Constant-Time Discrete Gaussian Sampling

Mark Schultz

March 25, 2019

1 Introduction

In this paper we'll develop ideas related to *discrete gaussian sampling*. This is a primitive of fundamental importance in lattice cryptography, which has received (comparatively) less research interest than other relatively-advanced constructions (such as FHE or IBE). The current state-of-the-art for DGS involves a variety of sampling methods (which we will summarize quickly), that all fall into one of the following categories:

1. They require *large pre-computed tables* to be stored
2. They run intrinsically in *non-constant time*
3. They are a *purely theoretical* construct

Our goal will to be implementing various forms of DGS in "constant-time", which will be a valuable tool for the wider cryptographic community to use. We'll first quickly summarize why each of the above bullet-points is problematic.

1.1 Why not large tables?

One major use-case of cryptography is in embedded systems/systems with small amounts of processing power. An easy example of this is the Internet of Things. These machines often can't store large tables of precomputed values, which is required for techniques based on Cumulative Distribution Tables (CDT). Moreover, for schemes that require DGS with respect to multiple parameters σ_i, naive methods would require a separate table for each i, and the table size will be dependent on σ (say $O(\log \sigma)$).

Another argument against large-memory methods is that of hardware implementation. DGS is a rather attractive primitive for hardware implementation in consumer-grade machines (similarly to how AES has hardware implementations). This would require storing any pre-computed data in the processor itself, where storage space is at a premium.

1.2 Why constant-time?

Here, constant-time is a little misleading of a signifier. This is because the input of a sampling algorithm isn't the most clear notion. We can assume it's some random bit-string, but then "constant-time" in the size of the input ends up being some statement along the lines of "constant-time in the amount of entropy used", which isn't really the notion we're interested in.

What we are interested in is for the distribution of *outputs* of the sampler to be "mostly independent" of the running time of the sampler. We currently don't have a great way to formalize this, and searching the literature/thinking about it seems like a good decision. One notion that still seems promising is:

$$\Delta(D, D \mid T = t) \leq \epsilon \tag{1}$$

Where D is the distribution of our sampler, and $D \mid T = t$ is the conditional distribution, conditioned on precise knowledge of its running time. Note that a truly constant-time sampler would clearly satisfy this, but non-constant time samplers that satisfy this should still be acceptable.

Note that this doesn't handle all side-channel attacks related to the running time. A CCS 2017 paper discussed running Linux's perf command to get additional information about sampling (such as the input to sub-routines, or which branches are taken in conditional statements). A broader model that encompasses these attacks may be interesting, but likely isn't strictly necessary. This is especially the case for embedded systems, where perf can likely be made admin-only with little cost.

1.3 Why discuss implementation details?

Often cryptographers give a theoretical construct, and leave implementation details to others. We aren't suggesting new theoretical constructs — why not leave the implementations to others? This is mostly due to the existence of good theory which appears to be ignored so far. Recent papers regarding constant-time DGS appear to appeal to various "minimization black-boxes", which require various heuristic assumptions to make any claims about good performance.

2 A Survey of Samplers

Initially, we'll restrict to a subset of known samplers which seem especially suitable for constant-time implementations. This will specifically exclude some samplers (such as Knuth-Yao) which are extremely efficient, but require large losses to be made constant time. We're making this decision to ignore them despite similar work in this direction for constant-time implementations due to the theoretical issues with the work. Specifically, they appeal to boolean formula minimization black boxes, which is Σ_2-hard to do within $O(n^\epsilon)$ factors.

Intrinsically, there are a few main types of samplers that we'll consider

2.1 CDT-Based Samplers

These pre-compute the CDT of the discrete gaussian, and store it in some table. Then, to sample from this table a random variable $U(0, 1)$ is sampled, which is then used to find the correct value in the table.

The DGS is usually approximated by some discrete gaussian with support on $[-k\sigma, k\sigma]$. This is standard (where k is known as the *tail-cut parameter*), and for $k = O(1)$ can lead to a distribution that's statistically close to the true DGS (essentially through gaussian concentration). The table must store both the value to return, and a high-precision cumulative distribution value corresponding to that row (which serves as the key into the table essentially).

A basic computation shows that if the key is required to be precise to b bits, then the number of bits per row is $\log_2 2k\sigma + b$. Moreover, the number of rows is $2k\sigma$, so we get a storage requirement of:

$$\approx 2k\sigma(\log_2 2k\sigma + b) \tag{2}$$

bits. An approximation of how large this can get is $\approx 224KB$, which is rather large for an embedded device. Moreover, this only allows for sampling from a single σ, and has the center implicitly set to 0. For each new σ and center, you'd need a new table, which would make this storage requirement grow rather quickly.

That being said, for $\sigma \approx 4$ this is much more reasonable, giving $4KB$. It's possible the tail-cut parameter ($k = 10$ above) can even be set to be lower in this case, but I didn't try to justify this.

For a CDT-based sampler, people recommend doing a linear scan to make it constant-time. While naive binary search (with early abort) appears to be clearly non-constant, I'm not clear why binary search (with the duration padded to take $O(\log n)$ time each search) isn't acceptable. This has data-dependent branching, which makes it susceptible to things like perf, but it's not clear to me that removing this susceptibility is with the:

$$O(\log n) \mapsto O(n) \tag{3}$$

blowup. This seems like a rather cheap exponential speedup to take, especially since embedded systems seem to have no reason to run perf when they're actually deployed.

This has apparently been done[1], and some people say that the memory-access pattern leaks information (that an entire linear scan doesn't). This could be fixed by appealing to ORAM (sublinear-overhead schemes exist, meaning $O(\log^2(n))$, where n is the number of items stored). This seems like it'd inflate the running time to $O(\log^3 n)$, which is asymptotically much better than $O(n)$, although implementing ORAM in hardware is a tougher sell. The client storage requirement is $O(\log n)$, with an additional server storage requirement of $O(k'n)$, where k' is some other small constant. This seems like it could be rather competitive. There are a few issues with this though — the CDT values

[1]On practical discrete gaussian samplers for lattice-based cryptography

are unencrypted (and what threat model are we even operating under? That an adversary can get memory-access patterns, but not read the memory? This sounds like the perf attack).

2.2 Knuth-Yao Samplers

This involves building some tree, and then a random (unbiased) walk on the tree recovers the value to return. This is rather entropy efficient, although it's unclear how to make it constant time. Truncating the gaussian can still be done, so that the tree is at least finite, but the tree will in general be *highly* unbalanced. It's possible some sort of rebalancing can be done, but that will distort the random walk from unbiased to biased (with the bias depending on which node you're at, so this will require additional storage at each node).

Some papers build this tree, pad it out to a full tree, then appeal to a black-box circuit minimization technique, but this seems suboptimal in several ways theoretically. It should likely still be considered to ensure that it isn't practically much more efficient.

2.3 Kearny Sampling/Rejection Sampling

These schemes

2.4 Convolution Samplers

Base sampler $\to$ Full sampler via convolution methods.

3 Existing Side-Channel Attacks

While designing a sampler without side-channel attacks, it makes sense to survey the existing attacks and suggested countermeasures.

3.1 Power Analysis

https://www.mdpi.com/2076-3417/8/10/1809/htm

3.2 TODO:

Read this survey https://eprint.iacr.org/2019/068.pdf