Pushing the speed limit of constant-time discrete Gaussian sampling. A case study on the Falcon signature scheme.

Angshuman Karmakar[1], Sujoy Sinha Roy[2], Frederik Vercauteren[1], Ingrid Verbauwhede[1]
[1]imec-COSIC, KU Leuven, Kasteelpark Arenberg 10, Bus 2452, B-3001 Leuven-Heverlee, Belgium
[2]School of Computer Science, University of Birmingham, United Kingdom
{firstname.lastname}@esat.kuleuven.be,s.sinharoy@cs.bham.ac.uk

ABSTRACT

Sampling from a discrete Gaussian distribution has applications in lattice-based post-quantum cryptography. Several efficient solutions have been proposed in recent years. However, making a Gaussian sampler secure against timing attacks turned out to be a challenging research problem. In this work, we present a toolchain to instantiate an efficient constant-time discrete Gaussian sampler of arbitrary standard deviation and precision. We observe an interesting property of the mapping from input random bit strings to samples during a Knuth-Yao sampling algorithm and propose an efficient way of minimizing the Boolean expressions for the mapping. Our minimization approach results in up to 37% faster discrete Gaussian sampling compared to the previous work. Finally, we apply our optimized and secure Gaussian sampler in the lattice-based digital signature algorithm Falcon, which is a NIST submission, and provide experimental evidence that the overall performance of the signing algorithm degrades by at most 33% only due to the additional overhead of 'constant-time' sampling, including the 60% overhead of random number generation. Breaking a general belief, our results indirectly show that the use of discrete Gaussian samples in digital signature algorithms would be beneficial.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; *Digital signatures*; *Hardware attacks and countermeasures*; Cryptography.

1 INTRODUCTION

In the past decade, the looming threat of quantum computers and widely believed resistance of lattice-based cryptosystems against quantum computers as opposed to popular RSA or elliptic curve based schemes, has given the research in lattice-based cryptography a major boost. Among the lattice-based schemes the cryptosystems based on LWE [22] (or its ring variant RLWE [17]) has emerged to be the most popular mostly due to their simple operations, security evaluation and strong worst case to average case reduction. Therefore, it is not a coincidence that in NIST's recent standardization call for post-quantum cryptographic protocols [1], a majority of the submitted protocols are based on LWE or RLWE.

These problems usually require *error* terms to hide the secret keys. Traditionally, these error terms are sampled from a discrete Gaussian distribution which itself is a non-trivial task. A lot of effort [11, 13, 19, 25] have been devoted into generating these Gaussian samples efficiently. However, in almost all these methods the sampling process runs in non-constant time which opens up new avenues for side-channel attacks. Unfortunately, except for a few simple countermeasures [6, 24], it was not known how to generate Gaussian samples that can resist these attacks. So the current trend is to generate samples from distributions e.g the binomial distribution [2, 5] or the uniform random distribution [8, 9] where it is easy to generate samples in constant-time and model those distributions as Gaussian distribution during security evaluation of the cryptosystem. This works well with encryption and key-exchange schemes but for signature schemes [3, 12] this leads to larger key-sizes and costlier computations. In this work, we revisit the constant-time discrete Gaussian sampler described in [15] and further improve its efficiency. Specifically our contributions in this paper can be summarized as follows,

(1) The work in [15] established the concept of constant-time discrete Gaussian sampling by evaluating Boolean expressions. Unlike the previous work, here we provide a detailed and generic description for generating the Boolean expression. We combine our methods to create a tool to instantiate a discrete Gaussian sampler of arbitrary standard deviation and precision.
(2) We observed a special property of the input random strings that generate samples. We show how we can leverage this property to minimize the Boolean functions efficiently. This minimization technique can speed up the sampling by up to 37% compared to the simple minimization technique in [15].
(3) Finally, we show that using Gaussian samples in a post-quantum signature algorithm can be both very practical and secure. We use the Falcon signature algorithm to show that the performance of the algorithm does not degrade much with respect to the fastest non constant-time sampler and stays within practical limit even after we replace the non constant-time sampler with our constant-time sampler.

2 PRELIMINARIES

In this section, we define different notations which we are going to use throughout this work. We define $\mathbb{Z}^* = \{0\} \cup \mathbb{Z}^+$. All the binary strings are read or processed in right-to-left direction, and the integer value of an n-bit binary string $b = b_{n-1}b_{n-2}\cdots b_1b_0$ is $b_{n-1}+\ldots+2^{n-2}\cdot b_1+2^{n-1}\cdot b_0$. Boolean `and`, `or`, and `not` operations

DAC '19, June 2–6, 2019, Las Vegas, NV, USA

ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317887

are denoted by the symbols &, |, ¯respectively. The terms Boolean function and Boolean expression are used interchangeably. We use f_η to denote Boolean functions that map η Boolean variables to a single Boolean variable. We denote repetition of Boolean variable b, i times $\underbrace{b \ldots b}_{\text{i}}$ by b^i. A Boolean string of length i where each variable is either 0 or 1 is denoted by $(0/1)^i$. For the binary trees, we denote the level where children of the root exist as the 0-th level, children of these nodes reside at 1-st level and so on. Hence, starting from root we need $i+1$ *steps* to reach nodes at level i. We assume that the standard deviations in our sampler are *small* and our sampler can be used as a base sampler in [18, 21] where samples from a discrete Gaussian distribution with large standard deviation are generated by combining samples from a discrete Gaussian distribution with small standard deviation. We also use σ to denote the standard deviation of a Gaussian distribution.

2.1 Discrete Gaussian distribution

The probability distribution function of a discrete Gaussian distribution of a random variable X defined over $\mathbb{Z}$ is given as,

$$\mathscr{D}_\sigma(X = z) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(z-c)^2/2\sigma^2}.$$

In this work, we always consider discrete Gaussian distributions that are centered around 0 *i.e* $c = 0$. Due to the symmetry of the probability density function it is sufficient to generate samples over $\mathbb{Z}^*$ and use a random bit to determine the sign. For most practical scenario, all the samples are generated in the interval $[0, \tau\sigma]$, where τ is a positive constant known as *tail-cut* factor and the probabilities $\mathscr{D}_\sigma(x)$ are calculated up to n-bit precision, we denote this as $\mathscr{D}^n_\sigma(x)$.

2.2 Knuth-Yao Sampling

Dwarakanath et al. [13] first used Knuth-Yao [16] sampling method to generate samples from discrete Gaussian distributions. In the precomputation stage of this method, for a particular standard deviation it first creates a probability matrix of dimension $(\lceil\tau\sigma\rceil + 1) \times n$ where a row consists of binary expanded $\mathscr{D}^n_\sigma(v)$ if $v = 0$ and $2 \cdot \mathscr{D}^n_\sigma(v)$ for all other $v \in [1, \tau\sigma]$. Using this probability matrix a binary tree called discrete distribution generation (DDG) tree is created such that the Hamming weight of the i-th column of the probability matrix equals the number of leaf nodes in the i-th level of the tree and each leaf node contains a sample value in the sample space $[0, \tau\sigma]$. An example is shown in Fig. 1. During sampling, a random walk is started from the root node and a random bit at every step is used to decide between the bottom or top subtree. The sampling stops when this random walk hits a leaf node and the value in the leaf node is returned as sample. It is worthwhile to note here that for a distribution D with finite support s.t $\sum_{x \in Supp(\mathrm{D})} \mathrm{P}^n(x) = 1$, where $\mathrm{P}^n(x)$ is the probability of x up to n-bit floating point precision under the distribution D, the DDG tree is finite and generates samples that follow the distribution D *exactly*. Whereas, for distributions like discrete Gaussian distribution with infinite support the DDG tree grows infinitely and it is not possible to generate samples that follow the discrete Gaussian distribution *exactly*. In this case, the τ and n are chosen such that the statistical distance between the generated distribution and the actual distribution is lower than $2^{-\lambda}$ where λ is the security parameter.

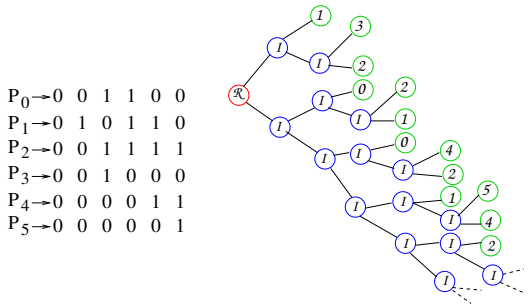

Figure 1: Probability matrix and corresponding DDG tree for $\sigma = 2$ and $n = 6$. Red, blue and green nodes denote the root, intermediate and leaf nodes respectively.

Algorithm 1: Column scanning Knuth-Yao algorithm [25]

```
input  : Probability matrix P
output : Sample value s
d ← 0; Hit ← 0; col ← 0;
for col=0 to n − 1 do
    b ← RandomBit();
    d ← 2 ∗ d + b;
    for row=⌈τσ⌉ down to 0 do
        d ← d − P[row][col];
        if d=-1 then
            s ← row; Hit ← 1;
            ExitForLoop();
if Hit=1 then
    return s
else
    restart
```

2.3 Column scanning Knuth-Yao sampling

The column scanning Knuth-Yao sampling algorithm from [25] generates the DDG tree on-the-fly, thus making the sampling process time and memory efficient. This is shown in Alg. 1. We denote the Hamming weight of column i of the probability matrix P as $\mathrm{h_i}$. Define $\mathrm{GAP^i}$ as,

$$\mathrm{GAP^i} = (b_i + \ldots + b_0 \cdot 2^i) - (\mathrm{h_i} + \ldots + \mathrm{h_0} \cdot 2^i) \tag{1}$$

here b_j is the output of *RandomBit()* during the j-th iteration of the **for** loop in Alg. 1. It is evident from the above algorithm that a sample is found in the i-th column if and only if $\mathrm{GAP^i} < 0$ and $\mathrm{GAP^{i'}} \geq 0, 0 \leq i' < i$. It is clear that Alg. 1 runs in non constant-time. Also, it is possible that with a small probability $O(2^{-n})$, a sample is not found ($Hit = 0$) in that case the algorithm is restarted.

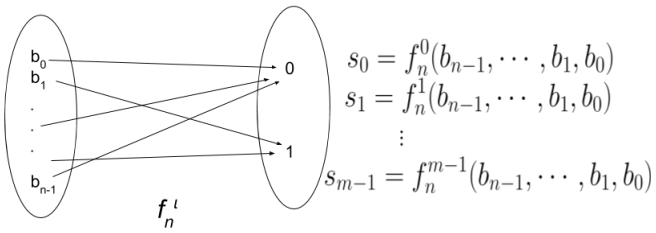

Figure 2: Mapping n random bits to output sample bits and corresponding Boolean functions.

3 PREVIOUS WORK

The bit-sliced discrete Gaussian sampler in [15] first proposed an efficient and constant-time discrete Gaussian sampler. The key observation in this work was that there exists a unique path from the root to each leaf of the DDG tree since it is a binary tree which is determined by the input random bits to the sampler. This implies that there exists a many-to-one mapping between the set of input random bit strings ($b_{n-1} \ldots b_0$) to the set of sample bits. Further, this mapping can be expressed as a set of Boolean functions $f_n^\iota,\ \iota \in [0, m-1]$, where m is the maximum possible bit length of any sample. This is shown in Fig. 2. Now, each sample bit can be calculated in constant-time by executing the corresponding Boolean function completely. This method though being a constant-time algorithm, has very poor efficiency. So, the next key observation was that multiple samples can be generated in a batch using a single instruction multiple data (SIMD) fashion by exploiting wide word length and bit wise Boolean operators of modern processors. In this method, assuming word length w, a variable $b_i^{var}, i \in [0, n-1]$ is packed with w input random bits $b_i^0, b_i^1, \cdots, b_i^{w-1}$. These random variables are then used to evaluate the Boolean function f_n^ι using bitwise Boolean operators to generate variables $s_\iota^{var}, \iota \in [0, m, -1]$ which are packed with w output sample bits $s_\iota^0, s_\iota^1, \cdots, s_\iota^{w-1}$. These variables are then unpacked to generate w samples at a time. Despite the overhead of packing and unpacking bits this method was shown to be approximately two times faster (for $\sigma = 6.15543$) than the best known alternative for constant-time sampling *i.e* cumulative distribution table (CDT) based sampling with linear search [6].

3.1 The minimization problem

The work in [15], however do not consider efficient minimization of Boolean expressions for their application. Indeed, the authors only used a simple and straightforward technique to generate the Boolean expressions. Hence, it is possible to improve the efficiency of the sampler by efficient minimization. Now to achieve this, we can use synthesis tools to minimize the expressions, since minimizing the Boolean expressions is equivalent to the well known circuit minimization problem which is a well known NP-complete problem. Now, as the number of variables n (usually ≈ 128) is large, these tools can only use heuristic minimization algorithms which can cause a number of complications. Firstly, these heuristics make the standard minimization approaches no longer applicable for our constant-time discrete Gaussian sampler because there is no guarantee for constant-time anymore, secondly minimization is not very efficient as the tools can use only heuristic algorithms, and finally most of these heuristic algorithms are intellectual property of their respective corporations and the output of these algorithms cannot be put in the public domain, rendering them unusable for discrete Gaussian samplers for lattice based cryptography implementations which are mostly open source. To overcome these problems we propose an alternate but efficient minimization strategy in the following sections.

4 OUR WORK

In this section, we first prove Theorem 1, which establishes an important property of the structure of the input random bit strings which generate the samples. We then describe our Boolean minimization strategy for discrete Gaussian sampling. We consider a Knuth-Yao sampler always takes input random bit strings of length n. However, it is possible that a sample is found on the c-th level of the DDG tree or equivalently the sampler needs only $c+1$ bits to generate the sample *i.e* we do not stop Alg. 1 when a sample is found continue without updating the sample value further. Hence, the remaining $n-(c+1)$ bits do not influence the outcome of the sampling, we call them *don't care*(x) bits. For instance, in Fig. 1 to hit the top most leaf with sample 1 or the leaf node with sample 0 (4-th from top) the random bit strings will be $xxxx00$ and $xxx001$ respectively.

Theorem 1. *All the random bit strings which generate samples are of the form $x^i(0/1)^j01^k$, where $i, j, k \in \mathbb{Z}^*$ and $i+j+k+1 = n$.*

Proof. It is enough to show that there is no random bit string of the form $\mathrm{x}^i 1^{k'}, k' \in \mathbb{Z}^*$ which generates a sample, *i.e* the sampling process does not hit any leaf node when the input random bit string is $1^{k'}$. We prove this by contradiction. Let us assume that the bit string $1^{k'}$ hits a leaf node at $k'-1$-th level of the DDG tree. So, $\mathrm{GAP}^{k'} < 0$ (see Eq. 1). Since the random bit string is $1^{k'}$ this implies that for all $2^{k'}$ different input random bit strings $\mathrm{GAP}^{k'} < 0$. In other words, all the random input bit strings generate some sample or hit some leaf nodes of the DDG tree. So, the DDG tree is a finite tree as the tree beyond $k'-1$-th level is not reachable by the Knuth-Yao sampling anymore and becomes redundant. Hence, by the property of Knuth-Yao sampling we can say that the DDG tree up to level $k'-1$ is able to generate samples *exactly* from the discrete Gaussian distribution of the given standard deviation. Or, in other words if the samples in the DDG tree up to $k'-1$-th level lie in the interval $[0, \tau'\sigma]$, then

$$\sum_{z=-\tau'\sigma}^{\tau'\sigma} \mathscr{D}_\sigma^n(z) = 1$$

But, this is impossible as discrete Gaussian distribution has infinite tail and the probabilities being real numbers have infinite precision, so the above summation will never be equal to 1. Hence, our assumption that the input bit string $x^i 1^{k'}, k' \in \mathbb{Z}^*$ generates a sample is therefore wrong. □

Also, experimentally we have seen that j is bounded by a *small* Δ i.e $j_{\max} \leq \Delta$. For example for $\sigma = 1, 2$, and 6.15543 the Δ is 4, 4, and 6 respectively. In the next sections, we will show that by using these two facts we can develop a very efficient minimization technique for a fast constant-time Gaussian sampler.

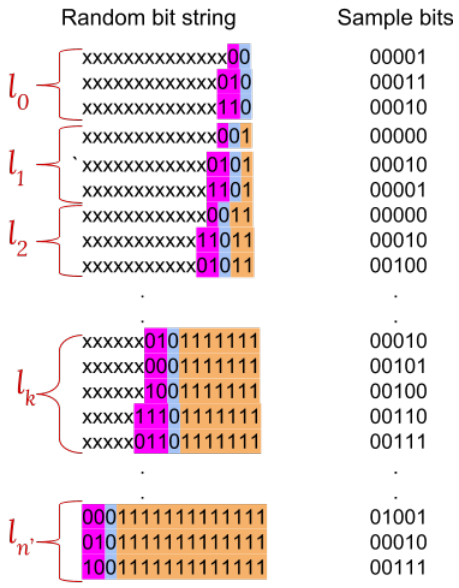

Figure 3: Dividing a List L in sublists $l_0, l_1, \cdots l_{n'}$ for $n = 16$ and $\sigma = 2$. The rightmost bit is the LSB in both the columns. During a random walk, the random bits are processed from the right side.

4.1 Efficient minimization

To instantiate an efficient discrete Gaussian sampler with a specific standard deviation, we first enumerate the number of leaves in the DDG tree and the random bit strings that hit those leaf nodes. We create a list L of these random bit strings $x^i(0/1)^j01^k$ with their corresponding binary decomposed sample values. It is evident from Sec. 2.2 that the size of this list is $\sum_{i=0}^{n-1} h_i$.

Then we sort the input random bit strings $x^i(0/1)^j01^k$ and their corresponding sample bits in the list L in ascending order of k. This makes all the input random bit strings with equal number of consecutive 1's from the right-most side become adjacent in the list L. This is shown in Fig. 3 for a discrete Gaussian sampler with $\sigma = 2$. In the next step, we divide the list L in n' sublists as $l_0, l_1, \cdots l_{n'}$ such that all the input random bit string in the sublist l_κ has $\kappa \in [0, n']$ consecutive 1's from the LSB or is of the form $x^i(0/1)^j01^\kappa$. As in the sublist l_κ, the $\kappa + 1$ least significant bits are fixed, the output sample bits in this sublist are determined by the next j random bits only. Now, we recall that $j_{\max} \leq \Delta$. Hence, we can generate Boolean functions f_Δ^ι, that maps Δ input random bits to the output sample bits for each sublist. Since, Δ is *small*, we can now use very efficient minimization techniques to minimize the Boolean expressions f_Δ^ι. It is even practically feasible to use Karnaugh map or brute force techniques to minimize the expressions. In this work we used the open source tool `Espresso` with `-Dso -S1` options for *exact* minimization of each expression. We generate these Boolean expressions $f_\Delta^{\iota,\kappa}, \kappa \in [0, n']$ for all the sublists $l_0, l_1, \cdots l_{n'}$ and for all $\iota \in [0, m-1]$. In the next section, we will discuss how we can join these Boolean expressions together to create a constant-time discrete Gaussian sampler.

4.2 Constant-time sampling

We first recall a method to execute a non constant-time *if-else* block $v = \alpha\ ?\ \beta_0 : \beta_1$ in constant-time as $v = (\alpha\&\beta_0)|(\bar{\alpha}\&\beta_1)$ where α, β_i are binary variables. It is easy to extend this to a long *if-elseif-...-else* block as $v = (\alpha_0\&\beta_0)|(\bar{\alpha}_0\&((\alpha_1\&\beta_1)|(\bar{\alpha}_1\&(\ldots|(\bar{\alpha}_n\&\beta_{n+1})))))$. Such method for constant-time execution of *if-else* blocks has beeen known since as early as constant-time bit sliced implementation of DES [4] and their constant-time behaviour has been studied well in the literature.

Now, consider the binary variable $c_\kappa = \bar{b}_\kappa\&b_{\kappa-1}\ldots\&b_0$ where the b_i's are the input random bits to the Gaussian sampler. Also, recall from the previous section that the input random bits of sublist l_κ are of the form $x^i(0/1)^j01^\kappa$. In Fig. 3, $b_{\kappa-1}\ldots b_0$ represent the κ-length chain of '1's and b_κ represent the first '0' (in blue) following the chain of 1s. Now, we make the following claim.

Claim 1. *c_κ equals 1 if and only if the random bit string belongs to the sublist l_κ.*

The above claim can be proven easily by using the structure of input random bits of list l_κ and the definition of the variable c_κ. We leave this to the reader. Using, Claim 1 and the constant-time *if-elseif-...-else* blocks as discussed in the beginning of this section we can combine the Boolean expressions $f_\Delta^{\iota,\kappa}, \kappa \in [0, n']$ to create a constant time Gaussian sampler with precision n as shown in Eq. 2,

$$f_n^\iota = c_0\ ?\ f_\Delta^{\iota,0} : (c_1\ ?\ f_\Delta^{\iota,1} : (\ldots : (c_{n'-1}\ ?\ f_\Delta^{\iota,n'-1} : f_\Delta^{\iota,n'}))) \quad (2)$$

For correctness of the construction of f_n^ι in Eq. 2, it can be easily verified that the variable c_κ ensures that if an input random bit string is in l_κ the output of f_n^ι becomes equal to the output of $f_\Delta^{\iota,\kappa}$. So far, we have used only binary variables $\alpha, \beta, c_\kappa, b$ etc., to describe our methods. All of these methods can be trivially transformed into the bit-sliced SIMD setting as described in Sec.3.2 of [15] by using wider variables instead of binary variables and replacing single bit Boolean operators to their bit wise counterparts. We assert that our construction of the Boolean expression f_n^ι runs in constant-time as long as each of the Boolean expressions $f_\Delta^{\iota,t}$ runs in constant-time. Now, each of $f_\Delta^{\iota,t}$ is a Boolean expression that computes the ι-th bit of a sample from Δ random bits. Constant-time behaviour of such functions has been proven and analyzed rigorously in [15]. Moreover, we used the tool `"dudect"` described in [23] to affirm the constant running time of our algorithm. The whole process for efficient minimization of f_n^ι is shown as a flowchart in Fig. 4. We will make all the source codes and a tool that implements the strategies mentioned here publicly available.[1] Fig. 5 shows histogram plots constant-time discrete Gaussian sampling of $\sigma = 2$ and 6.15543 using the methods described above.

5 RESULTS AND APPLICATION TO FALCON SIGNATURES

In this section, we provide performance results of our constant-time discrete Gaussian sampler. Since the lattice based signature scheme Falcon [14] uses discrete Gaussian samples during signing, we chose this scheme as our target application. We give below

[1] Tool and the code available at https://github.com/Angshumank/const_gauss_split

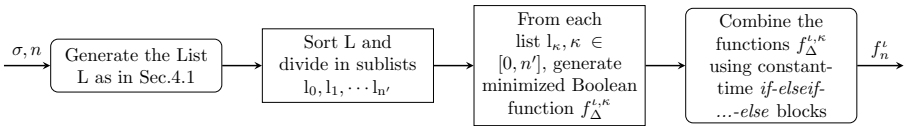

Figure 4: Flowchart for efficient minimization of Boolean expressions f^ι for constant-time discrete Gaussian sampling.

Security level	Non constant-time (Signs/Sec)		Constant-time (Signs/Sec)	
	Byte-scanning CDT	CDT	Linear search CDT	This Work
Level 1 (N=256)	10327	8041	6080	7025
Level 2 (N=512	5220	4064	3027	3527
Level 3 (N=1024)	2640	2014	1519	1754

Table 1: Comparing performances of Falcon-sign with non constant-time and constant-time samplers with `ChaCha` as the pseudo random number generator. N is a security parameter which refers to the degree of quotient polynomial used to define the number field used in Falcon.

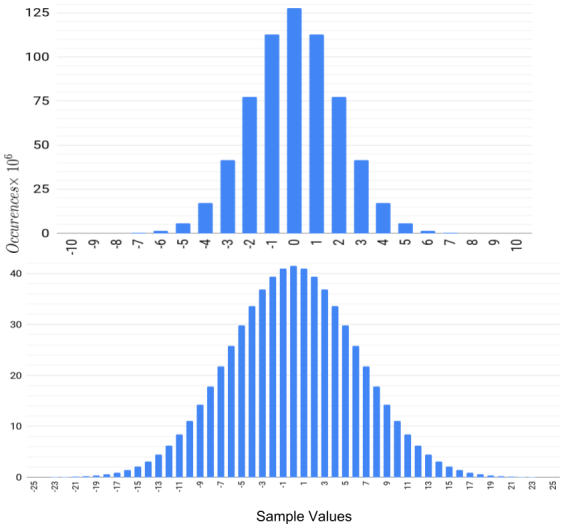

Figure 5: Histogram plot for $\sigma = 2$ and $\sigma = 6.15543$ using 64×10^7 samples.

a small description of the Falcon signature and the necessity of constant-time Gaussian sampling in the scheme.

Falcon signature scheme : The post-quantum signature scheme Falcon [14] has been submitted to NIST's [1] ongoing effort to standardize post-quantum protocols. Among the lattice based signature schemes, Falcon has the smallest combined bitsize of public key and signature, this is partly due to the choice of Gaussian distribution and NTRU lattices. During the signing of messages Falcon requires samples from a discrete Gaussian sampler with σ either 2 or $\sqrt{5}$ depending on the number field used. In our work, we only considered the instance of Falcon with $\sigma = 2$, the other instance can be realized using the same methods described above. For more details about the Falcon signature scheme we refer the interested readers to the detailed documentation [14].

The Falcon signature scheme is a relatively newer scheme and robustness against side channel attacks of this scheme has not been analyzed rigorously like older lattice based signature schemes [11]. Though at this moment we are not aware of any attacks like [7, 20] on Falcon which exploits a non constant-time Gaussian sampler, but the authors of Falcon have specifically mentioned that the use of a non constant-time Gaussian sampler during signing can be potentially harmful to the security of the scheme and serious effort should be dedicated for implementing constant-time Gaussian sampler.

For comparison, we plugged in our constant-time Gaussian sampler in the implementation [14] of Falcon provided by the authors and compare our results with two of their fastest non constant-time samplers, CDT sampler [19] and the byte-scanning CDT sampler [10]. Additionally, we also provided a comparison with the linear search based constant-time CDT sampler [6] with our sampler. The precision $n(= 128)$, the tail-cut $\tau(= 13)$ and the pseudo-random number generator have been kept same for all the instances. We compiled all implementations using `gcc-5.4` with flags `-O3 -fomit-frame-pointer -march=native -std=c99`. The computation times are measured on a single core of a Intel(R) Core(TM) i7-6600U processor running at 2.60GHz and disabling hyper-threading, Turbo-Boost, and multi-core support as standard practice on Ubuntu 16.04 running on a Dell Latitude E7470 laptop. As our target processor has 64-bits our sampler can generate 64 samples in a batch. The results are shown in Table. 1

We can see from Table. 1 that replacing the fastest non constant-time sampler with our constant-time sampler reduces the performance of signing algorithm by approximately 33% at worst, whereas replacing the constant-time CDT sampler with our sampler slows down the signing algorithm by at most 13%. Also, the signing algorithm with our sampler is at least 15% faster than the linear search

	Constant-time sampler in [15]	This work	Improvement
$\sigma = 2$	3,787	2,293	37%
$\sigma = 6.15543$	11,136	9,880	11%

Table 2: Comparing discrete Gaussian sampler with our efficient minimization with the previous described in [15]. The numbers in the table are in clockcycles and do not include the overhead for generating the pseudorandom numbers.

based CDT sampling. It is very important to note that the table based methods such as byte-scanning CDT, CDT or linear search CDT also gets advantage of data caching in our target Intel processors for fast table search due to the small size of the tables. Overall, these results show that removing side-channel vulnerabilities of non constant-time discrete Gaussian sampler with our constant-time Gaussian sampler does not hamper the performance of signing algorithm to a large extent.

In Table 2, we compare the performance of the Gaussian sampler with our efficient minimization technique and the Gaussian sampler with the one described in [15]. We can see that for $\sigma = 2$ we get around 37% improvement, but the improvement for $\sigma = 6.15543$ is approximately 11%, the reason behind this is that for $\sigma = 6.15543$ in [15], the output of the minimization tool has been manually optimized further for better efficiency.

6 DISCUSSION

In this work, we provided a generic tool to generate efficient discrete Gaussian samplers. We also demonstrated the efficiency of our sampler using two examples. Similar to previous work, we have also seen that there a large percentage ($\approx 60\%$) of total time is spent on generating the pseudorandom bits alone. One of the reasons the centered binomial distribution [2] sampler and the byte-scanning CDT sampler are efficient is that they need very few pseudorandom bits to generate a sample. Our experiments suggest that improvements in the direction of reducing the requirement of pseudorandom bits per sample using better statistical measures like Rényi [21] divergences or max-log [18] distances and efficient generation of these pseudorandom bits combined with the methods described here can make the discrete Gaussian sampling very competitive.

7 ACKNOWLEDGEMENTS

This work was supported in part by the Research Council KU Leuven: C16/15/058, European Commission through the Horizon 2020 research and innovation programme under grant agreement Cathedral ERC Advanced Grant 695305 and by EU H2020 project FENTEC (Grant No. 780108) and by the Hercules Foundation AKUL/11/19.

REFERENCES

[1] 2017. NIST Post-Quantum cryptography Round 1 submissions. (2017). https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions [Online; accessed 12-April-2018].

[2] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. 2016. Post-quantum key exchange – a new hope. In *USENIX Security 2016*.

[3] Paulo S. L. M. Barreto, Patrick Longa, Michael Naehrig, Jefferson E. Ricardini, and Gustavo Zanon. 2016. Sharper Ring-LWE Signatures. Cryptology ePrint Archive, Report 2016/1026. (2016). https://eprint.iacr.org/2016/1026.

[4] Eli Biham. 1997. A fast new DES implementation in software. In *Fast Software Encryption*, Eli Biham (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 260–272.

[5] Joppe Bos, Léo Ducas, Eike Kiltz, Tancréde Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, and Damien Stehlé. 2017. CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM. Cryptology ePrint Archive, Report 2017/634. (2017). http://eprint.iacr.org/2017/634.

[6] J. W. Bos, C. Costello, M. Naehrig, and D. Stebila. 2015. Post-Quantum Key Exchange for the TLS Protocol from the Ring Learning with Errors Problem. In *2015 IEEE Symposium on Security and Privacy*. 553–570. https://doi.org/10.1109/SP.2015.40

[7] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. 2016. Flush, Gauss, and Reload – A Cache Attack on the BLISS Lattice-Based Signature Scheme. Cryptology ePrint Archive, Report 2016/300. (2016). https://eprint.iacr.org/2016/300.

[8] Jung Hee Cheon, Duhyeong Kim, Joohee Lee, and Yongsoo Song. 2016. Lizard: Cut off the Tail! Practical Post-Quantum Public-Key Encryption from LWE and LWR. Cryptology ePrint Archive, Report 2016/1126. (2016). http://eprint.iacr.org/2016/1126.

[9] Jan-Pieter DÁnvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. 2018. Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. Cryptology ePrint Archive, Report 2018/230. (2018). https://eprint.iacr.org/2018/230.

[10] Chaohui Du and Guoqiang Bai. 2015. Towards efficient discrete Gaussian sampling for lattice-based cryptography. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. 1–6. https://doi.org/10.1109/FPL.2015.7293949

[11] Léo Ducas, Alain Durmus, Tancréde Lepoint, and Vadim Lyubashevsky. 2013. Lattice Signatures and Bimodal Gaussians. Cryptology ePrint Archive, Report 2013/383. (2013). https://eprint.iacr.org/2013/383.

[12] Leo Ducas, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehle. 2017. CRYSTALS – Dilithium: Digital Signatures from Module Lattices. Cryptology ePrint Archive, Report 2017/633. (2017). https://eprint.iacr.org/2017/633.

[13] Nagarjun C. Dwarakanath and Steven D. Galbraith. 2014. Sampling from discrete Gaussians for lattice-based cryptography on a constrained device. *Applicable Algebra in Engineering, Communication and Computing* 25, 3 (2014), 159–180. https://doi.org/10.1007/s00200-014-0218-3

[14] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. 2018. Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU. (2018). https://falcon-sign.info/ [Online; accessed 10-October-2018].

[15] A. Karmakar, S. S. Roy, O. Reparaz, F. Vercauteren, and I. Verbauwhede. 2018. Constant-Time Discrete Gaussian Sampling. *IEEE Trans. Comput.* 67, 11 (Nov 2018), 1561–1571. https://doi.org/10.1109/TC.2018.2814587

[16] D. Knuth and A. Yao. 1976. *Algorithms and Complexity: New Directions and Recent Results*. Academic Press, Chapter The complexity of nonuniform random number generation.

[17] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2012. On Ideal Lattices and Learning with Errors Over Rings. Cryptology ePrint Archive, Report 2012/230. (2012). https://eprint.iacr.org/2012/230.

[18] Daniele Micciancio and Michael Walter. 2017. Gaussian Sampling over the Integers: Efficient, Generic, Constant-Time. Cryptology ePrint Archive, Report 2017/259. (2017). https://eprint.iacr.org/2017/259.

[19] Chris Peikert. 2010. An Efficient and Parallel Gaussian Sampler for Lattices. In *Advances in Cryptology – CRYPTO 2010: 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, Tal Rabin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 80–97. https://doi.org/10.1007/978-3-642-14623-7_5

[20] Peter Pessl. 2017. Analyzing the Shuffling Side-Channel Countermeasure for Lattice-Based Signatures. Cryptology ePrint Archive, Report 2017/033. (2017). https://eprint.iacr.org/2017/033.

[21] Thomas Pöppelmann, Léo Ducas, and Tim Güneysu. 2014. Enhanced Lattice-Based Signatures on Reconfigurable Hardware. Cryptology ePrint Archive, Report 2014/254. (2014). https://eprint.iacr.org/2014/254.

[22] Oded Regev. 2004. New Lattice-based Cryptographic Constructions. Vol. 51. ACM, New York, NY, USA, 899–942. https://doi.org/10.1145/1039488.1039490

[23] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. 2016. Dude, is my code constant time? Cryptology ePrint Archive, Report 2016/1123. (2016). https://eprint.iacr.org/2016/1123.

[24] Sujoy Sinha Roy, Oscar Reparaz, Frederik Vercauteren, and Ingrid Verbauwhede. 2014. Compact and Side Channel Resistant Discrete Gaussian Sampling. Cryptology ePrint Archive, Report 2014/591. (2014). https://eprint.iacr.org/2014/591.pdf.

[25] Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. 2014. High Precision Discrete Gaussian Sampling on FPGAs. In *Revised Selected Papers on Selected Areas in Cryptography – SAC 2013 - Volume 8282*. Springer-Verlag New York, Inc., 383–401.