Sampling from Discrete Distributions in Combinational Hardware with Application to Post-Quantum Cryptography

Michael X. Lyons and Kris Gaj
Cryptographic Engineering Research Group
George Mason University
Fairfax, Virginia, U.S.A.
{mlyons3, kgaj}@gmu.edu

***Abstract*—Random values from discrete distributions are typically generated from uniformly-random samples. A common technique is to use a cumulative distribution table (CDT) lookup for inversion sampling, but it is also possible to use Boolean functions to map a uniformly-random bit sequence into a value from a discrete distribution. This work presents a methodology for deriving such functions for any discrete distribution, encoding them in VHDL for implementation in combinational hardware, and (for moderate precision and sample space size) confirming the correctness of the produced distribution. The process is demonstrated using a discrete Gaussian distribution with a small sample space, but it is applicable to any discrete distribution with fixed parameters. Results are presented for sampling schemes from several submissions to the NIST PQC standardization process, comparing this method to CDT lookups on a Xilinx Artix-7 FPGA. The process produces compact solutions for distributions up to moderate size and precision.**

***Index Terms*—Boolean functions, centered binomial, combinational logic, constant time, DDG-tree, discrete Gaussian, FPGA, logic minimization**

I. Introduction

Random values available in hardware and software are typically distributed uniformly, but many applications require random values from other distributions. For example, lattice-based post-quantum cryptography (PQC) schemes often use random samples drawn from discrete Gaussian distributions, which maximize entropy for fixed parameters, but on-the-fly calculations for them require high-precision calculations of exponentials. A common solution is to use a CDT lookup for inversion sampling, but it is also possible to use Boolean functions to map a uniformly-random bit sequence into a value from a discrete distribution.

This work presents a methodology for sampling from any discrete distribution in combinational hardware. The process is described in detail for a discrete Gaussian distribution with a small sample space, and results are reported for selected key encapsulation mechanisms (KEMs) and signature schemes submitted to Round 2 of the NIST PQC standardization process. This method produces viable solutions that require fewer resources in the target FPGA than typical CDT lookups when the number of table entries is moderate.

II. Background

A. Definitions

When logic is described as being implemented in *software* this means in the form of instructions executed on a general purpose CPU or embedded processor; in *hardware* means in the form of logic elements in devices that are reconfigurable (e.g. FPGAs) or custom designed (e.g. ASICs).

Logic whose output depends only on the current inputs is called *combinational*; if the output also depends on the current state, it is called *sequential* as its output is dependent on the sequence of input states. Combinational circuits may be used as components in larger circuit assemblies.

An implementation is described as being *constant-time* if it always takes a fixed number of clock cycles to produce a valid output sample, and *time-independent* if the number of cycles required may vary but is not related to the value of a valid sample produced. In this context, combinational logic is constant-time as the output is available in the same clock cycle when inputs change.

B. Sampling Methods

Random bits available in software or hardware are typically uniformly distributed. Several methods are available to transform sequences of such bits into values from a different type of distribution (a process which we call *redistribution*), including rejection sampling, Karney's method, Knuth-Yao sampling, and inversion sampling. This work uses a technique derived from Knuth-Yao sampling, and implementations of it are compared to a typical CDT-based inversion sampling method.

III. Previous Work

Knuth and Yao [1] showed that any discrete distribution can be modeled as a binary tree, with a path from the root to a terminal node representing the probability of the associated sample value. A path can be viewed as the sum of selected negative powers of 2 represented by each level in the tree. Sampling of values according to the distribution can be performed by a "random walk", with a uniform random bit

directing the choice at each branch, until a terminal node is reached.

Dwarakanath and Galbraith [2] applied the Knuth-Yao algorithm to discrete distributions with the probabilities expressed as binary fractions to some fixed precision. They demonstrated the process for a discrete Gaussian distribution, noting that it is necessary to sample only from the non-negative portion[1] with the probability of 0 halved, and use an additional uniform bit to select the sign of the result.

Karmakar, Roy, Reparaz, Vercauteren, and Verbauwhede [3] observed that redistribution based on a Knuth-Yao *discrete distribution generating tree* (DDG-tree) is a unique mapping[2] between the bits of the input sequence and those of the output sample; each output sample bit can be produced by a Boolean function of the input bit sequence. They used bit-slicing to calculate multiple samples in parallel in software, and in [4] they describe a method to evaluate the Boolean functions in constant time in software.

The inherently parallel nature of hardware makes it an obvious candidate platform for calculating these output samples. This work presents a methodology for determining the bit-mapping functions for a particular discrete distribution, minimizing the required logic, generating VHDL code to evaluate the functions in hardware, and confirming that the distribution of the output sample values is correct.

IV. Completeness of Discrete Distributions

We call a representation of a discrete distribution *complete* if the probabilities of the values in its sample space sum exactly to 1, and *incomplete* otherwise. If a distribution is complete then every input bit sequence maps to a valid output sample value; if it is incomplete then some input bit sequences must be mapped to an invalid output value/s, and sequential logic at a higher level will be needed to repeat the process until a valid value is produced; such higher-level logic is not constant-time, but is time-independent as the number of iterations required is independent of the valid sample value eventually produced.

To demonstrate the methodology in this work, a very narrow discrete Gaussian distribution is used; to the selected precision it is incomplete, and thus requires handling of the possibility that an output value is invalid. It has mean $\sigma = 1.5$, and probabilities are represented with 5-bit precision. For values outside the range $[-3, 3]$ the probabilities in this distribution are 0 when truncated to 5 bits; this is equivalent to selecting a *tail cut* of 2σ. For each value in the tails the probability expressed to the chosen precision is 0, but those probabilities cannot be ignored because the sum of the expressed probabilities is less than 1, making this distribution incomplete. We assign the shortfall 0.001 as the probability of generating an invalid value, and map it to a value outside the implemented sample space.

[1]This is sometimes called a "half-Guassian".

[2]The mapping for a particular DDG-tree is unique; many equivalent trees can be constructed for a given distribution by varying the branching and the ordering of terminal node values across a level.

TABLE I
Implemented Distribution

Variable		Probability	
binary	*decimal*	*binary*	*decimal*
000	0	0.0100	0.2500
001	1	0.0110	0.3750
010	2	0.0011	0.1875
011	3	0.0001	0.0625
100	4	0.0010	0.1250
	sum	**1.0000**	**1.0000**

Table I shows the distribution to be implemented, with positive probabilities doubled and the invalid probability assigned to the first value outside the sample space. The most-significant bit (MSB) of the input bit sequence is used to determine whether the result should be negated. A set of discrete distribution probabilities in this form is sometimes called a *probability matrix*[3].

V. Implementation in Hardware

Our methodology begins with a process described by Dwarakanath and Galbraith [2], which in turn adopts the process described by Knuth and Yao [1] to derive a *discrete distribution generating tree* (DDG-tree) from the probability matrix.

A. DDG-tree

A DDG-tree contains *branch nodes* (each of which has two subordinate[4] nodes) and *terminal nodes*[5] (each of which has an associated output sample value). A given output sample value will be associated with one terminal node for each 1 bit in its probability as a binary fraction.

Fig. 1 shows the DDG-tree for the implemented distribution. This work follows the convention from [3], where 0 is assigned

[3]This is not the same type of data structure as a *stochastic matrix*, which sometimes is called a probability matrix (among other names).

[4]Knuth and Yao call them *son* nodes, and also *descendants*.

[5]This work uses the terminology introduced by Knuth and Yao. Dwarakanath and Galbraith call these *internal vertices* and *leaves*, respectively. Karmakar et al. call them *intermediate positions* and *leaf nodes*, respectively.

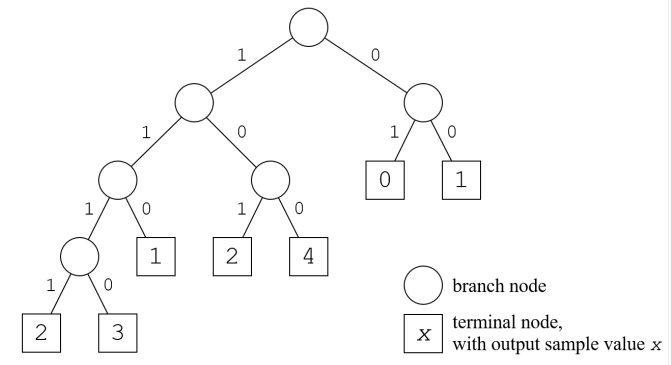

Fig. 1. Discrete distribution generating tree (DDG-tree).

TABLE II
COMBINATIONAL LOGIC AFTER MINIMIZATION

Output sample bit	Input bit string(s)
$o_2 = 1$	100x
$o_1 = 1$	1x1x
$o_0 = 1$	00xx, 110x, 1110

to the right-hand side of a branch, and terminal node values at a level are assigned in ascending order left-to-right.

B. Bit strings

The DDG-tree provides one or more paths to each output sample value, with the probability of each value according to our modified distribution (assuming the probability of taking one side of a branch is). For each terminal node an associated bit string is derived by concatenating the bits in the path leading to that node.

C. Mapping and Logic minimization

The bit strings derived from the DDG-tree are mapped to the bit positions in the output sample value. The bit string(s) for the 0 output sample value are not mapped; that value will be generated when the input bit string does not match any of the mapped expressions.

To minimize the area and the critical path delay of a hardware implementation, it is necessary to minimize the logic in the Boolean functions. For this work, a custom Python script was developed to read a probability matrix, construct a DDG-tree, derive Boolean functions, minimize the functions using the Espresso heuristic tool[6], and generate VHDL code for the minimized functions. Table II shows the bit strings for the example after logic minimization; ''x" indicates a "don't care" condition in a bit position.

D. VHDL code generation

A custom Python script was created to generate combinational VHDL code from the minimized Boolean functions, represented as bit strings as described above.

The example distribution is incomplete, so the interface includes a single-bit flag to indicate the validity of the output value; this simplifies higher-level logic by avoiding a numeric comparison to test for the special invalid value.

Multiple sets of output can be generated by instantiating more than one unit of this code (assuming uniformly-random input bits can be supplied at that rate).

E. Confirmation of the distribution

To confirm that the output sample values are distributed correctly a custom VHDL testbench was created to generate every possible input bit sequence and record the input and output values[7]. A spreadsheet used to analyze the recorded data showed that the number of occurrences of each value in the output range (including the invalid value) was exactly as expected. Table III shows the results for the example distribution.

[6]Espresso was also used by Karmakar et al. in [3]. In this work we used the Espresso implementation from the PyEDA package.

[7]This process is feasible for small-to-moderate precision and sample space size, e.g. for input sequences up to 16 bits in length.

VI. IMPLEMENTATION OF SELECTED PQC SCHEMES

Howe, Khalid, Rafferty, Regazzoni, and O'Neill [5] determined that a variation on Peikert's CDT method [6] provided the best balance of throughput and low area among the independent-time sampling methods considered, under the constraint that no RAM units are used. They compared their method to that of Pöppelmann and T. Güneysu [7], which is constant-time (producing one sample per clock cycle) but requires about 5x the area and operates at about the speed.

The focus of this work is sampling from reasonably narrow discrete distributions, using only combinational hardware, and requiring neither block RAMs nor DSP units. Such hardware can be incorporated into more complex circuit assemblies, e.g. to produce multiple samples in parallel by using multiple instances of the sampling component. In order to compare implementations of our method against a typical CDT lookup we chose the error distributions from the FrodoKEM submission to the second round of the NIST post-quantum cryptography (PQC) selection process [8]. FrodoKEM "approximates a rounded continuous Gaussian distribution" and is complete, so there is no need to handle out-of-range values. The specification lists integer values for the probabilities (scaled by 2^{16}) of output samples in the range [0, 12], where the probabilities for negative values in the range are the same as for their positive counterparts.

Bernstein [9] identified all the key encapsulation mechanisms (KEMs) submitted to Round 2 of the NIST PQC standardization process which are lattice-based and target indistinguishability under adaptive chosen ciphertext attack (IND-CCA2). In Table 8.7 he shows the distribution of "short elements" (polynomial coefficients, vector elements, or matrix elements) for each. To evaluate the effectiveness of our methodology, we applied it to all the KEMs which use a non-uniform distribution. Of these only FrodoKEM uses a discrete Gaussian distribution, so the methodology is also applied to three signature schemes from two other submissions to Round 2, each of which samples from a discrete Gaussian with a much larger sample space and with much higher precision than the FrodoKEM variants.

Each distribution tested was implemented in Xilinx Vivado 2017.4.1 (64-bit), with default options for synthesis and for

TABLE III
DISTRIBUTION COUNTS FROM THE TESTBENCH

Output sample value (*decimal*)	Frequency	Probability (*binary*)
0	8	0.0100
–1, 1	6, 6	0.0110
–2, 2	3, 3	0.0011
–3, 3	1, 1	0.0001
–4 (invalid)	4	0.0010
	sum = 32	**sum = 1.0000**

TABLE IV
IMPLEMENTATION RESULTS FOR XILINX ARTIX-7 FPGA

Scheme	Variant	Sample space	Precision *(input size)*	Output size	Architecture	LUTs	MUXes	Slices	Depth
CRYSTALS-KYBER			4	3	Boolean functions	2		2	1
NewHope			8	4	Boolean functions			5	2
Saber	LightSaber		10	4	Boolean functions	16		4	3
	Saber		8	4	Boolean functions	5		2	2
	FireSaber		6	3	Boolean functions	2		1	1
LAC	LAC-128		2	2	Boolean functions	1		1	1
	LAC-192		3	2	Boolean functions	1		1	1
	LAC-256		2	2	Boolean functions	1		1	1
ThreeBears	BabyBear		7	3	Boolean functions	3		1	2
	MamaBear		6	2	Boolean functions	2		1	1
	PapaBear		5	2	Boolean functions	1		1	1
FrodoKEM	Frodo-640	[-12, 12]	16	5	Boolean functions	30		9	4
					CDT lookup	35		10	5
	Frodo-976	[-10, 10]	16	5	Boolean functions	36	3	12	5
					CDT lookup	39		12	4
	Frodo-1344	[-6, 6]	16	4	Boolean functions	24		7	4
					CDT lookup	22		7	4
Falcon		[-19, 19]	72	5	Boolean functions	590		168	8
					CDT lookup	655		168	13
qTESLA	qTESLA_p_I	[-78, 78]	63	7	Boolean functions	2316		609	7
					CDT lookup	1019		273	7
qTESLA	qTESLA_I	[-208, 208]	63	8	Boolean functions	6775		1757	4
					CDT lookup	2365	1	623	5

implementation, and the target device xc7a15tcpg236-3[8]. For each scheme sampling from a discrete Gaussian distributions a CDT equivalent also was coded in VHDL, along with a testbench to exercise both implementations. We do not claim that the CDT implementations are optimal, but given the small tables used in the FrodoKEM variants (with 12, 10, and 6 rows, respectively,) it seems likely that techniques like those in [5] would provide only limited improvement. The signature schemes tested (Falcon, qTESLA) had much larger tables and much higher precision, so specialized CDT implementations would likely be more compact than the generic implementations used here.

The KEM schemes using centered binomial distributions (CBDs) or similar all had smaller sample spaces, ranging from $\{$-1, 1$\}$ for LAC and two of the Three Bears to $\{$-5, 5$\}$ for LightSaber. For the narrowest of these, our method produced correct results with very few logic units, but it would be easy to produce comparable results using VHDL selected signal assignment (`with ... select`) statements or indexed lookups of tables of constants.

Table IV shows the results for implementations of the selected PQC algorithms, in terms of *lookup table units* (LUTs), *multiplexers* (MUXes), and *slices* used on the target device, and the *circuit depth* - the maximum number of LUTs and/or MUXes between any input bit and any output bit, which gives a good indication of the relative latency. Output size is the number of bits needed to represent the largest value in the sample space.

[8]The target device was arbitrarily chosen – it is the smallest member of the Xilinx Artix-7 family that has the highest speed grade "3". Artix-7 is the hardware family NIST has asked developers to focus on in its PQC standardization process.

VII. CONCLUSION

The methodology presented in this work produces correct combinational hardware implementations for sampling from arbitrary discrete distributions. For distributions with small-to-moderate precision and sample space size, these implementations are comparable to and in some cases better than typical CDT lookups.

REFERENCES

[1] D. E. Knuth and A. Yao, "The complexity of nonuniform random number generation," in *Algorithms and Complexity: New Direction and Recent Results*, J. Traub, Ed. New York, NY: Academic, 1976, pp. 357–428.

[2] N. C. Dwarakanath and S. D. Galbraith, "Sampling from discrete Gaussians for lattice-based cryptography on a constrained device," *Applicable Algebra in Engineering, Communication and Computing*, vol. 25, no. 3, pp. 159–180, Jun. 2014.

[3] A. Karmakar, S. S. Roy, O. Reparaz, F. Vercauteren, and I. Verbauwhede, "Constant-Time Discrete Gaussian Sampling," *IEEE Transactions on Computers*, vol. 67, no. 11, pp. 1561–1571, Nov. 2018.

[4] A. Karmakar, S. S. Roy, F. Vercauteren, and I. Verbauwhede, "Pushing the speed limit of constant-time discrete Gaussian sampling. A case study on the Falcon signature scheme," in *56th Annual Design Automation Conference 2019, DAC 2019*. Las Vegas, NV, USA: ACM Press, 2019, pp. 1–6.

[5] J. Howe, A. Khalid, C. Rafferty, F. Regazzoni, and M. O'Neill, "On Practical Discrete Gaussian Samplers for Lattice-Based Cryptography," *IEEE Transactions on Computers*, vol. 67, no. 3, pp. 322–334, Mar. 2018.

[6] C. Peikert, "An Efficient and Parallel Gaussian Sampler for Lattices," in *Advances in Cryptology – CRYPTO 2010*, vol. 6223. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 80–97.

[7] T. Pöppelmann and T. Guneysu, "Towards Practical Lattice-Based Public-Key Encryption on Reconfigurable Hardware," in *Selected Areas in Cryptography – SAC 2013*, vol. 8282. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 68–85.

[8] "Post-Quantum Cryptography: Round 2 Submissions," https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions, Apr. 2019.

[9] D. J. Bernstein, "Comparing proofs of security for lattice-based encryption," Cryptology ePrint Archive 2019/691, Jul. 2019.