Harvard Privacy Tools Project

# Noise Generation Notes

Christian Covington

February 11, 2020

## 1 Overview

This document is a write-up of extra notes regarding the ways in which we sample noise in yarrow.

## 2 Random Number Generation

All of our random number generation involves uniform random sampling of bits via OpenSSL. We will take as given that OpenSSL is cryptographically secure, and talk about how it forms the basis for various functions in the library. When we refer to floating-point numbers, we specifically mean the IEEE 754 floating-point standard.

### 2.1 Uniform Number Generation

#### 2.1.1 $sample\_uniform(min : f64, max : f64)$

In this method, we start by generating a floating-point number in $[0, 1)$, where each is generated with probability relative to its unit of least precision.[1] That is, we generate $x \in [2^{-i}, 2^{-i+1})$ with probability $\frac{1}{2^i}$ for all $i \in \{1, 2, \ldots, 1022\}$ and $x \in [0, 2^{-1022})$ for $i = 1023$.
Within each precision band (the set of numbers with the same unit of least precision), numbers are sampled uniformly. We achieve this sample our exponent from a geometric distribution with parameter $p = 0.5$ and a mantissa uniformly from $\{0, 1\}^{52}$. Let $e$ be a draw from $Geom(0.5)$ (truncated such that $e \in \{1, 2, \ldots, 1023\}$) and $m_1, m_2 \ldots, m_{52}$ be the bits of our mantissa. At the end, we will scale our output from $[0, 1)$ to be instead in $[min, max)$. Then our function outputs $u$, where

$$u = (1.m_1 m_2 \ldots m_{52})_2 * 2^{-e} * (max - min) + min.$$

When $i = 1023$ we are sampling from subnormal floating-point numbers. Because processors do not typically support subnormals natively, they take much longer to sample and

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

open us up to an easier timing attack, as seen in [AKM+15]. Protecting against timing attacks is mostly seen as out of scope for now, but I wanted to bring this up anyway.

## 2.2 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.

### 2.2.1 $sample\_bit(prob : f64)$

This function uses the unbiased bit generation from OpenSSL to return a single bit, where $\Pr(bit = 1) = prob$. I was introduced to the method for biasing an unbiased coin from a homework assignment given by Michael Mitzenmacher, and I later found a write-up online here. We will give a general form of the algorithm, and then talk about implementation details.

**Algorithm 1** Biasing an unbiased coin

1: $p \leftarrow \Pr(bit = 1)$
2: Find the infinite binary expansion of $p$, which we call $b = (b_1, b_2, \dots, )_2$. 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: **if** $b_k = 1$ **then**
5: return 1
6: **else**
7: return 0
8: **end if**

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

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

This is consistent with the statement in Algorithm 1, 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 hsa $\mathbb{E}(\#flips) = 2$.

The part that is a bit more difficult is 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 in $[0, 1]$ are represented as

$$(1.m_1m_2 \ldots m_{52})_2 * 2^{(e_1e_2 \ldots e_{11})_2 - 1023}.$$

Then, our binary expansion is just our mantissa $(1.m_1m_2 \ldots m_{52})_2$, 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 $k$th element.

## 3 Truncation vs. 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.

## 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.