Harvard Privacy Tools Project

Randomness and Noise

Contents

1 Overview

This document describes the strategies the library uses for generation of randomness and noise.

2 Preliminaries

Definition 1. *Differential Privacy [DMNS06]*
For $\epsilon, \delta \geq 0$, a randomized mechanism $\mathcal{M} : \mathcal{X}^n \times \mathcal{Q} \rightarrow \mathcal{Y}$ is (ϵ, δ)-DP if, for every pair of neighboring data sets $X, X' \in \mathcal{X}^n$ and every query $q \in \mathcal{Q}$ we have

$$\forall \mathcal{T} \subseteq \mathcal{Y} : \Pr[\mathcal{M}(X, \epsilon, \delta, q) \in \mathcal{T}] \leq e^{\epsilon} \Pr[\mathcal{M}(X', \epsilon, \delta, q) \in \mathcal{T}] + \delta.$$

If $\delta = 0$, we call this *Pure DP*. If $\delta > 0$, we call this *Approximate DP*. Note that, in practice, differential privacy could be thought of a bit more broadly – as a bounded distance between joint distributions over all entities that could be observable to an adversary.[1] The primary focus of the WhiteNoise library is to respect the traditional notion of differential privacy as defined in 1. However, we are also working to reduce the possibility of information leakage via differences in computational runtime. We will touch on this when relevant during throughout the document.

Definition 2. *Exact Rounding*
Let $S \subset \mathbb{R}$ be some set. Let $\phi : \mathbb{R}^n \rightarrow \mathbb{R}$ be a function on the reals and $\phi_S : S^n \rightarrow S$ be its implementation over S. Then, ϕ_S respects exact rounding *for (ϕ, S) if*

$$\forall s \in S : \phi_S(s) = round_S[\phi(s)],$$

where $round_S(\cdot)$ rounds a real number to a member of S according to some rounding rule.

For our purposes, we will care only about the case where $S = \mathbb{F}$, the set of IEEE-754 floating-point numbers.

Definition 3. *Truncation and Censoring*
Throughout our noise functions, we use the terms "truncated" and "censored". Both are means of bounding the support of the noise distribution, but they are distinct.

Truncating a distribution simply ignores events outside of the given bounds, so all probabilities within the given bounds are scaled up by a constant factor. One way to generate a truncated distribution is via rejection sampling. You can generate samples from a probability distribution as you normally would (without any bounding), and reject any sample that falls outside of your bounds.

Censoring a distribution, rather than ignoring events outside of the given bounds, pushes the probabilities of said events to the closest event within the given bounds. One way to generate a censored distribution would be to generate samples from a probability distribution as you typically would, and then clamp samples that fall outside of your bounds to the closest element inside your bounds.

3 Randomness

All of our random number generation involves uniform random sampling of bits via OpenSSL. We will take as given that OpenSSL is cryptographically secure. We intend to support a broader set of cryptographically secure sources of randomness at a later date.

4 Noise Generation

4.1 Introduction to MPFR

The GNU MPFR Library[FHL+07] is a C library with methods for carrying out a number of floating-point operations with *exact rounding*(see Definition 2). Among these are basic arithmetic operations and means of generating samples from basic probability distributions.

[1] For example, imagine that the US government uses $\epsilon = 1$ if the President is in the data and $\epsilon = 10$ if not – if anyone knew about this rule, the choice of epsilon would leak information not accounted for in the traditional definition of DP.

4.2 Biased bit sampling

Recall that we are taking as given that we are able to sample uniform bits from OpenSSL. For many applications, however, we want to be able to sample bits non-uniformly, i.e. where $\Pr(bit = 1) \neq \frac{1}{2}$. To do so, we use the *sample_bit* function. This function uses the unbiased bit generation from OpenSSL to return a single bit, where $\Pr(bit = 1) = prob$ – there is a nice write-up of the algorithm here. We will give a general form of the algorithm, and then talk about implementation details.

Algorithm 1 Biasing an unbiased coin (in theory)

1: $p \leftarrow \Pr(bit = 1)$
2: Find the infinite binary expansion of p, which we call $b = (b_1, b_2, \ldots,)_2$. Note that $p = \sum_{i=1}^{\infty} \frac{b_i}{2^i}$.
3: Toss an unbiased coin until the first instance of "heads". Call the (1-based) index where this occurred k.
4: return b_k

Let's first show that this procedure gives the correct expectation:

$$
\begin{aligned}
p &= \Pr(bit = 1) \\
&= \sum_{i=1}^{\infty} \Pr(bit = 1 | k = i) \Pr(k = i) \\
&= \sum_{i=1}^{\infty} b_i \cdot \frac{1}{2^i} \\
&= \sum_{i=1}^{\infty} \frac{b_i}{2^i}.
\end{aligned}
$$

This is consistent with the statement in Algorithm **??**, so we know that the process returns bits with the correct bias. In terms of efficiency, we know that we can stop coin flipping once we get a heads, so that part of the algorithm has $\mathbb{E}(\#flips) = 2$ because the probability of getting a heads on any given flip is 1/2.

We now move to constructing the infinite binary expansion of p. We start by noting that, for our purposes, we do not actually need an infinite binary expansion. Because p will always be a 64-bit floating-point number, we need only get a binary expansion that covers all representable numbers in our floating-point standard that are also valid probabilities. Luckily, the underlying structure of floating-point numbers makes this quite easy.

In the 64-bit standard, floating-point numbers are represented as

$$
(-1)^s (1.m_1 m_2 \ldots m_{52})_2 * 2^{(e_1 e_2 \ldots e_{11})_2 - 1023},
$$

where s is a sign bit we ignore for our purposes. Our binary expansion is just the mantissa $(1.m_1 m_2 \ldots m_{52})_2$, with the radix point shifted based on the value of the exponent. We can then index into the properly shifted mantissa and check the value of the kth element. We end up with the following algorithm:

Algorithm 2 Biasing an unbiased coin (in practice):
sample_bit(p: f64)

We know that p is respresentable as an IEEE-754 64-bit floating point number.
$m, x \leftarrow$ mantissa and exponent of the floating-point representation of p. We ensure that the mantissa gets the implicit leading 1 and the exponent is the "unbiased" version. So $m \in \{1\} \bigcup \{0,1\}^{52}$ and $x \in \{0, 1, \ldots, 2047\}$
Toss an unbiased coin until the first instance of "heads". Call the (0-based) index where this occurred k.
n_leading_zeros $\leftarrow \max(0, 1022 - x)$
if $k <$ n_leading_zeros **then**
    return 0
else
    $i \leftarrow$ n_leading_zeros $+k$
    return i^{th} element of m (using 0-based indexing)
end if

4.3 Sampling from censored Geometric

The Geometric distribution is a building block for many of our other mechanism, either as the basis of the noise distribution (as for the Geometric mechanism) or as a component of a larger algorithm (as we will show in section 4.4). For now, the library supports sampling only from a censored Geometric distribution.

The function accepts three arguments. p is the probability of success on any given trial. max_trials is the maximum number of trials for which the algorithm will run – if no success has occurred by the time max_trials has been reached, the algorithm will return max_trials as its answer. The *ect* boolean stands for *enforce constant time* and tells the algorithm to run for the maximum number of trials, regardless of when success is achieved. This is useful for reducing variability in algorithm runtime that could be used for a timing attack.

Algorithm 3 Generating draws from Censored Geometric:
sample_geometric_censored(p: f64, max_trials: i64, ect: bool)

```
trial_index ← 1
geom_return ← 0
while t dorial_index ≤ max_trials
    bit ← sample_bit(p)
    if bit == 1 then
        if geom_return == 0 then      ▷ Update result from Geometric only if we have not already seen a 1
            geom_return ← trial_index
            if ect == False then      ▷ If we do not care to enforce constant time...
                return geom_return    ▷ return the result
            end if
        end if
    else
        trial_index + = 1
    end if
end while
if geom_return == 0 then              ▷ If Geometric result > censoring bound...
    return max_trials                 ▷ have it return the value of the bound
end if
```

4.4 Sampling from Uniform[min, max)

In this method, we start by generating a floating-point number $y \in [0, 1)$, where each is generated with probability relative to its unit of least precision (ULP).[2] That is, we generate $y \in [2^{-g}, 2^{-g+1})$ with probability $\frac{1}{2^i}$ for all $g \in \{1, 2, \ldots, 1022\}$ and $y \in [0, 2^{-1022})$ for $g = 1023$. At the end, we will scale our output from $[0, 1)$ to be instead in $[min, max)$.

The algorithm is as follows:

Algorithm 4 Generating draws from Uniform[min, max)

1. $m \leftarrow \{0, 1\}^{52}$ from OpenSSL (or other cryptographically-secure RNG)
2. $g \leftarrow min(1023, sample\_geometric\_censored(p = 0.5, max\_trials = 1023, ect = True))$
3. $u \leftarrow (1.m_1 m_2 \ldots m_{52})_2 * 2^{-g} * (max - min) + min$
4. return u

This method was proposed in [Mir12] as a component of a larger attempt to create a version of the Laplace mechanism that is not susceptible to floating-point attacks. Note that the original method generates values $\in [0, 1)$ rather than arbitrary $[min, max)$ and does not give guidance on what to do if the sample from the Geometric is > 1023. There is no universally agreed upon method for generating uniform random numbers (for privacy applications or otherwise), but this method seems to approximate the real numbers better than many others because of the sampling relative to the ULP.

[2] The ULP is the value represented by the least significant bit of the mantissa if that bit is a 1.

Known Privacy Issues

When $g = 1023$ we are sampling from subnormal floating-point numbers. Because processors do not typically support subnormals natively, they take much longer to sample and open us up to an easier timing attack, as seen in [AKM+15].

We are incurring some floating-point error when converting from $[0, 1)$ to $[min, max)$ which could jeopardize privacy guarantees in ways that are difficult to reason about.[Mir12] [Ilv19]

We have a method for generating uniform samples via MPFR that respects exact rounding, but it is being used sparingly in the library. We are working to figure out if and how we can use this method as a building block for floating-point safe methods of drawing from other distributions.

4.5 Problems with sampling from other continuous distributions

In principle, we can generate draws from non-uniform continuous distributions (e.g. Laplace, Gaussian) by using inverse transform sampling. To draw from a distribution f with CDF F, we sample u from $Unif[0, 1)$ and return $F^{-1}(u)$.

Known Privacy Issues

Carrying out the inverse probability transform employs floating-point arithmetic, so we run into the same problems as were described in the uniform sampling section. This is potentially a very significant problem, and one for which we do not currently have a good general solution.

Because of the vulnerabilities inherent in using floating-point arithmetic, we would like to avoid using inverse transform sampling. We do not have a completely general way of ensuring that algorithms designed to be private when drawing noise from $\mathbb{R}$ remain private when they have access only to floating-point numbers. Instead, we will take each relevant distribution individually and discuss potential solutions.

4.5.1 Potential fixes for Laplace

We have a branch with an implementation of the Snapping mechanism from [Mir12]. We are currently working to verify the theory and associated implementation, as well as consider how to use it effectively in practice.

4.5.2 Potential fixes for Gaussian

As mentioned for the Uniform, we have a method for generating Gaussian samples via MPFR that respects exact rounding. It is not currently being used in the library.

[DKM+06] proposes using the binomial approximation to the Gaussian, and notes that an additive noise mechanism drawing noise from a $Binomial(n, p = 0.5)$ respects (ϵ, δ)-DP provided that $n \geq 64\frac{\log(2/\delta)}{\epsilon^2}$. This seems promising and we already have the infrastructure for generating a Binomial without floating-point arithmetic, but it involves manually sampling bits. For small ϵ, this method could quickly become computationally unwieldy.

For example, having the mechanism respect $(10^{-2}, 10^{-9})$-DP requires sampling nearly 14 million bits.

References

[AKM+15] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *2015 IEEE Symposium on Security and Privacy*, pages 623–639. IEEE, 2015.

[DKM+06] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our data, ourselves: Privacy via distributed noise generation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 486–503. Springer, 2006.

[DMNS06] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*, pages 265–284. Springer, 2006.

[FHL+07] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2):13–es, June 2007.

[Ilv19] Christina Ilvento. Implementing the exponential mechanism with base-2 differential privacy, 2019.

[Mir12] Ilya Mironov. On significance of the least significant bits for differential privacy. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 650–661, 2012.