4. First, unlike in 2HashDH [JKK14], applying a hash H_1 on the client input before the protocol starts is not necessary for the proof. This is because the protocol security does not rely on the hardness of the DLP of the group to which the client input belongs. Since it is impossible for the simulator to hide "DLP trapdoors" in H_1 as in 2HashDH, H_1 does not provide any advantage.

Despite the protocol relying on the security of the subprotocols UC-realizing $\mathcal{F}_{\text{OLE}}$ and $\mathcal{F}_{\binom{N}{1}\text{-OT}}$, the proof is not completely straightforward: the final syndrome $\mathsf{BCH}(\mathbf{d})$ is sent in clear, allowing the environment to modify it and obtain the evaluation of H_2, with some degree of freedom on the inputs, without contacting the oracle. To ensure that H_2 query responses and protocol outputs are

coherent, the simulator uses a family of fake sender identities $\{S_\Delta\}_\Delta$ where Δ represents the difference between the syndrome crafted by the adversary and the one sent by the server during a protocol execution.

For every H_2 query to the oracle on input (x, f), the simulator is able to retrieve the Δ such that $f = \mathcal{F}_{\mathbf{K}}(H_1(x)) + \Delta$ and return a coherent output.

One of the great advantages provided by the subprotocols is that generating "prefixes" becomes trivial. $\mathcal{F}_{\text{OPRF}}$ requires both participants to output a prefix, a partial transcription of the protocol, and if the two prefixes are identical than the adversary has no way to modify the protocol execution to obtain $\mathcal{F'}_{\mathbf{K}}(x')$ on x' of its choice. Thanks to the subprotocols, this kind of attack is never possible. For this reason, the protocol session ID *ssid* is used as a prefix.

4 Security Proof of LEAP

We now show that LEAP for SPRING's BCH mode UC-realizes the $\mathcal{F}_{\text{OPRF}}$ described in Figure 1, in the hybrid world thanks to the functionalities $\mathcal{F}_{\binom{N}{1}\text{-OT}}$ and $\mathcal{F}_{\text{OLE}}$, respectively described in Figure 2 and Figure 3. The following conventions will be used to improve the readability of the proof: $\mathcal{F}$ will be used to represent $\mathcal{F}_{\text{OPRF}}$; $+$ and $-$ will be used instead of $\oplus$ to explicit if we are adding or removing elements even though we are working in $\mathbb{Z}_2$; since H_1 is not modeled as a random oracle, but it is only used to map the client input to a string of fixed length, in the proof, we will not consider this computation and simply work with the client input belonging directly to $\{0,1\}^\kappa$. So, the OPRF output in the proof will be $\mathcal{F'}_{\mathbf{K}}(x) = H_2\left(x, \mathcal{F}_{\mathbf{K}}\left(x\right)\right)$. Note also that despite $\mathcal{F}_{\text{OLE}}$ handles inputs that are vectors, the protocol calls the functionality multiple times to evaluate the OLE on elements.

Theorem 2. *The protocol in Figure 5 for $\mathcal{F'}_{\mathbf{K}}$ realizes the functionality $\mathcal{F}_{OPRF}$ if the OLE protocol realizes the functionality $\mathcal{F}_{OLE}$, and the two OT protocols realize the functionality $\mathcal{F}_{\binom{N}{1}\text{-}OT}$ in the ROM. More precisely, for every adversary $\mathcal{A}$, there exists a simulator* **SIM** *that produces a view in the ideal world that no environment $\mathcal{Z}$ can distinguish with advantage better than $\frac{h}{2^k}$ where h is the number of H_2 query performed by $\mathcal{Z}$ on an uncompromised server.*

Proof. We can always assume that $\mathcal{A}$ acts as a dummy adversary who follows $\mathcal{Z}$'s instructions and shares its view with $\mathcal{Z}$. For every possible environment $\mathcal{Z}$, we will use the simulator **SIM** described in Figure 6.

First, let us consider the messages that **SIM** can craft and are trivially identical to the protocol execution in the hybrid world. These messages are completely independent of any piece of information that is involved in the protocol except the identity of the participants and the session IDs *sid* and *ssid*. The messages are:

- STARTPROTOCOL from $\mathcal{C}$ (if $\mathcal{A}$ changes this message, the protocol cannot reach the end, otherwise the prefixes always match, and the evaluation of the final output is always permitted)

Components: Hash function $H_2 : \{0,1\}^* \times \{0,1\}^k \to \{0,1\}^\ell$, where ℓ polynomial in the security parameter. H_2 is specific to the OPRF instance initialized for a unique session ID *sid*, and they should be implemented by folding *sid* into their inputs.

Initialization: On (INIT, *sid*) from $\mathcal{Z}$, $\mathcal{S}$ picks $\mathbf{k}_0, \ldots, \mathbf{k}_\kappa \leftarrow\$ \mathbb{Z}_{q-1}^n$ and stores $(sid, (\mathbf{k}_0, \ldots, \mathbf{k}_\kappa))$.

Server compromise: On (COMPROMISE, $sid, \mathcal{S}$) from $\mathcal{A}$, $\mathcal{S}$ reveals $(\mathbf{k}_0, \ldots, \mathbf{k}_\kappa)$.

Offline evaluation: On (OFFLINEEVAL, $sid, \mathcal{S}, x$) from $\mathcal{Z}$, $\mathcal{S}$ recovers $(sid, (\mathbf{k}_0, \ldots, \mathbf{k}_\kappa))$ and returns (OFFLINEEVAL,$sid, H_2(x, S(\mathbf{k}_0 \prod_{i=1}^{\kappa} \mathbf{k}_i^{x_i})))$.

Client online evaluation:

- On (EVAL, $sid, ssid, \mathcal{S}', x$) from $\mathcal{Z}$, $\mathcal{C}$ stores $(sid, ssid, x)$; sends (STARTPROTOCOL, $sid, ssid$) to $\mathcal{S}'$, (PREFIX,$sid, ssid, ssid$) to $\mathcal{Z}$ and for $i \in [\kappa]$ sends (RECEIVE,$sid, (ssid, i), \mathcal{S}', \mathcal{C}, x_i$) to $\mathcal{F}_{\binom{2}{1}\text{-OT}}$.
- On (RECEIVED, $sid, (ssid, i), \mathcal{S}, \mathcal{C}, \mathbf{t}_i$) from $\mathcal{F}_{\binom{2}{1}\text{-OT}}$ for $i \in [\kappa]$, $\mathcal{C}$ evaluates $\mathbf{b} = \text{lookup}(\sum_{i=1}^{\kappa} \mathbf{t}_i)$ and for $j \in \{0, \ldots, n-1\}$, $\mathcal{C}$ sends (INPUTR, $sid, (ssid, j), \mathcal{S}, \mathcal{C}, b_j$) to $\mathcal{F}_{\text{OLE}}$.
- On (OUTPUT,$sid, (ssid, i), \mathcal{S}, \mathcal{C}, w_i$) from $\mathcal{F}_{\text{OLE}}$ for $i \in \{0, \ldots, n-1\}$, $\mathcal{C}$ stores $\mathbf{y} = \text{iNTT}(\sum_{i=0}^{n-1} w_i X^i)$. For $j \in [n]$, $\mathcal{C}$ sends (RECEIVE,$sid, (ssid, j), \mathcal{S}, \mathcal{C}, y_{j-1}$) to $\mathcal{F}_{\binom{q}{1}\text{-OT}}$.
- On (BCH, $ssid, \mathbf{s}$) from $\mathcal{S}$ and (RECEIVED,$sid, (ssid, i), \mathcal{S}, \mathcal{C}, u_i$) from $\mathcal{F}_{\binom{q}{1}\text{-OT}}$ for $i \in \{0, \ldots, n-1\}$; $\mathcal{C}$ sends (EVAL, $sid, ssid, H_2(x, \text{BCH}(\mathbf{u}) - \mathbf{s})$)

Server online evaluation: On (STARTPROTOCOL, $sid, ssid$) from $\mathcal{C}$ and (SNDRCOMPLETE, $sid, ssid'$) from $\mathcal{Z}$, $\mathcal{S}$ recovers $(sid, (\mathbf{k}_0, \ldots, \mathbf{k}_\kappa))$ and performs the following actions:

- $\mathcal{S}$ sends (PREFIX, $sid, ssid', ssid$) to $\mathcal{Z}$
- for $i \in [\kappa]$ $\mathcal{S}$ sends (SEND,$sid, (ssid, i), \mathcal{S}, \mathcal{C}, (\mathbf{r}_i, \mathbf{r}_i + \mathbf{k}_i)$) to $\mathcal{F}_{\binom{2}{1}\text{-OT}}$ and stores $\mathbf{a} = \text{lookup}(\mathbf{k}_0 - \sum_{i=1}^{\kappa} \mathbf{r}_i)$
- for $i \in \{0, \ldots, n-1\}$, $\mathcal{S}$ samples $\bar{e}_i \leftarrow\$ \mathbb{Z}_q$ and sends (INPUTS,$sid, (ssid, i), \mathcal{S}, \mathcal{C}, (a_i, \bar{e}_i)$) to $\mathcal{F}_{\text{OLE}}$. $\mathcal{S}$ stores $\mathbf{e} = \text{iNTT}(\sum_{i=0}^{n-1} \bar{e}_i X^i)$
- $\mathcal{S}$ samples $\mathbf{d} \leftarrow\$ \{0,1\}^n$ and for $i \in [n]$, $\mathcal{S}$ sends (SEND,$sid, (ssid, i), \mathcal{S}, \mathcal{C}, (\lfloor z - e_{i-1} \rceil_2 + d_i.\ z \in \mathbb{Z}_q)$) to $\mathcal{F}_{\binom{q}{1}\text{-OT}}$
- $\mathcal{S}$ sends (BCH, $ssid$, BCH($\mathbf{d}$)) to $\mathcal{C}$.

Fig. 5: LEAP protocol in UC $\mathcal{F}_{\binom{N}{1}\text{-OT}}, \mathcal{F}_{\text{OLE}}$−hybrid world

- PREFIX from $\mathcal{S}$ and $\mathcal{C}$ (through communications with $\mathcal{F}$)
- SEND, RECEIVE from $\mathcal{F}_{\binom{N}{1}\text{-OT}}$
- INPUT from $\mathcal{F}_{\text{OLE}}$
- BCH from $\mathcal{S}$ to $\mathcal{C}$ since $\mathbf{d}$ has uniform distribution and BCH$(\cdot)$ is a linear application.

For every session ID sid, which is uniquely bound to a key $\mathbf{k}_0, \ldots, \mathbf{k}_\kappa$, **SIM** keeps a list $L_{sid} = \langle(\Delta, \mathcal{S}_\Delta)\rangle$ where $\Delta \in \{0,1\}^k$ and $\mathcal{S}_\Delta$ is a fake sender identity.

Initialization: On (INIT, $\mathcal{S}, sid$) from $\mathcal{F}$, **SIM** picks $\mathbf{k}_0, \ldots, \mathbf{k}_\kappa \leftarrow\$\ \mathbb{Z}_{q-1}^n$ and stores $(\mathcal{S}, sid, (\mathbf{k}_0, \ldots, \mathbf{k}_\kappa))$.

Server compromise: On (COMPROMISE, $sid, \mathcal{S}$) from $\mathcal{A}$ meant for $\mathcal{S}$, **SIM** forwards it to $\mathcal{F}$ and reveals $(\mathbf{k}_0, \ldots, \mathbf{k}_\kappa)$.

H_2 queries response: On (H_2 QUERY, x, f, sid) from $\mathcal{Z}$ meant for the oracle, **SIM** recovers $\mathbf{k}_0, \ldots, \mathbf{k}_\kappa$ associated to sid and evaluates $\Delta = f - \mathcal{F}_{\mathbf{K}}(x)$:

- if $\Delta = 0$ and $\mathcal{S}$ is not compromised **SIM** sends (HALT) to $\mathcal{Z}$
- if $\Delta = 0$ and $\mathcal{S}$ is compromised, **SIM** sends (OFFLINEEVAL, sid, S, x) to $\mathcal{F}$ and shares its response $F_{sid,\mathcal{S}}(x)$ to $\mathcal{Z}$
- if $\Delta \neq 0$, **SIM** looks for Δ in L_{sid}. If it is not present, **SIM** picks a fresh identity $\mathcal{S}_\Delta$ and add $(\Delta, \mathcal{S}_\Delta)$ to L_{sid}. In both cases sends (OFFLINEEVAL, $sid, \mathcal{S}_\Delta, x$) to $\mathcal{F}$ and shares its response $F_{sid,\mathcal{S}_\Delta}(x)$ to $\mathcal{Z}$.

Client online evaluation:

- On (EVAL, $sid, ssid, \mathcal{C}, \mathcal{S}'$) from $\mathcal{F}$, **SIM** sends ($ssid$) to $\mathcal{F}$ as $\mathcal{C}$'s prefix. Then **SIM** shows $\mathcal{A}$ (STARTPROTOCOL, $ssid$) as a message from $\mathcal{C}$ to $\mathcal{S}'$ and (RECEIVE, $sid, (ssid, i), \mathcal{S}', \mathcal{C}$) for $i \in [\kappa]$ as messages from $\mathcal{F}_{\binom{2}{1}\text{-OT}}$
- On (RECEIVED, $sid, (ssid, i), \mathcal{S}, \mathcal{C}$) from $\mathcal{A}$ meant for $\mathcal{F}_{\binom{2}{1}\text{-OT}}$ for $i \in [\kappa]$ and only if $\mathcal{A}$ didn't change $ssid$ in the STARTPROTOCOL message, **SIM** shows $\mathcal{A}$ (INPUT, $sid, (ssid, j)$) for $j \in \{0, \ldots, n-1\}$ as messages from $\mathcal{F}_{\text{OLE}}$
- On (DELIVER, $sid, (ssid, i), \mathcal{S}, \mathcal{C}$) from $\mathcal{A}$ meant for $\mathcal{F}_{\text{OLE}}$ for $i \in \{0, \ldots, n-1\}$, **SIM** shows $\mathcal{A}$ (RECEIVE, $sid, (ssid, j), \mathcal{S}, \mathcal{C}$) for $j \in [n]$ as messages from $\mathcal{F}_{\binom{q}{1}\text{-OT}}$
- On (RECEIVED, $sid, (ssid, i), \mathcal{S}, \mathcal{C}$) from $\mathcal{A}$ meant for $\mathcal{F}_{\binom{q}{1}\text{-OT}}$ for $i \in [n]$ and on (BCH, $ssid, \mathbf{s}'$) from $\mathcal{A}$ as a message from $\mathcal{S}$ to $\mathcal{C}$, **SIM** retrieves the stored syndrome $\mathbf{s}$ and $\mathcal{A}$ modified and evaluates $\Delta = \mathbf{s}' - \mathbf{s}$:
  - if $\Delta = 0$ **SIM** sends (RCVCOMPLETE, $sid, ssid, \mathcal{C}, \mathcal{S}$) to $\mathcal{F}$
  - if $\Delta \neq 0$, **SIM** looks for Δ in L_{sid}. If it is not present,**SIM** picks a fresh identity $\mathcal{S}_\Delta$ and add $(\Delta, \mathcal{S}_\Delta)$ to L_{sid}. In both cases sends (RCVCOMPLETE, $sid, ssid, \mathcal{C}, \mathcal{S}_\Delta$) to $\mathcal{F}$

Server online evaluation: On (STARTPROTOCOL, $sid, ssid'$) from $\mathcal{A}$ as a message from $\mathcal{C}$ to $\mathcal{S}$ and (SNDRCOMPLETE, $sid, ssid, \mathcal{S}$) from $\mathcal{F}$, **SIM** performs the following actions:

- **SIM** sends ($ssid'$) to $\mathcal{F}$ as $\mathcal{S}$'s prefix
- for $i \in [\kappa]$ **SIM** sends (SEND,$sid, (ssid', i), \mathcal{S}, \mathcal{C}$) to $\mathcal{A}$ as a message from $\mathcal{F}_{\binom{2}{1}\text{-OT}}$.
- for $i \in [n]$, **SIM** sends (INPUT,$sid, (ssid', i-1), \mathcal{S}, \mathcal{C}$) to $\mathcal{A}$ as a message from $\mathcal{F}_{\text{OLE}}$.
- for $i \in [n]$, **SIM** sends (SEND,$sid, (ssid', i), \mathcal{S}, \mathcal{C}$) to $\mathcal{A}$ as a message from $\mathcal{F}_{\binom{q}{1}\text{-OT}}$
- **SIM** samples $\mathbf{s} \leftarrow\$\ \{0,1\}^k$, stores $(\mathbf{s}, ssid')$ and shows $\mathcal{A}$ (BCH, $ssid', \mathbf{s}$) as message from $\mathcal{S}$ to $\mathcal{C}$.

Fig. 6: **SIM** behavior in the ideal world.

Now, we analyze the messages that are not identical in the two worlds but indistinguishable to $\mathcal{Z}$.

Like in [BKLW22], every H_2 instance is uniquely bound to the session ID *sid*, so all queries have input (x, f, sid). The dependency of H_2 from *sid* is kept silent as in [BKLW22].

Let us analyze the view of $\mathcal{Z}$ in the real world:

- On H_2 QUERY on input (x, f, sid) it receives $H_2(x, f)$.
- On COMPROMISE $\mathcal{S}$ from $\mathcal{A}$, $\mathcal{Z}$ recovers the secret keys $\mathbf{k}_0, \ldots, \mathbf{k}_\kappa$.
- On EVAL message to $\mathcal{C}$ on input x, with sender $\mathcal{S}$ who has keys $\mathbf{k}_0, \ldots, \mathbf{k}_\kappa$, $\mathcal{Z}$ can ask $\mathcal{A}$ to modify the syndrome $\mathbf{s}$ to $\mathbf{s}' = \mathbf{s} + \Delta$ for some Δ. The output from $\mathcal{C}$ is $H_2\left(x, \mathcal{F}_{\mathbf{K}}(x) + \Delta\right)$. Note that if the keys are compromised, then $\mathcal{Z}$ has full control over the second input of H_2.

What happens in the ideal world is:

- On H_2 QUERY on input (x, f, sid), **SIM** recovers $\mathbf{k}_0, \ldots, \mathbf{k}_\kappa$ that it has associated to *sid*, evaluates $\Delta = f - \mathcal{F}_{\mathbf{K}}(x)$ and if $\Delta \neq 0$, returns $F_{sid,\mathcal{S}_\Delta(x)}$ to $\mathcal{Z}$. This is always possible when sending an OFFLINEEVAL message to $\mathcal{F}$ since $\mathcal{S} \neq \mathcal{S}_\Delta$. If $\Delta = 0$ and the keys are compromised, then, thanks to OFFLINEEVAL, **SIM** can obtain $F_{sid,\mathcal{S}}(x)$ and share it with $\mathcal{Z}$. If $\Delta = 0$ and the keys are not compromised, then **SIM** cannot produce the correct output and must HALT the simulation.
- On COMPROMISE $\mathcal{S}$ from $\mathcal{A}$, **SIM** discloses to $\mathcal{Z}$ the keys $\mathbf{k}_0, \ldots, \mathbf{k}_\kappa$ associated to $\mathcal{S}$.
- On EVAL message to $\mathcal{C}$ on input x, with sender $\mathcal{S}$, $\mathcal{Z}$ can ask $\mathcal{A}$ to modify the syndrome $\mathbf{s}$ to $\mathbf{s}' = \mathbf{s} + \Delta$ for some Δ. **SIM** can clearly retrieve Δ. If the syndrome is modified $\mathcal{Z}$ receives $F_{sid,\mathcal{S}_\Delta}(x)$, otherwise $F_{sid,\mathcal{S}}(x)$.

So far, **SIM** is able to produce an output in most cases. Multiple EVAL outputs with the same interaction from $\mathcal{Z}$ are consistent, and the same works for H_2 queries. We have to verify that the output created with a H_2 QUERY is coherent with what is obtained during the protocol.

So suppose that $\mathcal{S}$ has uncompromised keys $\mathbf{k}_0, \ldots, \mathbf{k}_\kappa$. Let us assume that $\mathcal{Z}$ performs multiples H_2 queries on the same server $\mathcal{S}$ on the inputs $\{(x_i, f_i)\}_i$ and start the protocol on inputs $\{x_j\}_j$ and modifies the syndromes with $\{\Delta_j\}_j$. Out of all these Inputs, **SIM** lacks only the knowledge of $\{x_j\}_j$. Let us define $\Delta_i = f_i - \mathcal{F}_{\mathbf{K}}(x_i)$ and suppose for a moment that all of them are nonzero. We can also suppose that all Δ_j are different from 0, otherwise a correct $F_{sid,\mathcal{S}}(x_j)$ is returned.

From the set of H_2 queries performed on the inputs $\{x_i\}_i$, $\mathcal{Z}$ receives the messages $\{F_{sid,\mathcal{S}_{\Delta_i}}(x_i)\}_i$, whereas from the protocols executed on inputs $\{x_i\}_i$, $\mathcal{Z}$ obtains the messages $\{F_{sid,\mathcal{S}_{\Delta_j}}(x_j)\}_j$. Without knowing $\mathbf{k}_0, \ldots, \mathbf{k}_\kappa$, the second input of H_2 in a protocol execution is unknown to $\mathcal{Z}$, so it cannot do anything with these results.

If $\mathcal{Z}$ decides to COMPROMISE $\mathcal{S}$, then it has access to the keys $\mathbf{k}_0, \ldots, \mathbf{k}_\kappa$:

– for every x_j, $\mathcal{Z}$ can evaluate $f_j = \mathcal{F}_{\mathbf{K}}(x_j) + \Delta_j$ and ask for a H_2 query on input (x_j, f_j). At this point **SIM** will evaluate $\Delta = f_j - \mathcal{F}_{\mathbf{K}}(x_j) = \Delta_j$. So the result returned is exactly $F_{sid,\mathcal{S}_{\Delta_j}}(x_j)$.
– for every (x_i, f_i), $\mathcal{Z}$ can evaluate $\Delta_i^* = f_i - \mathcal{F}_{\mathbf{K}}(x_i) = \Delta_i$. In order to obtain $H_2(x_i, f_i)$ through the protocol, $\mathcal{Z}$ needs to modify the syndrome in $\mathbf{s}' = \mathbf{s} + \Delta_i$. **SIM**, following its procedure, will make $\mathcal{C}$ return $F_{sid,\mathcal{S}_{\Delta_i}}(x_i)$ to $\mathcal{Z}$.

All messages are coherent, and there is no way for $\mathcal{Z}$ to distinguish reality from simulation. If $\mathcal{Z}$ tries to evaluate H_2 on new inputs after the server compromise, the situation does not change since **SIM** is now in a stronger position.

Let us now consider the probability that the simulator shares the HALT message. This event happens if and only if $\mathcal{Z}$ perform a query to the H_2 oracle on input (x_i, f_i, sid) where sid is related to the uncompromised keys $\mathbf{k}_0, \ldots, \mathbf{k}_\kappa$ and $f_i = \mathcal{F}_{\mathbf{K}}(x_i)$ (which is to say, $\Delta_i = 0$). Since both participants are honest, through protocol executions $\mathcal{Z}$ can only recover x_i, $H_2(x_i, \mathcal{F}_{\mathbf{K}}(x_i))$ and the syndromes $\mathbf{s}$ which have uniform distribution (since $\mathbf{d}$ have). For this reason, it is impossible for $\mathcal{Z}$ to obtain information about $\mathcal{F}_{\mathbf{K}}(x_i)$. The only thing that $\mathcal{Z}$ can do is guess the correct output, which happens with a probability of $1/2^k$ per query. The total advantage $\mathcal{Z}$ has to distinguish the views is exactly $\frac{h}{2^k}$.

5 Instantiation and Implementation

We now turn to the concrete instantiation of Leap and present our reference implementation in C++. The reference implementation is research code and not side-channel resistant. In the implementation, we demonstrate that Leap stands as a competitive construction regarding computational resources. By leveraging the AVX-2 instruction set and the libOTe framework[10] for oblivious evaluation, the online phase of our implementation requires less than a millisecond to compute. Furthermore, the actual computation time of the client amounts to less than 100 μs, exclusive of communication overhead.

We start by discussing performance considerations for the implementation of Spring in Section 5.1, revisit the security analysis of the RLWR parameter selection in Section 5.2, and then move on to the optimizations for each phase depicted in Figure 4. Finally, we provide benchmarks for our reference implementation in Section 5.6, where we also discuss the communication complexity.

5.1 Performance of Spring

The reference implementation of Spring [BBL+15] was originally eight to ten times slower than AES when implemented with SSE instructions, mainly due to clever polynomial transformations, which rely on three key observations. First, Spring employs the Number-Theoretic Transformation (NTT) for the multiplication of the polynomials, which reduces the complexity from $O(n^2)$ to

[10] https://github.com/osu-crypto/libOTe

$O(n \log n)$. The NTT enables the multiplication of polynomials in element-wise point-value form. The main properties we need are:

- $\mathsf{NTT}(\mathbf{a}) \cdot \mathsf{NTT}(\mathbf{b}) = \mathsf{NTT}(\mathbf{a} * \mathbf{b})$
- $\mathsf{iNTT}(\mathbf{a} \cdot \mathbf{b}) = \mathsf{iNTT}(\mathbf{a}) * \mathsf{iNTT}(\mathbf{b})$
- $\mathsf{NTT}(\mathbf{a}) + \mathsf{NTT}(\mathbf{b}) = \mathsf{NTT}(\mathbf{a} + \mathbf{b})$
- $\mathsf{iNTT}(\mathbf{a}) + \mathsf{iNTT}(\mathbf{b}) = \mathsf{iNTT}(\mathbf{a} + \mathbf{b})$

In the case of the OPRF, the polynomials used as the server key need to be transformed only once and can be stored in NTT form. This is a standard technique in lattice-based schemes. To be even more efficient, the keys can be sampled directly in *NTT form*. The procedure remains the same as when sampling normal polynomials, with the additional constraint that the polynomials must have no zero coefficients so the polynomial is invertible.

This condition enables us to convert the subset-product to a subset-sum using the observation that for some generator g for $\mathbb{Z}_q^*$, $g^a g^b = g^{a+b}$. Two conversion steps would be necessary to replace the multiplication with addition: The first conversion is to the subset-sum form to map an element x to its subset-sum representation g^a and a second step to map the result g^{a+b} back to the subset-product representation. Both steps can be computed with a simple table lookup.

Note that by Fermat's theorem, $g^{q-1} \equiv g^0 \equiv 1 \mod q$ for a prime q, so we can sample from a uniform distribution $\mod q - 1$. For the parameters of Spring, 3 is used as the generator g. Since the polynomial may not contain zeros, the table lookup to compute the inverse discrete logarithm substitutes the modular exponentiation of 1 to $q - 1$, thus preventing the polynomial from having zero coefficients. The polynomial coefficients now range from $[0, 255]$ and fit perfectly in a single byte. In addition, using $q = 257$ allows for free modular operations with wrapping arithmetic. Therefore, the polynomials (and later blinding elements) are sampled in Subset-Sum NTT form for the implementation, saving a table lookup and an NTT computation. The optimized Spring PRF proceeds as follows:

$$\mathcal{F}_{\mathbf{K}}(c_1, \cdots, c_\kappa) = S\left(\mathsf{iNTT}\left(\mathsf{lookup}\left(\mathbf{k}_0 + \sum_{i=1}^{\kappa} \mathbf{k}_i^{c_i}\right)\right)\right)$$

5.2 Security Analysis of Spring

Compared to other protocols and cryptographic schemes built from lattice assumptions, Spring has a much smaller modulus q. The predecessing BPR protocol [BPR12] requires an exponentially large modulus relative to the PRF input length, which is prohibitive for the performance of concrete instantiations. The authors point out that this may be a proof artifact. Thus, the parameters of Spring are based on heuristic analysis of the hardness of the underlying problem.

Related work extending Spring [BDFK17] to a PRG gives a more detailed security analysis, concluding that the parameters selected for Spring ensure that it is a secure PRF. More concretely, the authors describe a potential birthday

attack targeting a small portion of the PRF's internal state, primarily affecting its counter mode. The paper also covers Gröbner basis attacks, where the algorithm remains secure even when utilizing the BKW attack with a small p. The most efficient attack against the BCH mode involves detecting the small bias in the output bits.

We now discuss the parametrization (including modulus q and dimension n) of related cryptosystems to explain their choice, particularly in comparison to SPRING. In particular, we look at ML-KEM amid the ongoing NIST competition for post-quantum primitives standardization [Moo22]. Moreover, we check with the lattice estimator [APS15, ACD+18] to ensure there are no known lattice attacks against the parameters.

Modulus q. Setting q to 257 or 514 is significantly lower than, for example, ML-KEM's[11] $q = 3329$ [RL23]. However, ML-KEM requires a larger modulus to decrease the failure probability for the CCA security requirement. A similarly small modulus is chosen by the FALCON-based KEM BAT [ETWY22], which sets $q = \{128, 257, 769\}$ for their NTRU-based KEM. The modulus $q = 257$ is used for their target level of 128 bits, which is equal to the target level of SPRING.

Dimension n. In ML-KEM, the dimension is chosen to be $n = 128$ to encapsulate 256-bit keys. BAT sets $n = \{256, 512, 1024\}$, with $n = 512$ being the target dimension for a security level of 128 bits. To show $n = 128$ is sufficient, we estimate the security for $n = \{128, 256, 512\}$.

Robustness against lattice attacks. As the Lattice Estimator [APS15, ACD+18] currently does not provide estimations for (R)LWR, we rely on the fact that for specific choices of parameters (R)LWR is as hard as (R)LWE [BGM+16]. Specifically, when setting the secret distribution X_s to be uniform over $\mathbb{Z}_q$ and the noise distribution X_e is uniform over the integers in the range $[-\frac{q}{2p}, \ldots, \frac{q}{2p})$, provided q is a multiple of $p = 2$. In case of uneven $q = 257$, we round the term down to $\lfloor \frac{q}{p} \rfloor$. With this choice of distributions, the results from the Lattice Estimator provide lower bounds for the corresponding RLWR instance. The Lattice Estimator[12] [APS15, ACD+18] output presented in Table 2 shows that common lattice attacks are not feasible for both BCH and CRT instantiations of SPRING.

5.3 Efficient Blind Subset Computation

The subset-sum computation step from Section 3.1 can be computed efficiently using the OT extensions from Section 2.5. First, the server computes the uniformly random polynomials $\mathbf{r}_{0,i}, \mathbf{r}_{1,i}$ from the base OT strings $r_{0,i}, r_{1,i}$ using an extendable output function (XOF), squeezing 32 bytes per polynomial. The unblinding polynomial $\mathbf{a}$ is set to zero. The client expands all their OT results $r_{c_\$,i}$

[11] ML-KEM (derived from Kyber [SAB+20]) is secure under the Module Learning With Error (MLWE) assumption. Note that Ring-LWE is a special case of MWLE. The same is true for the Learning with Rounding assumptions.

[12] Using commit ID `a7738f4cf9d985bf7d7e063320d8e0763daf6ac8`

Table 2: Estimations of attack complexity against different instantiations of the SPRING PRF using the Lattice Estimator, called with `LWE.Parameters(n=n, q=q, Xe=NoiseDistribution.UniformMod((q//2)), Xs=NoiseDistribution.UniformMod(2))`. The USVP output was inifinity for all parameter inputs.

		dual hybrid		arora-gb	
q	n	rop	mem	rop	mem
257	128	2^{867}	2^{866}	$2^{503.3}$	$2^{503.3}$
	256	$2^{1764.1}$	$2^{1763.1}$	$2^{696.2}$	$2^{696.2}$
	512	$2^{3557.1}$	$2^{3556.1}$	$2^{914.7}$	$2^{914.7}$
514	128	$2^{994.8}$	$2^{993.8}$	2^{∞}	2^{∞}
	256	$2^{2020.6}$	$2^{2019.6}$	$2^{1016.3}$	$2^{1016.3}$
	512	2^{4071}	2^{4070}	2^{∞}	2^{∞}

to $\mathbf{r}_{c_\$,i}$ and sets a polynomial $\mathbf{a}$ to zero. After the computation, $\mathbf{a}$ will contain the subset-sum.

To compute the subset-sum, the client and the server engage in the following protocol for all $i \in (1, \ldots, \kappa)$: First, the client computes a correction vector where each correction bit b_i is computed as $b_i = c_i \oplus c_{\$,i}$. The server takes each b_i and adds $\mathbf{r}_{b,i}^{-1}$ to $\mathbf{a}$. Then, the server replies with the polynomial $\mathbf{r}_i = \mathbf{r}_{1-b_i} \oplus (\mathbf{r}_{b,i} + \mathbf{k}_i)$. On the client side, if $c_i = 0$, $\mathbf{r}_{c_\$,i}$ is added to $\mathbf{a}$. If $c_i = 1$, the client adds $\mathbf{r}_{c_i} \oplus \mathbf{r}_i$ to $\mathbf{a}$ instead.

Performance. The blinding polynomials are derived from the OT extension strings. Since the extraction step is performed in the base phase, the online subset-sum computation only requires to compute the correction elements and some additions, which can be performed using vector instructions and can be efficiently pipelined.

Implementation Considerations. The subset-sum computation requires a table lookup to transition to subset-product form. There are two possible ways of obtaining an implementation without side channels: Either by bit-slicing the lookup table or by foregoing the subset-sum representation and conducting the Naor-Reingold aggregation multiplicatively in subset-product form.

5.4 OLE Computation

In our implementation, we use Gilboa's product-sharing protocol [Gil99], which is a specific OLE. This protocol computes the product of two m-bit numbers using m OT calls. It works as follows:

- The client sets $y = 0$, and the server sets $e = 0$.
- The client's input to the OT is the bitwise decomposition of b such that $b = \sum_{i=1}^{m} 2^{i-1} b_i$.

- The client and the server engage in m OT calls. The server samples $e_i \in \mathbb{Z}_p$ and computes $c_i = a(1 \ll i) + e_i \mod p$. The server inputs to the OT are (e_i, c_i). The server updates $e = e + e_i$.
- From the $\binom{2}{1}$-OT, the client obtains $y_i = e_i + a(1 \ll (b_i i))$, where b_i is the ith bit of b, and updates $y = y + y_i$.

The subset-sum polynomials are lifted to subset-product polynomials. Again, we use the OT extension to obtain masking elements. Computing Gilboa's OLE with OT extensions is very similar to computing the blinded subset-sum. The client again computes the correction. If the ith correction bit is set, the server responds with $r1[i] \oplus (((1 \ll k)a) + r0[i])$, and with $r0[i] \oplus (((1 \ll k)a) + r1[i])$ otherwise. The OLE evaluation is carried out for each polynomial coefficient $a \in \mathbf{a}$, in the case of SPRING, where $n = 128$, 128 times.

Performance. The OLE protocol can be implemented efficiently using only basic arithmetic. Our reference implementation consumes more randomness than necessary since libOTe only supports extracting 128-bit secrets while we only need nine bits for the masking term e_i, discarding the remaining 119 bits. To reduce the networking load in the preprocessing phase, the OT would need to be tailored to the smaller output. A small speedup comes from preprocessing the extraction of the random bits, which is performed in the preprocessing phase using rejection sampling and an extensible output function.

5.5 Rounding and BCH code

The linearly shared polynomials are now put through the inverse NTT transformation. The NTT itself is already heavily optimized in the original paper. We use a simplified version of the NTT in our implementation and point out that an efficient NTT will likely give another performance boost to the protocol.

For the $\binom{N}{1}$-OT, we originally wanted to use the KKRT OT protocol [KKRT16]. A quick test showed that the libOTe implementation of the KKRT protocol takes between 6 and 7 milliseconds for 128 OT calls, which is over 80% of the total execution time for the OPRF. The overhead is caused by choosing one of only 128 entries, as KKRT is usually used for private set intersection algorithms, where the set size is much larger. Using OT extensions and the Naor-Pinkas approach for Polynomial Evaluation [NP99], as described in Section 2.5, we are able to bring down the runtime significantly. A nice feature of this approach is that we were able to use the base OT for all protocol steps, which significantly reduces the overhead in the base phase and also makes the implementation simpler.

Performance. The server packs the results in 128 bit blocks. The client holding the OLE output y_i can access the result in the $128y + i^{th}$ bit. The data structure is drawn in Figure 7. Similar to the OLE, we waste 127 bits of ran-

domness as we only need a single bit from the 128-bit OT extension output.

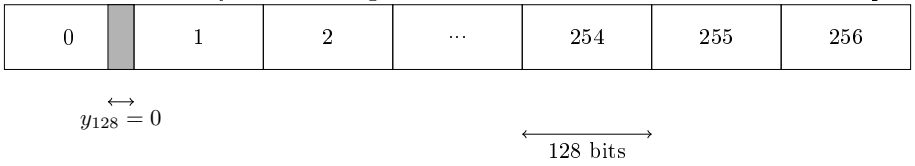

Fig. 7: Representation of rounding results in memory. The client picks the index of the encrypted rounding result depending on their OLE result. For example, if y_{128} is zero, the client takes the 127th bit of the array.

5.6 Communication Evaluation and Benchmarks

Based on our reference implementation[13] in C++ using libOTe for the oblivious primitives, we evaluate both the computational and communication overhead. The OPRF input is 128 bits for all benchmarks. All benchmarks were performed on a computer running Ubuntu Linux version 22.04.1 with the 6.2.0-37-generic kernel release. The processor is an AMD Ryzen 9 7900X 12-Core Processor, with the processor frequency fixed at 4.7 MHz. The computer has 128 GiB of RAM.

The evaluation of the OPRF can be divided into two distinct phases: an input-independent preprocessing phase and a fast, input-dependent online phase.

Preprocessing Phase. The online phase requires some preprocessing in the form of base OT. We consider two potential OT candidates: The SimplestOT protocol [CO15] over elliptic curves, which may be vulnerable to Shor's algorithm [Sho94], and the KEM-based endemic OT using ML-KEM (KyberOT), which is secure against quantum adversaries [MR19], but not in the UC model. We stress that any OT may be used for the OT evaluation, which makes our construction very flexible. Both protocols are used to generate base OT that is then extended using either the IKNP protocol [IKNP03] or SilentOT [BCG+19].

In Table 3, we present the numbers obtained from the different combinations. A clear trade-off between communication overhead and computation time is visible. Although the KyberOT protocol introduces a large constant to the communication, the shorter communication time may be beneficial if only a few OPRF evaluations are needed. The communication/computation trade-off may be optimized further using SoftSpokenOT [Roy22], depending on the application using our OPRF.

Online Phase. In Table 4, we show the overall communication cost and the communication cost per protocol phase, as well as the computational costs. The theoretical client communication cost is very low with 304 bytes, which are correction elements for the base OT. In the implementation, libOTe tacks on 8 bytes per phase for synchronization, bringing the total communication to 328 bytes for the client.

[13] https://github.com/meyira/leap

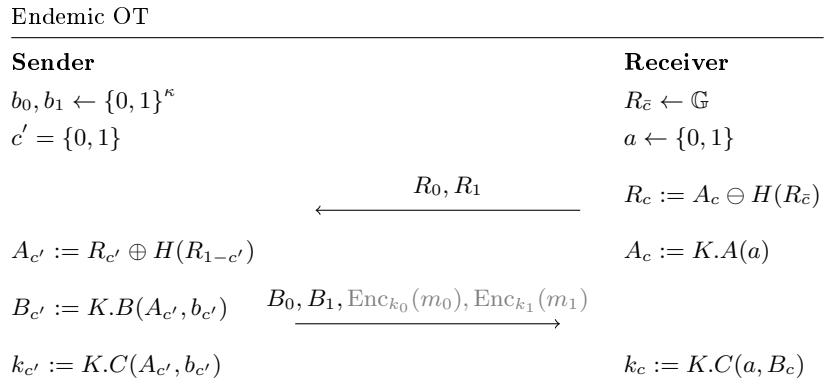

Fig. 8: KEM-agnostic description of endemic OT protocol from key agreement in a group $\mathbb{G}$ used for KyberOT. $K.A(x)$ is an algorithm generating a public key from a private key, usually from some starting element. $K.B(X, x)$ is used to derive another group element, starting from a specific element. $K.C(X, x)$ uses x to unmask X. Note that the sender computes two private keys A_0, A_1 and encapsulates them in two messages B_0, B_1. The encrypted messages are grey to denote they are optional in a random OT setting.

Table 3: Communication and computation overhead of OT protocols of the preprocessing step for $1 = 2^0$ and 2^{13} batched OPRF evaluations. K.OT is the quantum-secure KyberOT of [MR19], S.OT is the SimplestOT from [CO15] based on classical assumptions. The measurements are the median taken over 100 runs.

#	**Protocol**	**Communication**		**Computation**	
		Client	Server	Client	Server
2^0	S.OT+IKNP	39 kB	**4 kB**	63 ms	63 ms
	K.OT+IKNP	465 kB	328 kB	**10 ms**	**10 ms**
	S.OT+Silent	**22 kB**	22 kB	68 ms	68 ms
	K.OT+Silent	448 kB	392 kB	**10 ms**	10 ms
2^{13}	S.OT+IKNP	319 MB	**4.26 kB**	420 ms	463 ms
	K.OT+IKNP	319 MB	328 kB	**311 ms**	**423 ms**
	S.OT+Silent	**46 kB**	155 kB	2065 ms	3371 ms
	K.OT+Silent	487 kB	559 kB	2130 ms	3496 ms

The server needs to send 128 subset polynomials to the client in the subset-sum phase, each of which is 128 bytes long. In the OLE step, the server computes the encryption of nine bits for each of the 257 possible outputs, 128 times. The implementation does not implement bit packing here and sends each nine-bit number in 16 bits, which adds around 1 kB of communication overhead. In the

rounding step, each of the 257 possible client results are added as a single bit for each of the 128 polynomial coefficients. Finally, the random BCH code adds another 8 bytes. Due to synchronization, the libOTe communication overhead is then at 22840 bytes or 22.8 kilobytes.

Table 4: Communication and Computation of our protocol per phase and overall. The *Overall, Network* row includes the overhead for network synchronization and the padding for the OLE step.

Phase	**Client**			**Server**		
	Comm.	Comp.	Idle	Comm.	Comp.	Idle
Subset-Sum	16 bytes	5 μs	21 μs	16 384 bytes	863 μs	46 μs
OLE	144 bytes	4 μs	3 μs	1 296 bytes	72 μs	88 μs
Rounding	144 bytes	2 μs	726 μs	4 118 bytes	814 μs	67 μs
BCH	0 bytes	1 μs	0 μs	8 bytes	/	26 μs
Overall (Network)	304 bytes (328 bytes)	11 μs	750 μs	21 806 bytes (22 840 bytes)	1.7 ms	227 μs

From a computational perspective, the rounding phase seems to offer the most optimization potential, as computing all $n \cdot q$ possible results takes a significant amount of time. The BCH step of the client includes a call to an XOF to post-process the OPRF output.

These numbers can be improved further. Due to implementation constraints imposed by libOTe, which considers blocks of 128 bits for OLE and $\binom{q}{1}$-OT, whereas nine bits would be sufficient for Leap. Hence, our proof-of-concept implementation is unable to use approximately 3808 bytes of randomness in the online phase per OPRF computation, as $128 - 9 = 119$ bits are discarded per OT output used for either the OLE or the $\binom{q}{1}$-OT.

6 Application: Private Set Intersection

In private set intersection (PSI), two parties each hold a set of data. They aim to compute the intersection of their sets without revealing the items that are not shared. Use cases include *malware detection, checking compromised credentials, private database querying, contact discovery* [MAL23], and many more.

In the balanced PSI case, where both set sizes are equal, the best protocol is Vole-PSI [RS21], which uses a programmable OPRF to absorb multiple inputs at once. The authors can achieve a set intersection computation in less than a second locally for set sizes of 2^{16}.

OPRFs are particularly useful for the unbalanced case, where one set is significantly larger than the other. Here, the server samples a key and uses it to compute a PRF over each item in the larger set. The PRF outputs are then inserted in an efficient data structure, e.g., a Cuckoo filter [FAKM14]. The Cuckoo

Table 5: Communication and computation in PSI using ML-KEM+IKNP for the Base phase to illustrate the flexibility and trade-off of our protocol. The computation times are the median over 10 runs.

|S|	|C|	Setup C	Setup S	Base C	Base S	Online C	Online S
2^{20}	2^{1}	4.5 s 0 s	4.35 s 4.2 MiB	0.1 s 0.48 MiB	0.1 s 0.3 MiB	0.003 s 0.4 kiB	0.01 s 44 kiB
	2^{10}	4.2 s 0 s	4.33 s 4.2 MiB	0.5 s 38.4MiB	1 s 0.3 MiB	1.7 s 0.3 MiB	1.2 s 22.3 MiB
2^{24}	2^{15}	68 s 0 s	68 s 67 MiB	16.4 s 1.18 GiB	27s 0.3 MiB	51.4 s 9.5 MiB	38.4 s 712.8 MiB

filter is then sent to the client. In order to check whether a client element is contained in the set, the client and the server engage in an OPRF protocol with the client input and the server key. The resulting PRF value can be used by the client to query the Cuckoo filter. Currently, the fastest protocol employing this method is DisCo [HSW23], which is built upon the work of Kales et al. [KRS+19]. Both PSI protocols send the Cuckoo filter in a Setup phase. Then, data-independent base OT and OT extension computations are performed during the base phase. During the following online phase, the client and server engage in the online phase of the OPRF computation. The client checks if the results are in the Cuckoo filter obtained in the setup phase.

The OPUS paper [HHM+24b] also demonstrates private set intersection using their isogeny-based OPRFs. However, our approach (using Leap) performs significantly better across all metrics except online server communication, where OPUS is a bit better. However, OPUS requires a linear number of rounds concerning the input size and involves expensive isogeny computations.

To demonstrate the effectiveness of our OPRF, we instantiate the implementation of [KRS+19] with our OPRF[14] In comparison, DisCo uses database partitioning and client query scheduling techniques to enhance the protocols performance. We stress that, to the best of our knowledge, the techniques from DisCo can be directly applied to our protocol. However, as our current focus is on demonstrating the efficiency of our OPRF compared to others, we have not implemented these techniques.

In Table 5, we show that Leap is a competitive construction for private set intersection. The client's setup time is negligable in practice, as it only has to receive the Cuckoo filter. For simplicity, the client waits synchronously for the server in our proof of concept implementation. However, in practice, the client would not receive the entire Cuckoo filter and instead use a hybrid approach similar to DisCo [HSW23].

[14] The implementation is available on request and will be made public with the release of this paper.

Table 6: Communication and computation in PSI using SimplestOT+SilentOT for the Base phase to illustrate the flexibility and trade-off of our protocol. The computation times are the median over 10 runs.

|S|	|C|	Setup C	Setup S	Base C	Base S	Online C	Online S
2^{20}	2^1	4.2 s	4.2 s	0.005 s	0.009s	0.03 s	0.01 s
		0 s	4.2 MiB	0.02 MiB	0.06 MiB	0.4 kiB	44 kiB
	2^{10}	4.3 s	4.2 s	0.8 s	1.3 s	1.7 s	1.2 s
		0 s	4.2 MiB	0.33 MiB	0.16 MiB	0.3 MiB	22.3 MiB
2^{24}	2^{15}	68.9 s	68.9 s	26.4 s	43.9 s	56 s	38.6 s
		0 s	67 MiB	9.5 MiB	0.16 MiB	9.5 MiB	712 MiB

7 Conclusion and Future Work

In this work, we introduced – Leap –, a protocol for the efficient oblivious evaluation of Spring in BCH mode. In Leap, we show how to leverage the design ideas of Spring to construct an efficient OPRF that we believe to be secure against quantum adversaries. Concretely, the OPRF uses the Learning with Rounding hardness assumption and a BCH code for bias reduction in the PRF output. Beyond applying our OPRF in the context of unbalanced PSI, we envision our OPRF also to be useful for various other applications. Specifically, applications that require multiple OPRF evaluations can benefit from batching the preprocessing phase and a very efficient online phase.

Beyond the applications of our protocol, our construction also gives rise to a new approach in constructing OPRFs from PRFs that follow the Naor-Reingold paradigm in principle but where the OT-based transform [FIPR05] is not applicable due to some deviation.

While our reference implementation is already efficient, it can be significantly improved by using both more of the strategies used in the original paper and by optimizing the underlying primitives. In addition, mismatch in output sizes supported by the libOTe and required by Leap leaves potential room for optimizations with targeted OT and OLE constructions and implementations. Finally, further investigation into Spring-CTR and the counter mode, as well as investigating alternatives to using the BCH code for bias reduction, to allow more output bits would make the protocol more flexible as well.

Acknowledgments. We thank Lukas Helminger for fruitful discussions during the design phase of Leap, Martin Albrecht for discussions concerning the security of SPRING. Sebastian Felix and Mateusz Zalega helped with some debugging of the reference implementation. We also thank the anonymous reviewers at Eurocrypt 2025 and NDSS 2024 for helpful comments on the paper.

This work is partially funded by the Digital Europe Program under grant agreement number 101091642 ("QCI-CAT"), the European Union's Horizon Europe research and innovation program under the project "Quantum Security

	Parameters		Setup		Online	
	$\|S\|$	$\|C\|$	$\|S\|$	$\|C\|$	$\|S\|$	$\|C\|$
ECNR	2^0	2^0	10 ms 133 bytes	0 s 0 bytes	0.23 s 12.04 kiB	0.05 s 16 bytes
	2^5	2^5	0.02 s 262 bytes	0 s 0 bytes	0.21 s 137.05 kiB	0.06 s 512 bytes
	2^{10}	2^{10}	0.3 s 4.36 kiB	0 s 0 bytes	0.64 s 4.04 MiB	0.57 s 16 kiB
NR-OT	2^0	2^0	0.26 s 134 bytes	0.51 s 1 byte	0.06 s 128 kiB	0.10 s 0.75MiB
	2^5	2^5	1.63 s 263 bytes	1.88 s 1 byte	3.11 s 4MiB	3.15 s 8.5 MiB
	2^{10}	2^{10}	45.04 s 4.31 MiB	45.28 s 1 byte	99.66 s 128 MiB	99.71 s 256.6 MiB
OPUS	2^0	2^0	0.26 s 133 bytes	0.26 s 0 bytes	15.47 s 17.07 kiB	15.91 s 9.04 kiB
	2^5	2^5	8.71 s 262 bytes	8.71 s 0 bytes	328.46 s 546.25 kiB	329.14 s 290.26 kiB
	2^{10}	2^{10}	303.38 s 4.31 kiB	303.38 s 0 bytes	16367.12 s 34.14 MiB	16367.60 s 18.08 MiB
LEAP	2^0	2^0	1 ms 117 bytes	1 ms 0 bytes	0.2s 73.4 kiB	6 ms 20 KiB
	2^5	2^5	1 ms 246 bytes	1 ms 0 bytes	0.08 s 802 kiB	0.07 s 47.5 kiB
	2^{10}	2^{10}	4 ms 4.30 kiB	5 ms 0 bytes	2.4 s 22.39 MiB	2.46 s 646.5 kiB

Table 7: Performance of Naor-Reingold OPRFs for PSI. Concretely, the performance of OPUS, isogeny-based NR-OT using their reference implementation with an additive homomorphic encryption OT [BDK+20] and LEAP using IKNP+KyberOT are compared to the ECNR protocol that is not secure against quantum computers.

Networks Partnership" (QSNP, grant agreement number 101114043), the European Union's Horizon Europe project SUNRISE (project no. 101073821), by REMINDER, a project funded by the Austrian Science Fund (FWF) under project number I 6650-N, and by PREPARED, a project funded by the Austrian security research programme KIRAS of the Federal Ministry of Finance (BMF).

References

ABB+13. Michel Abdalla, Fabrice Benhamouda, Olivier Blazy, Céline Chevalier, and David Pointcheval. SPHF-friendly non-interactive commitments. In Kazue

Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part I*, volume 8269 of *LNCS*, pages 214–234. Springer, Berlin, Heidelberg, December 2013.

ACD+18. Martin R. Albrecht, Benjamin R. Curtis, Amit Deo, Alex Davidson, Rachel Player, Eamonn W. Postlethwaite, Fernando Virdia, and Thomas Wunderer. Estimate all the LWE, NTRU schemes! In Dario Catalano and Roberto De Prisco, editors, *SCN 18*, volume 11035 of *LNCS*, pages 351–367. Springer, Cham, September 2018.

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

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

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

Ajt96. Miklós Ajtai. Generating hard instances of lattice problems (extended abstract). In *28th ACM STOC*, pages 99–108. ACM Press, May 1996.

APRR24. Navid Alamati, Guru-Vamsi Policharla, Srinivasan Raghuraman, and Peter Rindal. Improved alternating-moduli PRFs and post-quantum signatures. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part VIII*, volume 14927 of *LNCS*, pages 274–308. Springer, Cham, August 2024.

APS15. Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Math. Cryptol.*, 9(3):169–203, 2015.

Bas24a. Andrea Basso. Poke: A framework for efficient pkes, split kems, and oprfs from higher-dimensional isogenies. Cryptology ePrint Archive, Paper 2024/624, 2024. https://eprint.iacr.org/2024/624.

Bas24b. Andrea Basso. A post-quantum round-optimal oblivious PRF from isogenies. In Claude Carlet, Kalikinkar Mandal, and Vincent Rijmen, editors, *SAC 2023*, volume 14201 of *LNCS*, pages 147–168. Springer, Cham, August 2024.

BBL+15. Abhishek Banerjee, Hai Brenner, Gaëtan Leurent, Chris Peikert, and Alon Rosen. SPRING: Fast pseudorandom functions from rounded ring products. In Carlos Cid and Christian Rechberger, editors, *FSE 2014*, volume 8540 of *LNCS*, pages 38–57. Springer, Berlin, Heidelberg, March 2015.

BCG+19. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 489–518. Springer, Cham, August 2019.

BDFK17. Charles Bouillaguet, Claire Delaplace, Pierre-Alain Fouque, and Paul Kirchner. Fast lattice-based encryption: Stretching spring. In Tanja Lange and Tsuyoshi Takagi, editors, *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017*, pages 125–142. Springer, Cham, 2017.

BDK+20. Niklas Büscher, Daniel Demmler, Nikolaos P. Karvelas, Stefan Katzenbeisser, Juliane Krämer, Deevashwer Rathee, Thomas Schneider, and Patrick Struck. Secure two-party computation in a quantum world. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *ACNS 20International Conference on Applied Cryptography and Network Security, Part I*, volume 12146 of *LNCS*, pages 461–480. Springer, Cham, October 2020.

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

BGM+16. Andrej Bogdanov, Siyao Guo, Daniel Masny, Silas Richelson, and Alon Rosen. On the hardness of learning with rounding over small modulus. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A, Part I*, volume 9562 of *LNCS*, pages 209–224. Springer, Berlin, Heidelberg, January 2016.

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

BKLW22. Daniel Bourdrez, Dr. Hugo Krawczyk, Kevin Lewi, and Christopher A. Wood. The OPAQUE Asymmetric PAKE Protocol. Internet-Draft draft-irtf-cfrg-opaque-09, Internet Engineering Task Force, July 2022. Work in Progress.

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, Cham, December 2021.

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

BPR12. Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 719–737. Springer, Berlin, Heidelberg, April 2012.

Can01. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

CDGS19. Jan Camenisch, Angelo De Caro, Esha Ghosh, and Alessandro Sorniotti. Oblivious PRF on committed vector inputs and application to deduplication of encrypted data. In Ian Goldberg and Tyler Moore, editors, *FC 2019*, volume 11598 of *LNCS*, pages 337–356. Springer, Cham, February 2019.

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

CO15. Tung Chou and Claudio Orlandi. The simplest protocol for oblivious transfer. In Kristin E. Lauter and Francisco Rodríguez-Henríquez, editors, *LATINCRYPT 2015*, volume 9230 of *LNCS*, pages 40–58. Springer, Cham, August 2015.

DFK+23. Luca De Feo, Tako Boris Fouotsa, Péter Kutas, Antonin Leroux, Simon-Philipp Merz, Lorenz Panny, and Benjamin Wesolowski. SCALLOP: Scaling the CSI-FiSh. In Alexandra Boldyreva and Vladimir Kolesnikov, editors, *PKC 2023, Part I*, volume 13940 of *LNCS*, pages 345–375. Springer, Cham, May 2023.

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

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.

DP24. Cyprien Delpech de Saint Guilhem and Robi Pedersen. New proof systems and an OPRF from CSIDH. In Qiang Tang and Vanessa Teague, editors, *PKC 2024, Part II*, volume 14603 of *LNCS*, pages 217–251. Springer, Cham, April 2024.

DRRT18. Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. PIR-PSI: Scaling private contact discovery. *PoPETs*, 2018(4):159–178, October 2018.

ETWY22. Thomas Espitau, Mehdi Tibouchi, Alexandre Wallet, and Yang Yu. Shorter hash-and-sign lattice-based signatures. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 245–275. Springer, Cham, August 2022.

FAKM14. Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. Cuckoo filter: Practically better than bloom. In *CoNEXT*, pages 75–88. ACM, 2014.

FHV13. Sebastian Faust, Carmit Hazay, and Daniele Venturi. Outsourced pattern matching. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *ICALP 2013, Part II*, volume 7966 of *LNCS*, pages 545–556. Springer, Berlin, Heidelberg, July 2013.

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

FOO23. Sebastian H. Faller, Astrid Ottenhues, and Johannes Ottenhues. Composable oblivious pseudo-random functions via garbled circuits. In *LATINCRYPT 2023*, 2023.

GGM86. Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, October 1986.

Gil99. Niv Gilboa. Two party RSA key generation. In *CRYPTO '99*, volume 1666 of *LNCS*, pages 116–129. Springer, 1999.

GNN17. Satrajit Ghosh, Jesper Buus Nielsen, and Tobias Nilges. Maliciously secure oblivious linear function evaluation with constant overhead. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 629–659. Springer, Cham, December 2017.

HHM+24a. Lena Heimberger, Tobias Hennerbichler, Fredrik Meisingseth, Sebastian Ramacher, and Christian Rechberger. OPRFs from isogenies: Designs and analysis. In Jianying Zhou, Tony Q. S. Quek, Debin Gao, and Alvaro A. Cárdenas, editors, *ASIACCS 24*. ACM Press, July 2024.

HHM+24b. Lena Heimberger, Tobias Hennerbichler, Fredrik Meisingseth, Sebastian Ramacher, and Christian Rechberger. Oprfs from isogenies: Designs and analysis. In *AsiaCCS*. ACM, 2024.

HL08. Carmit Hazay and Yehuda Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In Ran Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 155–175. Springer, Berlin, Heidelberg, March 2008.

HSW23. Laura Hetz, Thomas Schneider, and Christian Weinert. Scaling mobile private contact discovery to billions of users. In Gene Tsudik, Mauro Conti, Kaitai Liang, and Georgios Smaragdakis, editors, *ESORICS 2023, Part I*, volume 14344 of *LNCS*, pages 455–476. Springer, Cham, September 2023.

IKNP03. Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Berlin, Heidelberg, August 2003.

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

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

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

KKRT16. Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious PRF with applications to private set intersection. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 818–829. ACM Press, October 2016.

KLS+17. Ágnes Kiss, Jian Liu, Thomas Schneider, N. Asokan, and Benny Pinkas. Private set intersection for unequal set sizes with mobile applications. *PoPETs*, 2017(4):177–197, October 2017.

KRS+19. Daniel Kales, Christian Rechberger, Thomas Schneider, Matthias Senker, and Christian Weinert. Mobile private contact discovery at scale. In Nadia Heninger and Patrick Traynor, editors, *USENIX Security 2019*, pages 1447–1464. USENIX Association, August 2019.

MAL23. Daniel Morales, Isaac Agudo, and Javier Lopez. Private set intersection: A systematic literature review. *Computer Science Review*, 49:100567, 2023.

Moo22. Dustin Moody. Parameter selection for the selected algorithms. NIST PQ forum, nov 2022. https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/4MBurXr58Rs.

MR19. Daniel Masny and Peter Rindal. Endemic oblivious transfer. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 309–326. ACM Press, November 2019.

NP99. Moni Naor and Benny Pinkas. Oblivious transfer and polynomial evaluation. In *31st ACM STOC*, pages 245–254. ACM Press, May 1999.

NR04. Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. *Journal of the ACM*, 51(2):231–262, March 2004.

PSSW09. Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, Berlin, Heidelberg, December 2009.

Reg05. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th ACM STOC*, pages 84–93. ACM Press, May 2005.

RL23. Gina M Raimondo and Laurie E Locascio. Module-lattice-based key-encapsulation mechanism standard. *National Institute of Standards and Technology, Gaithersburg*, 2023.

Roy22. Lawrence Roy. SoftSpokenOT: Quieter OT extension from small-field silent VOLE in the minicrypt model. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 657–687. Springer, Cham, August 2022.

RS21. Peter Rindal and Phillipp Schoppmann. VOLE-PSI: Fast OPRF and circuit-PSI from vector-OLE. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, pages 901–930. Springer, Cham, October 2021.

SAB+20. Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

SHB23. István András Seres, Máté Horváth, and Péter Burcsi. The legendre pseudorandom function as a multivariate quadratic cryptosystem: security and applications. *Applicable Algebra in Engineering, Communication and Computing*, pages 1–31, 2023.

Sho94. Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*, pages 124–134. IEEE Computer Society, 1994.

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, Cham, May / June 2022.