qFALL – Rapid Prototyping of Lattice-based Cryptography

Marvin Beckmann[1], Phil Milewski[2], Laurens Porzenheim[3], Marcel Luca Schmidt[3], and Jan Niklas Siemer[4]

[1] Technical University Denmark
mabeck@dtu.dk
[2] Technical University Delft
p.milewski@tudelft.nl
[3] Paderborn University
laurens.porzenheim@upb.de, marcells@campus.upb.de
[4] King's College London
jan.siemer@kcl.ac.uk

Abstract. We introduce qFALL, an open-source library for rapid prototyping of lattice-based cryptography written in Rust. qFALL is designed to bridge the gap between theory and practice by offering a modular architecture that provides a theory-affine, flexible, high-level interface for mathematics and common algorithms in lattice-based constructions with representative runtime performance. This enables researchers to rapidly assemble minimal working prototypes that are easily auditable, modifiable, and allow users to assess algorithmic trade-offs as well as the viability of their constructions early in the development cycle. Furthermore, the library supports an incremental optimization workflow, allowing users to replace bottlenecks with optimized modules to evolve the codebase toward a fully optimized implementation. We demonstrate that qFALL allows for efficient assembly of auditable cryptographic constructions that approximate the performance of optimized implementations and serve as a reusable resource to the scientific community.

Keywords: Prototype · Lattice-based cryptography

1 Introduction

The advancement of quantum computing poses a risk to classical public-key cryptography (PKC) and thus, necessitates a fundamental shift in PKC, driving the development and standardization of post-quantum cryptography (PQC). Among the candidates to replace classical primitives, lattice-based cryptography has emerged as one of the most promising fields, with three out of five selected schemes relying on lattice assumptions in the National Institute of Standards and Technology (NIST) standardization process.[5] This is not only due to strong

[5] https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization

security guarantees derived from reductions to worst-case lattice problems – such as the Shortest Vector Problem (SVP) – but also from the versatility of lattice-based constructions, which extend beyond basic primitives to enable advanced functionalities like Fully Homomorphic Encryption (FHE) [Gen09], Attribute-Based Encryption [GVW13], or anonymous credentials [BCK+14].

As lattice-based cryptography originally imposed high communication overhead and significant computational effort, the focus of the community has shifted to improve practicality of lattice-based algorithms in the past decade. Modern lattice-based constructions rely increasingly on novel cryptographic assumptions or assumptions based on algebraically-structured lattices to improve practicality. With the same goal, increasingly optimized algorithms for arithmetic operations, (preimage) sampling or compression are introduced.

Theory-to-Practice Gap Consequently, the gap between a theoretical algorithm description and a practical, efficient implementation has widened. Validating these new constructions requires more than formal verification; it requires prototyping to assess runtime performance and the viability of sub-algorithms (such as preimage sampling) before committing to a highly optimized implementation.

Turning theoretical constructions or pseudocode into working prototypes is non-trivial and often poses fiddly engineering challenges. Theoretical specifications frequently abstract away the specificities of sub-algorithms, relying on asymptotic bounds that may not translate well to concrete parameters. For instance, the choice between discrete Gaussian samplers (e.g., rejection sampling [GPV08] vs. Knuth-Yao [DG14]), or the selection of specific polynomial multiplication techniques (e.g., the Number-Theoretic Transform (NTT) [CT65] vs. Karatsuba [KO62]), can have profound impacts on the runtime, memory footprint, and side-channel resilience of a prototype.

Furthermore, formal specifications or proofs may change during the design phase, requiring an implementation methodology that is flexible enough to adapt to theoretical changes. In this environment, the ability to rapidly assess the practicality of a design is crucial. Researchers need to answer fundamental questions early in the development cycle. For example, is the preimage sampling algorithm too slow in practice? These questions require a minimal working prototype (MWP) – an implementation that is correct and testable, yet flexible enough to accomodate significant changes quickly.

Ideally, such prototyping would be facilitated by a modular, high-level library that provides standard lattice techniques out-of-the-box while enabling easy optimization and interchangeability of components, allowing researchers to determine optimal trade-offs experimentally. This kind of tool would allow researchers to assemble schemes using high-level abstractions, conduct standardized benchmarks, and iteratively optimize critical sub-modules.

Existing Libraries Currently, the tooling landscape for lattice-based cryptography presents researchers with a stark dilemma between flexibility and performance. On one end of the spectrum are flexible, general-purpose libraries for algebra like *sagemath* [The25] or *MATLAB* [Inc25]. While these tools offer a

vast array of mathematical functions, they are often too slow to handle cryptographic parameters with interesting sizes, rendering them unsuitable for performance benchmarking. On the other end are highly optimized libraries such as *OpenFHE* [BAB+22], *SEAL* [SEA23], *LattiGo* [Lat24], or *lazer* [LSS24]. These libraries are primarily designed for FHE or zero-knowledge arguments and prioritize execution speed over API flexibility. Their rigid structures and complex optimizations make them difficult to adapt for novel, general-purpose experiments with lattice-cryptography, which increases the duration of development.

A 'middle ground' specifically targeting lattice research has historically been sparse. Libraries such as $\Lambda \circ \lambda$ [CP16] and *NFLlib* [AMBG+16] attempted to fill this niche but suffer from significant drawbacks up to the extent that both projects appear to have been discontinued since 2021 and 2019 respectively. We discuss existing libraries extensively in Section 1.2.

Consequences of Missing Prototyping Libraries The absence of an established, maintained, and accessible prototyping library often leads to researchers bypassing the MWP stage entirely, i.e., they build high-performance implementations from scratch or on top of a rigid highly optimized library. This approach yields two primary issues.

- *Premature Optimization.* As the implementation of different variants of modules or sub-procedures is labor-intensive, researchers are discouraged from exploring the space of potential optimizations. Sub-optimal subroutines are often 'baked in' because the cost refactoring the optimized code to test an alternative is too high.
- *Opaqueness and Lack of Reusability.* Optimized prototypes written under the pressure of publication deadlines often lack readability, documentation, and modularity. They become monolithic 'write-once' artifacts, useful for generating numbers in a specific paper, but difficult for other researchers to read, audit, modify, or extend [KSSW22]. This hampers the scientific process as prototypes serve as proof-of-concept rather than a reusable resource for the community that follow-up work can build upon.

1.1 Our Contribution

We introduce *qFALL*[6], an open-source prototyping library designed to bridge the gap between theoretical specification and highly optimized implementation. To address the dilemma between flexibility and performance, qFALL provides efficient, generalized implementations of mathematical structures and algorithms commonly used in lattice-based cryptography, exposing them through a descriptive and flexible API. This enables the rapid assembly of sound MWPs, allowing researchers to test prototypes extensively, swap sub-modules efficiently, and identify optimization targets early in the development cycle. Our contributions center on three design pillars:

[6] `https://qfall.github.io/`

Usability and Modularity To combat the opaqueness of 'write-once' research code and enable rapid prototyping, qFALL prioritizes readability and intuitive discovery through descriptive but concise naming of features. We split qFALL into three distinct libraries, which naturally define the location of any feature:

qFALL-math for fundamental mathematical structures, such as $\mathbb{Z}, \mathbb{Z}_q, \mathbb{Q}$, polynomial rings, and matrices over them;

qFALL-tools for commonly used procedures in lattice-based constructions, such as preimage-sampleable functions [GPV08], encodings of polynomial rings, and G-trapdoors [MP12];

qFALL-schemes for prototypes of cryptographic constructions, such as GPV signatures [GPV08], Regev and Dual Regev encryption [Reg05,GPV08].

Each library is structured in a modular way, function names and parameters are chosen to closely follow conventions found in cryptographic papers, and everything is documented extensively in a standardized way. This design ensures that source code remains comprehensible, easy to debug, and close to notation in pseudocode. An example can be seen in Figure 1. Consequently, it renders

```
fn key_gen() -> (MatZq, MatZq) {
    // A ←$ Z_q^{nxm}
    let mat_a = MatZq::sample_uniform(n, m, q);
    // x ←$ Z_2^m
    let vec_x = MatZ::sample_uniform(m, 1, 0, 2).unwrap();

    // u = A · x
    let vec_u = &mat_a * &vec_x;

    // A = [A  u]
    let mat_a = mat_a.concat_horizontal(&vec_u).unwrap();

    // pk = A, sk = x
    (mat_a, vec_x)
}
```

Figure 1. Implementation of the key generation algorithm of Dual Regev Encryption [GPV08] without `self`-calls.

prototypes more easily auditable, modifiable, and extendable, making them a reusable resource to the scientific community.

Recognizing that extensive libraries can present a steep learning curve, we further lower the barrier to entry by providing a comprehensive tutorial that guides users through library structures, essential features, and benchmarking capabilities.

Rust for Correctness and Ease of Experimentation While Rust is typically chosen for strict memory safety and C-level efficiency, we leverage it to create a 'safe playground' for cryptographic experimentation. qFALL utilizes Rust's ecosystem with its unified package manager, *cargo*, and its standard benchmarking libraries, *criterion*[7] and *flamegraph*[8], to streamline the build, test, and benchmarking process. To enhance theory-affine coding capabilities, we relax certain rigid conventions of Rust; for example, by implementing seamless type interoperability to allow operations between integers `Z` and rationals `Q` without explicit typecasting.

Crucially, this architecture supports *incremental optimization*: researchers can start with high-level, generic reference implementations and progressively replace bottlenecks with highly optimized and even hardware-specific Rust-components. This enables a smooth transition from mathematical reference implementations to high-performance prototypes within a single framework.

Representative Performance As execution times and communication overhead are crucial characteristics of cryptographic algorithms, qFALL is developed with performance in mind. Especially since lattice-based cryptography has struggled with its reputation for being slow and bulky in the past, fast algorithm runtimes are highly valued by the community. Therefore, qFALL implements and uses state-of-the-art algorithms so that prototype execution times approximate the performance of optimized implementations. By comparing prototypes using qFALL, researchers can assess the practicality and efficiency of new lattice-based constructions early in the development cycle, while significantly lowering the required effort to implement a minimal working prototype (MWP).

1.2 Related Work

There are several libraries, which can be utilized to implement lattice-based cryptography. These can be classified in four categories: General purpose math-libraries, general purpose libraries specialized for lattice-based cryptography, libraries focusing on and optimizing for a specific field of lattice-based cryptography, and highly optimized single-purpose libraries.

The open-source math-library *sagemath* [The25] is often used in lattice-based cryptography to derive parameters for schemes or run small scale experiments. However, its fragmented backend architecture requires constant conversion between different data layouts of optimized backends (e.g., *NTL*, *FLINT*). In combination with the interference of Python's garbage collection, this renders it unreliable for accurate performance analysis, limiting its utility to small-scale verification of lattice-based cryptography. Similarly, commercial competitors like *MATLAB* [Inc25] or *Mathematica* [Inc] offer wide arrays of mathematical functions, yet their high density of features makes them difficult to navigate and hinders accessibility. Besides research-oriented math-libraries, there are *Number Theory*

[7] `https://crates.io/crates/criterion`
[8] `https://crates.io/crates/flamegraph`

Library (NTL) [S+25] and *Fast Library for Number Theory (FLINT)* [tea25] as highly optimized libraries for number theory with smaller scopes.

The class of general-purpose libraries for lattice-based cryptography contains two libraries other than ours. *$\Lambda \circ \lambda$: Functional Lattice Cryptography* [CP16] is written in the functional programming language Haskell and provides a high-level interface to implement FHE and Learning with Errors (LWE)-based primitives. It is targeted at researchers and focuses on efficiency. It does not offer a tutorial on how to use the library, but a brief documentation and some examples provide some intuition for each function. $\Lambda \circ \lambda$ offers an abstraction from the underlying math-layer, which itself is not documented extensively. The last update to this library was published in March 2021. The other library in this class is called *NTT-based Fast Lattice Library (NFLlib)* [AMBG+16]. It is a C++ library based around core-optimizations using the Chinese Remainder Theorem (CRT) and NTT. The last update to this library was published in November 2019 and in constrast to $\Lambda \circ \lambda$ and our library, it only offers polynomial rings as an underlying data-structure.

Highly optimized libraries focusing on a specific field of lattice-cryptography are more common. Specifically, there are several competitors to implement FHE, e.g., *OpenFHE* [BAB+22], *LattiGo* [Lat24], *HElib* [HS20], and *SEAL* [SEA23]. *OpenFHE* enables using some of their features in Rust while other competitors are directly written in Rust, such as *tfhe-rs* [Zam22] or *fheanor* [OPP25]. Except for *fheanor*, this class of libraries is not targeted at researchers and experiments. These libraries are used to build highly efficient implementations of FHE schemes and provide very little functionalities aside from this purpose. Two mentionable outliers focusing on another sub-field of lattice-cryptography are *lazer* [LSS24] and *Lattirust* [Kna25], which provide fundamental features for implementations of (succinct) zero-knowledge proofs.

Finally, there are a few highly optimized libraries such as *HEXL* [BKS+21] or *concrete-ntt* [Zam24] to perform polynomial multiplication using AVX2 or AVX512 hardware-acceleration. These libraries are often baked into the design of the aforementioned class of libraries focused on a specific sub-area of lattice-based cryptography. They are great to test the efficiency boundaries of any prototype, but it can also lead to overestimations of performance on hardware such as embedded devices.

2 Background

By convention, vectors are in column form and denoted by bold lower-case letters, e.g., $\mathbf{x}$. The i-th component of $\mathbf{x}$ is represented by x_i. Matrices are written as bold capital letters, e.g., $\mathbf{X}$, and their i-th column is denoted by $\mathbf{x}_i$. Transposed matrices or vectors are described by $\mathbf{X}^t$. Special square matrices are the zero matrix $\mathbf{0}_n$ and the identity matrix $\mathbf{I}_n$ of size n with column vectors $\mathbf{0}$ and unit vectors $\mathbf{e}_i$ respectively. If the dimensions are clear from context, we may omit the size n. The norm of an integer vector $\|\cdot\|$ denotes the Euclidean norm $\|\cdot\|_2$ if

not specified otherwise. The length of an integer matrix is defined by the largest norm of its column vectors.

The natural security parameter throughout this paper is λ, and all other quantities are implicit functions of λ. We use standard Landau notation to classify the growth of functions. Denote by $\mathsf{negl}(\lambda)$ an unspecified function that is negligible in λ.

If a value x is sampled from some distribution χ, this is denoted by $x \leftarrow\!\!\$\ \chi$. We overload notation by denoting $x \leftarrow\!\!\$\ X$ as $x \leftarrow\!\!\$\ U(X)$, i.e., sampling uniformly from a finite set X.

The *statistical distance* between two distributions X and Y over some finite set Ω is defined by $\Delta(X;Y) \coloneqq \frac{1}{2}\sum_{s\in\Omega}|X(s) - Y(s)|$. Two ensembles of distributions $\{X_\lambda\}$ and $\{Y_\lambda\}$ are *statistically close* if their statistical distance $\Delta(X_\lambda;Y_\lambda)$ is negligible in λ.

For any square real matrix $\mathbf{X}$, denote $\mathbf{X}^+$ as the Moore-Penrose pseudoinverse, which is the unqiue matrix that satisfies $(\mathbf{X}\mathbf{X}^+)\mathbf{X} = \mathbf{X}$ and $\mathbf{X}^+(\mathbf{X}\mathbf{X}^+) = \mathbf{X}^+$. A symmetric matrix $\mathbf{\Sigma}$ is positive semi-definite, written as $\mathbf{\Sigma} \geq 0$, if $\mathbf{x}^t\mathbf{\Sigma}\mathbf{x} \geq 0$ for all non-zero real $\mathbf{x}$.

2.1 Lattices

Lattices provide the mathematical structure on which the hardness of all presented problems as well as the security of all presented constructions rests.

Definition 1 (Lattice). *An n-dimensional lattice Λ is an additive subgroup of $\mathbb{R}^n$ and discrete, i.e., for every $\mathbf{x} \in \Lambda$ there exists an n-dimensional ball around $\mathbf{x}$ that contains no other lattice point.*

Let $\mathbf{b}_1, \ldots, \mathbf{b}_m \in \mathbb{R}^n$ be linearly independent vectors and the matrix $\mathbf{B} \coloneqq \begin{bmatrix}\mathbf{b}_1 & \ldots & \mathbf{b}_m\end{bmatrix} \in \mathbb{R}^{n\times m}$. Then, $\mathbf{B}$ forms a basis for the set

$$\Lambda = \mathcal{L}(\mathbf{B}) \coloneqq \left\{\sum_{i=1}^{m} a_i \cdot \mathbf{b}_i | a_i \in \mathbb{Z}\right\}$$

of all integer combinations, which forms a lattice of dimension n and rank m. The quality of a basis is defined by $\|\mathbf{B}\|$.

Almost any lattice-based construction is based on q-ary lattices as described in Definition 2.

Definition 2 (q-ary Lattice). *Let $n, m, q \in \mathbb{N}$ with $m \geq n$. Define $\Lambda_q^\perp(\mathbf{A})$, a q-ary lattice for $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$, and a lattice coset $\Lambda_q^{\mathbf{u}}(\mathbf{A})$ for $\mathbf{u} \in \mathbb{Z}_q^n$:*

$$\Lambda_q^\perp(\mathbf{A}) \coloneqq \{\mathbf{e} \in \mathbb{Z}^m : \mathbf{A}\cdot\mathbf{e} = \mathbf{0} \mod q\}$$
$$\Lambda_q^{\mathbf{u}}(\mathbf{A}) \coloneqq \{\mathbf{e} \in \mathbb{Z}^m : \mathbf{A}\cdot\mathbf{e} = \mathbf{u} \mod q\}.$$

Observe that if $\mathbf{t} \in \Lambda_q^{\mathbf{u}}(\mathbf{A})$, then $\Lambda_q^{\mathbf{u}}(\mathbf{A}) = \Lambda_q^\perp(\mathbf{A}) + \mathbf{t}$. Hence, $\Lambda_q^{\mathbf{u}}(\mathbf{A})$ denotes a shift of $\Lambda_q^\perp(\mathbf{A})$.

Most lattice-based constructions rely on the hardness of two computational problems (or variants thereof). These are the Short Integer Solution (SIS) problem and the LWE problem.

Definition 3 ($\mathsf{SIS}_{n,m,q,\beta}$)**.** *A probabilistic polynomial-time (ppt) adversary $\mathcal{A}$ is given a uniformly random matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$ and asked to find a short, non-zero vector $\mathbf{x} \in \mathbb{Z}_q^m$ with $0 < \|\mathbf{x}\| < \beta$ such that $\mathbf{A}\cdot\mathbf{x} = \mathbf{0} \mod q$.*

Definition 4 ($\mathsf{LWE}_{n,m,q,\chi}$)**.** *The decisional LWE problem asks an adversary $\mathcal{A}$ to distinguish the distributions $U\left(\mathbb{Z}_q^{n\times m} \times \mathbb{Z}_q^{1\times m}\right)$ from $(\mathbf{A}, \mathbf{s}^t\cdot\mathbf{A} + \mathbf{e}^t)$ with $\mathbf{A} \hookleftarrow \mathbb{Z}_q^{n\times m}, \mathbf{s} \hookleftarrow \mathbb{Z}_q^n$, and $\mathbf{e} \hookleftarrow \chi^m$.*

The hardness of these problems is related to the hardness of problems defined over lattices, e.g., SVP [Ajt96,Reg05].

2.2 Discrete Gaussians

The previously presented problems highlight that the shortness of vectors is essential for lattice-based problems. Such short vectors are usually drawn from specific discrete Gaussian distributions.

For a matrix $\mathbf{\Sigma} = \mathbf{B}\mathbf{B}^t \geq 0$, $m \geq 1$ and $\mathbf{x} \in \mathbb{R}^m$, define the Gaussian function on $\mathbb{R}^m$

$$\rho_{\sqrt{\mathbf{\Sigma}}}(\mathbf{x}) := \exp(-\pi\cdot\mathbf{x}^t\mathbf{\Sigma}^+\mathbf{x}).$$

The matrix $\mathbf{\Sigma}$ is taken to be $\mathbf{I}$ when omitted, so $\rho_{\sqrt{\mathbf{\Sigma}}}(\mathbf{x}) = \rho(\mathbf{B}^+\mathbf{x})$. When $\mathbf{\Sigma} = s^2\mathbf{I}_m$, we simplify the notation to ρ_s and refer to s as the Gaussian parameter.

For any $\mathbf{c} \in \mathbb{R}^m$, positive semidefinite $\mathbf{\Sigma} \geq 0$ with $(\Lambda - \mathbf{c}) \cap \mathrm{span}(\mathbf{\Sigma}) \neq \emptyset$, and m-dimensional lattice Λ, the *discrete Gaussian distribution* over Λ is defined as

$$\forall \mathbf{x} \in \Lambda : D_{\Lambda,\sqrt{\mathbf{\Sigma}},\mathbf{c}}(\mathbf{x}) = \frac{\rho_{\sqrt{\mathbf{\Sigma}}}(\mathbf{x}-\mathbf{c})}{\rho_{\sqrt{\mathbf{\Sigma}}}(\Lambda-\mathbf{c})},$$

where $\rho_{\sqrt{\mathbf{\Sigma}}}(\Lambda - \mathbf{c}) = \sum_{\mathbf{x}\in\Lambda}\rho_{\sqrt{\mathbf{\Sigma}}}(\mathbf{x}-\mathbf{c})$ denotes the normalization factor.

2.3 Lattice Trapdoors

Many lattice-based constructions depend on the ability to sample short vectors $\mathbf{x}$ given target vector $\mathbf{u}$ and matrix $\mathbf{A}$ such that $\mathbf{A}\mathbf{x} = \mathbf{u}$. This is typically achieved using a trapdoor hidden in $\mathbf{A}$, of which there are two types.

For the first type, one keeps a *short basis* [GGH97] of a lattice, keeps it secret, and hands out a basis with large norm or another representation to use as the matrix $\mathbf{A}$. This works for any lattice of suitable dimension.

The other type is only applicable to q-ary lattices. It is called *gadget* or **G**-trapdoor [MP12], at least as powerful and computationally more efficient than short basis trapdoors. This technique uses the fact that it is easy to find short, non-zero vectors $\mathbf{z} \in \mathbb{Z}^m$ for any vector $\mathbf{u} \in \mathbb{Z}_q^n$ in the lattice coset $\Lambda_q^{\mathbf{u}}(\mathbf{G})$ defined by the gadget matrix $\mathbf{G} = \mathbf{I}_n \otimes \mathbf{g}^t$, where $k = \lceil \log q \rceil, m = nk$, and

$$\mathbf{g}^t = \begin{bmatrix} 1 & 2 & 4 & \dots & 2^{k-1} \end{bmatrix}.$$

The matrix $\mathbf{G}$ is then hidden inside a public matrix distributed statistically close to uniform by sampling two matrices $\bar{\mathbf{A}} \hookleftarrow \mathbb{Z}_q^{n\times\bar{m}}$ and $\mathbf{R} \hookleftarrow \{0,1\}^{\bar{m}\times nk}$.

The public matrix is defined as $\mathbf{A} = \begin{bmatrix}\bar{\mathbf{A}} & \mathbf{G} - \bar{\mathbf{A}}\mathbf{R}\end{bmatrix}$ s.t. the matrix $\mathbf{G}$ can be uncovered by multiplying $\mathbf{A}$ with $\begin{bmatrix}\mathbf{R} & \mathbf{I}_n\end{bmatrix}^t$. Thus, knowledge of the matrix $\mathbf{R}$ allows to solve SIS easily and also enables mapping short vectors from $\Lambda_q^{\mathbf{u}}(\mathbf{G})$ to $\Lambda_q^{\mathbf{u}}(\mathbf{A})$. Lemma 1 formalizes the existence of an algorithm that samples short preimages using $\mathbf{G}$-trapdoors.

Lemma 1 (Theorem 5.1 in [MP12]). *Let $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ be a matrix with $\mathbf{G}$-trapdoor $\mathbf{R}$. For any $\mathbf{u} \in \mathbb{Z}_q^n$ and large enough $m, s \in \mathcal{O}(\sqrt{n \log q})$, there is a ppt algorithm $\mathsf{SampleD}(\mathbf{A}, \mathbf{R}, \mathbf{u}, s)$ that generates a sample from $D_{\Lambda_q^{\mathbf{u}}(\mathbf{A}),s}$ up to negligible statistical distance.*

2.4 Structured Lattices

To reduce execution times and communication overhead of cryptographic algorithms, lattice-based constructions often rely on structured lattices. A frequently used structure is called an anti-cyclic lattice as Lyubashevsky et al. [LPR10] showed that it is particularly easy to work with. Elements in an anti-cyclic lattice correspond to elements in a ring a over polynomials $\mathcal{R} \coloneqq \mathbb{Z}[X]/f(X)$ for $f(X) = X^N + 1$ with N being a power of two. In general, any monic polynomial f, where the leading coefficient is 1, can be used to define a ring $\mathcal{R}$. For any element $a \in \mathcal{R}$ with $a = \sum_{i=0}^{N-1} a_{i+1} X^i$, we define the coefficient embedding as $\theta : \mathcal{R} \to \mathbb{Z}^N$ mapping $a \in \mathcal{R}$ to a vector in $\mathbb{Z}^N$ with $\theta(a) = \begin{bmatrix}a_1 & \dots & a_N\end{bmatrix}^t$. Thus, the norm of a ring element $\|a\|$ can be defined as the norm of its coefficient embedding $\|\theta(a)\|$.

Corresponding to the idea of q-ary lattices, we define the ring $\mathcal{R}_q \coloneqq \mathcal{R}/q\mathcal{R} = \mathbb{Z}_q[X]/f(X)$ with $q \geq 2$. Similar to the definition of SIS and LWE over q-ary lattices, there exist corresponding versions of SIS and LWE over structured lattices [Mic02,LM06,LS15], known as ring and module versions of these problems. Module-SIS (M-SIS) introduces the ring $\mathcal{R}$ as an additional public parameter and exchanges the integers $\mathbb{Z}$ by the ring $\mathcal{R}$ when compared to SIS as in Definition 3. The same changes apply to Module-LWE (M-LWE) when compared to LWE. Ring-SIS (R-SIS) and Ring-LWE (R-LWE) can be described as special cases of M-SIS and M-LWE with $n = 1$.

Number Theoretic Transform For polynomials of degree at most N, naïve polynomial multiplication has complexity $\mathcal{O}(N^2)$. The NTT, a special case of the Discrete Fourier Transform (DFT) over polynomial rings, optimizes this asymptotically to $\mathcal{O}(N \log N)$ using the CRT [PSD96].

The NTT over polynomial rings $\mathbb{Z}_q[X]/f(X)$ relies on choosing the degree N of $f(X)$ and q such that $f(X)$ can be factored as $f(X) = \prod_{i=1}^{k} f_i(X) \mod q$, where each factor is a small degree polynomial. Multiplying two polynomials a, b over $\mathbb{Z}_q[X]/f(X)$ can be accomplished by first computing $a_i = a \mod f_i(X)$ and $b_i = b \mod f_i(X)$ for $i \in [k]$, then computing the component-wise product and then inverting the transformation, i.e., finding a polynomial c such that $c \mod f_i(X) = a_i b_i \mod f_i(X)$. If each factor $f_i(X)$ is of degree 1, we call

it a full split. If two polynomials are then represented as their N coefficients, the point-wise multiplication only has complexity $\mathcal{O}(N)$. The dominating cost is then described by the transformation itself.

Consider the polynomial ring defined by $f(X) = X^N + 1$ (and $f(X) = X^N - 1$), where N is a power of two. For appropriate choices of q, $f(X)$ splits into N factors of degree 1. The transformation can then be expressed using the $2N$-th (and N-th) root of unity. Using the transformations by Cooley-Tukey [CT65] and Gentleman-Sande [GS66], the transformation complexity can be reduced to $\mathcal{O}(N \log N)$, which is asymptotically faster than the naïve transformation that takes time $\mathcal{O}(N^2)$.

2.5 Cryptographic Primitives and Security Notions

In this section, we briefly summarize primitives and security notions used in this paper. A notion often used in lattice-based cryptography are preimage sampleable functions (PSFs) [GPV08] as defined in Definition 5. They describe a trapdoored function $f_\kappa : D \to R$ with additional properties on f_κ and its preimage sampling algorithm $\mathsf{SamplePre}$. The hardness of preimage sampleable functions (PSFs) relies on the one-wayness of f_κ without access to the trapdoor.

Definition 5 (Preimage Sampleable Function [GPV08]). *A PSF is a triple of ppt algorithms* $\mathcal{F} = (\mathsf{TrapGen}, \mathsf{SampleDom}, \mathsf{SamplePre})$*, where*

- $\mathsf{TrapGen}(1^\lambda)$ *outputs a pair* (κ, td)*, where* κ *describes an efficiently computable function* $f_\kappa : D \to R$ *for* D, R *only depending on* λ*,*
- $\mathsf{SampleDom}(1^\lambda)$ *outputs an* $x \leftarrow\!\!\$\ \chi$*, where* χ *denotes a distribution over* D*,*
- $\mathsf{SamplePre}(td, y)$ *outputs an element* $x \in D$ *so that* $f_\kappa(x) = y$ *for any* $y \in R$*.*

We require for all $\lambda \in \mathbb{N}, (\kappa, td) \leftarrow \mathsf{TrapGen}(1^\lambda), y \in R$*:*

$$\Pr\big[f_\kappa(x) = y \;\big|\; x \leftarrow \mathsf{SampleDom}(1^\lambda)\big] = 1/|R| \tag{1}$$

$$\forall x \in D : \Pr[\mathsf{SamplePre}(td, y) = x] = \Pr\big[\mathsf{SampleDom}(1^\lambda) = x \;\big|\; f_\kappa(x) = y\big] \tag{2}$$

3 Quality Assurance and Implementation Standards

Software and product quality is internationally accepted to be assessed based on ISO/IEC 25010:2023. It categorizes nine top-level qualities: functional suitability, performance efficiency, compatibility, interaction capability, reliability, security, maintainability, flexibility and safety. Quality assurance was a high priority throughout the design process of qFALL, prompting us to regularly reevaluate our architectural and implementation decisions until the current robust concept was reached.

3.1 Library Structure

We identify three core components essential for implementing cryptographic schemes and define separate libraries: (1) qFALL-math: Fundamental mathematical types, e.g., large integers, and operations over them; (2) qFALL-tools: Cryptographic building blocks (e.g., PSFs) that operates on aforementioned data types and are used in explicit constructions; (3) qFALL-schemes: Complete, higher-level constructions (e.g., signature schemes).

The logical separation enables users to easily find desired features and enforces a principle of minimal dependency. Specifically, internal dependencies follow a strict hierarchical structure. qFALL-schemes depends on qFALL-math and qFALL-tools, qFALL-tools only depends on qFALL-math and qFALL-math only depends on external crates.

Internally, each library is structured hierarchically into modules. For our biggest library qFALL-math, modularization is based on the base-types (as detailed in Section 4.1). Each base type (e.g., integers, rationals) forms the foundation and is extended in underlying modules to more complex type structures, such as matrices and polynomials. The lowest layer is separated by explicit functionality (e.g., arithmetic, matrix manipulation).

We deem this fine-grained structure necessary for proper modularization, maintainability and code discoverability. This structure also has the notable advantage that each individual file remains manageable in length, allowing us to include unit tests alongside the respective functions. This co-location simplifies maintaining proper testing and ensures high code quality.

3.2 Code Quality and Consistency

While often of secondary importance in prototypes [KSSW22], in prototyping libraries, where users interact directly with the codebase and interfaces, *code quality* is paramount. To ensure high code quality we enforce strict guidelines for code, detailed in our `Contributing.md` file, which mandates the design requirements for adding consistent code to the existing codebase. Key to consistency is also the comprehensive use of (standard) Rust traits[9] where applicable, as well as the enforcement of consistent naming conventions throughout different base types.

Both qFALL-math and qFALL-tools are designed to be included in new projects as they are; therefore, it is important to ensure high quality of the public interface. The naming is done descriptively making it easy for users to navigate our library. Each function provided through the public interface is extensively documented. It consists of five mandatory components: (1) a detailed explanation of the behavior; (2) a complete list and explanation of all parameters; (3) a concise summary that explains the intended result (e.g., the return value); (4) at least one contextual usage example; and (5) an section discussing potential errors and failures.

[9] `https://doc.rust-lang.org/book/ch10-02-traits.html`

To ensure implementation quality, a strict code review was enforced. Pull requests utilized a merging template containing checklists for the developers. Among others, this checklist requires rigorous testing of the intended and potentially unintended behavior. All libraries contain unit tests, and qFALL-tools and qFALL-schemes have additional integration tests. In these crates, we focus on testing functional correctness, rather than capturing absurd edge cases such as $n = -1$. and we put less emphasis on capturing all edge cases. In contrast, several unit tests in qFALL-math try to capture the behavior of functions as well as previously mentioned edge cases s.t. accidental changes are spotted with high certainty. This leads to test coverage of $93\%, 95\%$, and 89% across our math, tools, and schemes crate, respectively. All tests are run as part of the continuous integration pipeline to prevent unintended side effects. Furthermore, the pipeline also checks Rust's own formatter and linter *clippy*[10], which ensures that the code is concise and that basic style guidelines are followed.

With the quality standards set for our libraries, we are following the call from [KSSW22] to improve software quality in cryptographic prototypes.

3.3 Preventing Unintended Use

Tracing and debugging errors is highly frustrating, especially in low-level implementations written in languages like C or C++, where errors can be of various natures. These can be memory leaks, but also general errors that are not properly propagated through different layers. Particularly for prototypes developed solely to produce benchmarks for a paper, error handling and documentation are frequently deprioritized or entirely omitted.

Rust's compiler ensures memory safety utilizing Rust's ownership model (unless developers explicitly disable it using the keyword `unsafe`), preventing memory leaks when using the interfaces of our libraries. To ensure robust behavior, proactive input validation is needed. This input validation is of two categories: (1) *logic errors* (irrecoverable): Fundamental errors causing impossible states. We expect these to be rare (e.g., instantiating a matrix with a negative number of rows) and have decided to automatically halt execution. (2) *API errors* (recoverable): Errors that are expected during normal programming (e.g., concatenating two matrices with different dimensions). These return a custom error type that allows the developer to evaluate failure and implement recovery logic. The custom error type is consistent and descriptively named, structured as an `enum`. It is divided into distinct subtypes allowing developers to use pattern matching for precise error handling. Additionally, each error instance provides explicit information about the content or context in which the failure occurred, simplifying the debugging process.

[10] `https://crates.io/crates/clippy`

4 Math Crate

In Rust, libraries are commonly called *crates*. Our qFALL-math crate serves as the fundamental binding layer for all arithmetic operations within the system. It provides a memory-safe, user-friendly and idiomatic Rust interface to selected, highly-optimized features of the C-library *FLINT* [tea25]. The design is guided by two primary goals: safe, usable APIs and high performance (where possible), directly supporting functional suitability and reliability (memory safety). The crate specifically includes mathematical types essential for lattice-based cryptography with features promoting usability, as well as useful algorithms for lattice cryptography missing from *FLINT*.

4.1 Supported Types

qFALL-math provides the basic building blocks required for lattice-based cryptography, encompassing integers $\mathbb{Z}$, quotient rings over integers $\mathbb{Z}_q$, rationals $\mathbb{Q}$, and the quotient of a polynomial ring over integers $\mathbb{Z}_q[X]/f(X)$ for any monic polynomial f. All implemented types support arbitrary sizes, i.e., arbitrarily big integers $\mathbb{Z}$ or arbitrary-precision rationals $\mathbb{Q}$. *FLINT* provides optimized support for these base types, which are extended to polynomials and matrices. This crate wraps these types in a memory-safe way to make them available in Rust.

Table 1 provides an overview of the supported features and the underlying *FLINT* types. Notably, $\mathbb{Z}_q[X]/f(X)$ is the composition of `fmpz_poly` and a context object. The operations and additional algorithms use the base operations on the polynomials and apply *FLINT*'s context reduction.

Basetypes	**qFALL Types**	**FLINT Types**
$\mathbb{Z}$	`Z, MatZ, PolyOverZ, MatPolyOverZ`	`fmpz, fmpz_mat, fmpz_poly, fmpz_poly_mat`
$\mathbb{Z}_q$	`Zq, MatZq, PolyOverZq`	`fmpz_mod, fmpz_mod_mat, fmpz_mod_poly`
$\mathbb{Q}$	`Q, MatQ, PolyOverQ`	`fmpq, fmpq_mat, fmpq_poly`
$\mathbb{Z}_q[X]/f(X)$	`PolynomialRingZq, MatPolynomialRingZq`	

Table 1. An overview of the types that the math crate supports. Types defined over $\mathbb{Z}_q$ and $\mathbb{Z}_q[X]/f(X)$ have additional context objects from *FLINT* included and are considered separately.

Context Objects In *FLINT*, context objects to define quotient rings or moduli must be managed separately from the value itself. For an element in $\mathbb{Z}_q$, the user needs to store the integer representative (`fmpz`) and the large context

object (`fmpz_mod_ctx`). Manual management is tedious and risks silent, mathematically unsound errors if an incorrect context (or modulus) is accidentally used in computation.

To overcome these challenges, qFALL-math utilizes smart pointers (specifically, reference counting) to safely manage and share expensive context objects. Each `Zq` internally stores a `Z` element and a `Modulus` context object. The context-object hidden inside the modulus is only stored once and then reference-counted. This design has two benefits:

- Memory Efficiency: The large `Modulus` object is stored once and shared across all elements of the same ring. The memory is automatically released only when the last reference is dropped.
- Reliability: Since the modulus is fixed and tied to the element, no accidental switching between moduli can occur. This ensures high correctness of the program.

The `ModulusPolynomialRingZq` context object follows the same principles, but also encapsulates precomputed values required for specialized algorithms like the NTT (as detailed in Section 4.2).

Support for *FLINT*'s algorithms & *FLINT*-backend *FLINT* offers many features beyond those incorporated in qFALL-math. This raises a conflict between full access and idiomatic Rust safety via a public interface. The qFALL-math crate prioritizes safety and usability, yet we provide a workaround to access *FLINT*'s underlying structures in an `unsafe` way, allowing users to call functions from *FLINT* that may not be wrapped safely in qFALL-math. This way, the user can leverage the foreign function interface (FFI) and wrap these functions themselves for their specific needs.

We rely on the community-maintained *flint-sys* crate[11] as a FFI, and it currently supports bindings to (most of) *FLINT* 2.8.5. The FFI needs to be updated to support more recent versions of *FLINT*.

Crucially, excluding the explicit access structures (marked also as `unsafe` in the code), users can develop without direct usage of the FFI. This guarantees that changes to the FFI can be captured within qFALL-math guaranteeing that the public interface of qFALL-math remains stable.

We also considered and evaluated using Rug[12] instead of wrapping *flint-sys* ourselves. However, it lacks support for quotient rings, polynomial arithmetic and matrix support. Therefore, it is not suited for prototyping of lattice-based cryptography.

Memory Management of *FLINT* in Rust In Rust's ownership model, each value has a single owner, and the owner is responsible for the memory

[11] `https://crates.io/crates/flint-sys`
[12] `https://crates.io/crates/rug`

management when leaving the scope. In C, the `fmpz` needs to be manually initiated (`fmpz_init()`) and freed (`fmpz_clear()`) by the user. In qFALL-math this behavior is abstracted and Rust's `Drop` trait automatically performs *FLINT*'s cleanup functions. This, among other measures, eliminates memory leaks as a major source of implementation errors. Notably, the `fmpz` values may be more than 63 bits, i.e., an `fmpz` can also be a pointer, which when dropped, needs to free the associated memory referenced by the pointer. Another source of memory leaks can occur by addressing memory cells that are not formally bound to an object. In *FLINT*, this can for example occur by instantiating a matrix, but addressing indices outside the dimensions of the matrix. We prevent these types of errors via input validation, while also providing explicitly named methods without input validation for high performance prototypes.

Type Interoperability Designed for seamless prototyping, qFALL-math is designed to minimize friction when combining its arbitrary-precision types with native Rust primitives (e.g., `i64`). We facilitate this through:

1. Implementation of the standard `From` trait for implicit conversions of Rust's native types to ours.
2. Explicit additional implementations if it improves performance and/or there are dedicated *FLINT* functions.
3. Convenient macros for common operations not capturable by implicit type conversions.

While this approach is user-friendly, it departs from strict native Rust type-system behavior, where combining different number types (e.g., floats and integers) is typically disallowed without explicit type casting. qFALL-math mimics the mathematical intuition of number fields, e.g., combining an integer `Z` and a rational `Q` returns a rational number `Q`. This removes the need for manual type casting in most cases. As our types enable arbitrarily big integers and arbitrary-precision rationals, this crate does not stumble into overflow problems, which are a crucial reason to require explicit type casting in lower level languages otherwise.

4.2 Additional Algorithms and Features

FLINT is not constructed explicitly for lattice-based cryptography and thus, lacks some important features for prototyping. We implemented some of these features such as finding solutions for linear equations, sampling over base types for certain distributions, and the NTT transform for $\mathbb{Z}_q[X]/f(X)$.

Solving Linear Equations Systems of linear equations $\mathbf{Ax} = \mathbf{y}$ are easily solved over $\mathbb{Q}$ or prime fields $\mathbb{Z}_q$, using Gaussian elimination [Hig11]. But for lattice-cryptography, we also need to look at systems in $\mathbb{Z}_q$ where q may be composite (e.g., a power of 2). These rings contain zero divisors, which we cannot

use as pivots for Gaussian elimination, as they have no inverse. Instead, we split the modulus into co-prime factors, compute solutions to $\mathbf{Ax} = \mathbf{y}$ under these co-prime moduli and combine them using the Chinese Remainder Theorem [PSD96]. The implementation uses factor refinement from *FLINT* and an algorithm for Hensel Lifting from [Dix82]. In some cases, the Hensel Lifting and thus, solving the system of linear equations may fail (if $\mathbf{A}$ does not contain an invertible submatrix), but since we typically use matrices of dimensions $n \times m$ with $m \geq 2n \lg q$ in lattice cryptography, this is very unlikely to happen [GPV08].

We then use the algorithm for solving linear equations to provide an algorithm that can invert matrices over rings $\mathbb{Z}_q$ for q non-prime.

Sampling Cryptographically secure random sampling is a fundamental requirement for cryptographic schemes. Sampling according to the discrete Gaussian or binomial distribution is usually required to instantiate lattice-based problems like LWE. However, sampling from distributions is not a cryptographic feature. Thus, we provide sampling algorithms in qFALL-math and implemented binomial, uniform, and spherical as well as non-spherical discrete Gaussian samplers.

Uniform. One of Rust's most used crates is called *rand*[13]. It offers two pseudo-random number generators (PRNGs), which are considered to be cryptographically secure. We use the *rand*-crate to generate random bytes and cut off any bits after $\lceil \log(interval_size) \rceil$ bits. Then, a `Z` is created from the uniformly random sampled bytes and checked whether the resulting value is bigger than the specified interval size. Hence, we use rejection sampling with probability larger than $\frac{1}{2}$ for the instantiation of `Z`s and `Zq`s.

Upper and lower bounds are used to compute the interval size and shift the samples into the specified interval. If a matrix or a polynomial is sampled uniformly at random, its entries or coefficients are sampled uniformly at random according to the specified parameters.

Discrete Gaussians. For the base types, we implement `SampleZ` from [GPV08] utilizing rejection sampling. We support both on-the-fly and precomputed lookup tables for the underlying Gaussian function. In terms of efficiency, other solutions like [MW17] can be more desirable. However, due to the versatility and the lack of precomputation requirements (useful for single instance sampling for a parameter set), we consider rejection sampling a reasonable choice. To sample lattice vectors (represented using matrices of dimension $n \times 1$), we implement `SampleD` from [GPV08] for spherical discrete Gaussians. For non-spherical discrete Gaussian distributions, we use a version of `SampleD` from [Pei10] using continuous Gaussian sampling over vectors of $\mathbb{Q}$ and randomized rounding as described in [Pei10]. We discuss the performance of our implementation in Section 7.

[13] `https://crates.io/crates/rand`

Binomial. The binomial distribution's shape can be close to a discrete Gaussian distribution for an appropriate choice of parameters. It is often chosen as a replacement for discrete Gaussian sampling in explicit constructions like ML-KEM [Nat24], as it is easier to sample according to a binomial distribution in constant time.

We implemented binomial sampling for all supported data types in qFALL-math by utilizing a cryptographically secure PRNG from the *rand*-crate combined with a module for sampling according to the binomial distribution from the *rand_ distr*-crate[14].

NTT The NTT transform, formally introduced in Section 2.4, is defined for $\mathbb{Z}_q[X]/f(X)$ and we have implemented two versions in this library. We *only* implemented a full split NTT for $f(X) = X^N + 1$ and $f(X) = X^N - 1$. We implement the transformation iteratively with precomputation of the powers of the root of unity, and then use the Cooley-Tukey butterfly [CT65]. Similarly, for the inverse transformation, we implement an iterative version using the Gentleman-Sande butterfly [GS66] with precomputation of the powers of the inverse of the root of unity. The precomputed powers are stored inside the context object, which itself can be shared over different instances of objects over $\mathbb{Z}_q[X]/f(X)$.

To simplify usage of the NTT, we added two explicit data types called `NTTPolynomialRingZq` and `MatNTTPolynomialRingZq`, that support basic operations such as addition, subtraction and multiplication. The NTT and its inverse are used, e.g., to convert a `PolynomialRingZq` into a `NTTPolynomialRingZq`. We discuss the performance of our implementation in Section 7.

Miscellaneous All types are serializable, making it possible to store and restore our data types from memory. This can be useful for Known Answer Tests (KATs) and benchmarking as well as any other deterministically repeatable task.

5 Tools-Crate

The qFALL-tools library provides primitives, interfaces and helper functions that are frequently used in lattice cryptography, but are not schemes themselves. These building blocks are essential for implementing various kinds of schemes.

5.1 Primitive Implementation: Gadget Trapdoors

We implement the gadget trapdoors explained in Section 2.3. We provide two implementation domains.

- Trapdoors over $\mathbb{Z}_q$: The gadget trapdoor construction is implemented using a centered binomial distribution with parameter $n = 2$.

[14] `https://crates.io/crates/rand_distr`

– Ring Trapdoors: For polynomial ring settings, we followed methods detailed in [EB14] and [GPR+17] using a version of `SampleZ` for the trapdoor generation.

In both implementations, the underlying probability distribution is modular and can be dynamically modified. Furthermore, we added an algorithm to compute a short basis for the related q-ary lattice from the gadget trapdoor using [CHKP10, Lemma 3.2]. Generating a new gadget trapdoor with our library suite can be done as intuitive and simple as in Fig. 2.

```
fn gen_gtrapdoor(n: u32, m: u32, k: u32) -> (MatZq, MatZ) {
    let base = Z::from(2);
    let q = base.pow(k).unwrap();

    // Ā ←$ ℤ_q^{n×m}
    let a_bar = MatZq::sample_uniform(n, m, q);
    // gadget matrix G = I_n ⊗ g^t for gadget vector g
    let g = gen_gadget_vec(k, &base);
    let i_n = MatZ::identity(n, n);
    let g_mat = i_n.tensor_product(&g.transpose());
    // R ←$ Bin(n = 2, p = 0.5, c = 0)^{n×m}
    let r = MatZ::sample_binomial_with_offset(
        n, m, -1, 2, 0.5).unwrap();

    // A = [Ā   G − ĀR]
    let a = a_bar.concat_horizontal(&(g - (&a_bar * &r)))
        .unwrap();
    (a, r)
}
```

Figure 2. Implementation of a gadget trapdoor generation with base 2, $q = 2^k$ and the binomial distribution with $p = 0.5$, two trials for the sampling of the trapdoor and centered around 0.

5.2 Abstraction and Implementation of Preimage Sampleable Functions

Since PSFs [GPV08] are a commonly used building block in other cryptographic schemes, we abstract the behavior of PSFs as a trait. The trait is designed to be flexible, and treated as a black-box. This enables implementation of schemes using a mostly generic PSF and without much effort, evaluate the implementation with different underlying PSFs.

We provide two key sets of implementations: a classical and a polynomial ring PSF using gadget trapdoors for preimage sampling. Both utilize the `SampleD` algorithm from qFALL-math for preimage sampling. Additionally, we implemented

a classical PSF using perturbation sampling and non-spherical discrete Gaussian sampling from [Pei10, Algorithm 1] for the classical gadget trapdoor.

5.3 Utility Functions and Structure Helpers

qFALL-tools includes a variety of useful algorithms that might not be immediate tools themselves, but useful utility arguments. These functions are not contained in qFALL-math, as they deal with explicit cryptography-relevant tasks, rather than the underlying math foundation. Currently, these helpers include bit-wise encoding of polynomial ring elements, methods to easily instantiate polynomial rings that are regularly found in practice, and rotation matrices for vectors.

6 Schemes-Crate

In addition to some core and refining concepts of lattice-based cryptography, we provide a few prototypes of well-known constructions. We focused on relevant constructions such as LPR encryption [LP11] or GPV-signatures [GPV08], which are frequently used as building blocks for more involved constructions or protocols such as [CHK04] or [AKSY22]. All prototypes are based on modules from the qFALL-tools and qFALL-math crates, and are designed specifically for good readability, reusability, and easy modification. Examples showcasing the previously named properties and the usability of our crates overall are given in Fig. 3 and Appendix A. Note the similarities between each comment and the following code, showcasing that our API is theory-affine and descriptive. Furthermore, the examples show that our API provides several commonly used tools out-of-the-box, such as sampling, modification and transposing of matrices. The example in Appendix A also displays some of our extensive documentation, containing function descriptions, parameter explanations, and usage examples.

In contrast to schemes directly built from mathematical primitives, we also provide multiple schemes built from other schemes. One such example is the CCA-secure encryption scheme from [CHK04], which requires an Identity-Based Encryption (IBE) scheme and a signature scheme. Our implementation of the CCA-secure encryption scheme uses our traits for IBEs and signature schemes in a black-box way. It is then concretely instantiated with a combination of our implementation of the Dual Regev-based IBE scheme [GPV08] and the probabilistic full-domain hash signature scheme [GPV08]. Thus, we also showcase how to use our implemented schemes and interfaces as building blocks for other schemes. A list of implemented schemes can be found in Table 2.

The modular architecture of qFALL-schemes, along with its trait definitions and existing implementations, simplifies the construction of compatible prototypes. Consequently, integrating these new prototypes into the schemes crate becomes a seamless step.

```
fn enc(pk: &MatZq, msg: &Z) -> MatZq {
        // s ←$ Z_q^n
        let vec_s_t = MatZq::sample_uniform(1, n, q);
        // e^t ←$ D_{α·q}^{1×(m+1)}
        let vec_e_t = MatZq::sample_discrete_gauss(
                1, m + 1, q, 0, alpha * q).unwrap();

        // c^t = s^t · A + e^t + [0^{1×m}   μ · ⌊q/2⌋]
        let mut vec_c = (vec_s_t * pk + vec_e_t).transpose();

        // Instantiate [0^{1×m}   μ · ⌊q/2⌋]^t
        let mut vec_msg = MatZ::new(m + 1, 1);
        vec_msg.set_entry(-1, 0, msg * q / 2).unwrap();

        vec_c += vec_msg;

        vec_c
}
```

Figure 3. Implementation of the encryption algorithm of Dual Regev Encryption [GPV08] without `self`-calls.

Scheme	**Reference**
Public-Key Encryption	
Regev	[Reg05]
Dual Regev	[GPV08]
LPR	[LP11]
Ring-based LPR	[LPR10,LPR13]
CCA-secure from Identity-Based Encryption	[CHK04]
K-PKE (Foundation of ML-KEM)	[BDK+18,Nat24]
Signature	
Full-Domain Hash	[GPV08]
Probabilistic Full-Domain Hash	[GPV08]
Ring-based Full-Domain Hash	[GPV08]
Hash Function	
Ajtai SIS-Hash	[Ajt96]
SHA-256-based Hash	[Nat15]
Identity-Based Encryption	
IBE from Dual Regev	[GPV08]

Table 2. List of implemented prototypes in qFALL-schemes.

7 Benchmarks

In this section, we present benchmarks for three algorithms of specific interest for lattice-based cryptography. To benchmark these algorithms, we use Rust's standard library *criterion*[15]. Furthermore, any qFALL-library provides support for the integration of *flamegraph*[16] to profile execution times. All benchmarks presented in this section were conducted on a personal laptop with an Intel Core Ultra 7 165U 4.9 GHz CPU with 32 GB of RAM running Ubuntu 24.04.3 with kernel 6.14.0-36-generic.

Discrete Gaussian Sampling As described in Section 4, sampling from a discrete Gaussian distribution is provided by implementing rejection sampling with the opportunity to employ a lookup table. In Fig. 4, we compare our implementation using a precomputed lookup table with the implementation given in *dgs* [AW18], an open-source C-library specialized for discrete Gaussian sampling. All algorithms in Fig. 4 use precomputed lookup tables and double precision, i.e., they operate on 64-bit floating point arithmetic to compute lookup tables. A major difference between *dgs* and our library is the source of uniform randomness used by the discrete Gaussian sampler. The library *dgs* relies on randomness generated by the library *gmp*, which uses either the Mersenne Twister or a linear congruential generator as PRNG. Both of these are unsuitable for cryptographic applications according to the documentation of *gmp*[17]. Contrary, any randomness in qFALL-math is generated using the ChaCha block cipher [B+08] with 12 rounds, which is considered a cryptographically secure PRNG.

Fig. 4 shows that our implementation of rejection sampling is more efficient than the implementation in *dgs*. Nevertheless, more efficient algorithms using alias sampling or convolution techniques outperform our implementation of rejection sampling with a lookup table. Despite the existence of (theoretically) more efficient sampling techniques [Pei10,DDLL13,BCG+14,Kar16,MW17], not every technique is as versatile as rejection sampling. Almost all previously listed techniques require comparatively heavy precomputations to perform well. These algorithms are optimized for sampling from a single discrete Gaussian distribution whose Gaussian width and center are predetermined. However, especially lattice trapdoor sampling [GPV08,MP12] requires sampling from discrete Gaussians with several different Gaussian widths and centers, which are only known at runtime. In these cases, filling a lookup table for rejection sampling on-the-fly becomes quite efficient compared to the execution of a more computationally-intensive precomputation phase for each sample.

NTT Multiplication Table 3 compares our implementation of NTT to the one of the highly-specialized library *concrete-ntt* [Zam24]. We provide measurements for two choices of rings. The first ring is used by Dilithium [LDK+22] and defined

[15] `https://crates.io/crates/criterion`
[16] `https://crates.io/crates/flamegraph`
[17] `https://gmplib.org/manual/Random-Number-Algorithms`

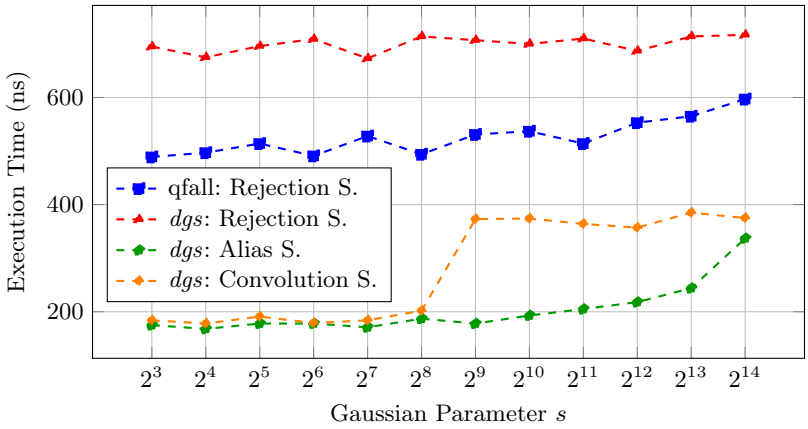

Figure 4. Average execution times to generate a single sample chosen from $D_{\mathbb{Z},s}$.

by $X^{256}+1 \bmod 2^{23}-2^{13}+1$. The second ring is used by HAWK [BBD+24] and defined by $X^{1024}+1 \bmod 12289$. Both rings use NTT-friendly primes and we compare against the 64-bit prime-plan of *concrete-ntt* with hardware acceleration using the AVX2 instruction set.

Table 3 shows that the elementwise multiplication of *concrete-ntt* is about three times faster than ours. Taking into account the transforms, *concrete-ntt* outperforms our implementation by a factor of about twenty. However, multiplying two polynomials using *FLINT*'s version of Karatsuba [KO62] is currently 4-5 times faster than our implementation of NTT. While this feature is not optimized yet, it enables users to access the NTT transform in case algorithms require it, e.g., in case of using the canonical embedding. Even though it seems reasonable to shrink the gap in performance compared to *concrete-ntt*, we will not be able to close it entirely, as we are not planning to introduce hardware-accelerated instructions. Furthermore, notice that our current implementation of the NTT is capable of optimizing execution times of some algorithms by keeping the polynomials in NTT representation across multiple multiplications and thus, reducing the number of computationally-intensive transforms.

Kyber The key exchange mechanism (KEM) Kyber [SAB+22] is based on a public-key encryption (PKE) scheme called K-PKE, which can be used in a more versatile way than the KEM itself. Thus, we provide a naïve prototype of K-PKE in qFALL-schemes. Our implementation currently does not utilize the NTT as the ring defined by Kyber's specification does not split fully, which is not supported by our implementation of the NTT yet. Furthermore, our implementation ignores smaller optimizations such as matrix generation from seeded PRNGs. However, it uses centered binomial distributions as well as the specified lossy (de-)compression technique, i.e., any features required to satisfy security requirements are implemented.

Library	Algorithms	Duration (μs)	
		Dilithium	**HAWK**
qFALL	NTT + Multiplication + InvNTT	60.60	297.82
	Elementwise Multiplication	1.56	6.28
	Karatsuba	15.73	66.00
concrete-ntt [Zam24]	NTT + Multiplication + InvNTT	3.26	14.90
	Elementwise Multiplication	0.51	2.04

Table 3. Average execution times to multiply two uniformly sampled polynomials from rings used in Dilithium [LDK+22] and HAWK [BBD+24], i.e., $\mathcal{R} = \mathbb{Z}[X]/f(X)$ with $f(X) = X^{256} + 1 \bmod 2^{23} - 2^{13} + 1$ and $f(X) = X^{1024} + 1 \bmod 12289$ respectively.

In Table 4, we provide benchmarks for our implementation as well as Kyber's reference implementation. Note that *Enc* and *Dec* denote *Encapsulate* and *Decapsulate* for Kyber's reference implementation while they refer to *Encrypt* and *Decrypt* for ours. The differences between Kyber's encapsulation and K-PKE's encryption are negligible in terms of performance. However, the decapsulation algorithm performs K-PKE's decryption and verifies the decrypted key by re-encrypting with the same randomness. Thus, we need to compare Kyber's decapsulation benchmark to the sum of K-PKE's encryption and decryption benchmarks.

Overall, Table 4 shows that the reference implementation of Kyber outperforms our naïve implementation of K-PKE by a factor of 5-6. Profiling our implementation shows that the algorithms *KeyGen* and *Enc* are dominated by sampling random matrices, which consumes about 65% and 40% of their execution time respectively. Performing lossy compression and decompression takes roughly 25% and 35% of the runtimes of *Enc* and *Dec*. Even though our implementation does not use NTT for multiplication of polynomials, arithmetic operations only consume 30-35% of each algorithm's execution time. Thus, the generated flamegraph shows that our implementation would profit from a more optimized memory layout, binomial sampler, and polynomial multiplication in decreasing order of potential for improvement. In this way, users of qFALL can identify the greatest possible potential for optimization in their prototypes and replace submodules of qFALL with optimized components while retaining a functional prototype.

8 Supporting Resources

As already mentioned, in order to maintain a transparent structure and the modularity of qFALL, we have split the project into three libraries. As each library comes with its own documentation, this keeps each library as well as our extensive documentation of each library manageable and easy to navigate. The

Implementation	Specification	Duration (μs) KeyGen	Enc	Dec
qFALL-schemes K-PKE	512	104.48	134.57	55.43
	768	182.66	207.17	70.34
	1024	305.78	299.40	91.93
Kyber [SAB+22] Reference Impl.	512	21.14	25.94	31.53
	768	35.97	41.83	48.60
	1024	55.01	58.97	67.44

Table 4. Average execution times of algorithms related to Kyber.

documentation itself is created using *rustdoc*[18], which is a Rust tool to automatically create documentation from code. Thus, the documentation is located right besides the documented code and our documentation follows the same modularization as our library. Every module page has a description, followed by a list of modules, structs, traits and enums, whereas the pages of structs, traits and enums contain functionalities specific to the corresponding struct, trait or enum. Additionally, the documentation includes a search function to find features quickly.

Nevertheless, the separation into three libraries also introduces the need for a general page to collect information about all libraries that are part of qFALL. Our website[19] provides a brief overview of the project and acts as a collection of relevant information and links.

A second challenge for projects with a lot of features is their steep learning curve. Learning to work with a library offering hundreds of functions for any data-type can be overwhelming for beginners. Thus, we provide a step-by-step tutorial[20] to understand the most fundamental features of qFALL, understand our datatypes, implemented features, etc. Our tutorial is structured as follows. First, we provide an installation guide specifying the prerequisites to use our library as well as options to embed qFALL in other projects. Then, we provide a small overview of qFALL-math. Among other features, this includes basic instantiations, arithmetics and further operations such as sampling according to common distributions. Next, we show how the features in qFALL-tools can simplify implementations and provide an overview of all implemented schemes in qFALL-schemes. The latter section is completed by an interactive step-by-step tutorial on implementing a private key Ring-LWE encryption scheme with qFALL. Finally, we briefly explain how prototypes can utilize benchmarking as well as serialization of our datatypes. In conclusion, the tutorial provides a less overwhelming starting point to use qFALL and allows users to gain some

[18] `https://doc.rust-lang.org/`
[19] `https://qfall.github.io/`
[20] `https://qfall.github.io/book/`

intuition to use our libraries before they start experimenting with qFALL themselves.

9 Future Directions

While qFALL provides a lot of features that enable rapid prototyping, not all of them are fully optimized. For example, the discrete Gaussian sampler uses rejection sampling, which could be replaced by a more efficient sampler for discrete Gaussian distributions such as Micciancio and Walter's sampler [MW17] utilizing convolution techniques. Furthermore, our implementation of NTT currently lacks the efficiency to outperform Karatsuba's polynomial multiplication algorithm.

Furthermore, there are potential extensions of functionalities or entirely new features that are interesting to qFALL. One is to support floating point arithmetic and real values by moving to *FLINT* 3 to utilize its new structs `arf` and `arb` respectively. Furthermore, we plan to provide an API to use optimized samplers by enabling the use of precomputed tables or seeded samplers for any datatype. Another idea to sustainably extend our implementation of the NTT is to enable support for non-fully splitting rings as well as support for moduli q, where q is the product of NTT-friendly primes but not NTT-friendly itself. Additionally, there are a few features such as preimage sampling or pruned gadget matrices, which we can extend from the ring setting to the module setting. Moreover, we investigated multi-threading and may provide an API to utilize it in the future. Another option to enrich qFALL-tools is to integrate further commonly-used optimizations used in Kyber, Dilithium, and Falcon such as support for approximate equalities. In combination with this, we plan to extend qFALL-schemes with several prototypes of relevant schemes such as Dilithium, Falcon, and HAWK. A feature with a bigger scope is to offer support for generic compilers for commonly used transforms such as the Fiat-Shamir transform [FS87] or the Naor-Yung transform [NY90].

Apart from the aforementioned features, we also see other possibilities for qFALL. As our notation is theory-affine and helps users to abstractly use core concepts of lattice-cryptography, it provides a good environment for students to prototype first lattice-based constructions or attacks. One way to integrate practical implementations into lectures could utilize qFALL by deploying a CTF-platform with lattice-based challenges that can be solved using qFALL. We may provide a platform and challenges as well as a few sample tasks to support lecturers with educational material.

Acknowledgements We thank Sven Moog for his valuable contribution to the initial development of the qFALL library suite and for his conceptual insights that continue to shape our libraries. Marvin Beckmann acknowledges support from the Danish Ministry of Defense Acquisition and Logistics Organization (FMI). The work of Jan Niklas Siemer was partially funded by the EPSRC Research Council, part of the EPSRC/UKRI Horizon Europe Grant Studentship, Grant Ref: EP/Y02432X/1.

References

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

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

AMBG+16. Carlos Aguilar-Melchor, Joris Barrier, Serge Guelton, Adrien Guinet, Marc-Olivier Killijian, and Tancrede Lepoint. Nfllib: Ntt-based fast lattice library. In *Cryptographers' Track at the RSA Conference*, pages 341–356. Springer, 2016. `https://github.com/quarkslab/NFLlib`.

AW18. Martin R. Albrecht and Michael Walter. dgs, Discrete Gaussians over the Integers. `https://bitbucket.org/malb/dgs`, 2018.

B+08. Daniel J Bernstein et al. Chacha, a variant of salsa20. In *Workshop record of SASC*, volume 8, pages 3–5. Lausanne, Switzerland, 2008. `https://cr.yp.to/chacha.html`.

BAB+22. Ahmad Al Badawi, Andreea Alexandru, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Carlo Pascoe, Yuriy Polyakov, Ian Quah, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. OpenFHE: Open-source fully homomorphic encryption library. Cryptology ePrint Archive, Paper 2022/915, 2022. `https://eprint.iacr.org/2022/915`, `https://github.com/openfheorg/openfhe-development`.

BBD+24. Joppe W. Bos, Olivier Bronchain, Léo Ducas, Serge Fehr, Yu-Hsuan Huang, Thomas Pornin, Eamonn W. Postlethwaite, Thomas Prest, Ludo N. Pulles, and Wessel van Woerden. HAWK. Technical report, National Institute of Standards and Technology, 2024. available at `https://csrc.nist.gov/Projects/pqc-dig-sig/round-2-additional-signatures`.

BCG+14. Johannes Buchmann, Daniel Cabarcas, Florian Göpfert, Andreas Hülsing, and Patrick Weiden. Discrete ziggurat: A time-memory trade-off for sampling from a Gaussian distribution over the integers. In Lange et al. [LLL14], pages 402–417.

BCK+14. Fabrice Benhamouda, Jan Camenisch, Stephan Krenn, Vadim Lyubashevsky, and Gregory Neven. Better zero-knowledge proofs for lattice encryption and their application to group signatures. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part I*, volume 8873 of *LNCS*, pages 551–572. Springer, Berlin, Heidelberg, December 2014.

BDK+18. Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS - Kyber: A CCA-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy*, pages 353–367. IEEE Computer Society Press, April 2018.

BKS+21. Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe DM de Souza, Vinodh Gopal, et al. Intel HEXL (release 1.2). `https://github.com/intel/hexl`, 2021.

CHK04. Ran Canetti, Shai Halevi, and Jonathan Katz. Chosen-ciphertext security from identity-based encryption. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 207–222. Springer, Berlin, Heidelberg, May 2004.

CHKP10. David Cash, Dennis Hofheinz, Eike Kiltz, and Chris Peikert. Bonsai trees, or how to delegate a lattice basis. In Gilbert [Gil10], pages 523–552.

CP16. Eric Crockett and Chris Peikert. $\Lambda o\lambda$: Functional Lattice Cryptography. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 993–1005, 2016. `https://github.com/cpeikert/Lol`.

CT65. James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

DDLL13. Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal Gaussians. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 40–56. Springer, Berlin, Heidelberg, August 2013.

DG14. Nagarjun C Dwarakanath and Steven D Galbraith. Sampling from discrete gaussians for lattice-based cryptography on a constrained device. *Applicable Algebra in Engineering, Communication and Computing*, 25(3):159–180, 2014.

Dix82. John D Dixon. Exact solution of linear equations using p-adic expansions. *Numerische Mathematik*, 40(1):137–141, 1982.

EB14. Rachid El Bansarkhani and Johannes Buchmann. Improvement and efficient implementation of a lattice-based signature scheme. In Lange et al. [LLL14], pages 48–67.

FS87. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Berlin, Heidelberg, August 1987.

Gen09. Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st ACM STOC*, pages 169–178. ACM Press, May / June 2009.

GGH97. Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Public-key cryptosystems from lattice reduction problems. In Burton S. Kaliski, Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 112–131. Springer, Berlin, Heidelberg, August 1997.

Gil10. Henri Gilbert, editor. *EUROCRYPT 2010*, volume 6110 of *LNCS*. Springer, Berlin, Heidelberg, May / June 2010.

GPR+17. Kamil Doruk Gür, Yuriy Polyakov, Kurt Rohloff, Gerard W. Ryan, and Erkay Savaş. Implementation and evaluation of improved Gaussian sampling for lattice trapdoors. Cryptology ePrint Archive, Report 2017/285, 2017.

GPV08. Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In Richard E. Ladner and Cynthia Dwork, editors, *40th ACM STOC*, pages 197–206. ACM Press, May 2008.

GS66. W Morven Gentleman and Gordon Sande. Fast fourier transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, fall joint computer conference*, pages 563–578, 1966.

GVW13. Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Attribute-based encryption for circuits. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th ACM STOC*, pages 545–554. ACM Press, June 2013.

Hig11. Nicholas J Higham. Gaussian elimination. *Wiley Interdisciplinary Reviews: Computational Statistics*, 3(3):230–238, 2011.

HS20. Shai Halevi and Victor Shoup. Design and implementation of HElib: a homomorphic encryption library. *Cryptology ePrint Archive*, 2020. `https://github.com/homenc/HElib`.

Inc. Wolfram Research, Inc. Mathematica. Version 14.3, `https://www.wolfram.com/mathematica`.

Inc25. The MathWorks Inc. Matlab version: 25.1, 2025. `https://www.mathworks.com`.

Kar16. Charles FF Karney. Sampling exactly from the normal distribution. *ACM Transactions on Mathematical Software (TOMS)*, 42(1):1–14, 2016.

Kna25. Christian Knabenhans. Lattirust: Lattice zero-knowledge/succinct arguments, and more, 2025. Version 0.0.1-alpha, `https://github.com/lattirust`.

KO62. Anatolii Alekseevich Karatsuba and Yu P Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145, pages 293–294. Russian Academy of Sciences, 1962.

KSSW22. Matthias J Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. Improving software quality in cryptography standardization projects. In *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 19–30. IEEE, 2022.

Lat24. Lattigo v6, 2024. EPFL-LDS, Tune Insight SA, `https://github.com/tuneinsight/lattigo`.

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

LLL14. Tanja Lange, Kristin Lauter, and Petr Lisonek, editors. *SAC 2013*, volume 8282 of *LNCS*. Springer, Berlin, Heidelberg, August 2014.

LM06. Vadim Lyubashevsky and Daniele Micciancio. Generalized compact Knapsacks are collision resistant. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *ICALP 2006, Part II*, volume 4052 of *LNCS*, pages 144–155. Springer, Berlin, Heidelberg, July 2006.

LP11. Richard Lindner and Chris Peikert. Better key sizes (and attacks) for LWE-based encryption. In Aggelos Kiayias, editor, *CT-RSA 2011*, volume 6558 of *LNCS*, pages 319–339. Springer, Berlin, Heidelberg, February 2011.

LPR10. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Gilbert [Gil10], pages 1–23.

LPR13. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. A toolkit for ring-LWE cryptography. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 35–54. Springer, Berlin, Heidelberg, May 2013.

LS15. Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *DCC*, 75(3):565–599, 2015.

LSS24. Vadim Lyubashevsky, Gregor Seiler, and Patrick Steuer. The LaZer library: Lattice-based zero knowledge and succinct proofs for quantum-safe privacy. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 3125–3137, 2024. https://github.com/lazer-crypto/lazer.

Mic02. Daniele Micciancio. Generalized compact knapsacks, cyclic lattices, and efficient one-way functions from worst-case complexity assumptions. In *43rd FOCS*, pages 356–365. IEEE Computer Society Press, November 2002.

MP12. Daniele Micciancio and Chris Peikert. Trapdoors for lattices: Simpler, tighter, faster, smaller. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 700–718. Springer, Berlin, Heidelberg, April 2012.

MW17. Daniele Micciancio and Michael Walter. Gaussian sampling over the integers: Efficient, generic, constant-time. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 455–485. Springer, Cham, August 2017.

Nat15. National Institute of Standards and Technology (NIST). FIPS 180-4: Secure Hash Standard (SHS), 2015. https://doi.org/10.6028/NIST.FIPS.180-4.

Nat24. National Institute of Standards and Technology (NIST). FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM), 2024. https://doi.org/10.6028/NIST.FIPS.203.

NY90. Moni Naor and Moti Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *22nd ACM STOC*, pages 427–437. ACM Press, May 1990.

OPP25. Hiroki Okada, Rachel Player, and Simon Pohmann. Fheanor: a new, modular FHE library for designing and optimising schemes. Cryptology ePrint Archive, Paper 2025/864, 2025. https://github.com/FeanorTheElf/fheanor.

Pei10. Chris Peikert. An efficient and parallel Gaussian sampler for lattices. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 80–97. Springer, Berlin, Heidelberg, August 2010.

PSD96. Dingyi Pei, Arto Salomaa, and Cunsheng Ding. *Chinese remainder theorem: applications in computing, coding, cryptography*. World Scientific, 1996.

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.

S$^+$25. Victor Shoup et al. NTL: A library for doing number theory, 2025. Version 11.6, https://libntl.org/.

SAB$^+$22. Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

SEA23. Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL, 2023. Microsoft Research, Redmond, WA.

tea25. The FLINT team. FLINT: Fast Library for Number Theory, 2025. Version 3.3.1, https://flintlib.org.

The25. The Sage Developers. Sagemath, the Sage Mathematics Software System (Version 10.5), 2025. https://www.sagemath.org.
Zam22. Zama. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data, 2022. https://github.com/zama-ai/tfhe-rs.
Zam24. Zama AI, Inc. concrete-ntt, 2024. Version 0.2.0, https://github.com/zama-ai/concrete-ntt.

A Code Example of naïve Implementation of K-PKE

The following example code shows our naïve implementation of the K-PKE algorithms, including inline and documentation comments. Apart from different line breaks and removed `self`-calls, this piece of code can be found verbatim in qFALL-schemes. The numbers in the comments correspond to the line numbers of the respective pseudocode from [Nat24].

```
/// Generates a `(pk, sk)` pair by following these
/// steps:
/// - A <- R_q^{k x k}
/// - s <- Bin(eta_1, 0.5)^k centered around 0
/// - e <- Bin(eta_1, 0.5)^k centered around 0
/// - t = A * s + e
///
/// Then, `pk = (A^T, t)` and `sk = s` are returned.
///
/// # Examples
/// ```
/// use qfall_schemes::pk_encryption::
///     {PKEncryptionScheme, KPKE};
/// let k_pke = KPKE::ml_kem_512();
///
/// let (pk, sk) = k_pke.key_gen();
/// ```
fn key_gen() -> (Self::PublicKey, Self::SecretKey) {
    // 5 A[i, j] <-$ SampleNTT(φ||j||i)
    // Reminder: NTT is not part of this prototype
    let mat_a =
      MatPolynomialRingZq::sample_uniform(k, k, &q);
    // 9 s[i] <-$ SamplePolyCBD_η1(PRF_η1(σ, N))
    let vec_s =
        MatPolynomialRingZq::sample_binomial_with_offset(
          k,
          1,
          &q,
          -eta_1,
          2 * eta_1,
          0.5
        ).unwrap();
    // 13 e[i] <-$ SamplePolyCBD_η1(PRF_η1(σ, N))
```

```
    let vec_e =
        MatPolynomialRingZq::sample_binomial_with_offset(
          k,
          1,
          &q,
          -eta_1,
          2 * eta_1,
          0.5,
        ).unwrap();

    // 18 t = A·s + e
    let vec_t = &mat_a * &vec_s + vec_e;

    let pk = (mat_a.transpose(), vec_t);
    let sk = vec_s;
    (pk, sk)
}

/// Encrypts a `message` with the provided public key
/// by following these steps:
/// - y <- Bin(eta_1, 0.5)^k centered around 0
/// - e_1 <- Bin(eta_2, 0.5)^k centered around 0
/// - e_2 <- Bin(eta_2, 0.5) centered around 0
/// - u = A^T * y + e_1
/// - v = t^T * y + e_2 + μ, where μ is the {q/2, 0}
///   encoding of the bits of `message`
/// - Compress u and v
///
/// Then, ciphertext `(u, v)` is returned.
///
/// Parameters:
/// - `pk`: specifies the public key `pk = (A, t)`
/// - `message`: specifies the message that should be
///   encrypted, which should not extend 256 bits
///
/// Returns a ciphertext `(u, v)` of type
/// [`MatPolynomialRingZq`] and [`PolynomialRingZq`].
///
/// # Examples
/// ```
/// use qfall_schemes::pk_encryption::
///     {PKEncryptionScheme, KPKE};
/// let k_pke = KPKE::ml_kem_512();
/// let (pk, sk) = k_pke.key_gen();
///
/// let c = k_pke.enc(&pk, 1);
/// ```
fn enc(pk: &Self::PublicKey, message: impl Into<Z>)
-> Self::Cipher {
    // 10 y[i] ←$ SamplePolyCBD_{η1}(PRF_{η1}(r, N))
```

```
        let vec_y =
            MatPolynomialRingZq::sample_binomial_with_offset(
              k,
              1,
              &q,
              -eta_1,
              2 * eta_1,
              0.5,
            ).unwrap();
        // 14 e_1[i] ←$ SamplePolyCBD_{η_2}(PRF_{η_2}(r, N))
        let vec_e_1 =
            MatPolynomialRingZq::sample_binomial_with_offset(
              k,
              1,
              &q,
              -eta_2,
              2 * eta_2,
              0.5,
            ).unwrap();
        // 17 e_2 ←$ SamplePolyCBD_{η_2}(PRF_{η_2}(r, N))
        let e_2 = PolynomialRingZq::sample_binomial_with_offset(
            &q,
            -eta_2,
            2 * eta_2,
            0.5,
        ).unwrap();

        // 19 u = NTT^{-1}(A^t · y) + e_1
        let vec_u = &pk.0 * &vec_y + vec_e_1;

        // 20 μ = Decompress_1(ByteDecode_1(m))
        let mu = encode_value_in_polynomialringzq(
            message,
            2,
            &q
        ).unwrap();

        // 21 v = NTT^{-1}(t^t · y) + e_2 + μ
        let v = pk.1.dot_product(&vec_y).unwrap() + e_2 + mu;

        // 22 c_1 = ByteEncode_{d_u}(Compress_{d_u}(u))
        let vec_u = vec_u.lossy_compress(d_u);
        // 23 c_2 = ByteEncode_{d_v}(Compress_{d_v}(v))
        let v = v.lossy_compress(d_v);

        (vec_u, v)
    }

    /// Decrypts the provided ‘cipher‘ using the secret
```

```
/// key ‘sk‘ by following these steps:
/// - Decompress u and v
/// - w = v - s^T * u
/// - returns the decoding of ‘w‘ with 1 and 0 set in the
///   returned [‘Z‘] instance if the corresponding
///   coefficient was closer to q/2 or 0 respectively
///
/// Parameters:
/// - ‘sk‘: specifies the secret key ‘sk = s‘
/// - ‘cipher‘: specifies the ciphertext
///   containing ‘cipher = (u, v)‘
///
/// Returns the decryption of ‘cipher‘ as a
/// [‘Z‘] instance.
///
/// # Examples
/// ‘‘‘
/// use qfall_schemes::pk_encryption::
///     {PKEncryptionScheme, KPKE};
/// let k_pke = KPKE::ml_kem_512();
/// let (pk, sk) = k_pke.key_gen();
/// let c = k_pke.enc(&pk, 1);
///
/// let m = k_pke.dec(&sk, &c);
///
/// assert_eq!(1, m);
/// ‘‘‘
fn dec(
    sk: &Self::SecretKey,
    (u, v): &Self::Cipher
) -> Z {
    // 3 u' = Decompress_{d_u}(ByteDecode_{d_u}(c_1))
    let u =
        MatPolynomialRingZq::lossy_decompress(u, d_u, &q);
    // 4 v' = Decompress_{d_v}(ByteDecode_{d_v}(c_2))
    let v = PolynomialRingZq::lossy_decompress(v, d_v, &q);

    // 6 w = v' - NTT^{-1}(s^t · NTT(u'))
    let w = v - sk.dot_product(&u).unwrap();

    // 7 m = ByteEncode_1(Compress_1(w))
    decode_value_from_polynomialringzq(&w, 2).unwrap()
}
```