Vectorized Discrete Gaussian Sampling with SIMD Support

Jia-jia LU and Yu-song DU*

School of Data and Computer Science, Sun Yat-sen University, Guangzhou, 510006, China
*Corresponding author

Keywords: Lattice-based cryptosystem, Discrete Gaussian sampling, SIMD.

Abstract. Discrete Gaussian sampling is a fundamental building block of lattice-based cryptography. Sampling from a Gaussian distribution $D_{\mathbb{Z},\sigma,c}$ over the integers $\mathbb{Z}$ is an important sub-problem of discrete Gaussian sampling, where parameter $\sigma > 0$ and center $c \in \mathbb{R}$. In this paper, we show that two common sampling algorithms for discrete Gaussian distribution over the integers can be implemented more efficiently by using vectorization with SIMD (Single Instruction Multiple Data) support. Specifically, we use the VCL (C++ vector class library) by Agner Fog, which offers optimized vector operations for integers and floating-point numbers with the support of SIMD. The VCL is also a simple tool for constant-time implementations, which helps prevent the information leakage caused by the timing attacks on sampling operations.

Introduction

Discrete Gaussian sampling, which is to sample from a discrete Gaussian distribution $D_{\mathbb{Z},\sigma,c}$ with parameter $\sigma > 0$ and center $c \in \mathbb{R}^n$ over an n-dimensional lattice $\Lambda$, plays a fundamental role in lattice-based cryptography [1,2]. An important sub-problem of discrete Gaussian sampling, denoted by Sample$\mathbb{Z}$, which is to sample from a discrete Gaussian distribution $D_{\mathbb{Z},\sigma,c}$ over the integers $\mathbb{Z}$ with parameter $\sigma > 0$ and center $c \in \mathbb{R}$, is usually one of the subroutines in discrete Gaussian sampling algorithms for distributions over a general n-dimensional lattice $\Lambda$ [3]. Furthermore, Sample$\mathbb{Z}$ is much more efficient and simpler than discrete Gaussian sampling over a general lattice, in some lattice-based cryptosystems, such as the ones shown in [4,5], the operations involving discrete Gaussian sampling are nothing but Sample$\mathbb{Z}$. The methods of sampling from a continuous Gaussian distribution are not trivially applicable for the discrete case. Therefore, how to design and implement good sampling algorithms, especially for Sample$\mathbb{Z}$, has received attentions in recent years.

Sample$\mathbb{Z}$ usually involves floating-point arithmetic because of the exponent operation in Gaussian function. This implies that online high-precision floating-point computation may be the biggest implementation bottleneck [6]. We can see that most of improved Sample$\mathbb{Z}$ algorithms manage to use high-precision floating-point arithmetic only at offline time [3,7]. Fortunately, the later work on discrete Gaussian sampling precision suggested that significand precision of 53 bits provided by double-precision floating arithmetic is sufficient for most of the security applications [2,8]. Thus, it is feasible to design Sample$\mathbb{Z}$ algorithms with the standard double precision floating-point arithmetic.

The side-channel leakage of discrete Gaussian sampling algorithms has also been recognized as an important problem. Bruinderink et al. and Espitau et al. presented (timing) side-channel attacks on the sampling algorithms used in BLISS signature scheme respectively [9,10]. In order to resist timing attacks, all the operations involving secret information in a discrete Gaussian sampling algorithm should be executed in a time which is independent from the secret data. This property of discrete Gaussian sampling algorithms is called time independence, which can be achieved by ensuring constant execution time [11] or randomly shuffling the secret values [12].

In this paper, we show that two common sampling algorithms for discrete Gaussian distribution over the integers can be more efficiently implemented by using vectorization with SIMD (Single Instruction Multiple Data) support. Specifically, we use the VCL (C++ vector class library) by
Agner Fog, which offers optimized vector operations for integers and floating-point numbers with the support of SIMD [13]. The two sampling algorithms are the GPV algorithm given by Gentry et al. in [1] and the inversion sampling algorithm (using a cumulative distribution table) which was suggested to be used by Peikert in [3]. We select these two algorithms, since the GPV algorithm supports varying parameters (including parameter \( \sigma > 0 \) and center \( c \in \mathbb{R} \)), while inversion sampling is widely used as a base sampler with a fixed and relatively small \( \sigma \) \([2,3]\). Furthermore, the VCL is a simple tool for constant-time implementations [14]. One can see that the two algorithms we implemented in this paper are constant-time or at least time-independent, helping mitigate the information leakage caused by the timing attacks on sampling operations.

Preliminaries

We denote the set of real numbers by \( \mathbb{R} \) and the set of integers by \( \mathbb{Z} \). We extend any real function \( f(\cdot) \) to a countable set \( A \) by defining \( f(A) = \sum_{x \in A} f(x) \). The Gaussian function on \( \mathbb{R} \) with parameter \( \sigma > 0 \) and center \( c \in \mathbb{R} \) evaluated at \( x \in \mathbb{R} \) can be defined by

\[
\rho_{\sigma,c}(x) = e^{-(x-c)^2/2\sigma^2}.
\]

For real \( \sigma > 0 \) and \( c \in \mathbb{R} \), the discrete Gaussian distribution over the integers \( \mathbb{Z} \) is defined by

\[
D_{\mathbb{Z},\sigma,c}(x) = \frac{\rho_{\sigma,c}(x)}{\rho_{\sigma,c}(\mathbb{Z})}
\]

for \( x \in \mathbb{Z} \). By convention, the subscript \( c \) is omitted when it is taken to be 0. For \( \sigma > 0 \) and \( c \in \mathbb{R} \), sampling an integer \( x \) from \( D_{\mathbb{Z},\sigma,c} \) is equivalent to sampling an integer \( x - \lfloor c \rfloor \) according to \( D_{\mathbb{Z},\sigma,\{c\}} \), where \( \{c\} \) is the fractional part of \( c \) such that \( 0 \leq \{c\} < 1 \).

The GPV Sampling Algorithm

The GPV sampling algorithm was given by Gentry et al. in [1], which uses rejection sampling from the uniform distribution over \([c - \tau \sigma, c + \tau \sigma]\) by outputting a uniform integer \( x \) with probability \( \rho_{\sigma,c}(x) = e^{-(x-c)^2/2\sigma^2} \). It needs high-precision floating-point arithmetic to compute the value of \( \rho_{\sigma,c}(x) \) online for every integer \( x \) uniformly taken from \([c - \tau \sigma, c + \tau \sigma]\). By the principle of rejection sampling, one can see that GPV algorithm requires about \( 2\tau/\sqrt{2\pi} \) trails on average. A few years ago, it was believed that \( \tau \) should be not less than 12, which implies that the GPV sampling algorithm cannot be very efficient because it requires \( 2\tau/\sqrt{2\pi} \approx 10 \) trials on average. The later works on discrete Gaussian sampling precision suggested that the number of trials could be decreased by using more cryptographically efficient measures, such as Rényi divergence [8] and max-log distance [2], and the significand precision of 53 bits provided by double-precision floating arithmetic is sufficient for most of the security applications. So, as in [2], one can take \( \tau = 6 \) (i.e., \( 2\tau/\sqrt{2\pi} \approx 5 \)), and uses standard double-precision floating arithmetic to compute \( \rho_{\sigma,c}(x) \), which certainly gives a performance boost for the GPV sampling algorithm.

The Inversion Sampling Algorithm based on Cumulative Distribution Tables

By using a pre-computed cumulative distribution table (CDT), inversion sampling can generate sample numbers at random from any probability distribution given its cumulative distribution function. We take the discrete (and finite) case as an instance. Let \( x_1, x_2, \ldots, x_n \) be all the possible value of the random variate \( X \). The probability of \( X = x_i \) is denoted by \( Pr(X = x_i) = p_i \), where \( i = 1, 2, \ldots, n \) and \( \sum_{i=1}^{n} p_i = 1 \). Thus, the CDT can be represented by \( \{F(x_1), F(x_2), \ldots, F(x_n) = 1\} \) with \( F(x_j) = \sum_{i=1}^{j} p_i \). When generating a sample number from \( X \), inversion sampling takes a uniform deviate \( u \in [0, 1) \), and then returns \( X = x_j \) such that \( F(x_{j-1}) < u \leq F(x_j) \), i.e., \( \sum_{i=1}^{j-1} p_i < u \leq \sum_{i=1}^{j} p_i \).

SIMD and the VCL Vector Class Library

Modern CPU’s have “Single Instruction Multiple Data” (SIMD) instructions for handling vectors of multiple data elements in parallel. The VCL vector class library is a tool that allows C++
programmers to speed up their code by handling multiple data in parallel. The compiler may be able to use SIMD instructions automatically in simple cases, but at the same time, a human programmer is more likely to do it better by organizing data into vectors that fit the SIMD instructions. With the VCL library, it is easier for programmers to write vector code instead of using assembly language or intrinsic functions [13].

Implementing the GPV Algorithm with VCL

In this section, we implement the GPV algorithm with the VCL. Since most of modern CPU’s support AVX (AVX2) instruction set but not AVX512, we use Vec4d class which handles vectors with 4 elements of double floating-point precision in parallel with the support of AVX. Moreover, the GPV algorithm needs a large amount of uniformly (pseudo-)random numbers. We generate these random numbers by using AES256-CTR with AES-NI.

We divide the GPV algorithm into two sub-functions, PopGPVPool() and SamplerGPV() (see the pseudocode in Appendix). The for loop in the PopGPVPool() is a time independent implementation of sampling $x$ uniformly from $[c - \tau \sigma, c + \tau \sigma] \cap \mathbb{Z}$. Specifically, it is to sample an integer $r$ between 0 and $2^{\log_2 l} - 1$ and accept $r$ if $r < l$ to generate a uniformly random number from $[0, l]$, where $l = \text{interval\_length} = 2 \cdot \lceil \tau \sigma \rceil + 1$. This is also a rejection sampling procedure. It is not constant-time but actually time independent. It is clear that $x = r - \text{half\_interval\_length} = r - \lceil \tau \sigma \rceil$ is a uniformly random integer in $[c - \tau \sigma, c + \tau \sigma]$.

The for loop gives 4 uniformly random integers from $[c - \tau \sigma, c + \tau \sigma]$ at a time after which these four integers are stored into the variable vx as a vector with 4 double precision floating-point numbers. The following statements in PopGPVPool() are designed to parallelly compute the exponential function $\rho_{\sigma, c}(x) = e^{-(x-c)^2/2\sigma^2}$. The four values of exponential function are then stored into the variable vprob and array_prob. Thus, PopGPVPool() generates 4 prospective outputs at a time.

Next, SamplerGPV() carries out the acceptance-rejection operation for each integer in vx until all the 4 prospective outputs in vx are exhausted, and then calls PopGPVPool() again. In particular, the integer $w$ in SamplerGPV() can be seen as the binary representation of the probability value. If the random integer $u w \leq w$, the corresponding $x$ can be returned as a sample number from $D_{\mathbb{Z}, \sigma, c}$.

According to the implementation detail of the VCL, most of arithmetic operators and mathematical functions in VCL have constant runtime, including multiplication, division, square and the exponential function, i.e. there is no timing difference dependent on the input [14]. This makes the VCL ideal for constant-time implementations, so we can see that PopGPVPool() is time independent. It is clear that SamplerGPV() is not constant-time because of its acceptance rejection operations. Fortunately, the timing difference from the acceptance-rejection operations is independent on the final outputs. Therefore, we say that SamplerGPV() is time-independent, i.e., adversary cannot obtain any more information on samples generated by SamplerGPV() through the time difference.

Implementing the Inversion Sampling Algorithm with VCL

The support of a discrete Gaussian distribution over the integers is infinite, which leads to a CDT of infinite size. In practice, one has to truncate the distribution table but ensure adequate precision at the same time. Following the idea of setting parameter $\tau$ in the GPV sampling algorithm, we can also take $\tau = 6$. Specifically, by using the MPFR library, we compute the probability density of $x$ offline, which is $D_\sigma(x) = e^{-(x-c)^2/2\sigma^2} / \sum_{y=-\lceil \tau \sigma \rceil}^{\lceil \tau \sigma \rceil} e^{-(y-c)^2/2\sigma^2}$ for $x = 0, \pm 1, \pm 2, \pm 3, \cdots, \pm \lceil \tau \sigma \rceil$, then use $2\lceil \tau \sigma \rceil + 1$ uint64_t integers to store the binary expansions of these probabilities and get the CDT $\{ F(x_1), F(x_2), \cdots, F(x_{2\lceil \tau \sigma \rceil + 1}) = 1 \}$. After establishing the CDT for given parameter $\sigma$ and center $c$, the inversion sampling algorithm is just to traverse the whole CDT linearly, and implementing it with the VCL is only a few lines of code (see the pseudocode in Appendix).
The variable $v_u$ and $v_{output}$ are both declared as Vec4q. The variable $v_u$ contains 4 random 64-bit integer, corresponding to 4 uniform deviates that are required for inversion sampling. The variable $v_{output}$, which is initialized with the vector $(-\lceil \tau \sigma \rceil, -\lceil \tau \sigma \rceil, -\lceil \tau \sigma \rceil, -\lceil \tau \sigma \rceil)$, handles vectors with 4 elements of uint64_t integers in parallel with the support of AVX. We use the ‘select(*,Vec4q, Vec4q)’ statement that is provided by the VCL so that each component of $v_{output}$ is updated independently. Finally, the SamplerCDT() generates 4 separate sample numbers according to the CDT.

**Experimental Results**

On a laptop computer (Intel i7-8550U, 16GB RAM), using the g++ compiler and enabling -O3 optimization option, we tested the performance of the GPV sampling algorithm for discrete Gaussian distribution $D_{\mathbb{Z}, \sigma, c}$ with $\sigma = 10, 50, 100, 200, 500, 1000$ and $c$ picked uniformly from $[0, 1)$.

| $\tau$ = 30 | $\tau$ = 60 | $\tau$ = 120 | $\tau$ = 250 | $\tau$ = 500 | $\tau$ = 1000 |
|-------------|-------------|-------------|-------------|-------------|-------------|
| GPV        | 3.426       | 3.434       | 3.408       | 3.438       | 3.447       | 3.442       |
| GPV+VCL    | 9.638       | 9.633       | 9.628       | 9.679       | 9.643       | 9.689       |

| $\tau$ = 5 | $\tau$ = 10 | $\tau$ = 10 | $\tau$ = 20 | $\tau$ = 25 | $\tau$ = 30 |
|------------|-------------|-------------|-------------|-------------|-------------|
| Inv.       | 14.234      | 6.892       | 4.507       | 3.295       | 2.567       | 2.239       |
| Inv.+VCL   | 38.163      | 24.134      | 17.789      | 13.856      | 11.513      | 9.726       |

As shown in Table 1, one can get about $3.4 \times 10^6$ samples per second by using the (standard) GPV algorithm. The performance gains from the VCL is substantial, and one can get about $9.6 \times 10^6$ samples per second. At the same implementation environment, Table 2 shows the performance of inversion sampling algorithm for discrete Gaussian distribution $D_{\mathbb{Z}, \sigma, c}$ with $\sigma = 5, 10, 15, 20, 25, 30$ and $c = 0$. We can see that the performance gains from the VCL is also substantial, though its sampling speed is directly bound up with the value of $\sigma$.

**Conclusion**

There have been several theoretical and practical attempts to replace discrete Gaussian distributions in lattice-based cryptography with more implementation friendly ones. However, they only show in some specific lattice-based cryptosystems (such as Bliss signature [5]) that discrete Gaussians could be replaced by more easily samplable distributions with almost no security penalty. The impact on the whole lattice-based cryptography, especially on some advanced cryptographic applications of lattices, still needs to be assessed. For example, it is suggested to replace discrete Gaussians with rounded Gaussians, which can be sampled very efficiently by using Box-Muller transform and rounding to the nearest integer, but the security analysis of rounded Gaussians is only confined to the Bliss signature [14]. Moreover, the VCL is also suggested to be used in the rounded Gaussians to get a constant-time and efficient implementation.

The results presented in paper means that one may not need to replace discrete Gaussian distributions in lattice-based cryptography. Sampling algorithms for discrete Gaussian distribution over the integers can be implemented simply and more efficiently by using the VCL (C++ vector class library) with SIMD (Single Instruction Multiple Data) support. The VCL can also help give constant-time (at least time-independent) implementations of sampling algorithms for Gaussian distribution over the integers.
Acknowledgments

This work was supported by National Key R&D Program of China (2017YFB0802500), Natural Science Foundation of Guangdong Province (2016A030313298) and the Opening Fund of Chongqing Key Lab of Computer Network and Communication Technology (CY-CNCL-2017-04).

Appendix

double stddev, center, prob, tmp; double array_x[4], array_y[4], array_s[4]; Vec4d vx, vy, vs, vz, vtmp, array_prob;
uint16_t half_interval_length = stddev * tau; uint16_t interval_length = 2 * uint16_t(half_interval_length) + 1;
void Sampler::PopGPVPool() {
    uint16_t r;
    for (int i = 0; i < 4; i++) {
        do {r = (random->getRandomShort()&mask_interval_length);} while(r>=interval_length);
        array_x[i] = double(r) - half_interval_length;
    }
    vx.load(array_x);
    vtmp = vx - center; vtmp = square(vtmp); vtmp = vtmp / (2 * stddev * stddev); vtmp = - vtmp;
    vprob = exp(vtmp); vprob.store(array_prob); posGPVPool = 0;
}
int16_t Sampler::SamplerGPV() {
    for(;;) {
        if (posGPVPool>=4) PopulateGPVPool();
        prob = array_prob[posGPVPool] * (uint64_t(1)<<52);
        w = uint64_t(prob); uw = random->getRandomLong() & mask;  // mask = (uint64_t(1)<<52) -1
        if (uw <= w)   {posGPVPool++; return int16_t(array_x[posGPVPool]); }
        else if (uw > w) {posGPVPool++; continue;}
        std::cout << "not enough precision, fail to generate a sample, restart." << std::endl; }
}
Vec4q Sampler::ConstTimeCDT_VCL() {
    random->getRand256bits(rndBytes); rnd256bits = (uint64_t*)((void*)rndBytes);
    vu.load(rnd256bits); voutput = -(int64_t(range));  // range = stddev * tau;
    for (int i = -range; i < range; i++) voutput = voutput + select(vu > CDT[i+range], v_one, v_zero); return voutput;
}

References

[1] C. Gentry, C. Peikert, V. Vaikuntanathan, (2008) Trapdoors for hard lattices and new cryptographic constructions. In: STOC 2008. ACM, Proceedings, pp. 197–206.
[2] D. Micciancio, M. Walter, Gaussian sampling over the integers: Efficient, generic, constant-time. In: CRYPTO 2017. Springer, Part II, Volume 10402 of LNCS, pp. 455–485.
[3] C. Peikert, An efficient and parallel gaussian sampler for lattices. In: CRYPTO 2010. Springer, Volume 6223 of LNCS, pp. 80–97.
[4] R. Lindner, C. Peikert, Better key sizes (and attacks) for lwe-based encryption. In: CT-RSA 2011. Springer, Volume 6558 of LNCS, pp. 319–339.
[5] L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky, Lattice signatures and bimodal gaussians. In: CRYPTO 2013. Springer, Part I, Volume 8042 of LNCS, pp. 40–56.
[6] L. Ducas, P.Q. Nguyen, Faster gaussian lattice sampling using lazy floating-point arithmetic. In: ASIACRYPT 2012. Springer, Volume 7658 of LNCS, pp. 415–432.
[7] N.C. Dwarakanath, S.D. Galbraith, Sampling from discrete gaussians for lattice-based cryptography on a constrained device. Applicable Algebra in Engineering, Communication and Computing, 2014, 25(3):159.

[8] T. Prest, Sharper bounds in lattice-based cryptography using the rényi divergence. In: ASIACRYPT 2017. Springer, Part I, Volume 10624 of LNCS, pp. 347–374.

[9] L.G. Bruinderink, A. Hülsing, T. Lange, Y. Yarom, Flush, gauss, and reload - A cache attack on the BLISS lattice-based signature scheme. In: CHES 2016. Springer, Volume 9813 of LNCS, pp. 323–345.

[10] T. Espitau, P.A. Fouque, B. Gérard, M. Tibouchi, Sidechannel attacks on bliss lattice-based signatures: Exploiting branch tracing against strongswan and electromagnetic emanations in microcontrollers. In: ACM SIGSAC 2017. ACM, Proceedings, pp. 1857–1874.

[11] A. Karmakar, S.S. Roy, O. Reparaz, F. Vercauteren, I. Verbauwhede, Constant-time discrete gaussian sampling. IEEE Trans. Computers, 2018, 67(11):1561–1571.

[12] T. Pöppelmann, L. Ducas, T. Güneysu, Enhanced lattice-based signatures on reconfigurable hardware. In: CHES 2014. Springer, Volume 8731 of LNCS, pp. 353–370.

[13] A. Fog, VCL, C++ vector class library. https://www.agner.org/optimize/.

[14] A. Hülsing, T. Lange, K. Smeets, Rounded Gaussians. In: PKC 2018. Springer, Volume 10770 of LNCS, pp 728-757.