By Jacob Alperin-Sheriff

# Discrete Gaussian Sampling-Techniques and Dangers

04/21/2017

# Why is Discrete Gaussian Sampling Necessary?

- For key exchange-it's not!

# Why is Discrete Gaussian Sampling Necessary?

- For key exchange-it's not!
- Can replace by centered binomial distribution $\psi_k$ (New Hope etc).
- Sampleable with $2k$ uniform bits $b_i, b'_i$:

$$Y \leftarrow \sum_{i=0}^{k} (b_i - b'_i)$$

# Why is Discrete Gaussian Sampling Necessary?

- For key exchange-it's not!
- Can replace by centered binomial distribution $\psi_k$ (New Hope etc).
- Sampleable with $2k$ uniform bits $b_i, b_i'$:

$$Y \leftarrow \sum_{i=0}^{k}(b_i - b_i')$$

- Close enough for LWE - small number of samples

# Why is Discrete Gaussian Sampling Necessary?

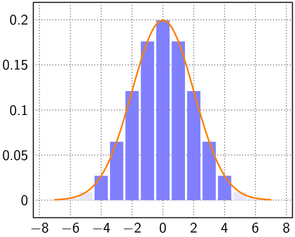
- For key exchange-it's not!
- Can replace by centered binomial distribution $\psi_k$ (New Hope etc).
- Sampleable with $2k$ uniform bits $b_i, b'_i$:

$$Y \leftarrow \sum_{i=0}^{k} (b_i - b'_i)$$

- Close enough for LWE - small number of samples
- For (SIS-based) signatures - large number of samples per instance
- Can't just approximate

# Discrete Gaussian Distribution

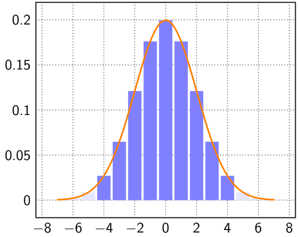
- Discrete Gaussian $D_{\mathbb{Z},\sigma}$ for $\sigma = 2$



# Discrete Gaussian Distribution

- Discrete Gaussian $D_{\mathbb{Z},\sigma}$ for $\sigma = 2$
- Each point in $\mathbb{Z}$ chosen with probability proportional to

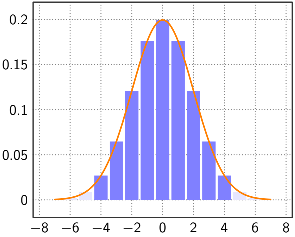
$$\rho_\sigma(x) = \exp(-x^2/2)$$



# Discrete Gaussian Distribution

- Discrete Gaussian $D_{\mathbb{Z},\sigma}$ for $\sigma = 2$
- Each point in $\mathbb{Z}$ chosen with probability proportional to

$$\rho_\sigma(x) = \exp(-x^2/2)$$

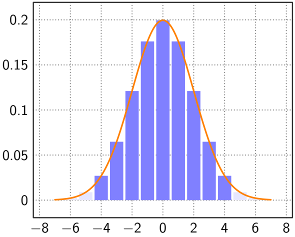



- Discrete Gaussians maintain many properties of normal distribution

# Discrete Gaussian Distribution

- Discrete Gaussian $D_{\mathbb{Z},\sigma}$ for $\sigma = 2$
- Each point in $\mathbb{Z}$ chosen with probability proportional to

$$\rho_\sigma(x) = \exp(-x^2/2)$$



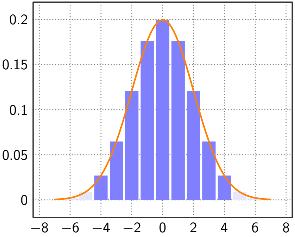

- Discrete Gaussians maintain many properties of normal distribution
- Sums of discrete Gaussians are still discrete Gaussians,

$$\sigma = \sqrt{\sigma_x^2 + \sigma_y^2}$$

# Discrete Gaussian Distribution

- Discrete Gaussian $D_{\mathbb{Z},\sigma}$ for $\sigma = 2$
- Each point in $\mathbb{Z}$ chosen with probability proportional to

$$\rho_\sigma(x) = \exp(-x^2/2)$$




- Discrete Gaussians maintain many properties of normal distribution
- Sums of discrete Gaussians are still discrete Gaussians,

$$\sigma = \sqrt{\sigma_x^2 + \sigma_y^2}$$

- Actual sampling: ignore the (very unlikely) points outside $[-\tau\sigma, \tau\sigma]$

# “Basic” SIS-based Signature Scheme [L’12]

- Public key: uniform $\mathbf{A}$, $\mathbf{T} := \mathbf{A}\mathbf{S}$ for short secret key $\mathbf{S}$

# “Basic” SIS-based Signature Scheme [L’12]

- Public key: uniform $\mathbf{A}$, $\mathbf{T} := \mathbf{A}\mathbf{S}$ for short secret key $\mathbf{S}$
- Cryptographic hash function H hashing input to short vectors

# "Basic" SIS-based Signature Scheme [L'12]

- Public key: uniform $\mathbf{A}$, $\mathbf{T} := \mathbf{AS}$ for short secret key $\mathbf{S}$
- Cryptographic hash function H hashing input to short vectors
- **Sign**$(\mu)$:
  1. Sample $\mathbf{y} \leftarrow D_{\mathbb{Z}^n,\sigma}$.
  2. Hash $\mathbf{c} \leftarrow \mathsf{H}(\mathbf{Ay}, \mu)$.
  3. Apply rejection sampling to $\mathbf{z} := \mathbf{Sc} + \mathbf{y}$
  4. Output $(\mathbf{z}, \mathbf{c})$ as signature.

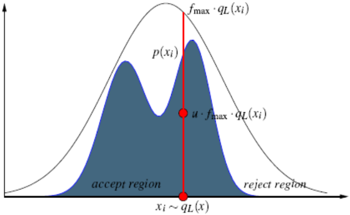
# “Basic” SIS-based Signature Scheme [L’12]

- Public key: uniform $\mathbf{A}$, $\mathbf{T} := \mathbf{A}\mathbf{S}$ for short secret key $\mathbf{S}$
- Cryptographic hash function H hashing input to short vectors
- **Sign**$(\mu)$:
  1. Sample $\mathbf{y} \leftarrow D_{\mathbb{Z}^n,\sigma}$.
  2. Hash $\mathbf{c} \leftarrow \mathsf{H}(\mathbf{A}\mathbf{y}, \mu)$.
  3. Apply rejection sampling to $\mathbf{z} := \mathbf{S}\mathbf{c} + \mathbf{y}$
  4. Output $(\mathbf{z}, \mathbf{c})$ as signature.
- **Verify**$((\mathbf{z}, \mathbf{c}), \mu)$:
  1. Verify that $\mathbf{z}$ is sufficiently short (under Euclidean norm)
  2. Verify that $\mathsf{H}(\mathbf{A}\mathbf{z} - \mathbf{T}\mathbf{c}, \mu) = \mathbf{c}$

# "Basic" SIS-based Signature Scheme [L'12]

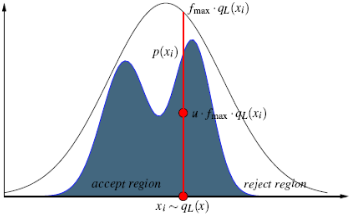
- Public key: uniform $\mathbf{A}$, $\mathbf{T} := \mathbf{A}\mathbf{S}$ for short secret key $\mathbf{S}$
- Cryptographic hash function H hashing input to short vectors
- **Sign**$(\mu)$:
  1. Sample $\mathbf{y} \leftarrow D_{\mathbb{Z}^n,\sigma}$.
  2. Hash $\mathbf{c} \leftarrow \mathsf{H}(\mathbf{A}\mathbf{y}, \mu)$.
  3. Apply rejection sampling to $\mathbf{z} := \mathbf{S}\mathbf{c} + \mathbf{y}$
  4. Output $(\mathbf{z}, \mathbf{c})$ as signature.
- **Verify**$((\mathbf{z}, \mathbf{c}), \mu)$:
  1. Verify that $\mathbf{z}$ is sufficiently short (under Euclidean norm)
  2. Verify that $\mathsf{H}(\mathbf{A}\mathbf{z} - \mathbf{T}\mathbf{c}, \mu) = \mathbf{c}$
- Key Step: rejection sampling – hides $\mathbf{S}$ contribution to signature.

# Rejection Sampling




- Standard general technique (due to von Neumann) to sample $f(x)$ given access to easily sampleable $g(x)$

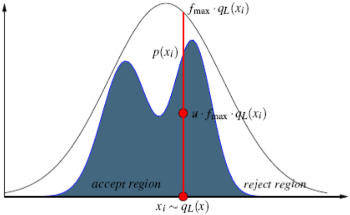
# Rejection Sampling




- Standard general technique (due to von Neumann) to sample $f(x)$ given access to easily sampleable $g(x)$

1. Sample $Y \leftarrow g$

# Rejection Sampling



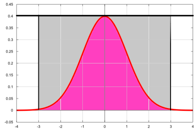

- Standard general technique (due to von Neumann) to sample $f(x)$ given access to easily sampleable $g(x)$

1. Sample $Y \leftarrow g$
2. Accept $Y$ with probability $\min(f(Y)/(Mg(Y), 1)$.
   - Need $f(x) \leq Mg(x)$ (except with negligible probability over $x$)

# Rejection Sampling for Discrete Gaussian Distributions

- For param $\sigma$, sample probabilities must be proportional to

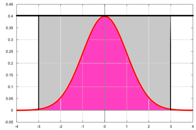
$$\rho_\sigma(x) = \exp(-x^2/(2\sigma^2))$$



# Rejection Sampling for Discrete Gaussian Distributions

- For param $\sigma$, sample probabilities must be proportional to

$$\rho_\sigma(x) = \exp(-x^2/(2\sigma^2))$$

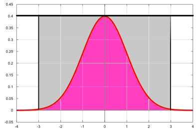


1. Sample $Y \leftarrow [-\tau\sigma, \tau\sigma]$ uniformly.

# Rejection Sampling for Discrete Gaussian Distributions

- For param $\sigma$, sample probabilities must be proportional to

$$\rho_\sigma(x) = \exp(-x^2/(2\sigma^2))$$

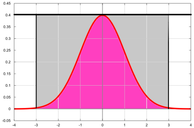


1. Sample $Y \leftarrow [-\tau\sigma, \tau\sigma]$ uniformly.
2. Accept with probability $\rho_\sigma(Y)/\rho_{\sigma(\mathbb{Z})}$, otherwise resample.

# Rejection Sampling for Discrete Gaussian Distributions

- For param $\sigma$, sample probabilities must be proportional to

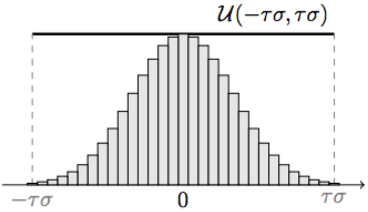
$$\rho_\sigma(x) = \exp(-x^2/(2\sigma^2))$$



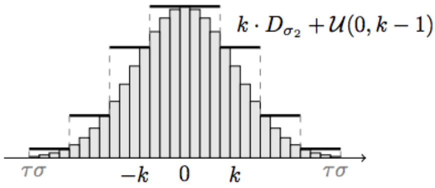
1. Sample $Y \leftarrow [-\tau\sigma, \tau\sigma]$ uniformly.
2. Accept with probability $\rho_\sigma(Y)/\rho_{\sigma(\mathbb{Z})}$, otherwise resample.

- Problems:
  - High rejection rate
  - Computing $\rho_\sigma$ to high precision is expensive

# Bernoulli Rejection Sampling [DDLL' 12]




(a) from uniform distribution (repetition rate $\approx 10$)




(b) from our adapted distribution (repetition rate $\approx 1.47$)

- "Core sampler" of $D_{\sigma_2}^{+}$ where $\sigma_2 = \sqrt{1/(2\ln(2))}$.
  - ⋆ $\rho_{\sigma_2}(x) = 2^{-x^2}, x \in \mathbb{Z}$
  - ⋆ In DDLL'12, binary-style rejection sampler given access to uniform bits.

# Bernoulli Rejection-Core Sampler

## Sampling $D^{+}_{\sigma_2}$

Draw random bit $b$.
**if** random bit $b = 0$ **then** return $0$
**for** $i = 1$ to $\infty$ **do**
  Draw random bits $b_1, \ldots, b_k$ for $k = 2i - 1$
  **if** $b_1 \ldots, b_{k-1} \neq 0 \ldots 0$ **then** restart
  **if** $b_k = 0$ **then** return $i$

# Bernoulli Rejection-Core Sampler

**Sampling $D^+_{\sigma_2}$**

Draw random bit $b$.
**if** random bit $b = 0$ **then** return $0$
**for** $i = 1$ to $\infty$ **do**
  Draw random bits $b_1, \ldots, b_k$ for $k = 2i - 1$
  **if** $b_1 \ldots, b_{k-1} \neq 0 \ldots 0$ **then** restart
  **if** $b_k = 0$ **then** return $i$

- Why it works: binary expansion of $\rho_{\sigma_2}(\{0, \ldots, j\})$ is

$$\rho_{\sigma_2}(0, \ldots, j) = \sum_{i=0}^{j} 2^{-i^2} = 1.100100001\underbrace{0\ldots0}_{6}1\ldots\underbrace{0\ldots0}_{2(j-1)}1$$

# Bernoulli Rejection (Full Algorithm)

**Sampling $D^+_{k\sigma_2}$, $k \in \mathbb{Z}$**

Sample $x \leftarrow D^+_{\sigma_2}$.
Sample $y \leftarrow \{0, \ldots, k-1\}$.
Let $z \leftarrow kx + y$.
Sample $b$ with probability $\exp(-y(y + 2kx)/(2(k\sigma_2)^2))$
**if** $b = 0$ **then** restart.
return $z$.

# Bernoulli Rejection (Full Algorithm)

**Sampling $D^{+}_{k\sigma_2}$, $k \in \mathbb{Z}$**

Sample $x \leftarrow D^{+}_{\sigma_2}$.
Sample $y \leftarrow \{0, \ldots, k-1\}$.
Let $z \leftarrow kx + y$.
Sample $b$ with probability $\exp(-y(y+2kx)/(2(k\sigma_2)^2))$
**if** $b = 0$ **then** restart.
return $z$.

- Sampling the exponential distribution can be done efficiently

# Bernoulli Rejection (Full Algorithm)

**Sampling $D^+_{k\sigma_2}$, $k \in \mathbb{Z}$**

Sample $x \leftarrow D^+_{\sigma_2}$.
Sample $y \leftarrow \{0, \ldots, k-1\}$.
Let $z \leftarrow kx + y$.
Sample $b$ with probability $\exp(-y(y + 2kx)/(2(k\sigma_2)^2))$
**if** $b = 0$ **then** restart.
return $z$.

- Sampling the exponential distribution can be done efficiently
  - Takes time $O(\log k)$.
  - Needs small lookup table with

$$\mathsf{ET}[i] := \exp(-2^i/(2(k\sigma_2)^2)), i \in [0, O(\log k)]$$

# Bernoulli Rejection-Timing Attacks

**Sampling $D^+_{\sigma_2}$**

Draw random bit $b$.
**if** random bit $b = 0$ **then** return $0$
**for** $i = 1$ to $\infty$ **do**
  Draw random bits $b_1, \dots, b_k$ for $k = 2i - 1$
  **if** $b_1 \dots, b_{k-1} \neq 0 \dots 0$ **then** restart
  **if** $b_k = 0$ **then** return $i$

- Problem-Information Revealed by Timing Attacks!

# Bernoulli Rejection-Timing Attacks

**Sampling $D^+_{\sigma_2}$**

Draw random bit $b$.
**if** random bit $b = 0$ **then** return $0$
**for** $i = 1$ to $\infty$ **do**
  Draw random bits $b_1, \ldots, b_k$ for $k = 2i - 1$
  **if** $b_1 \ldots, b_{k-1} \neq 0 \ldots 0$ **then** restart
  **if** $b_k = 0$ **then** return $i$

- Problem-Information Revealed by Timing Attacks!
- When **for** loop not entered, algorithm always outputs $0$

# Bernoulli Rejection-Timing Attacks

**Sampling $D^+_{\sigma_2}$**

Draw random bit $b$.
**if** random bit $b = 0$ **then** return $0$
**for** $i = 1$ to $\infty$ **do**
  Draw random bits $b_1, \ldots, b_k$ for $k = 2i - 1$
  **if** $b_1 \ldots, b_{k-1} \neq 0 \ldots 0$ **then** restart
  **if** $b_k = 0$ **then** return $i$

- Problem-Information Revealed by Timing Attacks!
- When **for** loop not entered, algorithm always outputs $0$
- Algorithm for $D^+_{\sigma_2}$ is slow in worst case.
- Can mitigate with batching

# CDT Sampling

- For each $y \in [-\tau\sigma, \tau\sigma]$, compute $\lambda$ bit precision

$$p_y := \Pr[x \leq y \mid x \leftarrow D_\sigma]$$

# CDT Sampling

- For each $y \in [-\tau\sigma, \tau\sigma]$, compute $\lambda$ bit precision

$$p_y := \Pr[x \leq y \mid x \leftarrow D_\sigma]$$

- Store in (large) table

# CDT Sampling

- For each $y \in [-\tau\sigma, \tau\sigma]$, compute $\lambda$ bit precision

$$p_y := \Pr[x \leq y \mid x \leftarrow D_\sigma]$$

- Store in (large) table
- To sample $D_\sigma$:
  - Sample (sufficient approximation of) uniform $r \in [0, 1)$

# CDT Sampling

- For each $y \in [-\tau\sigma, \tau\sigma]$, compute $\lambda$ bit precision

$$p_y := \Pr[x \leq y \mid x \leftarrow D_\sigma]$$

- Store in (large) table
- To sample $D_\sigma$:
  - ⋆ Sample (sufficient approximation of) uniform $r \in [0, 1)$
  - ⋆ Binary search to find $y \in [-\tau\sigma, \tau\sigma]$ such that $r \in [p_{y-1}, p_y)$.

# CDT Sampling

- For each $y \in [-\tau\sigma, \tau\sigma]$, compute $\lambda$ bit precision
$$p_y := \Pr[x \leq y \mid x \leftarrow D_\sigma]$$
- Store in (large) table
- To sample $D_\sigma$:
  - Sample (sufficient approximation of) uniform $r \in [0, 1)$
  - Binary search to find $y \in [-\tau\sigma, \tau\sigma]$ such that $r \in [p_{y-1}, p_y)$.
- Can be sped up with additional guide table

# CDT Sampling

- For each $y \in [-\tau\sigma, \tau\sigma]$, compute $\lambda$ bit precision

$$p_y := \Pr[x \leq y \mid x \leftarrow D_\sigma]$$

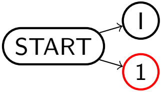
- Store in (large) table
- To sample $D_\sigma$:
  - Sample (sufficient approximation of) uniform $r \in [0, 1)$
  - Binary search to find $y \in [-\tau\sigma, \tau\sigma]$ such that $r \in [p_{y-1}, p_y)$.
- Can be sped up with additional guide table
- Problems: Table is quite large; infeasible for constrained devices.

# Knuth-Yao Sampling

START

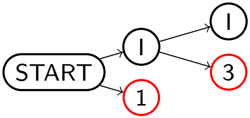
| **Val** | **Prob (binary)** |
|---|---|
| 1 | 0.10010 |
| 2 | 0.00011 |
| 3 | 0.01011 |

# Knuth-Yao Sampling



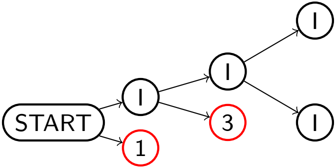

| Val | Prob (binary) |
|---|---|
| 1 | 0.**1**0010 |
| 2 | 0.**0**0011 |
| 3 | 0.**0**1011 |

# Knuth-Yao Sampling



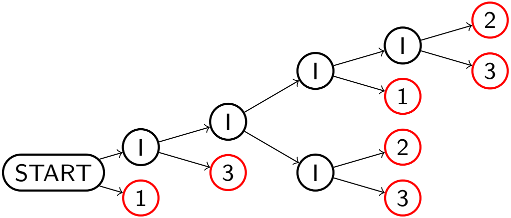

| Val | Prob (binary) |
|---|---|
| 1 | 0.1**0**010 |
| 2 | 0.0**0**011 |
| 3 | 0.0**1**011 |

# Knuth-Yao Sampling



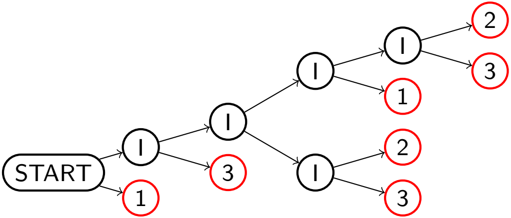

| Val | Prob (binary) |
|---|---|
| 1 | 0.10**0**10 |
| 2 | 0.00**0**11 |
| 3 | 0.01**0**11 |

# Knuth-Yao Sampling




| Val | Prob (binary) |
|---|---|
| 1 | 0.10010 |
| 2 | 0.00011 |
| 3 | 0.01011 |

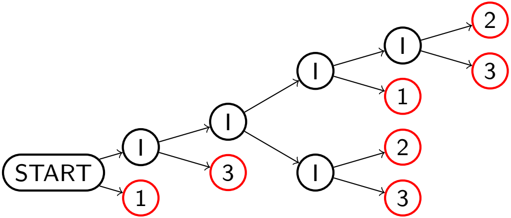
# Knuth-Yao Sampling




| Val | Prob (binary) |
|---|---|
| 1 | 0.10010 |
| 2 | 0.00011 |
| 3 | 0.01011 |

- Designed to minimize (average) number of bits required to sample

# Knuth-Yao Sampling




| Val | Prob (binary) |
|---|---|
| 1 | 0.10010 |
| 2 | 0.00011 |
| 3 | 0.01011 |

- Designed to minimize (average) number of bits required to sample
- Theorem: Knuth-Yao requires at most 2 more than entropy of dist.

# Knuth-Yao Sampling (Gaussians) [Dwarakanath/Galbraith]

| x | $\sigma_{10}(x)$ | Binary expansion of $\sigma_{10}(x)$ |
|---|---|---|
| 0 | 0.01994711 | 0.00000101000110110100 |
| $\pm 1$ | 0.03969525 | 0.00001010001010010111 |
| $\pm 10$ | 0.02419707 | 0.00000110001100011100 |
| $\pm 20$ | 0.00539909 | 0.00000001011000011101 |
| $\pm 30$ | 0.00044318 | 0.00000000000111010001 |
| $\pm 40$ | 0.00001338 | 0.00000000000000001110 |

- Need to store table of probabilities, which is large.

# Knuth-Yao Sampling (Gaussians) [Dwarakanath/Galbraith]

| x | $\sigma_{10}(x)$ | Binary expansion of $\sigma_{10}(x)$ |
|---|---|---|
| 0 | 0.01994711 | 0.00000101000110110100 |
| $\pm 1$ | 0.03969525 | 0.00001010001010010111 |
| $\pm 10$ | 0.02419707 | 0.00000110001100011100 |
| $\pm 20$ | 0.00539909 | 0.00000001011000011101 |
| $\pm 30$ | 0.00044318 | 0.00000000000111010001 |
| $\pm 40$ | 0.00001338 | 0.00000000000000001110 |

- Need to store table of probabilities, which is large.
- Can cut down by performing Knuth-Yao in “blocks”

# Knuth-Yao Sampling (Gaussians) [Dwarakanath/Galbraith]

| x | $\sigma_{10}(x)$ | Binary expansion of $\sigma_{10}(x)$ |
|---|---|---|
| 0 | 0.01994711 | 0.00000101000110110100 |
| $\pm 1$ | 0.03969525 | 0.00001010001010010111 |
| $\pm 10$ | 0.02419707 | 0.00000110001100011100 |
| $\pm 20$ | 0.00539909 | 0.00000001011000011101 |
| $\pm 30$ | 0.00044318 | 0.00000000000111010001 |
| $\pm 40$ | 0.00001338 | 0.00000000000000001110 |

- Need to store table of probabilities, which is large.
- Can cut down by performing Knuth-Yao in "blocks"
  1. Partition into disjoint sets with almost equivalent probabilities

# Knuth-Yao Sampling (Gaussians) [Dwarakanath/Galbraith]

| x | $\sigma_{10}(x)$ | Binary expansion of $\sigma_{10}(x)$ |
|---|---|---|
| 0 | 0.01994711 | 0.00000101000110110100 |
| $\pm 1$ | 0.03969525 | 0.00001010001010010111 |
| $\pm 10$ | 0.02419707 | 0.00000110001100011100 |
| $\pm 20$ | 0.00539909 | 0.00000001011000011101 |
| $\pm 30$ | 0.00044318 | 0.00000000000111010001 |
| $\pm 40$ | 0.00001338 | 0.00000000000000001110 |

- Need to store table of probabilities, which is large.
- Can cut down by performing Knuth-Yao in "blocks"
  1. Partition into disjoint sets with almost equivalent probabilities
  2. Pick a set

# Knuth-Yao Sampling (Gaussians) [Dwarakanath/Galbraith]

| x | $\sigma_{10}(x)$ | Binary expansion of $\sigma_{10}(x)$ |
|---|---|---|
| 0 | 0.01994711 | 0.00000101000110110100 |
| $\pm 1$ | 0.03969525 | 0.00001010001010010111 |
| $\pm 10$ | 0.02419707 | 0.00000110001100011100 |
| $\pm 20$ | 0.00539909 | 0.00000001011000011101 |
| $\pm 30$ | 0.00044318 | 0.00000000000111010001 |
| $\pm 40$ | 0.00001338 | 0.00000000000000001110 |

- Need to store table of probabilities, which is large.
- Can cut down by performing Knuth-Yao in "blocks"
  1. Partition into disjoint sets with almost equivalent probabilities
  2. Pick a set
  3. Perform Knuth-Yao within the set

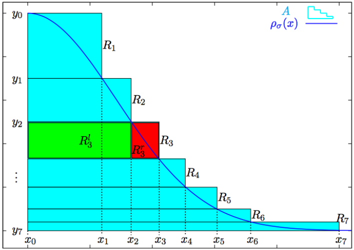
# Knuth-Yao Sampling (Gaussians) [Dwarakanath/Galbraith]

| x | $\sigma_{10}(x)$ | Binary expansion of $\sigma_{10}(x)$ |
|---|---|---|
| 0 | 0.01994711 | 0.00000101000110110100 |
| $\pm 1$ | 0.03969525 | 0.00001010001010010111 |
| $\pm 10$ | 0.02419707 | 0.00000110001100011100 |
| $\pm 20$ | 0.00539909 | 0.00000001011000011101 |
| $\pm 30$ | 0.00044318 | 0.00000000000111010001 |
| $\pm 40$ | 0.00001338 | 0.00000000000000001110 |

- Need to store table of probabilities, which is large.
- Can cut down by performing Knuth-Yao in "blocks"
  1. Partition into disjoint sets with almost equivalent probabilities
  2. Pick a set
  3. Perform Knuth-Yao within the set
- Knuth-Yao is not constant time!
- Can be mitigated by batching

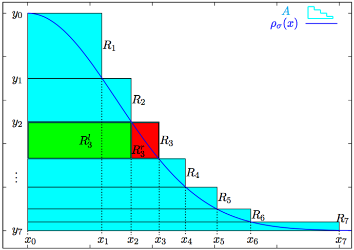
# Discrete Ziggurat Sampling

1. Partition density function into $m$ rectangles of equal probability



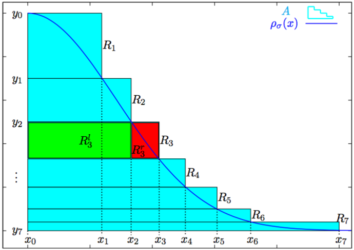
# Discrete Ziggurat Sampling

1. Partition density function into $m$ rectangles of equal probability
2. Choose a rectangle unif. at random



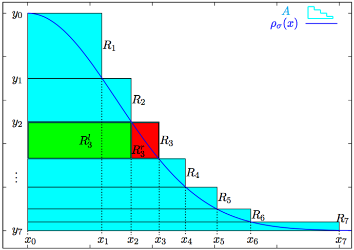
# Discrete Ziggurat Sampling

1. Partition density function into $m$ rectangles of equal probability
2. Choose a rectangle unif. at random
3. Choose point $x' \leftarrow [0, x_i]$



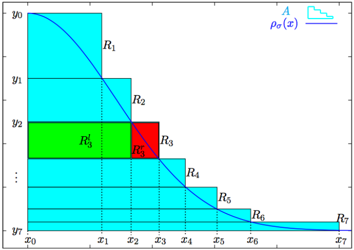
# Discrete Ziggurat Sampling

1. Partition density function into $m$ rectangles of equal probability
2. Choose a rectangle unif. at random
3. Choose point $x' \leftarrow [0, x_i]$
   1. If $x' \leq x_{i-1}$, accept.
   2. Otherwise, do rejection sampling.



# Discrete Ziggurat Sampling

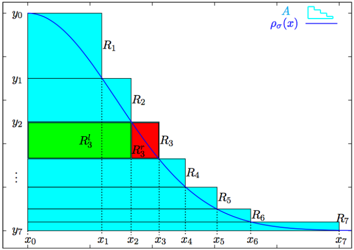
1. Partition density function into $m$ rectangles of equal probability
2. Choose a rectangle unif. at random
3. Choose point $x' \leftarrow [0, x_i]$
   1. If $x' \leq x_{i-1}$, accept.
   2. Otherwise, do rejection sampling.




- Sampling in discrete case requires some care

# Discrete Ziggurat Sampling

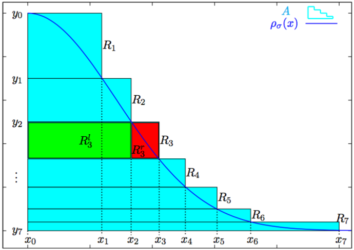
1. Partition density function into $m$ rectangles of equal probability
2. Choose a rectangle unif. at random
3. Choose point $x' \leftarrow [0, x_i]$
   1. If $x' \leq x_{i-1}$, accept.
   2. Otherwise, do rejection sampling.




- Sampling in discrete case requires some care
  - ⋆ Partitioning can't be done by "area", but by probability

# Discrete Ziggurat Sampling

1. Partition density function into $m$ rectangles of equal probability
2. Choose a rectangle unif. at random
3. Choose point $x' \leftarrow [0, x_i]$
    1. If $x' \leq x_{i-1}$, accept.
    2. Otherwise, do rejection sampling.




- Sampling in discrete case requires some care
    - ⋆ Partitioning can't be done by "area", but by probability
- No clear vulnerability to timing attacks.

# Batching

- Technique to make algorithm (e.g. Knuth-Yao) constant time

# Batching

- Technique to make algorithm (e.g. Knuth-Yao) constant time
- Signatures need large (linear) number of Gaussian samples anyway

# Batching

- Technique to make algorithm (e.g. Knuth-Yao) constant time
- Signatures need large (linear) number of Gaussian samples anyway
- Use Hoeffding-type bounds
  - Each sample takes on average $c$ random bits, max of $n$ WHP
  - All $n$ samples take combined time $cn$ on average
  - With overwhelming prob, all $n$ samples take at most $cn + n$ time.

# Batching

- Technique to make algorithm (e.g. Knuth-Yao) constant time
- Signatures need large (linear) number of Gaussian samples anyway
- Use Hoeffding-type bounds
  - Each sample takes on average $c$ random bits, max of $n$ WHP
  - All $n$ samples take combined time $cn$ on average
  - With overwhelming prob, all $n$ samples take at most $cn + n$ time.
- Have algorithm run in "time" proportional to $cn + n$ being used.

# Batching

- Technique to make algorithm (e.g. Knuth-Yao) constant time
- Signatures need large (linear) number of Gaussian samples anyway
- Use Hoeffding-type bounds
  - ⋆ Each sample takes on average $c$ random bits, max of $n$ WHP
  - ⋆ All $n$ samples take combined time $cn$ on average
  - ⋆ With overwhelming prob, all $n$ samples take at most $cn + n$ time.
- Have algorithm run in "time" proportional to $cn + n$ being used.
- Pitfall: Making sure implementation is constant time is extremely hard

# Walter-Micciancio Sampling

$\textsc{SampleZ}_{b,k,\max}(c, \sigma)$
  $x \leftarrow \textsc{SampleI}(\max)$
  $K \leftarrow \sqrt{\sigma^2 - \bar{\sigma}^2}/\sigma_{\max}$
  $c' \leftarrow \lfloor c + Kx \rceil_k$
  $y \leftarrow \textsc{SampleC}_{b,\sigma_0}(c')$
  **return** $y$

$\textsc{SampleI}(i)$
  **if** $i = 0$
    $x \leftarrow \textsc{SampleB}_{\sigma_0}(0)$
    **return** $x$
  $x_1 \leftarrow \textsc{SampleI}(i-1)$
  $x_2 \leftarrow \textsc{SampleI}(i-1)$
  $y = z_i x_1 + \max(1, z_i - 1) x_2$
  **return** $y$

$\textsc{SampleC}_b(c \in b^{-k}\mathbb{Z})$
  **if** $k = 0$
    **return** 0
  $g \leftarrow b^{-k+1}\textsc{SampleB}_{\sigma_0}(b^{k-1}c)$
  **return** $g + \textsc{SampleC}_b(c - g \in b^{-k+1}\mathbb{Z})$

Algorithm 1: A sampling algorithm for $\mathcal{D}_{\mathbb{Z}+c,\sigma}$ for arbitrary $c$ and $\sigma$. Definitions for $z_i$ and $\sigma_i$ as in (3) and (4) and $\bar{\sigma}$ as in (5). SampleB is an arbitrary base sampler for fixed $\sigma_0$ and small number of cosets $c + \mathbb{Z}$, where $c \in \mathbb{Z}/b$.

$$z_i = \lfloor \sigma_{i-1}/\sqrt{2\eta_\epsilon(\mathbb{Z})} \rfloor, \sigma_i^2 = (z_i^2 + \max((z_i - 1)^2, 1))\sigma_{i-1}^2$$

- New algorithm with constant-time online phase

# Walter-Micciancio Sampling

$\textsc{SampleZ}_{b,k,\max}(c,\sigma)$
- $x \leftarrow \textsc{SampleI}(\max)$
- $K \leftarrow \sqrt{\sigma^2 - \bar{\sigma}^2}/\sigma_{\max}$
- $c' \leftarrow \lfloor c + Kx \rceil_k$
- $y \leftarrow \textsc{SampleC}_{b,\sigma_0}(c')$
- **return** $y$

$\textsc{SampleI}(i)$
- **if** $i = 0$
  - $x \leftarrow \textsc{SampleB}_{\sigma_0}(0)$
  - **return** $x$
- $x_1 \leftarrow \textsc{SampleI}(i-1)$
- $x_2 \leftarrow \textsc{SampleI}(i-1)$
- $y = z_i x_1 + \max(1, z_i - 1)x_2$
- **return** $y$

$\textsc{SampleC}_b(c \in b^{-k}\mathbb{Z})$
- **if** $k = 0$
  - **return** $0$
- $g \leftarrow b^{-k+1}\textsc{SampleB}_{\sigma_0}(b^{k-1}c)$
- **return** $g + \textsc{SampleC}_b(c - g \in b^{-k+1}\mathbb{Z})$

Algorithm 1: A sampling algorithm for $\mathcal{D}_{\mathbb{Z}+c,\sigma}$ for arbitrary $c$ and $\sigma$. Definitions for $z_i$ and $\sigma_i$ as in (3) and (4) and $\bar{\sigma}$ as in (5). SampleB is an arbitrary base sampler for fixed $\sigma_0$ and small number of cosets $c + \mathbb{Z}$, where $c \in \mathbb{Z}/b$.

$$z_i = \lfloor \sigma_{i-1}/\sqrt{2\eta_\epsilon(\mathbb{Z})} \rfloor, \sigma_i^2 = (z_i^2 + \max((z_i - 1)^2, 1))\sigma_{i-1}^2$$

- New algorithm with constant-time online phase
- Works by recursively combining samples with smaller $\sigma$s.

# Walter-Micciancio Sampling

```
SampleZ_{b,k,max}(c, σ)
  x ← SampleI(max)
  K ← √(σ² − σ̄²)/σ_max
  c′ ← ⌊c + Kx⌉_k
  y ← SampleC_{b,σ0}(c′)
  return y

SampleI(i)
  if i = 0
    x ← SampleB_{σ0}(0)
    return x
  x1 ← SampleI(i − 1)
  x2 ← SampleI(i − 1)
  y = z_i x1 + max(1, z_i − 1) x2
  return y

SampleC_b(c ∈ b^{−k}ℤ)
  if k = 0
    return 0
  g ← b^{−k+1} SampleB_{σ0}(b^{k−1} c)
  return g + SampleC_b(c − g ∈ b^{−k+1}ℤ)
```

Algorithm 1: A sampling algorithm for $\mathcal{D}_{\mathbb{Z}+c,\sigma}$ for arbitrary $c$ and $\sigma$. Definitions for $z_i$ and $\sigma_i$ as in (3) and (4) and $\bar{\sigma}$ as in (5). SampleB is an arbitrary base sampler for fixed $\sigma_0$ and small number of cosets $c+\mathbb{Z}$, where $c \in \mathbb{Z}/b$.

$$z_i = \lfloor \sigma_{i-1}/\sqrt{2\eta_\epsilon(\mathbb{Z})} \rfloor, \sigma_i^2 = (z_i^2 + \max((z_i-1)^2, 1))\sigma_{i-1}^2$$

- New algorithm with constant-time online phase
- Works by recursively combining samples with smaller $\sigma$s.
- Assumes access to $D_{\mathbb{Z}+c,\sigma_0}$with small $\sigma_0 \geq \sqrt{2}\eta_\epsilon(\mathbb{Z})$

# Walter-Micciancio Sampling

$\textsc{SampleZ}_{b,k,\max}(c,\sigma)$
- $x \leftarrow \textsc{SampleI}(\max)$
- $K \leftarrow \sqrt{\sigma^2 - \bar{\sigma}^2}/\sigma_{\max}$
- $c' \leftarrow \lfloor c + Kx \rceil_k$
- $y \leftarrow \textsc{SampleC}_{b,\sigma_0}(c')$
- **return** $y$

$\textsc{SampleI}(i)$
- **if** $i = 0$
  - $x \leftarrow \textsc{SampleB}_{\sigma_0}(0)$
  - **return** $x$
- $x_1 \leftarrow \textsc{SampleI}(i-1)$
- $x_2 \leftarrow \textsc{SampleI}(i-1)$
- $y = z_i x_1 + \max(1, z_i - 1) x_2$
- **return** $y$

$\textsc{SampleC}_b(c \in b^{-k}\mathbb{Z})$
- **if** $k = 0$
  - **return** $0$
- $g \leftarrow b^{-k+1}\textsc{SampleB}_{\sigma_0}(b^{k-1}c)$
- **return** $g + \textsc{SampleC}_b(c - g \in b^{-k+1}\mathbb{Z})$

Algorithm 1: A sampling algorithm for $\mathcal{D}_{\mathbb{Z}+c,\sigma}$ for arbitrary $c$ and $\sigma$. Definitions for $z_i$ and $\sigma_i$ as in (3) and (4) and $\bar{\sigma}$ as in (5). SampleB is an arbitrary base sampler for fixed $\sigma_0$ and small number of cosets $c + \mathbb{Z}$, where $c \in \mathbb{Z}/b$.

$$z_i = \lfloor \sigma_{i-1}/\sqrt{2\eta_\epsilon(\mathbb{Z})} \rfloor, \sigma_i^2 = (z_i^2 + \max((z_i-1)^2, 1))\sigma_{i-1}^2$$

- New algorithm with constant-time online phase
- Works by recursively combining samples with smaller $\sigma$s.
- Assumes access to $D_{\mathbb{Z}+c,\sigma_0}$ with small $\sigma_0 \geq \sqrt{2}\eta_\epsilon(\mathbb{Z})$
  - Authors suggest generating these offline in “idle times”
  - Doesn't seem plausible for constrained devices
  - Relies on idle time (frequent queries could eliminate it)

# Walter-Micciancio (Runtime)

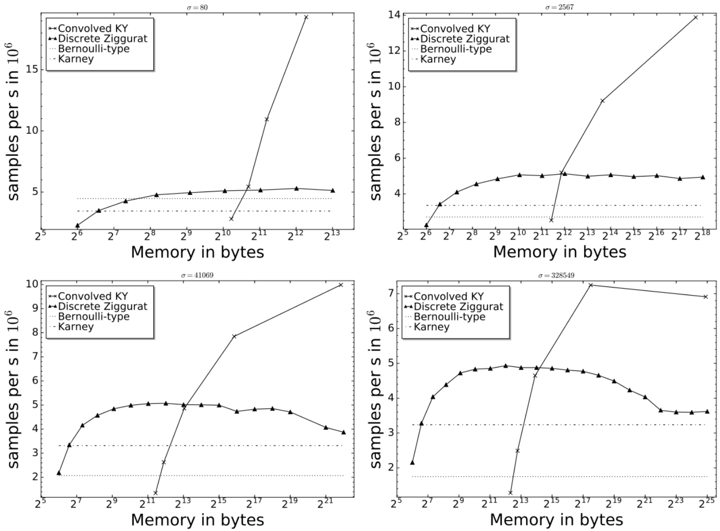



Figure 1: Time memory trade-off for combination sampler ("Convolved KY") and discrete Ziggurat compared to Bernoulli-type sampling for $\sigma \in \{2^5, 2^{10}, 2^{14}, 2^{17}\}\sqrt{2\pi}$. Knuth-Yao corresponds to right most point of "Convolved KY".

# Even Tables Are Vulnerable - To Cache Attacks

- Table access is constant-time. Or is it??

# Even Tables Are Vulnerable - To Cache Attacks

- Table access is constant-time. Or is it??
- FLUSH+RELOAD attack uses clflush instruction (on x86-64)

# Even Tables Are Vulnerable - To Cache Attacks

- Table access is constant-time. Or is it??
- FLUSH+RELOAD attack uses clflush instruction (on x86-64)
  1. Evicts memory block from cache,
  2. Lets victim execute
  3. Measures time to access same memory block

# Even Tables Are Vulnerable - To Cache Attacks

- Table access is constant-time. Or is it??
- FLUSH+RELOAD attack uses clflush instruction (on x86-64)
  1. Evicts memory block from cache,
  2. Lets victim execute
  3. Measures time to access same memory block
- Flush, Gauss and Reload uses this on Gaussian sampling in BLISS

# Even Tables Are Vulnerable - To Cache Attacks

- Table access is constant-time. Or is it??
- FLUSH+RELOAD attack uses clflush instruction (on x86-64)
  1. Evicts memory block from cache,
  2. Lets victim execute
  3. Measures time to access same memory block
- Flush, Gauss and Reload uses this on Gaussian sampling in BLISS
- Requires roughly 3000 signatures

# Even Tables Are Vulnerable - To Cache Attacks

- Table access is constant-time. Or is it??
- FLUSH+RELOAD attack uses clflush instruction (on x86-64)
  1. Evicts memory block from cache,
  2. Lets victim execute
  3. Measures time to access same memory block
- Flush, Gauss and Reload uses this on Gaussian sampling in BLISS
- Requires roughly 3000 signatures
- FLUSH+RELOAD must be run on same system as crypto

# Can We Avoid Discrete Gaussians?

- Seems like it should be easier for SIS-based cryptography

# Can We Avoid Discrete Gaussians?

- Seems like it should be easier for SIS-based cryptography
  - Unlike LWE, SIS problem is not defined with a noise distribution

# Can We Avoid Discrete Gaussians?

- Seems like it should be easier for SIS-based cryptography
  - ⋆ Unlike LWE, SIS problem is not defined with a noise distribution
  - ⋆ Just need to find short solution

# Can We Avoid Discrete Gaussians?

- Seems like it should be easier for SIS-based cryptography
  - Unlike LWE, SIS problem is not defined with a noise distribution
  - Just need to find short solution
- Discrete Gaussians do give tightest bounds, but how much tighter?

# Can We Avoid Discrete Gaussians?

- Seems like it should be easier for SIS-based cryptography
  - Unlike LWE, SIS problem is not defined with a noise distribution
  - Just need to find short solution
- Discrete Gaussians do give tightest bounds, but how much tighter?
- Would be nice to see concrete implementations without them