Miranda Christ
Rohan Jha

# Hardness vs. Randomness



We first present a brief summary of the paper; we then take a deep-ish dive into the construction of K-M designs; and we finally discuss our implementation of the Nisan-Wigderson pseudorandom generator (PRG).

## Summary

The aim here is to generate "many" pseudorandom bits, given "a few" truly random bits. By pseudorandom, we mean indistinguishable from truly random by any small circuit. We want to accomplish this efficiently, namely in exponential time with respect to the size of the generator's input. The existence of such a generator, with a further restriction on the size of its input, implies that we can deterministically simulate polynomial-time randomized decision algorithms in polynomial time. While this existence is contingent on an unproven assumption, this paper makes a significant step towards showing that P=BPP.

### Yao's lemma

Yao's lemma helps to motivate the results of this paper. We present the following generalization of the lemma that's included in the paper. Note that if a generator $G$ is *quick*, then $G$ is in $DTIME(2^{O(l)})$, where $l$ is the size of the random seed that $G$ takes as input.

**Lemma** (Yao's)**.** *If there exists a quick pseudorandom generator (PRG)* $G : l(n) \to n$ *then for any time constructible bound* $t = t(n) : RTIME(t) \subset DTIME(2^{O(l(t^2))})$

The intuition here is as follows. Assume that we have a PRG that fools circuits of some size into thinking that its outputs are truly random. Now, consider an arbitrary circuit $C$ of that size, and one of two cases: (1) we pass as input a truly random string, or (2) we pass as input a string that was generated by our PRG. If the outputs are different, on average, then we could use $C$ to distinguish between random and pseudorandom inputs, which contradicts the assumption that our PRG fools circuits of that size. Now with the assurance that the outputs of $C$ will be similar in both cases, we're able to run the PRG on each possible input to our generator, then pass each pseudorandom number into $C$ as input. It can then be shown that the 'majority vote' of the $2^l$ iterations of $C$ is surely the right answer.

A corollary, and the main takeaway, is that $BPP \subset P$ is implied by the existence of a quick PRG, for which the input size is $O(\log n)$. The goal of the paper is to build such a generator. It's also worth noting that the authors are able to relax the usual condition that the PRG runs in polynomial time by insisting that the inputs to the PRG are of size $O(\log n)$.

### Generator

Since our goal is to generate bits that are "hard" for a small circuit to differentiate from random bits, we need to make some sort of hardness assumption, namely the existence of a "hard" function.

**Definition** ($(\epsilon, S) - hard$)**.** Let $f : \{0,1\}^n \to \{0,1\}$ be a boolean function. We say that $f$ is $(\epsilon, S) - hard$ if for any circuit $C$ of size $S$,

$$\left| Pr[C(x) = f(x)] - \frac{1}{2} \right| < \frac{\epsilon}{2}$$

where $x$ is chosen uniformly at random in $\{0,1\}^n$. In other words, there is no sufficiently small circuit that can reproduce our function's behavior on significantly more than half the inputs.

In particular, we assume there exists a function $f$ that's $(\frac{1}{n^2}, S) - hard$, where $S$ is some polynomial. The construction of the generator follows directly from this hard function. We generate each pseudorandom bit by applying our hard function to a subset of $O(\log n)$ truly random bits. We reiterate that the existence of this generator implies $BPP \subset P$ by Yao's lemma.

In order to prove that our generator is pseudorandom, we derive a contradiction from the existence of a circuit that can distinguish our pseudorandom bits from truly random bits with greater than $\frac{1}{2}$ probability. More specifically, we use the existence of such a distinguisher to imply that $f$ is not hard. We give a short sketch of the proof, which relies on the choice of the subsets (we'll discuss this later).

Suppose we have a circuit whose output differs by at least $\frac{1}{n}$ given a pseudorandom input versus a truly random input. Applying Yao's lemma about hardness amplification and applying a hybridization argument allows us to convert this advantage to predict one pseudorandom bit based on the others. In other words, we can construct a circuit $D$ such that

$$Pr[D(y_1, ..., y_{i-1}) = y_i] - \frac{1}{2} > \frac{1}{n^2}$$

where the $y_i$'s represent pseudorandom bits and the probability is over the truly random bits used to generate them. Note that $y_i = f(x_1, ..., x_m)$ for some subset of the truly random $x$'s, and by another averaging argument, we can fix the other random $x$'s to constants and preserve our advantage. We observe that the bits that each $y_j$ depends on are fixed except for those in the intersection of its subset and $x_1, ..., x_m$. We've now written $f(x_1, ..., x_m)$ as a function of only these bits in the intersections, and with sufficiently small intersections, $f$ depends on few variables and can be computed with a small circuit, just by using CNF.

We therefore want to minimize the seed length, while keeping the intersection of any pair of subsets small. This nontrivial problem motivates K-M designs.

## K-M designs

We first present the definition of a K-M design from the paper.

**Definition** (K-M design)**.** A collection of sets $\{S_1, \ldots S_n\}$, where $S_i \subset \{1, \ldots l\}$ for all $i$, is called a (k-m)-design if $|S_i| = m$ for all $i$ and $|S_i \cap S_j| \leq k$ for all $i \neq j$.

We note that $l$ is a parameter in our control. In the context of the paper, however, $l$ is the size of the random seed that the generator takes as input. This implies that number of seeds (and the time to simulate a random algorithm) is exponential in $l$. It might be helpful to look at the generation of K-M designs in light of this context: we want the smallest possible $l$ such that we can satisfy the requirement that the intersections between $S_i$ and $S_j$ are small. It's also worth noting that this problem would be trivial if we didn't care about the size of $l$.

### Take 1

We'll now describe the method in the paper for generating these K-M designs for $l = O(m^2)$. This will prove the following lemma.

**Lemma.** *For all integers $n$ and $m$, such that $\log n \leq m \leq n$, there exists a $(\log n, m)$-design, where $S_i \subset \{1, \ldots l\}$ for all $i$ and $l = O(m^2)$.*

In broad terms, we'll define a polynomial (on a finite field) for each of the $n$ sets. We'll then apply each polynomial to the same set of $m$ inputs to yield $n$ sets of $m$ elements. These polynomials are helpful because we can bound the number of times that distinct polynomials can intersect, which implies a bound on the size of the intersection of any two sets.

We start by constructing a finite field. Note that we need a field that's large enough for the number of distinct polynomials to be larger than $n$ (we show below that this is the case). We find an $m'$ that's a prime power, such that $m \leq m' < 2m$. This can be done by finding a power of 2 in the interval $[m, 2m)$. We then have, by algebra, that the integers mod $m'$ are a field, under addition and multiplication. We'll refer to this field as $GF(m')$ (short for 'Galois field'). We finally set $l = (m')^2$, in order for the mapping below to be well-defined.

Next, we define a mapping $f$ from $\{1, \ldots l\}$ to $(a, b) | a, b \in GF(m')$. We'll use this to translate between the inputs and outputs of our polynomials and $\{1, \ldots l\}$. Because $l = (m')^2$, we have that elements in $\{1, \ldots l\}$ can be expressed as $a + b \cdot m'$ for a unique $(a, b) \, | a, b \in GF(m)$ (we can think of writing the elements in $\{1, \ldots l\}$ in a grid of dimension $m' \times m'$). We define $f(x)$ as this $(a, b)$, which yields that $f^{-1}((a, b))$ is $a + bm'$.

We now describe the algorithm. We select $n$ distinct polynomials $p_1, \ldots p_n$, on $GF(m')$ with degree at most $\log n$. The fact that $(m')^{\log n + 1} > n$ implies that there are enough such polynomials. We then build a set $S_i$ of our design by computing $b = p_i(a)$ for each $a$ in $\{1, \ldots m\}$ and adding each $f^{-1}((a, b))$ to $S_i$. Because $f^{-1}((a_1, b_1)) = f^{-1}((a_2, b_2))$ if and only if $a_1 = a_2$ and $b_1 = b_2$, having distinct $a$'s for each element of $S_i$ implies that the elements of $S_i$ are distinct.

We finally note that this is a valid K-M design. First, each set has $m$ elements. Second, the intersection between any two sets is at most $\log n$ because two distinct polynomials of degree $\log n$ can intersect at no more $\log n$ points (because their difference, which is also a polynomial of degree $\log n$, can have no more than $\log n$ roots).

**Take 2**

The authors also note that we can improve $l = O(m^2)$ to $l = O(m \log m)$. We'll discuss this briefly. Looking back on *Take 1*, $l$ is a function of the size of our field, which must be larger than $m$ in order for the number of distinct polynomials (with fixed degree) to be higher than $n$. The guiding principle here is to improve (decrease) the dependence of the number of distinct polynomials on the size of the field, and we find that this dependence is better for multivariate polynomials. We'll hand-wave through a short example. We consider the two following polynomials on the same arbitrary finite field:

$$f_1 = (a_1x + b_1)(a_2x + b_2)(a_3x + b_3)(a_4x + b_4)(a_5x + b_5)(a_6x + b_6)$$
$$f_2 = (a_1x + b_2)(a_2x + b_2)(a_3x + b_3)(a_4y + b_4)(a_5y + b_5)(a_6y + b_6)$$

We note that both have six roots, but $f_2$ yields more distinct polynomials because fewer terms combine. Therefore, if we're using multivariate polynomials, we're able to reduce the size of our field, while ensuring that the number of distinct polynomials remains high enough. This yields smaller $l$, assuming a strategy that's similar to that in *Take 1*.

## Building a PRG

In this section, we simulate a Nisan-Wigderson pseudorandom generator and compare the resulting strings to a truly random strings. Note that since the existence of a hard function hasn't been proven, we couldn't use one. Instead, we use $f : \{0,1\}^n \to \{0,1\}$ where $f(s) =$ the parity of the sum of the bits of $s$. $f$ is in fact very easy to compute, but it still helps us get some intuition. The bits it generates look "pretty random," but still significantly different from truly random. This indicates that Nisan-Wigderson's method is pretty good, while also motivating the need for a truly hard function. We also find that using more random bits leads to a better simulation of randomness.

In our plots below, we examine the distribution of the number of ones in pseudorandom vs. truly random strings. We do so by creating a $(\log n, m)$-design and applying $f$ to each resulting subset. We generate 10,000 each of pseudorandom and random strings of $n$ bits. We plot the number of strings containing $x$ ones vs. the number of ones. The truly random case should approach the binomial distribution, which we plot as well.

Note that in figures 1 and 2, we generate fewer pseudorandom bits than we have random bits. We include these figures because as the number of generated bits grows, the distribution becomes tighter, and it's more difficult to visually distinguish random from pseudorandom (though they remain significantly different). We include figure 3 to illustrate an example where we generate more pseudorandom bits (256) from fewer truly random bits (81). Though less striking, the difference in distributions is still apparent.

While clearly different from the truly random strings, our pseudorandom strings look fairly random. This gives some intuition that the Nisan-Wigderson generator performs reasonably

well, even with such an “easy” function. We also note that these plots imply that the outputs of our PRG are easily distinguished from truly random strings. A distinguisher might count the 1’s in the input (let this number be $N$), then look at the plots to determine whether $\Pr[N \mid \text{random}]$ is higher than $\Pr[N \mid \text{our PRG}]$, or vice versa. This is expected, since our function isn’t hard.

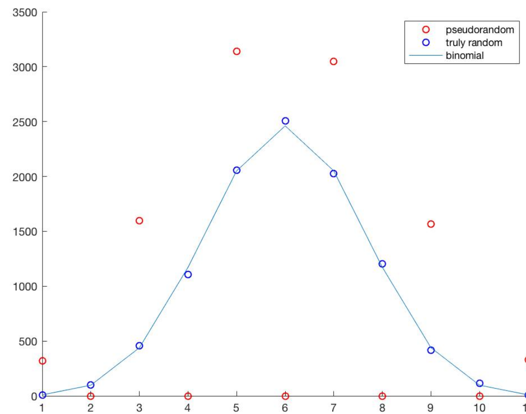


Figure 1: 10-bit strings, with 25 truly random bits

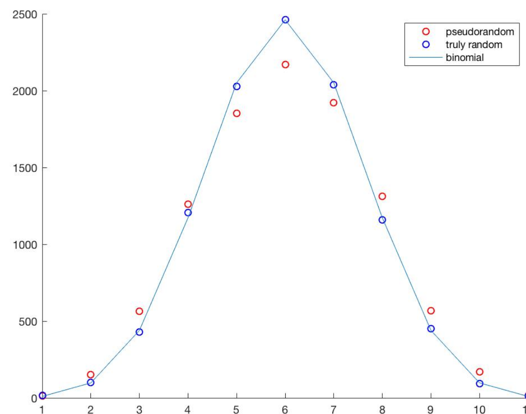


Figure 2: 10-bit strings, with 81 truly random bits

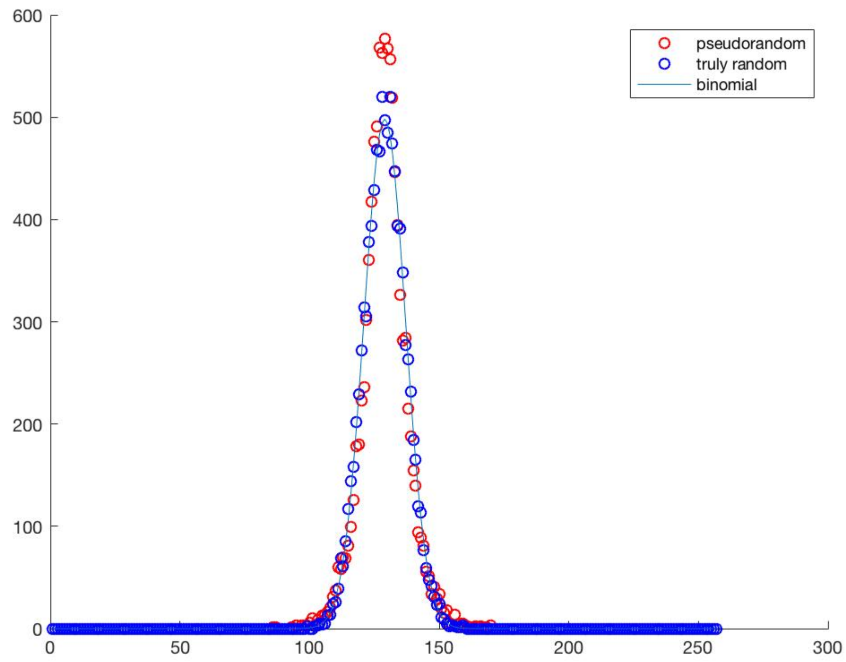


Figure 3: 256-bit strings, with 81 truly random bits