Implementation and Evaluation of Improved Gaussian Sampling for Lattice Trapdoors *

Kamil Doruk Gür[1,2], Yuriy Polyakov[2]**, Kurt Rohloff[2], Gerard W. Ryan[2], and Erkay Savaş[1,2]

[1] Sabancı University
Istanbul, Turkey
{dgur,erkays}@sabanciuniv.edu
[2] NJIT Cybersecurity Research Center
NJIT, Newark NJ, USA
{kg365,polyakov,rohloff,gwryan,savas}@njit.edu

Abstract. We report on our implementation of a new Gaussian sampling algorithm for lattice trapdoors. Lattice trapdoors are used in a wide array of lattice-based cryptographic schemes including digital signatures, attributed-based encryption, program obfuscation and others. Our implementation provides Gaussian sampling for trapdoor lattices with prime moduli, and supports both single- and multi-threaded execution. We experimentally evaluate our implementation through its use in the GPV hash-and-sign digital signature scheme as a benchmark. We compare our design and implementation with prior work reported in the literature. Evaluation shows that our implementation 1) has smaller space requirements and faster runtime, 2) does not require multi-precision floating-point arithmetic, and 3) can be used for a broader range of cryptographic primitives than previous implementations.

Keywords: lattice-based cryptography · trapdoor · Gaussian sampling · ring-LWE · digital signature

* Sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Army Research Laboratory (ARL) under Contract Numbers W911NF-15-C-0226 and W911NF-15-C-0233. The views expressed are those of the authors and do not necessarily reflect the official policy or position of the Department of Defense or the U.S. Government. Project sponsored by the National Security Agency under Grant H98230-15-1-0274. This research is based upon work supported in part by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either express or implied, of ODNI, IARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

** Corresponding Author

1 Introduction

Lattice-based cryptography is an increasingly common and important family of cryptostystems [21, 23, 25]. A motivation for the use of lattice-based cryptography is that lattice-based schemes are generally believed to be "post-quantum", meaning that they are resistant to quantum computing attacks [17, 19, 24]. Besides their security properties, lattice-based cryptographic schemes also have very attractive functional properties, such as the ability to support homomorphic encryption [3, 10, 14], attribute-based encryption [2], cryptographic software obfuscation [4], among many others. As such, lattice-based cryptography has become a subject of much interest for efficient implementation and application [7, 9, 15].

Lattice-based cryptographic protocols have been described as falling into two classes [18].

The first class utilizes direct application of hard mathematical problems, such as Learning With Errors (LWE) or the more efficient ring Learning With Errors (ring-LWE) variation. This first class of protocols involves the sampling of random polynomials and evaluation of linear functions to construct collision-resistant hash functions and public-key encryption schemes [3, 10, 14].

The second class provides a wider array of advanced lattice-based cryptographic schemes, such as "hash-and-sign" digital signatures [13], identity-based encryption [13], attribute-based encryption [2], and conjunction obfuscation [4]. This second class of schemes relies on a concept of a *strong lattice trapdoor*, which requires sampling from an n-dimensional lattice L with a Gaussian-like distribution [13]. This lattice trapdoor sampling operation is space- and compute-intensive and is often the primary computational bottleneck for implementations of this second class of schemes [9, 11, 13, 18].

There have been recent theoretical approaches to design algorithms for efficient Gaussian sampling [11], but there have been few attempts to implement and experimentally evaluate Gaussian sampling methods for lattice trapdoors.

Our Contribution We implement in software a variation of trapdoor sampling based on approaches in [11] for the case of power-of-two cyclotomic rings with a prime modulus. We evaluate the scalability and runtime performance of our implementation using the GPV hash-and-sign digital signature primitive developed in [13].

Our trapdoor sampling implementation has the following advantages over prior efforts:

- The trapdoor generation runtime is three orders of magnitude faster than prior results [9]. Our preimage sampling is faster and has two orders of magnitude smaller storage requirements.
- Our trapdoor implementation is based on cyclotomic rings with a prime modulus (rather than a power-of-two modulus as in prior work.) This implementation can be used for a broader range of cryptographic primitives [2, 4].

– Our trapdoor implementation does not rely on multiprecision floating-point arithmetic and does not depend on any external libraries, such as GMP[3] or MPFR[4]. All floating-point computations are performed using double-precision arithmetic.
– Our implementation supports multi-threaded execution on commodity computing hardware. We experimentally evaluate multi-threaded performance of our implemented trapdoor sampling capabilities on a commercial-off-the-shelf multi-core desktop computer.
– We are are adding our implementation as a trapdoor module to an open-source lattice-based cryptography library, making it available for practical use for applications such as digital signatures, identity-based encryption, and attribute-based encryption systems.

2 Related Work

An early concept of strong lattice trapdoor based on Gaussian-sampling-like approaches is explored in [13]. This seminal work provides a trapdoor construction which is arguably complex and not suitable for practical implementation.

Micciancio and Peikert [18] propose a more efficient trapdoor which uses samples around a target point $\mathbf{t}$ in lattice L is performed via an intermediate "primitive'" lattice G^n. The lattice L is first mapped to G^n, then a Gaussian sample is generated in G^n. The sample is then mapped back to L. The linear function T mapping G^n to L is used as the trapdoor. The main challenge of this approach is that the mapping T produces a lattice point in L with an ellipsoidal Gaussian distribution and covariance dependent on the transformation T. To generate spherical samples, the authors apply a perturbation technique that adds noise with complimentary covariance to the target point $\mathbf{t}$ prior to using it as the center for G^n sampling. From an implementation perspective, this approach decomposes the lattice trapdoor sampling into two phases: 1) a perturbation sampling stage, where target-independent perturbation vectors with a covariance matrix defined by the trapdoor mapping T are generated, and 2) a target-dependent stage where Gaussian samples are generated from lattice G^n. The authors suggest that the first phase, usually referred to as *perturbation generation* [11], can be performed offline as it does not depend on the target point $\mathbf{t}$. The second stage, referred to as *G-sampling* [11], is performed online as it depends on the target point.

Micciancio and Peikert [18] also provide an efficient algorithm for G-sampling for the case when the modulus q is a power of two. This approach runs in $O(\log q)$ for lattices over the cyclotomic ring with dimension n. (The full complexity is $O(n \log q)$ but the n factor can be dropped because all n integers can be sampled independently, i.e., in parallel.) At the same time for this approach, the G-sampling algorithm for an arbitrary modulus, such as for a prime modulus, has the computational complexity of $O\left(\log^3 q\right)$ (or $O\left(\log^2 q\right)$ for the on-line stage

[3] https://gmplib.org/
[4] http://www.mpfr.org/

when using pre-computation and additional large storage.) Their G-sampling algorithm for perturbation generation requires a pre-computation complexity of $O\left(n^3 \log^3 q\right)$ and storage of $O\left(n^2 \log^2 q\right)$ for the Cholesky decomposition matrix composed of multi-precision floating-points numbers. The time complexity of the main perturbation sampling computations in this case is $O\left(n^2 \log^2 q\right)$. The Cholesky decomposition matrix is the key time/space bottleneck of the lattice trapdoor sampling developed in [18]. We utilize a sampling approach which does not compute and store the Cholesky decomposition of the perturbation matrix and is hence much more time and space efficient.

Ducas and Nguyen [8] develop a more efficient perturbation generation algorithm for the power-of-two cyclotomic rings. This prior work uses a combination of lazy floating-point techniques and special square-root numerical algorithms (different from the Cholesky decomposition) that improves the expected running time of computations from quadratic to quasilinear. However, this method requires substantial pre-computation effort and significant storage (up to $O(n^2)$ bits) to store the result of the precomputation. Perturbation generation optimization techniques are presented in Section 6 of [8] at a high level, but the authors do not provide adequate detail to implement this perturbation method in software.

Bansarkhani and Buchmann [9] implement both matrix and ring versions of the trapdoor construction of [18] for the case when the modulus is a power of two. The trapdoor construction was used as part of the hash-and-sign digital signature primitive originally proposed in [13]. The authors also optimized the perturbation generation algorithm to work with a Cholesky decomposition matrix of size $2n \times 2n$ rather than $(k+2)n \times (k+2)n$, where $k = \log_2 q$ (the latter was used in [18]). The ring construction had a better computational and spatial efficiency compared to the matrix version. To the best of our knowledge, the ring implementation presented in [9] is the most efficient lattice trapdoor implementation available in literature and will be used as a benchmark to evaluate our implementation. We discuss designs and implementations of approaches which are more efficient than the results shown in [9], and do not rely on moduli which are a power of two.

Our designs and implementation build from the algorithms presented in [11]. This recent work substantially improves upon prior algorithms for both G-sampling and perturbation sampling. The G-sampling algorithm in [11] supports an abitrary modulus and has the same complexity, i.e., $O\left(\log q\right)$, as the algorithm developed in [18] for the case when q is a power of two. This allows one to apply the trapdoor construction to more advanced cryprographic primitives based on prime moduli, for example, the entropic ring-LWE conjunction obfuscator introduced in [4]. The perturbation generation algorithm from [11] for power-of-two cyclotomic rings (generalizable to arbitrary cyclotomic rings) takes full advantage of the algebraic structure of ring lattices to reduce the computational complexity to quasilinear, and does not require any precomputations or additional storage, in contrast to the methods developed in [8, 9, 18]. Both

algorithms from [11] are modified and implemented in this work and discussed in more detail in Section 3.

We implement in software a variation of trapdoor sampling algorithms developed in [11] for the case of power-of-two cyclotomic rings with a prime modulus. Our implementation of perturbation generation operation does not require any pre-processing and additional storage to store the result of precomputations in contrast to the Cholesky decomposition matrix that grows quadratically with ring dimension n. As a result, the trapdoor generation time is smaller by multiple orders of magnitude and the preimage sampling time has dramatically smaller storage requirements, as compared to the results reported in [9]. Our trapdoor implementation does not rely on multiprecision floating-point arithmetic and does not depend on any external libraries, such as GMP or MPFR. All floating-point computations are performed using double-precision arithmetic. In contrast to [9], our implementation supports multi-threaded execution on commodity computing hardware.

Table 1 compares runtimes achieved using our implementation with those reported in [9]. Both implementations used quad-core CPUs with comparable single-threaded performance (based on standard CPU benchmarks). Our key generation and verification runtimes are at least one order of magnitude faster for the case of single-thread execution. Our signing time is only slightly faster but our implementation supports parallelization. For the multi-threaded execution on a 4-core CPU we were able to achieve additional runtime improvement, which matches the number of cores in the case of batch parallelization. It should also be noted that our perturbation sampling implementation implicitly works the perturbation matrix, thus avoiding a storage of a large matrix of floating-point numbers, which is already 16MB for a practical digital signature setting of $n = 1024$.

Table 1. Comparison of our implementation for single- and multi-threaded runtimes to the results reported in [9]

Implementation	n	k	Runtime [ms]			Pert. matrix [kB]
			Key generation	Signing	Verification	
Single-threaded for $q = 2^k$ from [9]	512	24	4,562	27	3	4,100
Single-threaded for prime q	512	24	9.5	27	0.33	0
Multi-threaded with loop parallel	512	24	6.5	21	0.35	0
Multi-threaded with batch parallel	512	24	6.9	8.9	0.066	0
Single-thread for $q = 2^k$ from [9]	1024	27	28,074	74	10	16,392
Single-threaded for prime q	1024	27	17.2	62.5	0.68	0
Multi-threaded with loop parallel	1024	27	7.8	45.6	0.72	0
Multi-threaded with batch parallel	1024	27	7.8	19.8	0.15	0

3 Lattice Trapdoor Sampling Algorithms

3.1 Preliminaries

Our implementation utilizes cyclotomic polynomial rings $\mathcal{R} = \mathbb{Z}[x]/\langle x^n + 1\rangle$ and $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$, a special class of ideal lattices, where n is a power of 2 and q is prime. The order of cyclotomic polynomial $\Phi_m(x) = x^n + 1$ is $m = 2n$. The elements in these rings can be represented in coefficient or evaluation representation. The coefficient representation of polynomial $a(x) = \sum_{i<n} a_i x^i$ treats the polynomial as a list of all coefficients $\mathbf{a} = \langle a_0, a_1, \ldots, a_{n-1}\rangle \in (\mathbb{Z}/q\mathbb{Z})^n$. The evaluation representation, often also referred to as Chinese Remainder Transform (CRT) representation, computes the values of polynomial $a(x)$ at all primitive m-th roots of unity modulo q, i.e., $b_i = a(\zeta^i) \bmod q$ for $i \in (\mathbb{Z}/m\mathbb{Z})^*$. These cyclotomic rings support fast polynomial multiplication by transforming the polynomials from coefficient representation to the evaluation one in $O(n \log n)$ time using Fermat Theoretic Transform (FTT) and then performing component-wise multiplication.

The perturbation generation algorithm also utilizes cyclotomic fields $\mathcal{K}_{2n} = \mathbb{Q}[x]/\langle x^n + 1\rangle$, which are similar in their properties to the cyclotomic rings except that the coefficients/values of the polynomials in this case are rationals rather than integers. The elements of the cyclotomic fields also have coefficient and evaluation (CRT) representation, and support fast polynomial multiplication using variants of the Fast Fourier Transform (FFT). At the same time, the evaluation representation of such rational polynomials in our implementation works with complex primitive roots of unity rather than the modular ones.

Lattice sampling uses n-th dimensional discrete Gaussian distributions over lattice $\Lambda \subset \mathbb{R}^n$ denoted as $\mathcal{D}_{\Lambda,\mathbf{c},\sigma}$, where $\mathbf{c} \in \mathbb{R}^n$ is the center and σ is the distribution parameter. At the most primitive level, the lattice sampling algorithms work with discrete Gaussian distribution $\mathcal{D}_{\mathbb{Z},c,\sigma}$ over integers with floating-point center c and distribution parameter σ. If the center c is omitted, it is assumed to be set to zero. In the pseudocode we present, we use a subroutine SampleZ(σ, c) which returns a sample statistically close to $\mathcal{D}_{\mathbb{Z},c,\sigma}$. More details on our implementation of SampleZ are provided in Section 4.2.

We use U to denote discrete uniform distribution over $\mathbb{Z}_q$.

In the ring setting, preimage sampling is the procedure to generate a vector $\mathbf{x} \in \mathcal{R}_q^{\bar{m}}$ of sample polynomials such that $\mathbf{Ax} = u$, where $\mathbf{A} \in \mathcal{R}_q^{1\times\bar{m}}$ is the public key, $u \in \mathcal{R}_q$ is the target syndrome, and dimension $\bar{m}$ depends on the specific trapdoor construction.

We use the ring-LWE trapdoor construction proposed in [9] (depicted in Algorithm 1). In the pseudocode, $k = \lfloor \log_2(q) + 1 \rfloor$ is the bitwidth of modulus q, $\hat{\mathbf{r}}$ and $\hat{\mathbf{e}}$ are the row vectors of secret trapdoor polynomials generated using discrete Gaussian distribution, $\mathbf{A}$ is the public key, and $\mathbf{g}^t = \{g_1, g_2, \ldots, g_k\}$ is the primitive row vector corresponding to the primitive lattice G^n. The latter is often denoted as simply G because it is the orthogonal sum of n copies of a low dimensional lattice G. In our implementation, $\mathbf{g}^t = \left\{1, 2, 2^2, \ldots, 2^k\right\}$. For this trapdoor construction, $\bar{m} = 2 + k$.

Algorithm 1 Trapdoor generation using ring-LWE [9]

function TRAPGEN
  $a \leftarrow_U \mathcal{R}_q$
  $\hat{\mathbf{r}} := [\hat{r}_1, \ldots, \hat{r}_k] \leftarrow \text{SAMPLEZ}(\sigma) \in \mathcal{R}_q^{1\times k}$
  $\hat{\mathbf{e}} := [\hat{e}_1, \ldots, \hat{e}_k] \leftarrow \text{SAMPLEZ}(\sigma) \in \mathcal{R}_q^{1\times k}$
  $\mathbf{A} := [1, a, g_1 - (a\hat{r}_1 + \hat{e}_1), \ldots, g_k - (a\hat{r}_k + \hat{e}_k)] \in \mathcal{R}_q^{1\times(2+k)}$
  return $(\mathbf{A}, (\hat{\mathbf{r}}, \hat{\mathbf{e}}))$
end function

Algorithm 2 describes the high-level procedure for Gaussian preimage sampling. It calls perturbation generation function SampleP_Z and SAMPLEG as subroutines. The perturbation vector $\mathbf{p}$ is introduced to transform ellipsoidal Gaussian samples into spherical ones.

Algorithm 2 Gaussian preimage sampling [18]

function GAUSSSAMP$(\mathbf{A}, (\hat{\mathbf{r}}, \hat{\mathbf{e}}), u, \sigma, s)$
  $\mathbf{p} \leftarrow \text{SampleP}_Z(n, q, s, 2\sigma, (\hat{\mathbf{r}}, \hat{\mathbf{e}}))$ ▷ SampleP_Z is defined in Algorithm 4
  $\mathbf{z} \leftarrow \text{SAMPLEG}(\sigma, u - \mathbf{Ap}, q)$ ▷ SAMPLEG is defined in Algorithm 3
  `convert` $\mathbf{z} \in \mathbb{Z}^{k\times n}$ `to` $\hat{\mathbf{z}} \in \mathcal{R}_q^k$ ▷ CRT operations can be executed in parallel
  $\mathbf{x} := [p_1 + \hat{\mathbf{e}}\hat{\mathbf{z}}, p_2 + \hat{\mathbf{r}}\hat{\mathbf{z}}, p_3 + \hat{z}_1, \ldots, p_{k+2} + \hat{z}_k]$
  return x
end function

3.2 Sampling G-lattices

The G-lattice sampling problem, i.e., the problem of sampling the discrete Gaussian distribution on a lattice coset, is formulated as

$$\Lambda_v^{\perp}\left(\mathbf{g}^t\right) = \left\{\mathbf{z} \in \mathbb{Z}^k : \mathbf{g}^t\mathbf{z} = v \bmod q\right\},$$

where $q \leq b^k, v \in \mathbb{Z}$ and $\mathbf{g} = \left(1, b, b^2, \ldots, b^{k-1}\right)$. In our implementation, we use $b = 2$. The G-sampling problem is formulated for a single integer v rather than n-dimensional lattice because each of the n integers can be sampled in parallel.

We modify and implement a variation of the G-sampling algorithm developed in [11]. Algorithm 3 (seen in the appendix) shows our pseudocode variation of the G-sampling algorithm from [11] which we modify for more efficient implementation and easier translation into software as compared to the original design. Our variation from [11] reduces the number of calls to CRT operations and increases opportunities for parallel execution. Although our modification does not necessarily improve the computational complexity of the algorithm, it improves runtime in single- and multi-threaded modes of execution.

Algorithm 3 has complexity $O(\log q)$ for an arbitrary modulus. The main idea of the algorithm is not to sample $\Lambda_v^{\perp}(\mathbf{g}^t)$ directly, but to express the lattice

basis $\mathbf{B}_q = \mathbf{TD}$ as the image (using a transformation $\mathbf{T}$) of a matrix $\mathbf{D}$ with a sparse, triangular structure. This technique requires adding a perturbation with a complementary covariance to obtain a spherical Gaussian distribution, as in the case of the GaussSamp procedure described in Algorithm 2. In this prior work the authors select an appropriate instantiation of $\mathbf{D}$ that is sparse and triangular, and has a complementary covariance matrix with simple Cholesky decomposition $\Sigma_2 = \mathbf{L} \cdot \mathbf{L}^t$, where $\mathbf{L}$ is an upper triangular matrix, and find the entries of the $\mathbf{L}$ matrix in closed form.

We show the case of $b = 2$ in the G-sampling procedure in Algorithm 3. Further details and derivation of the original algorithm are in [11].

3.3 Perturbation sampling

The lattice preimage sampling algorithm developed in [18] requires the generation of $n(2+k)$-dimensional Gaussian perturbation vectors $\mathbf{p}$ with covariance

$$\Sigma_p := s^2 \cdot \mathbf{I} - \alpha^2 \begin{bmatrix} \mathbf{T} \\ \mathbf{I} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{T}^t \, \mathbf{I} \end{bmatrix},$$

where $\mathbf{T} \in \mathbb{Z}^{2n \times nk}$ is a matrix with small entries serving as a lattice trapdoor, s is the upper bound on the spectral normal of $\alpha[\mathbf{T}^t, \mathbf{I}]^t$ and α is a small factor discussed in the parameter selection section below.

When working with algebraic lattices, the trapdoor $\mathbf{T}$ can be compactly represented by a matrix $\widetilde{\mathbf{T}} \in R_n^{2\times k}$, where n denotes the rank (dimension) of the ring R_n. In our case, this corresponds to the cyclotomic ring of order $m = 2n$. For the ring-LWE trapdoor construction used in our implementation (Algorithm 1), the trapdoor $\widetilde{\mathbf{T}}$ is computed as $(\hat{\mathbf{r}}, \hat{\mathbf{e}})$. The main challenge with the perturbation sampling techniques developed in [9, 18] is the direct computation of a Cholesky decomposition of Σ_p that destroys the ring structure of the compact trapdoor and operates on matrices over $\mathbb{R}$.

Genise and Micciancio [11] provide an algorithm that leverages the ring structure of R_n and performs all computations either in cyclotomic rings or fields over $\Phi_{2n}(x) = x^n + 1$. The algorithm does not require any preprocessing/storage and runs with time and space complexity quasi-linear in n. The perturbation sampling algorithm can be summarized in a modular way as a combination of three steps [11]:

1. The problem of sampling a $n(2+k)$-dimensional Gaussian perturbation vector with covariance Σ_p is reduced to the problem of sampling a $2n$-dimensional integer vector with covariance expressed by a 2×2 matrix over R_n.
2. The problem of sampling with covariance in $R_n^{2\times 2}$ is reduced to sampling two n-dimensional vectors with covariance in R_n.
3. The sampling problem with covariance in R_n is reduced to sampling n-dimensional perturbation with covariance expressed by a 2×2 matrix over the smaller ring $R_{n/2}$ using an FFT-like approach.

The pseudocode for the perturbation generation algorithm with implementation notes are provided in Algorithm 4. As with Algorithm 3, we modify and implement a variation of the perturbation generation algorithm developed in [11]. Algorithm 4 (seen in the appendix) shows our pseudocode variation of the perturbation generation algorithm from [11] which we modify for more efficient implementation and easier translation into software as compared to the original design. Our variation from [11] reduces the number of calls to CRT operations and increases opportunities for parallel execution. Although our modification does not necessarily improve the computational complexity of the algorithm, it improves runtime in single- and multi-threaded modes of execution. Further details and the complete derivation of the algorithm can be found in [11].

4 Implementation

4.1 Cyclotomic rings and fields

The multiplication of elements in cyclotomic rings $\mathcal{R}_q$ and fields $\mathcal{K}_{2n}$ is performed using the Chinese Remainder Transform (CRT) [16].

For the case of $\mathcal{R}_q$ we use an implementation of Fermat Theoretic Transform (FTT) described in [1]. We implemented FTT with Number Theretic Transform (NTT) as a subroutine. For NTT, the iterative Cooley-Tukey algorithm with optimized butterfly operations was applied. We use native data types in our implementation whenever possible. When large ring moduli q are needed that exceed 32-bit representations, we use a generalized Barrett modulo reduction algorithm [6] for modulo reduction operations. This approach requires one pre-computation per NTT run and converts modulo reduction to roughly two multiplications.

For multiplications in $\mathcal{K}_{2n}$ we use the iterative Cooley-Tukey FTT algorithm over complex primitive roots of unity.

To convert elements of rings to fields, we switch the polynomials from the evaluation representation to the coefficient one as an intermediate step because the CRTs for rings operate with modular primitive roots of unity and CRTs for fields deal with complex primitive roots of unity.

Element transposition for a polynomial $f(x) = f_0 + f_1 x + \cdots + f_{n-1}x^{n-1}$ over cyclotomic polynomial $x^n + 1$ is expressed as $f^t(x) = f_0 - f_{n-1}x - \cdots - f_1 x^{n-1}$. This transposition technique was used for both rings and fields. In our implementation the transposition operation is performed directly in evaluation representation by applying an automorphism from $f(\zeta_{2n})$ to $f(\zeta_{2n}^{2n-1})$.

4.2 Integer sampling

Both G-sampling and perturbation algorithms call the integer sampling subroutine SampleZ(σ, c) that returns a sample statistically close to $\mathcal{D}_{\mathbb{Z},c,\sigma}$. When the center c does not change, our SampleZ implementation uses the inversion sampling method developed in [22]. When the center c varies, the rejection sampling method proposed in section 4.1 of [13] is applied. The inversion method is

significantly faster as it is based on a table lookup while the rejection method requires a computation of Gaussian Probability Distribution Function (PDF) for each integer sampling call, often multiple times.

A major bottleneck of integer sampling operations in lattice-based cryptography is associated with the use of multiprecision floating-point numbers, where the number of bits in the mantissa should roughly match the number of security bits supported by the cryptographic protocol. A recent theoretical result in [20] suggests that both the G-sampling and perturbation generation algorithms that are used in our implementation can support at least 100 bits of security using double-precision floating point arithmetic. More specifically, Lemma 3.2 in [20] states that $\kappa/2$ significant bits in a floating-point number is sufficient to maintain κ bits of security. As our goal is to support 100 bits of security, the significand precision of 53 bits provided by double-precision floating numbers is sufficient to achieve our security target.

We also perform the comparison of our implementation with the one described in [9] for the case when the G-sampling procedure of [11] and the perturbation generation procedure of [9] are used for sampling the lattices with a prime modulus. In this case we use quad-precision floating numbers in the computations related to the Cholesky decomposition matrix, which are exposed as __FLOAT128 in GCC, and lazy floating-point techniques from [8], which reduce most of the computations to double-precision floating-point arithmetic.

4.3 Parameter selection

To meet the ring-LWE security requirements for the trapdoor construction, we select the values of n and q using the inequality derived in [12], namely,

$$n \geq \frac{\log_2(q/\sigma)}{4\log_2(\delta)}. \tag{1}$$

Here, σ refers to the distribution parameter used in sampling the trapdoor $(\hat{\mathbf{r}}, \hat{\mathbf{e}})$ and δ is the root Hermite factor, a measure of lattice security that can be mapped to the number of bits of security. The value of $\delta < 1.006$ corresponds to at least 100 bits of security [5].

The smoothing (distribution) parameter σ can be estimated as

$$\sigma \approx \sqrt{\ln(2n_m/\epsilon)/\pi},$$

where n_m is the maximum ring dimension and ϵ is the bound on the statistical error introduced by each randomized-rounding operation [18]. For $n_m \leq 2^{14}$ and $\epsilon \geq 2^{-80}$, the value of $\sigma \approx 4.578$.

The value of α is taken as 2σ [18].

For the spectral norm parameter s we use [9, 18]:

$$s > s_1(\mathbf{X})\,\alpha,$$

where $\mathbf{X}$ is a subgaussian random matrix with parameter s.

Lemma 2.9 of [18] states that

$$s_1(\mathbf{X}) \leq C_0 \cdot \sigma \cdot \left(\sqrt{nk} + \sqrt{2n} + t\right),$$

where C_0 is a constant and t is at most 4.7.

We can now rewrite s as

$$s > C \cdot \sigma^2 \cdot \left(\sqrt{nk} + \sqrt{2n} + 4.7\right),$$

where $C = 2C_0$ is a constant that can be found empirically. In our experiments we used $C = 1.80$.

4.4 GPV signature as a benchmark

To evaluate the performance of our implementation of trapdoor sampling, we use the GPV hash-and-sign digital signature developed in [13] and implemented for the ring-LWE constuction in [9].

The key generation for this ring variant of the GPV scheme is exactly the same as the trapdoor generation depicted in Algorithm 1. In this case, the verification key is the public key $\mathbf{A}$ and the signing key is the trapdoor $(\hat{\mathbf{r}}, \hat{\mathbf{e}})$.

The signing operation of the GPV scheme is the Gaussian preimage sampling described in Algorithm 2. In this case, the syndrome $\mathbf{u} = H(\mu)$, where μ is the message being signed and $H(\mu)$ is the hash of the message, computed in our implementation by SHA-256 padded with random strings generated using a Pseudo-Random Number Generation (PRNG).

The verification operation is to check whether $\mathbf{Ax} \equiv H(\mu)$. If the equivalence relation is true, the verification is successful.

The GPV scheme is convenient for benchmarking because it wraps around the lattice trapdoor operations, and does not introduce any other compute- or space-intensive steps (the overhead of SHA-256 and random padding is negligible compared to the Gaussian preimage sampling).

4.5 Software implementation

We implement the trapdoor sampling algorithms and GPV scheme in a general-purpose portable multi-threaded C++ library. We design this library to be modular and provide three major software layers, each of which includes a collection of C++ classes to provide encapsulation, low inter-module coupling and high intra-module cohesion. The software layers are the (1) cryptographic primitives, (2) lattice constructs, and (3) arithmetic (primitive math) layers.

- The cryptographic primitives layer houses digital signature schemes through calls to objects in the lower layers.
- The lattice constructs layer provides support for power-of-two cyclotomic rings and fields (coefficient and CRT representations). Lattice operations are decomposed into primitive arithmetic operations on integers, vectors, and matrices in the arithmetic layer.

– The arithmetic layer provides basic modular operations, implementations of Number-Theoretic Transforms (NTT), Fermat-Theoretic Transform (FTT) and the Discrete Fourier Transform (DFT). Our discrete Gaussian samplers are implemented in this layer.

Our software library uses both native 64-bit math backend and a custom multi-precision math backend. For experiments with modulus q under 32 bits, we used the backend wrapped around the native C++ 64-bit unsigned integer data type. For computations dealing with the modulus higher than 32 bits, we relied on a custom multiprecision backend without external dependencies.

5 Experimental Results

5.1 Test bed

We conducted all experiments on a commodity desktop computing environment. The evaluation environment used an Intel Core i7-3770 CPU with four cores (eight logical processors) rated at 3.40GHz and 16GB of memory running CentOS 7.

We performed our experiments for five values of ring dimension n from 512 to 8192. This range covers most of the cryptography protocols based on strong lattice trapdoors. For $n = 512$ and $n = 1024$, we used the same modulus bit width as in [9]. For higher values of n, we used the median bit widths satisfying the security constraint $\delta < 1.006$ discussed in Section 4.3.

In the experiments for $n = 512$ and $n = 1024$, we used the native 64-bit mathematical backend. For higher values of ring dimension n, we relied on a custom multiprecision mathematical backend.

5.2 Single-threaded experiments

Table 2. Runtime and space requirements for single-threaded experiments

n	k	Runtime [ms]			Size [kB]		
		Key generation	Signing	Verification	Public key	Private key	Signature
512	24	9.5	27	0.33	73	57	55
1024	27	17.2	62.5	0.68	173	120	135
2048	55	283	629	23	1,439	489	587
4096	108	2,052	3,940	166	11,100	1,921	2,456
8192	214	16,560	28,360	1,236	87,160	7,613	10,351

The runtimes of key generation (TrapGen), signing (GaussSamp), and verification, and the file sizes of public/private keys and signature $\mathbf{x}$ for single-threaded experiments, are listed in Table 2. The file sizes were computed based on the serialized representation of the keys and signature. The values of $n = 512$

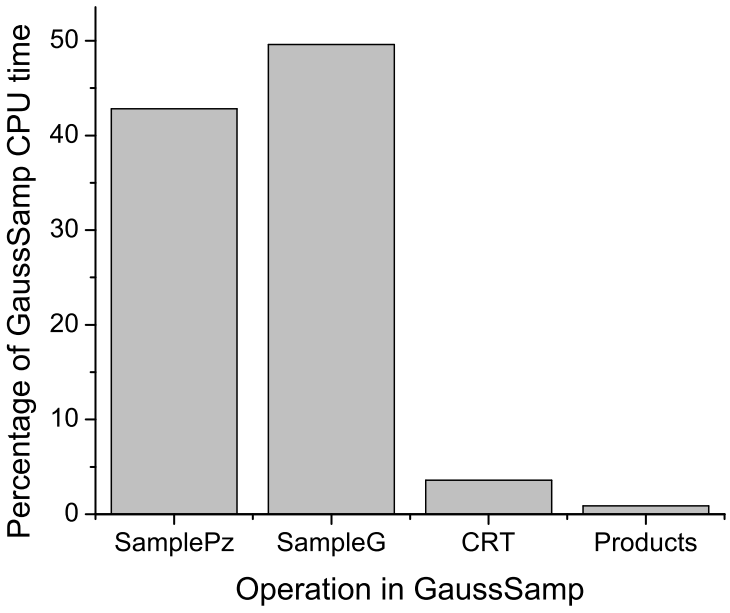

Fig. 1. Profile of the signing (preimage sampling) runtime for $n = 1024$ (native 64-bit mathematical backend)

and $n = 1024$ correspond to approximately 100-bit secure digital signatures. The higher values of n correspond to more advanced cryptographic protocols, such as attribute-based encryption and conjunction obfuscation.

Table 2 suggests that the signing (preimage sampling) time is the main runtime bottleneck, and public key requires the largest storage. Key generation is a one-time operation that runs quickly and requires no extra storage (for the Cholesky decomposition) in contrast to the results reported in [9]. Verification, which is based on a single inner product of vectors of polynomials, is more than one order of magnitude faster than the signing time.

Our key generation runtime is approximately 3 orders of magnitude smaller then the one reported in [9] (Table 1 includes both numbers). Our signing time is the same for $n = 512$ and slightly faster for $n = 1024$. Our verification time is approximately one order of magnitude smaller. Note that the implementation in [9] used a power-of-two modulus because an efficient algorithm for G-sampling in the case of a non-power-of-two modulus was not available. Although a power-of-two modulus can be used for the GPV signature, many advanced cryptographic primitives, such as attribute-based encryption [2] and conjunction obfuscation [4], are formulated for a prime modulus.

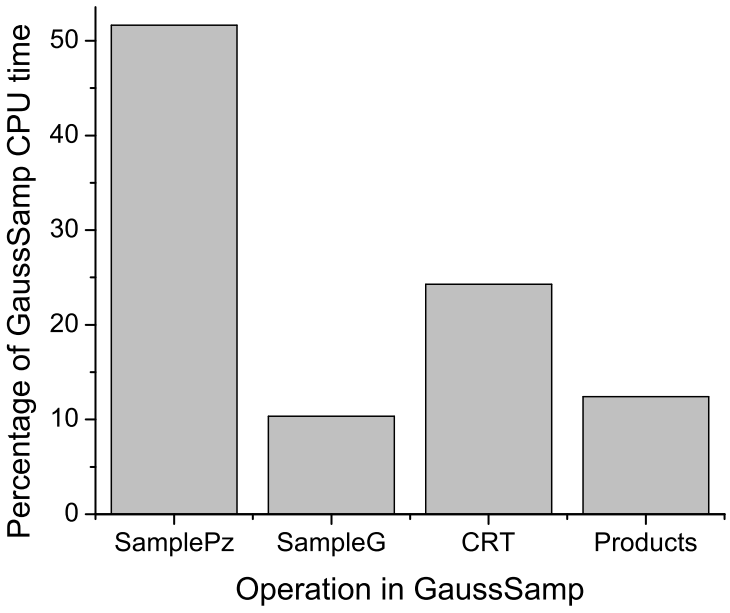

Fig. 2. Profile of the signing (preimage sampling) runtime for $n = 4096$ (multiprecision mathematical backend)

To the best of our knowledge, there are no benchmarks in literature for lattice trapdoor operations relying on $n > 1024$. Experiments for larger n would require significant storage for the Cholesky decomposition matrix when using the perturbation sampling methods developed in [18, 9].

As the preimage sampling GaussSamp is the main bottleneck in lattice trapdoor operations, we profiled it using Callgrind[5] for the cases of $n = 1024$ and $n = 4096$, which correspond to the native 64-bit and multiprecision mathematical backends, respectively. The profiles are depicted in Figures 1 and 2. As can be seen in Algorithm 2, the preimage sampling operation can be broken down into four major components: SampleP$_Z$, SampleG, CRT operations when converting $\mathbf{z}$ to $\hat{\mathbf{z}}$, and three polynomials products, namely, $\mathbf{Ap}$, $\hat{\mathbf{e}}\hat{\mathbf{z}}$, and $\hat{\mathbf{r}}\hat{\mathbf{z}}$. The combined contribution of these four components to preimage sampling in all our experiments was always above 95%.

Figure 1 shows that the perturbation sampling SampleP$_Z$ accounts for 43% of GaussSamp in the case of the native 64-bit mathematical backend. It should be noted that this operation is considered offline in [18] since it does not depend on the syndrome, i.e., a hash in the case of the GPV digital signature. This

[5] http://valgrind.org/docs/manual/cl-manual.html

operation can be performed independently, and the perturbation vectors can be fed to the preimage sampler on demand.

The G-sampling operation accounts for 50% of the GaussSamp runtime. Further analysis shows that 34% (with respect to GaussSamp) of CPU time is consumed by rejection sampling. This suggests that a more efficient integer sampling method, such as the one recently proposed in [20], can substantially improve the performance of G-sampling in this case. A further improvement can be achieved using loop parallelization. As noted in Algorithm 3, all coefficients in the sampled polynomial can be generated in parallel, thus creating an opportunity for loop parallelization in a multi-threaded configuration.

The combined contribution of CRT operations and polynomial products is only 4% as all modular arithmetic operations are performed using the native C++ uint64_t integer data type.

Figure 2 provides a different profile for the case of multiprecision mathematical backend. Although the contribution of SampleP$_Z$ is approximately the same (52%), the contribution of G-sampling is much smaller (only 10%) and the contributions of CRT and polynomial products operations are much larger, that is, 24% and 12%, respectively. This implies that further improvement can be achieved by optimizing the modular arithmetic operations of the multiprecision mathematical backend. Another opportunity is to explore loop parallelization for the CRT step in GaussSamp as the NTT operation is performed for k independent polynomials.

We also performed experiments for the case when the G-sampling procedure of [11] and the perturbation generation procedure of [9] are used for sampling the lattices with a prime modulus. Our results for $n \leq 1024$ showed that the runtime of G-sampling for lattices with a prime modulus was comparable to the one reported in [9] for a power-of-two modulus (the prime modulus variant was approximately three times slower than the power-of-two one). This supports the claim made in [11] that the complexity and expected runtime of their G-sampling for an arbitrary modulus is comparable to the G-sampling for a power-of-two modulus developed in [18].

5.3 Multi-threaded experiments

Multi-threading of our implementation was performed using OpenMP 4.0[6]. There are two approaches to parallelization of lattice trapdoor operations via multi-threading. The first one, which we call loop parallelization, focuses on the parallelization inside individual trapdoor generation, preimage sampling, and verification operations. The second approach, which we call batch parallelization, relies on parallel processing of a batch of preimage sampling or verification operations.

Loop parallelization inside lattice trapdoor operations The loop parallelization inside individual lattice trapdoor operations allows one to reduce the

[6] http://www.openmp.org/

actual runtime of a single trapdoor generation or preimage sampling on a multi-core machine. This approach to parallelization can be used for any cryptographic primitive based on lattice trapdoors.

In Algorithms 1 and 2 we identified two loops that deal with a large number of CRT operations that can be performed in parallel. In Algorithm 1, two row vectors of polynomials $\hat{\mathbf{r}}$ and $\hat{\mathbf{e}}$ need to be converted from the coefficient representation to the evaluation one. Each row vector contains k independent polynomials. In Algorithm 2 the conversion of $\mathbf{z}$ to $\hat{\mathbf{z}}$ requires k independent CRT operations. We used the OpenMP parallelization for these loops.

As the sampling for each coefficient of the syndrome u in Algorithm 3 can be performed in parallel, we used the OpenMP parallelization for this loop.

Figure 3 shows the effect of increasing the number of threads on the preimage sampling runtime on a 4-core machine with 8 threads. It can be seen that the maximum runtime reduction observed on a 4-core machine was by a factor of 2 (for $n = 2048 \dots 8192$). The runtime reduction for the cases of $n \leq 1024$ was roughly by one-fourth. The latter can be explained using Figure 1. In the case of $n \leq 1024$, the loop optimization primarly affected SampleG, and the CRT parallelization had a little effect as its contribution to GaussSamp is relatively small. Thus we observe only a net effect of the parallelization of SampleG, which is only 50% of the GaussSamp execution time.

In the case of larger n, the contribution of CRT operations becomes more significant, as can be seen in Figure 2. The net effect of CRT directly in GaussSamp is 24%. Another CRT for a vector of the same size is performed inside SamplePz. So the total contribution of CRT operations is roughly 50%. Moreover, SampleG accounts for additional 10%. Coupled with substantially larger running times for parallelized subroutines (primarily CRT), this provides a better runtime reduction.

Table 3 lists the runtimes for the optimal mode corresponding to 4 threads. The key generation time gets reduced more than by a factor of 2 for $n \geq 2048$ and by a factor of 1.5 for $n \leq 1024$. The verification runtime is mostly unaffected by loop parallelization because it does not contain any loop-parallelized operations.

Table 3. Runtime in the multi-threaded mode with loop parallelization inside lattice operations. All experiments were conducted for 4 threads (equal to number of cores), which is optimal for the loop parallelization mode

n	k	Runtime [ms]		
		Key generation	Signing	Verification
512	24	6.5	21	0.35
1024	27	7.8	45.6	0.72
2048	55	127	366	22.9
4096	108	815	2,135	160
8192	214	6,161	14,885	1,235

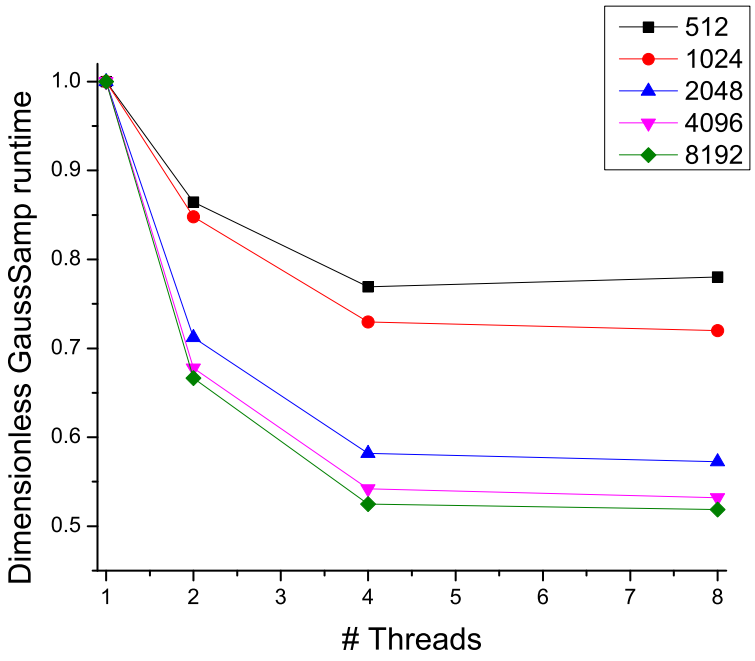

Fig. 3. Runtime improvement with increasing number of threads for the loop parallelization mode

Batch parallelization Batch parallelization deals with processing multiple preimage sampling or verification operations in parallel. This parallelization does not decrease the runtime of each individual operation but increases the throughput. This approach to parallelization can be used effectively when a batch of signatures is being generated or verified. It can also be used in lattice-based cryptography protocols that operate with matrices of private keys, such as conjunction obfuscation [4].

Figure 4 illustrates the effect of batch parallelization on preimage sampling time for different numbers of threads and ring dimensions. It can be seen that for $n \geq 2048$ we observe almost perfect runtime reduction (by a factor of 4 for a 4-core machine). The runtime reduction is slightly smaller for $n \leq 1024$ most likely due to inefficiencies in the PRNG used for integer sampling (a PRNG singleton is used by our implementation).

Table 4 lists the runtime for 8 threads, which is the optimal mode in the case of batch parallelization. The parallelization was only applied to the preimage sampling and verification operations. It can be seen that verification runtime also reduces by approximately a factor of 4.

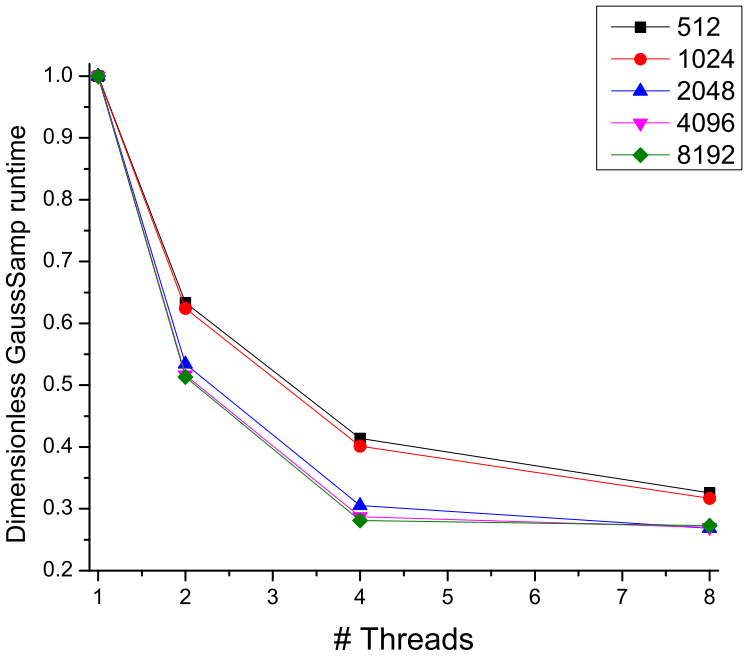

Fig. 4. Runtime improvement with increasing number of threads for the batch parallelization mode

Table 4. Runtime in the multi-threaded mode with batching of signing/verification operations. All experiments were conducted for 8 threads (equal to number of logical processors), which is optimal for the batch mode of parallelization

n	k	Normalized runtime [ms]		
		Key generation	Signing	Verification
512	24	6.9	8.9	0.066
1024	27	7.8	19.8	0.15
2048	55	127	169	6.2
4096	108	820	1,063	45
8192	214	6,160	7,740	359

6 Concluding remarks

In this paper we implement in software a variation of trapdoor sampling based on approaches in [11] for the case of power-of-two cyclotomic rings with a prime modulus. We evaluate the scalability and runtime performance of our implementation using the GPV hash-and-sign digital signature primitive developed in [13]. Our runtimes are substantially faster and storage requirements are dramatically smaller than for prior implementations [9].

Our experimental results for larger values of ring dimension and moduli suggest that this implementation can be applied to many other lattice cryptographic protocols based on power-of-two cyclotomic rings with prime moduli, including identity-based encryption [13], attribute-based encryption [2], and conjunction obfuscation [4].

Our analysis implies that the runtime performance of preimage sampling can be further improved by using faster integer sampling methods, such as the ones recently proposed in [20], more efficient multiprecision modular arithmetic implementations, and multi-threaded parallelization at a lower implementation level, for instance, by parallelizing vector operations. As our implementation operates on double-precision floating-point numbers and has no external dependencies, it can be applied to GPU systems to achieve a dramatic improvement in preimaging runtime.

Our implementation can be extended to support arbitrary moduli and rings over arbitrary cyclotomic polynomials as the underlying algorithms [11] support this more general configuration.

7 Acknolwedgements

We would like to gratefully acknowledge helpful input and feedback from Daniele Micciancio, Nicholas Genise and Michael Walter of University of California San Diego.

References

1. Aysu, A., Patterson, C., Schaumont, P.: Low-cost and area-efficient fpga implementations of lattice-based cryptography. In: Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on. pp. 81–86 (June 2013)
2. Boneh, D., Gentry, C., Gorbunov, S., Halevi, S., Nikolaenko, V., Segev, G., Vaikuntanathan, V., Vinayagamurthy, D.: Fully Key-Homomorphic Encryption, Arithmetic Circuit ABE and Compact Garbled Circuits, pp. 533–556. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
3. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. ACM Transactions on Computation Theory (TOCT) 6(3), 13 (2014)
4. Brakerski, Z., Vaikuntanathan, V., Wee, H., Wichs, D.: Obfuscating conjunctions under entropic ring LWE. In: Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science. pp. 147–156. ITCS '16, ACM, New York, NY, USA (2016)
5. Chen, Y., Nguyen, P.Q.: BKZ 2.0: Better lattice security estimates. In: ASIACRYPT. Lecture Notes in Computer Science, vol. 7073, pp. 1–20. Springer (2011)
6. Dhem, J.F., Quisquater, J.J.: Recent results on modular multiplications for smart cards. In: Quisquater, J.J., Schneier, B. (eds.) Smart Card Research and Applications, Lecture Notes in Computer Science, vol. 1820, pp. 336–352. Springer Berlin Heidelberg (2000)
7. Dowlin, N., Gilad-Bachrach, R., Laine, K., Lauter, K., Naehrig, M., Wernsing, J.: Manual for using homomorphic encryption for bioinformatics. Microsoft Research (2015)
8. Ducas, L., Nguyen, P.Q.: Faster Gaussian Lattice Sampling Using Lazy Floating-Point Arithmetic, pp. 415–432. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
9. El Bansarkhani, R., Buchmann, J.: Improvement and Efficient Implementation of a Lattice-Based Signature Scheme, pp. 48–67. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
10. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144 (2012), http://eprint.iacr.org/
11. Genise, N., Micciancio, D.: Faster Gaussian sampling for trapdoor lattices with arbitrary modulus. http://cseweb.ucsd.edu/~daniele/papers/Sampling.pdf (2017), In Preparation. Accessed: 2017-03-15
12. Gentry, C., Halevi, S., Smart, N.: Homomorphic evaluation of the AES circuit. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology–CRYPTO 2012, Lecture Notes in Computer Science, vol. 7417, pp. 850–867. Springer Berlin / Heidelberg (2012)
13. Gentry, C., Peikert, C., Vaikuntanathan, V.: Trapdoors for hard lattices and new cryptographic constructions. In: Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing. pp. 197–206. STOC '08, ACM, New York, NY, USA (2008)
14. Gentry, C., Sahai, A., Waters, B.: Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In: Advances in Cryptology–CRYPTO 2013, pp. 75–92. Springer (2013)
15. Halevi, S., Shoup, V.: Helib-an implementation of homomorphic encryption (2014)
16. Lyubashevsky, V., Peikert, C., Regev, O.: A toolkit for ring-LWE cryptography. In: EUROCRYPT. vol. 7881, pp. 35–54. Springer (2013)

17. Micciancio, D.: Lattice-based cryptography. In: Encyclopedia of Cryptography and Security, pp. 713–715. Springer (2011)
18. Micciancio, D., Peikert, C.: Trapdoors for lattices: Simpler, tighter, faster, smaller. In: EUROCRYPT. pp. 700–718 (2012)
19. Micciancio, D., Regev, O.: Lattice-based Cryptography, pp. 147–191. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
20. Micciancio, D., Walter, M.: Gaussian sampling over the integers: Efficient, generic, constant-time. In preparation (2017), personal communication
21. Peikert, C.: Public-key cryptosystems from the worst-case shortest vector problem. In: Proceedings of the forty-first annual ACM symposium on Theory of computing. pp. 333–342. ACM (2009)
22. Peikert, C.: An efficient and parallel Gaussian sampler for lattices. In: CRYPTO. pp. 80–97 (2010)
23. Peikert, C.: A decade of lattice cryptography. Foundations and Trends® in Theoretical Computer Science 10(4), 283–424 (2016)
24. Regev, O.: Quantum computation and lattice problems. SIAM J. Comput. 33(3), 738–760 (2004), preliminary version in FOCS 2002
25. Regev, O.: Lattice-based cryptography. In: Annual International Cryptology Conference. pp. 131–141. Springer (2006)

Appendix

Algorithm 3 G-sampling [11]

function SAMPLEG($s, u, \mathbf{q}$) ▷ $\mathbf{q} = [q]_2^k$ is the vector of bits in modulus q
  $\sigma := s/3$
  $l_0 := \sqrt{2\,(1 + 1/k) + 1}$
  $h_0 := 0$
  $d_0 := q_0/2$
  for $i = 1..k-1$ **do**
    $l_i := \sqrt{2\,(1 + 1/(k-i))}$ ▷ l_i, h_i are entries in sparse triangular matrix $\mathbf{L}$
    $h_i := \sqrt{2\,(1 - 1/\{k-(i-1)\})}$
    $d_i := (d_{i-1} + q_i)\,/2$ ▷ d_i are entries in the last column of matrix $\mathbf{D}$
  end for
  `Define` $\mathbf{Z} \in \mathbb{Z}^{k \times n}$ ▷ this vector will store the result of G-sampling
  for $i = 0..n-1$ **do** ▷ Iterate through all coefficients of polynomial. This loop can be parallelized.
    $\mathbf{v} := u(i)$ ▷ $\mathbf{v} = [v]_2^k$ is the vector of bits in coefficient $u(i) \in \mathbb{Z}_q$
    $\mathbf{p} \leftarrow$ PERTURB($\sigma, \mathbf{l}, \mathbf{h}$) ▷ $\mathbf{p} \in \mathbb{Z}^k; \mathbf{l}, \mathbf{h} \in \mathbb{R}^k$
    $c_0 := (v_0 - p_0)/2$
    for $j = 1..k-1$ **do**
      $c_j = (c_{j-1} + v_j - p_j)/2$
    end for
    $\mathbf{z} \leftarrow$ SAMPLED($\sigma, \mathbf{c}, \mathbf{d}$) ▷ $\mathbf{z} \in \mathbb{Z}^k; \mathbf{c}, \mathbf{d} \in \mathbb{R}^k$
    $t_0 := 2 \cdot z_0 + q_0 \cdot z_{k-1} + v_0$
    for $j = 1..k-2$ **do**
      $t_j := 2 \cdot z_j - z_{j-1} + q_j \cdot z_{k-1} + v_j$
    end for
    $t_{k-1} := q_{k-1} \cdot z_{k-1} - z_{k-2} + v_{k-1}$
    $\mathbf{Z}(:, i) := \mathbf{t}$ ▷ $\mathbf{t} = (t_0, t_1, \ldots, t_{k-1}) \in \mathbb{Z}^k$
  end for
  return Z
end function

function PERTURB($\sigma, \mathbf{l}, \mathbf{h}$) ▷ $\mathbf{l}, \mathbf{h} \in \mathbb{R}^k$ are the entries in matrix $\mathbf{L}$
  $\beta := 0$
  for $i = 0..k-1$ **do**
    $c_i := \beta/l_i$ and $\sigma_i := \sigma/l_i$
    $z_i \leftarrow$ SAMPLEZ(σ_i, c_i)
    $\beta_i = -z_i h_i$
  end for
  $p_0 := 5z_0 + 2z_1$
  for $i = 1..k-2$ **do**
    $p_i := 2\,(z_{i-1} + 2z_i + z_{i+1})$
  end for
  $p_{k-1} := 2\,(z_{k-2} + 2z_{k-1})$
  return p ▷ $\mathbf{p} = (p_0, p_1, \ldots, p_{k-1}) \in \mathbb{Z}^k$
end function

function SAMPLED($\sigma, \mathbf{c}, \mathbf{d}$) ▷ Sample from the lattice generated by matrix $\mathbf{D}$
  $z_{k-1} \leftarrow$ SAMPLEZ($\sigma_i/d_{k-1}, -c_{k-1}/d_{k-1}$)
  $\mathbf{c} := \mathbf{c} - z_{k-1}\mathbf{d}$
  for $i = 0..k-2$ **do**
    $z_i \leftarrow$ SAMPLEZ($\sigma, -c_i$)
  end for
  return z ▷ $\mathbf{z} = (z_0, z_1, \ldots, z_{k-1}) \in \mathbb{Z}^k$
end function

Algorithm 4 Perturbation generation [11]

function SAMPLEP$_Z(n, q, s, \alpha, (\hat{\mathbf{r}}, \hat{\mathbf{e}})$)
  $z := \left(\alpha^{-2} - s^{-2}\right)^{-1}$
  $a := s^2 - z\sum_{i=1}^{k} \hat{r}_i^t \hat{r}_i$ ▷ $a \in \mathcal{K}_{2n}$
  $b := -z\sum_{i=1}^{k} \hat{r}_i^t \hat{e}_i$ ▷ $b \in \mathcal{K}_{2n}$
  $d := s^2 - z\sum_{i=1}^{k} \hat{e}_i^t \hat{e}_i$ ▷ $d \in \mathcal{K}_{2n}$
  for $i = 0..nk-1$ **do**
    $q_i \leftarrow$ SAMPLEZ$(\sqrt{s^2 - \alpha^2})$
  end for
  `convert` $\mathbf{q} \in \mathbb{Z}^{k\times n}$ `to` $\hat{\mathbf{q}} \in \mathcal{R}_q^k$ ▷ CRT operations can be executed in parallel
  $\mathbf{c} := -\frac{-\alpha^2}{s^2-\alpha^2} \begin{bmatrix} \hat{\mathbf{r}} \\ \hat{\mathbf{e}} \end{bmatrix} \hat{\mathbf{q}}$ ▷ $\mathbf{c} \in \mathcal{K}_{2n}^2$
  $\mathbf{p} \leftarrow$ SAMPLE2$_Z(a, b, d, \mathbf{c})$ ▷ $\mathbf{p} \in \mathbb{Z}^{2\times n}$
  `convert` $\mathbf{p} \in \mathbb{Z}^{2\times n}$ `to` $\hat{\mathbf{p}} \in \mathcal{R}_q^2$
  return $(\hat{\mathbf{p}}, \hat{\mathbf{q}})$
end function

function SAMPLE2$_Z(a, b, d, c$)
  `let` $\mathbf{c} = (c_0, c_1)$
  $q_1 \leftarrow$ SAMPLEF$_Z(d, c_1)$ ▷ $q_1 \in \mathbb{Z}^n$
  `convert` $q_1 \in \mathbb{Z}^n$ `to` $\hat{q_1} \in \mathcal{K}_{2n}$
  $c_0 := c_0 + bd^{-1}(\hat{q_1} - c_1)$
  $q_0 \leftarrow$ SAMPLEF$_Z(a - bd^{-1}b^t, c_0)$ ▷ $q_0 \in \mathbb{Z}^n$
  return (q_0, q_1)
end function

function SAMPLEF$_Z(f, c$)
  if $\dim(f) = 1$ **then return** SAMPLEZ $\left(\sqrt{f}, c\right)$
  else
    `let` $f(x) = f_0(x^2) + x \cdot f_1(x^2)$ ▷ Extract even and odd componets of $f(x)$
    $\mathbf{c}' = P_{stride}(\mathbf{c})$ ▷ P_{stride} permutes coefficients $(a_0, a_1, \ldots, a_{n-1})$ to
      ▷ $(a_0, a_2, \ldots, a_{n-2}, a_1, a_3, \ldots, a_{n-1})$
    $(q_0, q_1) \leftarrow$ SAMPLE2$_Z(f_0, f_1, f_0, c')$
    `let` $q(x) = q_0(x^2) + x \cdot q_1(x^2)$
    return q
  end if
end function