Dream Investigation Results

Official Report by the Minecraft Speedrunning Team

Published: December 11, 2020
Updated: December 15, 2020

Contents

I Introduction 3 1 Mechanics 3 2 Motivation 3 3 Objectivity 4

II Data 5 4 Piglin Bartering 5 5 Blaze Rod Drops 6

III Analysis 6 6 Methodology 6

7 The Binomial Distribution 8 7.1 The Intuition . . . . 8 7.2 Generalizing the Binomial Distribution . . . . 8 7.3 The Cumulative Distribution Function (CDF) 9

8 Addressing Bias 10 8.1 Accounting for Optional Stopping . . . . 10 8.2 Sampling Bias in Stream Selection . . . . 11 8.3 Sampling Bias in Runner Selection . . . . 12 8.4 P-hacking 12

9 Code Analysis 14 9.1 Confirming the Probabilities . . . . 14 9.2 Setting RNG Seeds . . . . 15 9.3 Linear Congruential Generators . . . . 16 9.4 Periodicity . . . . 17 9.5 Bartering . . . . 18 9.6 Blaze Drops 19

IV Results 20

10 Computation 20 10.1 Naive Estimate . . . . 20 10.2 Full Computation 21
10.2.1 Pearls . . . . 21 10.2.2 Blaze Rods . . . . 21 10.2.3 Combined Number 22

11 Conclusion 23

A Raw Data 24 B Stopping Rule Computation Algorithm 25 C Probability Computations 27

Part I
Introduction

1 Mechanics

Note: This section exists to explain Minecraft 1.16 Random Seed Glitchless speedruns to the unfamiliar reader. Motivation begins a discussion of why this investigation took place, and is a suitable starting point for those already familiar with Minecraft speedruns.

Speedrunning is a hobby in which people compete to complete a video game as quickly as possible. This paper concerns speedruns of *Minecraft: Java Edition*, and, in particular, speedruns of the category known as "Any% Random Seed Glitchless" (RSG) performed on version 1.16. A brief summary of the relevant mechanics and speedrun strategies follows for the unfamiliar reader.

The final boss of Minecraft is located in an alternate dimension known as The End, which can be accessed using End Portals. An end portal consists of twelve End Portal Frame blocks, a random number (usually 10-12) of which must be filled with an Eye of Ender in order to activate the portal. Thus, the runner is required to have up to twelve eyes of ender when they arrive at the portal to be able to enter The End and complete the game.

In 1.16, the only way to obtain an eye of ender is by crafting it, which requires one Ender Pearl and one Blaze Powder. Ender pearls can be obtained in several ways, but the fastest is to use a mechanic known as Bartering. In a barter, the player exchanges a Gold Ingot with a Piglin (a humanoid creature in the Nether dimension) for a randomly chosen item or group of items. For each barter, there is about a 5% chance (in 1.16.1) the piglin will give the player ender pearls.

Blaze powder is crafted out of Blaze Rods, which are dropped by Blazes—a hostile mob. Upon being killed, each blaze has a 50% chance of dropping one blaze rod. The main focus during the beginning of a 1.16 RSG speedrun is to obtain (hopefully) 12 eyes of ender as quickly as possible, by bartering with piglins and killing blazes. These two parts of the speedrun route are the primary concern of this paper.

2 Motivation

Members of the Minecraft speedrunning community[a]reviewed six consecutive livestreams of 1.16 RSG speedrun attempts by Dream[b]from early October 2020. The data collected show that 42 of the 262 piglin barters performed throughout these streams yielded ender pearls, and 211 of the 305 killed blazes dropped blaze rods. These results are exceptional, as the expected proportion of ender pearl barters and blaze rod drops is much, much lower.

An initially compelling counterclaim is that top-level RSG runners must get reasonably good luck in order to get a new personal best time in the first place, so, while it is surprising to see such an unlikely event, it is perhaps not unexpected. However, upon further research, Dream's observed drop rates are substantially greater than those of other top-level RSG runners—including, Illumina, Benex, Sizzler, and Vadikus. If nothing else, the drop rates from Dream's streams are so exceptional that they ought to be analyzed for the sake of it, regardless of whether or not any one individual believes they happened legitimately.

[a]The data were originally reported by MinecrAvenger and danhem9.

[b]https://www.twitch.tv/dreamwastaken

3 Objectivity

The reader should note that the authors of this document are solely motivated by the presence of exceptional empirical data, and that any runner—regardless of popularity, following, or skill— observed experiencing such unlikely events would be held to the same level of scrutiny. The reader should also note that the data presented are extensively corrected for the existence of any bias. It would lack rigor and integrity for the conclusions made in this report to substantiate the moderation team's decision if they were merely based on a surface-level analysis of the data. Indeed, these corrections inherently skew the analysis *in Dream's favor*. We aim to calculate not the exact probability that this streak of luck occurred if Dream is innocent, but an upper bound on the probability; that is, we will arrive at a value which we are certain is *greater* than the true probability.

The goal of this document is to present the unbiased, rigorous statistical analysis of the data, as well as an analysis of the Minecraft source code, to conclusively determine whether or not such an event could be observed legitimately.

Part II
Data

The raw data (and its sources) from which the following graphs were derived can be found in Appendix A.

4 Piglin Bartering

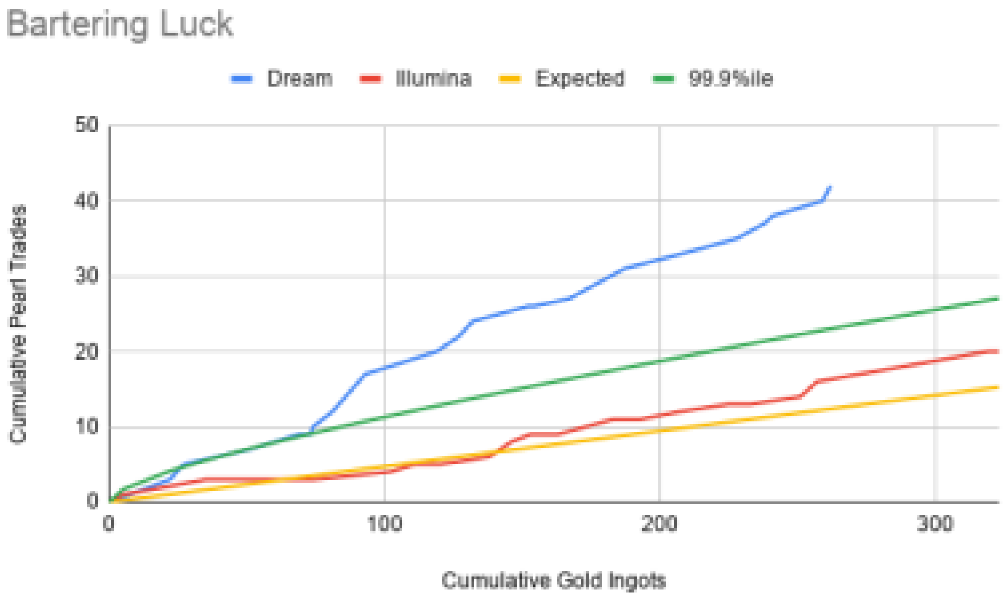


Figure 1: Dream’s pearl barters, charted alongside various comparisons. The 99.9th percentile line represents one-in-a-thousand luck (calculated using a normal approximation), which is *already* quite unlikely—if not necessarily proof of anything.

5 Blaze Rod Drops

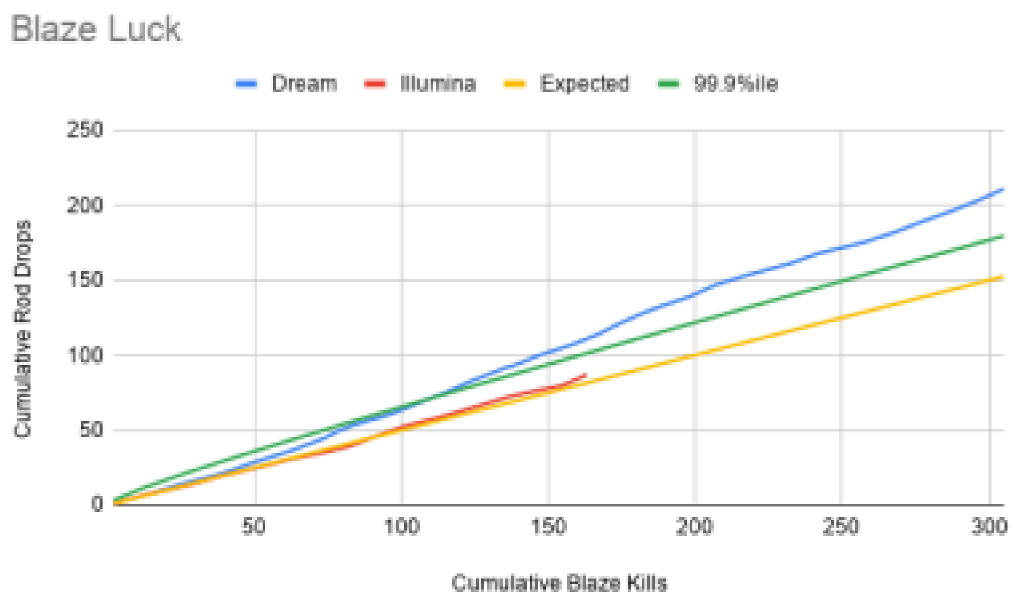


Figure 2: The same for blaze rod drops.

Part III
Analysis

6 Methodology

What follows is a thorough description of every aspect of our investigation in an accessible manner. We will begin with an introduction to the binomial distribution, and follow with adjustments to account for sampling bias and other biases lowering the accuracy of the binomial distribution. Finally, we will analyze Minecraft's code to justify the assumptions made in our statistical model. To strengthen our analysis to the skeptical reader, we now preemptively address expected criticisms and questions.

Why are you not analyzing all of Dream's runs? Doesn't that introduce sampling bias? Yes. There is clearly sampling bias in the data set, but its presence does not invalidate our analysis. Sampling bias is a common problem in real-world statistical analysis, so if it were impossible to account for, then every analysis of empirical data would be biased and useless. Consider flipping a coin 100 times and getting heads 50 of those times (a mostly unremarkable result). Within those 100 coin flips, however, imagine that 20 of the 50 heads occurred back-to back somewhere within the population. Despite the proportion overall being uninteresting, we

still would not expect 20 consecutive heads anywhere. Obviously, choosing to investigate the 20 heads introduces sampling bias—since we chose to look at those 20 flips *because* they were lucky, we took a biased sample.

However, we can instead discuss the probability that 20 or more back-to-back heads occur at *any point* in the 100 flips. We can use that value to place an upper bound on the probability that the sample we chose could possibly have been found with a fair coin, regardless of how biased a method was used to choose the sample.

It’s also worth noting that the choice to only consider Dream’s most recent streak of 1.16 streams is the least arbitrary distinction we could have made. The metaphor of "cherry-picking" usually brings to mind choosing from a wide number of options, but there were at most a small handful of options meaningfully equivalent to analyzing every stream since Dream’s return to public streaming. Note the importance of the restriction that we must analyze the entire six streams as a whole; true cherry-picking would specifically select individual barters to support a desired conclusion.

How do we know this investigation isn’t biased?

Concerns about the impartiality of the authors of this paper have been raised in discussion about the investigation. We do not think this is a significant issue; we have made an effort to be as fair to Dream and thorough as possible in our investigation. Regardless, it is a concern worth addressing.

This paper has been written to be as accessible as possible to an audience without in-depth knowledge of statistics or programming. This is primarily so that you do not have to take our word for its accuracy. By reading the analysis, you should be able to understand at least on a basic level why the statistical corrections we made account for all the relevant biases.

Additionally, as noted in Section 3: Objectivity, we aimed not to calculate the precise probability of Dream experiencing these events, but an *upper bound* on the probability. This makes it much more difficult for bias to have any effect; if we correct for the largest amount of bias in the data that there could possibly be, there is little risk our analysis will be skewed due to *our* bias causing us to underestimate how much we ought to correct.

We believe that, to the extent any bias exists, these measures should be more than sufficient to account for it. Additionally, note that we are not the only people capable of analyzing these events—if any unbiased third party points out a flaw in our statistical analysis or notes a glitch that could potentially cause these events, they would, of course, be taken seriously.

What if Dream’s luck was balanced out by getting bad luck off stream? This argument is sort of similar to the gambler’s fallacy. Essentially, what happened to Dream at any time outside of the streams in question is entirely irrelevant to the calculations we are doing. Getting bad luck at one point in time does not make good luck at a different point in time more likely.

We *do* care about how many times he has streamed, since those are additional opportunities for Dream to have been noticed getting extremely lucky, and if he had gotten similarly lucky during one of those streams an investigation still would have occurred. However, what luck Dream *actually got* in any other instance is irrelevant to this analysis, as it has absolutely no bearing on how likely the luck was in this instance.

7 The Binomial Distribution

Note: If the reader is equipped with a basic understanding of statistical analysis and the binomial distribution, they may skip to Section 8: Addressing Bias. Note that the explanations present here are sufficient for the probability calculations performed throughout the rest of the paper, but are not exhaustive. Supplemental reading is provided via footnotes where relevant.

7.1 The Intuition

Informally, if the outcome of a particular event can be described as "it either happens or it doesn't", then it can be modeled with the binomial distribution[c]. For example, imagine we wanted to compute the odds of flipping a fair coin[d] 10 times and having it land on heads exactly 6 of those times. Since a coin either lands on heads or it doesn't, we can use the formula for the binomial distribution[e]to determine the chance of this occurring.

Since we flip the coin 10 times, we say $n = 10$, and since we want exactly 6 of those flips to be heads, $k = 6$. The chance of a (fair) coin landing on heads is 50%, so $p = 0.5$. If we plug these values into the binomial distribution formula, we get

$$P\,(6; 0.5, 10) = \quad 10\ 6 \quad 0.5^6(1-0.5)^{10-6} \approx 0.205 \quad (1)$$

To interpret this value, if we flip a coin 10 times, we can expect to get exactly 6 heads about 20.5% of the time. To understand *why* this formula yields the probability of a binomial distribution, and how to generalize it, we break down each term.

7.2 Generalizing the Binomial Distribution

Generically, the probability of exactly k successes with probability p occurring in n trials (in our earlier example, $k = 6$ heads with probability $p = 0.5$ occurring in $n = 10$ flips) is given by

$$P\,(k; p, n) = \quad p^k(1-p)^{n-k} (2)\ k$$

$$n$$

We can deconstruct this formula term-by-term to understand why this represents the probability. Basically, this formula figures out how many distinct orderings of k successes and $n - k$ failures meet the criteria, and then sums the probability of each ordering[f].

The notation $^n{}_k$, read as "n choose k", represents the binomial coefficient[g], which is the number of ways we can observe k successes in n trials—the number of ways, with n options for trials to be successes, you could "choose" k of them. For example, there are two ways to observe $k = 1$ heads in $n = 2$ coin flips. The head could occur on the first flip, or it could occur on the

second flip. Therefore, $^2{}_1$ is equivalent to 2. With similar reasoning, $^4{}_2$ is equivalent to 6; there

[c]The binomial distribution also requires the assumption that we are observing discrete independent random variables. Since piglin bartering and blaze drops *are* discrete independent random variables (see Section 9: Code Analysis), we can safely make this assumption. There are other considerations about stopping rules which will be addressed in Section 8: Addressing Bias.

[d]A "fair coin" is defined as one whose probability of landing on heads is exactly the same as its probability of landing on tails. We are also not considering the probability that the coin lands on its side, which is entirely negligible for this introductory-level explanation to the binomial distribution.

[e]https://en.wikipedia.org/wiki/Binomial_distribution

[f]For an explanation of why this works, see https://www.youtube.com/watch?v=QE2uR6Z-NcU.

[g]https://en.wikipedia.org/wiki/Binomial_coefficient

are 6 unique ways to distribute 2 successes (heads) across 4 trials (coin flips). (These are 1&2, 1&3, 1&4, 2&3, 2&4, and 3&4.)

As the first term represents the number of distinct orderings, the next two terms represent the probability of any one order. To find this probability, we simply take the product of the probabilities of the events necessary to produce a given ordering; that is, the product of the probability of observing k successes and $n - k$ failures.

Since p is the probability of a given trial being successful, and there are k successful trials, we can account for the successful trials with the term p^k(p multiplied by itself k times)[h]. Similarly, we account for the failures by raising the probability of a failure to the power of the number of failures. As the only two possibilities in a given trial are success and failure, and the probabilities must sum to 1, the probability of a failure is $(1 - p)$. It follows that, since each trial that is not a success must be a failure, the number of failures is $(n - k)$. Thus, the final term is $(1 - p)^{n-k}$.

Multiplying all three terms together yields the probability of a binomial distribution with a given k, p, and n.

7.3 The Cumulative Distribution Function (CDF)

It would be helpful to have a way to compute the probability of observing k *or more* successes. Intuitively, we can expect the probability of observing *exactly* k successes in n trials to be smaller than the probability we observe k *or more* successes in the same n trials.

Referring back to the coin-flipping example, if we wanted to compute the probability of observing 6 *or more* heads within 10 trials, then we can simply add together the probabilities of observing exactly 6 heads, exactly 7 heads, (...), exactly 10 heads, given by

$$\sum_{k=6}^{10} \binom{10}{k} 0.5^k(1 - 0.5)^{10-k} \approx 0.377 \quad (3)$$

Indeed, this agrees with our intuition; it makes sense that it is more likely to get 6, 7, 8, 9, or 10 heads in 10 flips, than it is to get *exactly* 6 heads in 10 flips.

The chance of receiving k or more successes is often referred to as a p-value. More specifically, p-values are the chance of observing k or more successes *given the null hypothesis*. While that nuance is irrelevant if you already know for a fact the coin is fair, it is important to keep in mind in this scenario—our entire goal is, essentially, to analyze whether or not Dream is using a biased coin.

Armed with a basic understanding of the binomial distribution, we will now discuss how this initial calculation must be corrected in order to be applied to Dream's runs.

[h]For an explanation of why this works, see https://www.youtube.com/watch?v=xSc4oLA9e8o.

8 Addressing Bias

There are a few assumptions of the binomial distribution that are violated in this sample, some of which were noted in the document Dream published on October 27. This section accounts for these violated assumptions, and proves computations that account for these biases.

Note that some of these biases only apply to pearls, as blaze rod drops were examined in the same streams as pearls *due to* the pearl odds, which are independent of the blaze rod drop rate. This eliminates the sampling bias from the decision to investigate the pearl odds based on the fact that they are particularly lucky.

8.1 Accounting for Optional Stopping

The initial calculation for the p-value assumed that barters and rod drops within sequences of streams are binomially distributed, which is not precisely true (although likely a very good approximation). For the data to be binomially distributed, the *stopping rule*—the rule by which you decide when to stop collecting data—must be independent of the contents of the data.

For instance, Dream may be more likely to stop streaming for the day after getting a particularly good run, which is more likely to happen on a run with good barters and blaze rods. Indeed, Dream did stop speedrunning 1.16 RSG after achieving a new personal best time. This will result in the data being at least slightly biased towards showing better luck for Dream, and thus the data is not *perfectly* binomial.

To account for the stopping rule, we will correct for the worst possible (most biased) stopping rule. Imagine that this investigation was being conducted by *Shifty Sam*, a malicious investigator who is trying as hard as possible to report misleading data that will frame Dream. Since a lower p-value is more damning, Shifty Sam computes the cumulative p-value after every barter or after every blaze kill, and stops collecting data[i] once he deems the p-value "low enough" to make the strongest case against Dream. This is the worst possible stopping rule, since Shifty Sam will stop collecting data once the p-value is arbitrarily low enough (as deemed by him to be most convincing). It should be abundantly clear that this stopping rule is far worse than whatever stopping rule Dream actually followed during his runs.

It may not be immediately obvious how we can calculate a p-value under this stopping rule. We cannot look directly at the number of success in the data, as that is always going to be exceptional to this degree. What we can consider, however, is how quickly Shifty Sam reached his p-value cutoff. Intuitively, we might expect Shifty Sam to spend a long time waiting for the data to reach his p-value cutoff. To put it another way, it would certainly be surprising, regardless of how shifty Sam is, to hear that Dream got 30 successful barters in a row as soon as Shifty Sam started looking at the data. Knowing that Shifty Sam only decided to show you this data because it supported his argument would not really make that any less surprising (concerns about sampling bias aside—those will be addressed later).

Since the data reaching a p-value this extreme so soon is somewhat surprising even if we know the data comes from Shifty Sam, we will look at the probability that Shifty Sam stops collecting data at least as soon as Dream stopped. In other words, if n is the number of trials in Dream's data, our corrected p-value will be the probability that a series of trials will, at any point on or prior to the nth trial, have a binomial CDF p-value at least as small as the one for Dream's data.

[i]Since Shifty Sam here is supposed to represent *whatever* caused Dream to choose to stop running 1.16 RSG, suppose Shifty Sam is, say, Dream's manager, and can tell Dream when to stop or continue streaming.

Although that value could be computed through brute force, that approach would involve evaluating the probability and p-values for well over 2^{305} different sequences—which is obviously computationally intractable. As such, we used a method that allowed for dealing with multiple sequences at once. The exact algorithm is somewhat involved, so a description has been included in Appendix B for interested readers.

8.2 Sampling Bias in Stream Selection

As mentioned previously, we chose to analyze Dream's runs from the point that he returned to streaming rather than all of his runs due to a belief that, if he cheated, it was likely from the point of his return to streaming rather than from his first run. Although we cannot be entirely certain, it is also likely that MinecrAvenger decided to investigate Dream's streams due to noticing that they were unusually lucky. This, of course, means that the streams investigated are not actually a true random sample. Even if MinecrAvenger somehow chose streams to investigate at complete random, *we* are choosing to investigate these streams due to the fact that they are lucky. Thus, we cannot treat this as a true random sample.

To account for the maximum possible amount of sampling bias, imagine that Shifty Sam inspected every speedrun stream done by Dream and reported whatever sequence of consecutive streams was the most suspicious.[j] This would produce the strongest possible bias—or at least a bias much stronger than there actually is—from the choice of these particular Dream streams.

Recall the example of investigating the 20 back-to-back heads within 100 coin flips from earlier. Much like you could calculate the probability of 20 consecutive heads occurring at any point in the 100 flips, we can calculate the probability that Dream experienced bartering luck this unlikely in *any* series of consecutive streams. This would account for the bias from Shifty Sam, and thus more than account for the actual bias under consideration.

To calculate the chance that at least one sequence of streams is this lucky, we first calculate the chance that *no* sequence is. Assuming independence, we can do this by taking the chance that a given sequence isn't sufficiently lucky ($1 - p$) to the power of the number of sequences, m. If an event occurs more than zero times, then it must have occurred at least once, so we can then subtract $(1 - p)^m$ from one to get the chance that it occurs at least once, giving $1 - (1 - p)^m$. The number of consecutive sequences consisting of at least two streams from a set of n streams is $\binom{n}{2}$, as you choose two different streams to be the first and last. Adding in the n sequences consisting of only one stream, which were not included because the first and last stream are *the same stream*, you get $\binom{n}{2} + n$ which is equal to $\frac{n(n+1)}{2}$.

We can now get an upper bound p_n on the p-value across n streams, using the p-value derived from our sample.

$$p_n \leq 1 - (1 - p)^{\frac{n(n+1)}{2}} \tag{4}$$

At this point, let us go back and analyze an earlier assumption we made: that the p-values between sequences of streams are independent of one another. This assumption is false—however, it is not false in a way that could cause p_n to be greater than this upper bound.

Consider the exact way in which the sequences of streams are dependent on one another. Since they all contain streams from the same set (those from Dream), some of the data in each sequence will be identical to that in other sequences. This *lowers* the chance that Shifty Sam

[j]We can safely assume the streams reported would be consecutive—it would be extremely obvious that the streams were cherry-picked if Shifty Sam reported the luck in, say, Dream's first, seventh, and tenth streams. Non-consecutive streams could be reported credibly in unusual circumstances, but that possibility is essentially negligible.

could find misleading data, as he has less data to look through for unlikely events. In technical terms, we can say the p-values of the sequences of streams are *positively* dependent upon one another—they are positively correlated with each other. For this bound to fail, the sequences would need to be *negatively* dependent.

8.3 Sampling Bias in Runner Selection

In addition to these particular streams of Dream's being analyzed due to their high proportion of pearl barters, *Dream* was initially analyzed out of all runners due to his experiencing unusually good luck. Much like we calculated the chance of observing data as unlikely as the data in question in any sequence of streams, we will analyze the probability of observing data this unlikely from *any runner* in the Minecraft speedrunning community, using the same formula for the chance of something occurring at least once in a series of trials that we used earlier.

This results in the following correction, where p_n is the p-value corrected for a community with n runners, and p is the p-value for Dream in particular:

$$p_n \leq 1 - (1 - p)^n \quad (5)$$

Note that, as we are discussing the p-value for data this unlikely occurring to a runner within their entire speedrunning career, the *size* of their career is not relevant. Although a runner may be more likely to experience six exceptionally lucky streams if they stream more often, we already account for the amount they stream when calculating p—in other words, if someone streams more than Dream, they would need a *luckier* sequence of streams to have an equally low p.

8.4 P-hacking

Perhaps Shifty Sam examined multiple types of random events and only picked the most significant ones. For instance, there could have been analyses of flint drops or iron golem drops, and only pearls and rods were reported due to those being the most significant—indeed, some other barter items, as well as eye of ender breaking rates, actually *were* recorded.

To correct for this, we take the probability of finding each result at least once among an upper bound $\hbar$ on the different types of events that could have been analyzed. Unfortunately, the correction used for selection across individuals and streams will not work here. That correction requires either independent or positively dependent probabilities; however, there are negatively dependent probabilities involved here. For instance, the more pearl barters you receive, the less opportunities there are to receive an obsidian barter: your numbers of pearl and obsidian barters are negatively correlated.

We can still correct for this, but it will require a much looser upper bound than the ones we have used previously. Remember that the probability of any one of a number of *mutually exclusive* events occurring is the sum of their probabilities—for example, the chance of rolling either a two or a five on a six-sided die is $\frac{1}{6}+\frac{1}{6}=\frac{2}{6}$.

However, this is not the case for non-mutually exclusive events. Consider the chance of rolling either a number less than three or an even number. The chance of rolling a number less than three (1 or 2) is $\frac{2}{6}$and the chance of rolling an even number (2, 4, or 6) is $\frac{3}{6}$. Adding these together would produce $\frac{5}{6}$. But this counts rolling a two *twice*, producing a number *higher* than the true probability of $\frac{4}{6}$.

This double-counting problem is the reason why adding together fails for probabilities that are not mutually exclusive, so it is not a problem that our probabilities are not mutually exclusive:

the sum of the probabilities will still work as an upper bound. Thus, we have the following[k], where $p_\hbar$ is the p-value corrected for $\hbar$ comparisons, and p is the initial p-value:

$$p_\hbar \leq ph \quad (6)$$

We will choose values for these formulas and compute the final results in Part IV. However, to ensure these computations are not invalid due to unusual behavior of Minecraft's random number generation, we will first analyze Minecraft's code.

[k]This is commonly known as the Bonferroni correction.

9 Code Analysis

When discussing probabilities this low, concerns about edge-case behavior in Minecraft's random number generator (RNG) are relevant. We have been working under the assumption that the results of piglin bartering and blaze drops are independent random variables, as one would naively expect if Minecraft's RNG were truly random. This would mean that the variables cannot affect one another; that is, past piglin barters and blaze drops tell you precisely nothing about future ones. However, it may seem possible that, in some edge cases, piglin barters or blaze drops fail independence in ways which increase the probability of observing Dream's data. Here, we will analyze how likely that is by inspecting Minecraft's code.

Before beginning the analysis, it is worth noting that if Minecraft's RNG were to fail in such a way that piglin barters and blaze drops could not be said to be approximately independent, it would *still* be astonishingly unlikely for them to fail in exactly the way required to produce the observed data. The failure(s) would need to (1) occur repeatedly over the course of six separate play sessions for Dream, (2) only occur to Dream out of all runners, (3) affect both bartering and blaze drops, and (4) specifically bias the results towards piglins bartering ender pearls and blazes dropping blaze rods, rather than towards some other barter item or blazes *not* dropping rods. Although this may still be more likely than the data occurring without a flaw in Minecraft's RNG, even before analyzing the code it appears a priori extremely unlikely.

9.1 Confirming the Probabilities

Though the probabilities we have been using thus far for piglin and blaze drop rates in Minecraft 1.16.1 are publicly available information, it is important to identify exactly where these probabilities come from. The piglin bartering proportions are determined by the piglin_bartering.json file found in the 1.16.1 jar file[I]. As expected, exactly once each barter, the game selects an item from the following weighted table:

Item	Weight	Item	Weight
Book	5	Ender Pearl	20
Iron Boots	8	String	20
Potion	10	Fire Charge	40
Splash Potion	10	Gravel	40
Iron Nugget	10	Leather	40
Nether Quartz	20	Nether Brick	40
Glowstone Dust	20	Obsidian	40
Magma Cream	20	Crying Obsidian	40
		Soul Sand	40

Table 1: The simplified contents of piglin_bartering.json. Here an item of weight n is n times more likely than an item of weight one. Additional information regarding enchantments, stack sizes, and potion effects not shown.

Since the weights sum to 423, and ender pearls have a weight of 20, the probability of an ender pearl barter is indeed $\frac{20}{423}$ as expected (in 1.16.1, the version Dream used).

[I]To read these files on Windows, simply rename 1.16.1.jar to 1.16.1.zip and navigate to data\minecraft\loot_tables.

Blaze drops are specified by a file called blaze.json, an excerpt of which is included below:

```
" function ": " minecraft : set_count ",
" count ": {
        "min": 0.0 ,
        "max": 1.0 ,
        " type ": " minecraft : uniform "
}
```

One can see that, when the player’s weapon does not have a looting enchantment, blazes select between dropping either 0 or 1 rods using a uniform distribution. Thus, a rod drop occurs with probability 0.5 as expected.

9.2 Setting RNG Seeds

Failures of one of Minecraft’s RNGs to behave randomly are not unheard of—the most famous examples of these are the RNG manipulation exploits found in versions prior to 1.13. These all work on the same principle: some part of Minecraft’s code resets an RNG being used by other parts of the code, causing predictable behavior. For example, in 1.12, loading a chunk runs some woodland mansion code which resets an RNG also used for mob spawns, allowing them to be manipulated given controlled conditions. A careful investigation of how and when Minecraft’s RNGs are seeded is needed to ensure there is no bias that could contribute to the observed data.

Many of Minecraft’s RNGs are initially seeded in a similar manner when they are created, including the RNGs used for bartering and blaze drops. Two components are used: an RNG called the "seed uniquifier", and the system time. The seed uniquifier is a linear congruential generator (LCG) which is seeded at 8682522807148012 when Minecraft is started up, and used (as well as updated) many of the times an RNG is seeded. (We will describe how LCGs work in the next sections.) The system time is a value provided by the computer’s operating system which, on most modern devices, represents the time in nanoseconds since the computer was turned on.

These two components are then combined using an operation referred to as bitwise XOR to produce a seed. (Normal) XOR, or exclusive or, is an operation which takes two boolean (TRUE or FALSE) values as inputs and returns TRUE if either *but not both* of the values are TRUE. Otherwise, it returns FALSE. Another way of looking at XOR is that it returns TRUE if the input values are different from each other and FALSE if they are not. *Bitwise* XOR takes two input numbers, which it represents as a binary string of bits. It then performs a XOR on the bits of each input number in a certain place value (interpreting 0 as FALSE and 1 as TRUE) to decide on that bit for the output number. For example, the bitwise XOR of 1001 (9 in decimal) and 1100 (12) is 0101 (5).

Combining a value with another by bitwise XOR cannot make it less random, provided the two values are independent. Although system time may not be *absolutely* independent of the seed uniquifier, they are clearly not related enough to cause problems. If the "randomness" decreases, it is not by much. Thus, as long as at least one of the input values is sufficiently random—and *both* of them ought to be, for our purposes—this method of setting a seed cannot cause issues.

The code implementing this procedure is shown below.

```
private static long seedUniquifier = 8682522807148012 L;

// seed a new Random object randomly
public Random () {
        this ( seedUniquifier () ^ System . nanoTime () ) ;
}

// set the seed of a Random object to a predetermined value public Random (
long seed ) {
        // implementation not relevant here
}

private static long seedUniquifier () {
    seedUniquifier = seedUniquifier * 181783497276652981 L ; return
    seedUniquifier ;
}
```

The RNG used for bartering is known as the "world RNG" since it is attached to the world object in Minecraft's code. The world RNG is also used by random ticks[m], mob spawning, and several other miscellaneous features. The fact that many different things utilize this RNG could cause concern that something might be setting the RNG to a specific state at an improper time—as it is in 1.12.

However, a careful search of every instance in which Minecraft sets any RNG's seed or creates any new RNGs reveals that the "world RNG" is never reset in 1.16. This was confirmed by noting that the world object is final (final objects in Java cannot be replaced, only modified) and examining every single call to Random::setSeed in the Minecraft codebase. None of these calls affected the Nether world RNG. Ultimately, this RNG is only ever set upon world load using the bitwise XOR between the seed uniquifier and the system time.

The situation for blaze rods is much simpler: every time a blaze is killed it creates an entirely new RNG for the blaze's single drop, using the same bitwise XOR procedure. As no possibility exists that external factors may be setting the seeds of these generators, we conclude that any failure of randomness must lie with Java Random itself. Having eliminated the most obvious possible cause of a potential glitch that would result in the observed data, we will now analyze Java's RNG closely to see if there are any more subtle causes of bias.

9.3 Linear Congruential Generators

If the reader is unfamiliar with the properties of modular arithmetic, they are encouraged to read https: // en. wikipedia. org/ wiki/ Modular_ arithmetic for at least a conceptual understanding.

Nearly all of Minecraft's randomization is computed through Java's pseudo-random number generator (PRNG), which utilizes a linear congruential generator[n]. As the name suggests, an

[m] *Tick* is a term used to refer to each individual update of the program state, often referred to as *frames* in other games. In Minecraft, the distinction must be drawn because screen rendering can occur more than once per tick. 20 ticks occur in one second when there is no lag. *Random tick* is jargon for a certain type of event that occurs on, as the name suggests, a random tick.

[n] https://en.wikipedia.org/wiki/Linear_congruential_generator

LCG generates a sequence of numbers through a discontinuous linear equation in which each term in the sequence is computed by performing operations on the previous term. Each time a new random value is needed, the LCG seed is updated according to the following equation before being used to generate an integer (or other kind of variable).

$$\text{seed}_{n+1} = a \cdot \text{seed}_n + b \bmod m \quad (7)$$

In the case of Java's PRNG, the constants a, b, and m in the above equation are given by a = 25214903917, b = 11, and $m = 2^{48}$. Note that these constants are specific to the Java LCG and will vary depending on the implementation.

$$\text{seed}_{n+1} = 25214903917 \cdot \text{seed}_n + 11 \bmod 2^{48} \quad (8)$$

LCGs are designed to be fast and memory-efficient rather than cryptographically secure; it is crucial to appropriately use the sequence so as to not cause unwanted bias. Notably, some LCGs have bad periodicity in their lower bits, and some fail what is called the spectral test[o]. That said, cryptographic security is a *much* higher standard than required to avoid the kinds of bias witnessed here. LCGs see frequent use for small Monte Carlo simulations like this one—a common piece of folklore is that sampling less than the square root of your modulus (in this case 2^{24}, or about 17 million) gives good samples. They are the go-to random number generator in a number of compilers and languages, notably C++11, Java, and the GCC compiler for C. Furthermore, as we will soon see, Java Random was specifically designed to mitigate the known weaknesses of LCGs.

9.4 Periodicity

As the Java Random LCG can only hold 2^{48} different seeds, the sequence of seeds must loop in at most 2^{48} calls to the randomizer (Java's values for a and b were chosen in part to guarantee it only loops after $2^{48} \approx$ 281 trillion calls). This is a problem for large computer simulations, but this scenario involves a number of RNG calls extremely small compared to the period of 2^{48}. However, the choice of 2^{48} as the modulus also means the LCG still demonstrates periodicity in its lower bits.

If two numbers are equivalent mod 2^{48}, then they must also be equivalent mod 2. This is intuitively clear if you think of the numbers represented in binary: if the last 48 bits of two numbers are the same, then each bit within those last 48—including the very last one—must be the same. Thus, we have:

$$\begin{aligned} \text{seed}_{n+1} &= 25214903917 \cdot \text{seed}_n + 11 \bmod 2 \\ &= (25214903917 \bmod 2) \cdot \text{seed}_n + (11 \bmod 2) \bmod 2 \\ &= \text{seed}_n + 1 \bmod 2 \end{aligned}$$

This makes it clear that seeds must alternate between even and odd. In fact, all bits repeat in a similar way with different periods. The first (lowest) bit has a period of two, the second has a period of four, the third has a period of eight, and in general the nth bit has a period of 2^n. The advantages of having a large period suddenly break down when making use of the lower

[o]Java Random and its Seed Uniquifier do not fail the spectral test in any dimensions we care about. In fact, at least the uniquifier was chosen for its good results on the spectral test http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.99.6553&rep=rep1&type=pdf, suggesting the Java authors were aware of the issue. The spectral test is also irrelevant here - interference from other game mechanics would remove the problem.

bits. Fortunately, as this behavior is well known, Java generates random integers in a way that accounts for this.

nextInt(n)[p]is the Java method used to generate random integers. When called, it returns a random non-negative integer less than n. The exact process Java uses to account for the aforementioned problem differs depending on whether n is a power of two.

- For non-powers of two (such as n = 423, used in bartering), nextInt(n) takes the 31 highest bits of the seed, modulo n. This implies that calls with even bounds will have some very clear dependence on lower bits of the seed—the lowest relevant one being the 18th, which has a period of $2^{18} \approx$ 262 thousand. It does, however, mean the bits with *particularly* small periods are entirely irrelevant.

- For powers of two, nextInt(n) simply returns the highest $\log_2(n)$ bits of the seed. For example, nextInt(2), which is used for blaze drops, returns the highest $\log_2(2)$ = 1 bit (the 48th bit) of the seed, which has a period of 2^{48}. This is done because it is much more effective than simply applying the method used for non-powers of two—nextInt(2) would then return the 18th bit, and have a period of only 2^{18}.

9.5 Bartering

As stated earlier, bartering randomization takes place using the world RNG. Thus, periodicity could conceivably be relevant, depending on precisely how often world RNG is called. The first time concerns due to periodicity would come into play (recall that n = 423 for bartering, which is not a power of two) is when the 18th bit loops after 2^{18} calls to the LCG. For example, when n is even[q], the parity (whether a number is even or odd) of a nextInt(n) call is always the same as the parity of the 18th bit. If it were possible to call the LCG precisely 2^{18} times per game tick, one could potentially use that fact to guarantee nextInt(n)'s output to either always be even or always be odd.

This specific attack has seen no use in the RNG manipulation scene so far because of its stringent requirements, but it is not immediately obvious that 2^{18} LCG calls per tick could not occur naturally. However, (1) they do not, and (2) it would not matter if they did.

1. Empirical evidence shows that the world RNG is called 6000 to 9000 times per tick in the Nether[r]. This value may depend on render distance and what is present in the given chunks, but it should never come close to 2^{18} (262144) calls in a natural setting. This means that even in the best possible case you would only expect a period to complete in around 30 ticks.

2. When n shares no factors with 2^{31}, the output of nextInt(n) does not depend on any one bit of the LCG seed.

Even if one could set the bottom bits of the world RNG to only take on certain values, the fact that 423 is odd (invertible mod 2^k) means that the upper bits would foil almost all bias introduced—at worst some outputs have just one more seed mapping to them than the others. Periodicity cannot explain the observed bias in Dream's pearl trades.

[p]https://docs.oracle.com/javase/8/docs/api/java/util/Random.html#nextInt-int [q]n also must not be a power of two.

[r]Readers who have been following the investigation may have heard about lava holding some relevance to possible RNG failures. This is because the abundance of lava in the Nether contributes significantly to those 6000 to 9000 RNG calls.

Before abandoning our discussion of Bartering and world RNG, it is worth noting that the extra calls to the world random can be viewed as an extra source of randomness. The chaotic nature of when Minecraft needs numbers, *especially* taken across both runs and streams, only adds entropy to the system. To expect all of these factors to conspire to increase the probability of anything unrelated is absurd—they are far more likely to *decrease* the probability of some other RNG failure occurring.

9.6 Blaze Drops

When blazes die, the generic method for all mobs that drop items, LivingEntity.dropLoot(), is called. However, the Random object used to determine the number of dropped items is not seeded and is instead left as null (meaning it has no value whatsoever—not even zero).

```
protected void dropLoot ( DamageSource source , boolean causedByPlayer ) { Identifier
        identifier = this . getLootTable () ;
        LootTable lootTable = this . world . getServer () . getLootManager () . getTable (
                identifier ) ;
        LootContext . Builder builder = this . getLootContextBuilder ( causedByPlayer
                , source ) ;
        lootTable . generateLoot ( builder . build ( LootContextTypes . ENTITY ) , this ::
                dropStack ) ;
}
```

LootContext.Builder.build() catches the null object and sets it to a new Random instance. In layman's terms, it realizes the object has no value and then gives it one using the procedure described in Section 9.2: Setting RNG Seeds.

```
Random random = this . random ;

if( random == null ) {
        random = new Random () ;
}
```

Essentially, both the number of nanoseconds since computer start-up and the result of a different LCG are used to seed the LCG that decides whether or not to drop a blaze rod. Furthermore, the state of the Uniquifier when the seeding occurs in turn is dependent on how many Random objects have been created by Minecraft since it was started (the value persists across worlds), which adds more entropy to they system.

The code then runs nextInt(2) to determine the number of blaze rods that should be dropped (0 or 1).

As a thought experiment, one could imagine freezing nanoTime() value through the operating system, but since nextInt(2) returns the highest bit of the seed, there would still be no visible dependence; to be informal, the seed uniquifier alone provides enough "randomness". Likewise, if one were to fix the uniquifier and let nanotime vary normally, the value of nextInt(2) would change every few microseconds or so[s]- fast enough that factors such as lag and run length should be considered a further major source of randomness in blaze drops. Regardless, it is almost certain both of these components were working as expected, so the Random objects used for Dream's blaze drops should not have had unusual correlations.

[s]This estimate comes from dividing 2^{47}, the desired average size of a change of the seed needed for a change in value, by the multiplier for Java Random. This estimate is only rough due to the "scrambler" in the Random constructor, but is easily seen to hold up in practice.

Finally, given their vastly different implementations and separate code bases, it is reasonable to assume that a bug could not simultaneously affect both bartering and blaze rod drops. In fact, even in the (hilariously unlikely) occurrence of cosmic ray interference, two independent bit flips would need to occur at bare minimum for it to have an effect on both.

Part IV
Results

10 Computation

10.1 Naive Estimate

The initial computations performed by community members directly applied the binomial distri bution without considering the biases present in the data. To reiterate, what follows are naive estimations operating under faulty assumptions we have spent the entire paper accounting for. The criticisms of directly applying these computations to this situation are legitimate, but they are still meaningful and relevant *if they are properly interpreted and their inaccuracies are kept in mind*. First, for pearls:

$$P(X_{\text{pearls}} \geq 42) \approx \sum_{k=42}^{262} \binom{262}{k} (0.0473)^{k}(1 - 0.0473)^{262-k} \approx 5.65 \times 10^{-12} \quad (9)$$

This is Shell Guy's "1 in 40 billion" number often referenced, which more accurately should be 1 in 177 billion. Then, for rods:

$$P(X_{\text{rods}} \geq 211) \approx \sum_{k=211}^{305} \binom{305}{k} (0.5)^{k}(1 - 0.5)^{305-k} \approx 8.79 \times 10^{-12} \quad (10)$$

For independent discrete random variables, the probability of two events occurring simultaneously is the product of their individual probabilities. Thus,

$$P(X_{\text{pearls}} \geq 42 \cap X_{\text{rods}} \geq 211) \approx (0.00000000000565319)\,(0.00000000000879143) \approx 0.000000000000000000000049699624 \approx 4.9699624 \cdot 10^{-23} \quad (11)$$

This is 1 in 20 sextillion. The idea something this unlikely occurred is obviously ridiculous. The next section actually applies what we have discussed in the previous sections to compute a statistically rigorous upper bound on the probability.

10.2 Full Computation

The code implementing these calculations can be found in Appendix C.

10.2.1 Pearls

Using the method described in Section 8: Addressing Bias, we may compute an upper bound on the p-value. It is worth noting that this upper bound is conservative, and using it as if it were an exact p-value would result in an analysis that would be extremely biased in Dream's favor.

First, we examine the pearl odds. Using the stopping rule correction from Section 8.1, we obtain a value of 1.22×10^{-11}. Using this, we can bound the probability of getting this p-value or less in a sequence of 11 streams (the number of streams that Dream did), as follows:

$$1 - 1 - 1.22 \times 10^{-11} \quad {}_{11(11+1)}$$

$$^{2} = 8.04 \times 10^{-10} \quad (12)$$

8.04×10^{-10} represents the *highest possible probability* of Dream *ever*, while on stream, getting pearl barter rates as exceptionally good as the ones in the data. Finally, we account for the chance across one thousand runners, a generous upper bound on the number of runners who would be examined in a similar way:

$$1 - 1 - 8.04 \times 10^{-10\ 1000} = 8.04 \times 10^{-7} \quad (13)$$

8.04×10^{-7}represents the probability that *any active runner in the Minecraft speedrun ning community would ever experience events as rare as Dream*, at some point within his 11 streams, experiencing luck extreme enough that a malicious investigator would be able to portray it as being as rare as Dream's bartering luck[t].

This should not be equivocated with the probability that Dream experiences luck as extreme as he did on any *particular* occasion, which is far lower. This number describes how surprising it is for this to *ever* happen—not how surprising it is for it to happen in a given instance. Although events with a one in a million probability may occur sometimes, it is almost always when there are *multiple chances* for them to occur—this number has *already accounted* for the multiple chances for Dream (or anyone else) to experience luck this exceptional.

10.2.2 Blaze Rods

Like with pearls, we again apply the stopping rule correction. This yields a p-value of 4.72×10^{-11}. Unlike with the pearl drops, this is our final number. As mentioned previously, blaze rods are not subject to selection bias across streams or runners, as Dream's blaze rod drops were examined only because of his pearl rates. Potential selection across RNG types by a malicious investigator is still a problem, but this will be addressed in the combined p-value.

[t]As we account for p-hacking using the combined p-value rather than each individual p-value, technically you must include the assumption that the malicious investigator only looks at one type of RNG. This assumption will be addressed in the final p-value.

10.2.3 Combined Number

p-values may be combined by Fisher's method[u]. Fisher's method involves taking −2 times the sum of the *log*s of the p-values:

$$z = -2\log(p_{\text{rods}}) + \log p_{\text{pearls}} \quad (14)$$

This value is known to follow a χ^2 distribution with 4 degrees of freedom[v]. Hence, we may use the χ^2 CDF to get a combined p-value. Taking $\chi^2{}_k(z)$ to mean the CDF of a χ^2

distribution with k degrees of freedom at z, we have:

$$p_{\text{combined}} = 1 - \chi^2_4(z) \approx 1.472\,782 \times 10^{-15} \quad (15)$$

We may then correct this value for potential investigatory bias as described in Section 8.4. Note that there are not many obvious RNG targets for a runner to cheat. The two most obvious are pearls and blaze rods, but beyond that, it becomes less clear. Obsidian is an option, possibly allowing for nether travel. Another option is string, which runners have recently started hoping for with the advance of “hypermodern” strats that involve skipping villages—but hypermodern strats were not well-developed during the time that Dream ran, and Dream did not go for string. Other potential options include flint rates from gravel, iron amounts from iron golems, and eye of ender breaks, but these are not as obvious or as advantageous as the other options. As a generous number, we use 10 for the number of likely targets. We have 10 options for our first source of RNG, and another 9 for the second (not 10 anymore, as the same source cannot be used twice), giving us 90 possible options:

$$p_{\text{final}} = p_{\text{combined}} \times 90 = 1.325\,504 \times 10^{-13} \quad (16)$$

This is about 1 in 7.5 trillion. As stated earlier, this should not be equivocated to the probability Dream got this lucky in a given instance, as it already accounts for many other factors beyond that. This is a *loose* (i.e., almost certainly an overestimate) upper bound on the chance that *anyone* in the Minecraft speedrunning community would *ever* get luck comparable to Dream’s (adjusted for how often they stream).

[u]https://en.wikipedia.org/wiki/Fisher’s_method

[v]Fisher, R. A. (1992). Statistical methods for research workers. In Breakthroughs in statistics (pp. 66-70). Springer, New York, NY.

11 Conclusion

In our analysis, we were able to conclude the following statements:

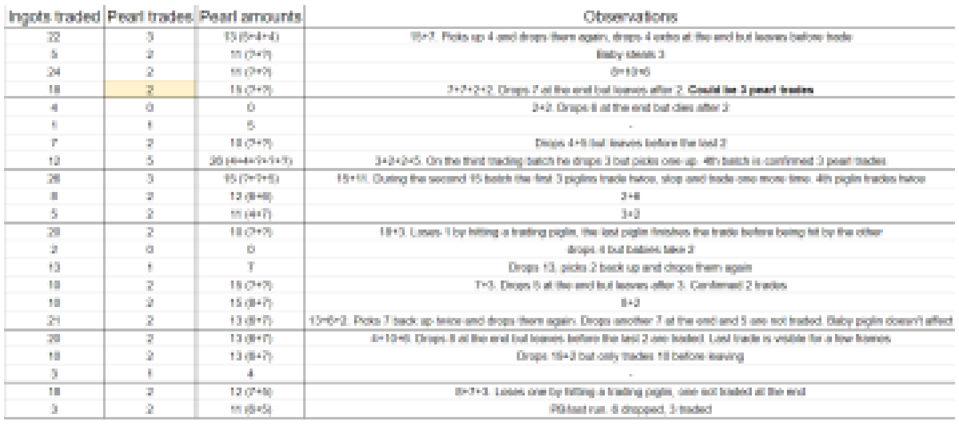
- The events that were observed on Dream’s stream cannot be modeled by any sensible, conventional probability distribution.
- After accounting for any contributors of bias, the likelihood of this occurring is still unfathomably small.
- There are no circumstances in a natural setting in which bartering and blaze drops could be dependent or biased to any notable degree, much less a degree strong

enough to produce this result.

• There is no way to accidentally manipulate these values in real time during an RSG speedrun, nor any conceivable way to do it intentionally using only conventional methods.

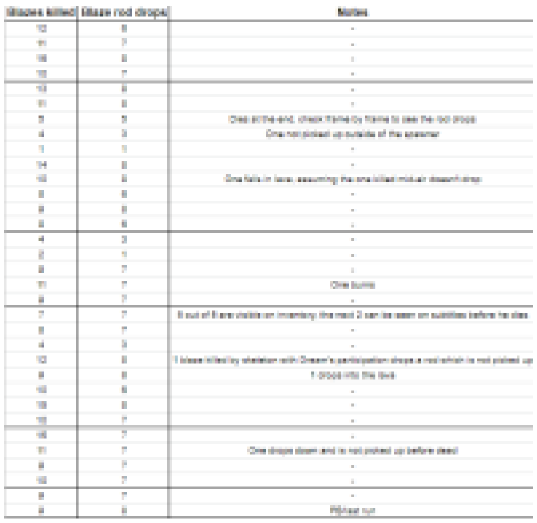
The only sensible conclusion that can be drawn after this analysis is that Dream's game was modified in order to manipulate the pearl barter and rod drop rates.

A Raw Data



Ingots traded	Pearl trades	Pearl amounts	Observations
22	3	13 (5+4+4)	15+7. Picks up 4 and drops them again, drops 4 extra at the end but leaves before trade
5	2	14 (7+7)	Baby steals 3
24	2	14 (7+7)	8+10+6
18	2	15 (7+8)	7+7+2+2. Drops 7 at the end but leaves after 2. **Could be 3 pearl trades**
4	0	0	2+2. Drops 6 at the end but dies after 2
1	1	5	-
7	2	14 (7+7)	Drops 4+5 but leaves before the last 2
12	5	26 (4+4+7+4+7)	3+2+2+5. On the third trading batch he drops 3 but picks one up. 4th batch is confirmed 3 pearl trades
26	3	15 (7+3+5)	15+11. During the second 15 batch the first 3 piglins trade twice, stop and trade one more time. 4th piglin trades twice
8	2	12 (6+6)	2+6
5	2	11 (4+7)	3+2
20	2	14 (7+7)	18+3. Loses 1 by hitting a trading piglin, the last piglin finishes the trade before being hit by the other
2	0	0	drops 4 but babies take 2
13	1	7	Drops 13, picks 2 back up and drops them again
10	2	15 (7+8)	7+3. Drops 5 at the end but leaves after 3. Confirmed 2 trades
10	2	15 (8+7)	8+2
21	2	13 (6+7)	13+6+2. Picks 7 back up twice and drops them again. Drops another 7 at the end and 5 are not traded. Baby piglin doesn't affect
20	2	13 (6+7)	4+10+6. Drops 8 at the end but leaves before the last 2 are traded. Last trade is visible for a few frames
18	2	13 (6+7)	Drops 18+2 but only trades 18 before leaving
3	1	4	-
18	2	12 (7+5)	8+7+3. Loses one by hitting a trading piglin, one not traded at the end
3	2	11 (6+5)	PB last run. 6 dropped, 3 traded

Figure 3: Dream’s pearl barter data. Timestamps and data from other runners at http://bombch.us/DPPU.



Blazes killed	Blaze rod drops	Notes
12	8	-
11	7	-
18	8	-
10	7	-
10	6	-
11	6	-
5	5	Dies at the end, check frame by frame to see the rod drops
4	3	One not picked up outside of the spawner
1	1	-
14	8	-
10	6	One falls in lava, assuming the one killed mid-air doesn't drop
8	6	-
8	6	-
8	6	-
4	3	-
2	1	-
8	7	-
11	7	One burns
8	7	-
7	7	6 out of 8 are visible on inventory, the next 2 can be seen on subtitles before he dies
8	7	-
4	3	-
12	8	1 blaze killed by skeleton with Dream's participation drops a rod which is not picked up
8	8	1 drops into the lava
10	6	-
18	8	-
10	7	-
16	7	-
11	7	One drops down and is not picked up before dead
8	7	-
10	7	-
8	7	-
8	6	PB last run

Figure 4: Dream’s blaze drop data. Timestamps and data from other runners at http://bombch.us/DPPV.

B Stopping Rule Computation Algorithm

This section will describe the algorithm used to calculate the probability of a sequence of some number of Bernoulli trials will have at any point had a binomial CDF p-value below a certain cutoff. As with the other computations, the code implementing this algorithm can be found in Appendix C.

The critical observation to understanding the algorithm used is that sequences with the same number of successes in the first n trials can be treated as essentially equivalent in some ways. Critically, their p-value (after the first n trials) will be equal: the binomial CDF calculation only considers the number of successes and the length of the sequence. Due to these equivalences, we can calculate these groups of sequences as a package-deal without considering the individual sequences.

1/1					
1/2	1/2				
1/4	2/4	1/4			
1/8	3/8	3/8	1/8		
1/16	4/16	6/16	4/16	1/16	
1/32	5/32	10/32	10/32	5/32	1/32

This table depicts the probability masses of various sequences of blaze rod drop trials, grouped based on how many successes they had in the first few trials of the sequence. Some readers may recognize a resemblance to Pascal's triangle, which famously is closely connected to the binomial coefficient.

Rows in this table, zero-indexed, represent the number n of trials at the start of a sequence which are being considered, and columns, also zero-indexed, represent the number k of successes which have occurred in the first n trials. The numbers in a certain row and column are the probability that, within the first n trials, k successes occur.

As you move down the table, you can imagine each number giving some of its probability mass to the number directly below it (which represents the next trial being a failure, and thus not increasing the number of successes) and the number below and to the right (which represents the next trial being a success, and thus increasing the number of successes by one). Since we are discussing blazes, it will be split evenly; with bartering, it is split about 95% down and 5% down-right.

To give a concrete example, the second "3/8" received a probability mass of $\frac{2}{4}\times\frac{1}{2}=\frac{2}{8}$from the $\frac{2}{4}$to the up-left and a probability mass of $\frac{1}{4}\times\frac{1}{2}=\frac{1}{8}$from the $\frac{1}{4}$directly up. This represents the two ways two successes in three trials can be achieved: having one success in two trials, followed by an additional success, or having two successes in two trials, followed by a failure. It should be fairly simple to check by hand that, of the eight possible sequences of three successes or failures a sequence might begin with, there are indeed three which have exactly two successes[w].

Next, consider the bolded 1/32 on the bottom. This passes a p-value cutoff of p = 0.05, which we will use for the purposes of this example due to the actual p-value being impractically small. Since all the sequences which have five successes in the first five trials will have met our p-value cutoff at some point, we will remove these sequences from the table before continuing and add the entire probability mass of $\frac{1}{32}$ to our stopping-rule corrected p-value before generating the next line.

[w]Since the chances of success and failure are equal for blaze rods, each sequence is equally likely. With bartering, on the other hand, not all sequences are equally likely and this method of checking would not work.

1/16	4/16	6/16	4/16	1/16		
1/32	5/32	10/32	10/32	5/32	0/32	
1/64	6/64	15/64	20/64	15/64	5/64	0/64

Notice how the table no longer follows Pascal's triangle; the last two fractions no longer represent the probability of a sequence with five or six successes in the first six trials, but the probability of such a sequence which also *did not have a p-value over the cutoff at any point in the past*. This is done to prevent double counting. For example, we already added the probability mass of the sequences beginning with six successes to the adjusted p-value when we added that of the sequences beginning with *five* successes, so we do not need to add that probability mass a second time.

This process continues until we reach the desired number of trials. An important note is that you cannot simply look at whether one of the fractions is below the p-value cutoff: that is not the probability that k or more successes are gotten in n trials. Instead, we calculate the number of successes required to get a p-value below the cutoff for every number of trials before the total amount, and then compare to the relevant number of successes at each step.

C Probability Computations

```
import java . math . BigDecimal ;
import java . math . RoundingMode ;
import org . apache . commons . math3 . distribution . ChiSquaredDistribution ;

class Corrections {

static int SCALE = 100;
static int NUM_TRIALS , NUM_SUCCESSES ;
static BigDecimal P_SUCCESS , P_FAIL ;

public static void main ( String [] args ) {
System . out . println (" --- ENDER PEARLS ---") ;
double pearlStoppingRule = shiftyInvestigator (262 , 42 , " 0.0473 ") ;
```

```
System . out . println (" Stopping rule : " + pearlStoppingRule ) ;
double pearlStreamBias = 1 - Math . exp ( Math . log1p ( - pearlStoppingRule ) * 66) ;
System . out . println (" Stopping rule + stream selection bias : " + pearlStreamBias ) ;
double pearlRunnerBias = 1 - Math . exp ( Math . log1p ( - pearlStreamBias ) * 1000) ;
System . out . println (" Stopping rule + stream selection bias + runner selection bias : " + pearlRunnerBias + "\n") ;

System . out . println (" --- BLAZE RODS ---") ;
double rodStoppingRule = shiftyInvestigator (305 , 211 , " 0.5") ;
System . out . println (" Stopping rule : " + rodStoppingRule + "\n") ;

System . out . println (" --- FINAL PROBABILITY ---") ;
double fisherMethod = -2 * ( Math . log ( pearlRunnerBias ) + Math . log ( rodStoppingRule ) ) ;
ChiSquaredDistribution csd = new ChiSquaredDistribution (4) ;
double cdf = 1 - csd . cumulativeProbability ( fisherMethod ) ;
double pHacking = cdf *= 90;
System . out . println ( pHacking ) ;
}

static double shiftyInvestigator ( int trials , int successes , String p ) {
Corrections . NUM_TRIALS = trials ;
Corrections . NUM_SUCCESSES = successes ;
Corrections . P_SUCCESS = new BigDecimal ( p ) . setScale ( SCALE , RoundingMode . CEILING ) ;
Corrections . P_FAIL = BigDecimal . ONE . subtract ( P_SUCCESS ) . setScale ( SCALE , RoundingMode . CEILING ) ;

BigDecimal targetP = BigDecimal . ONE . subtract ( binomCDF ( NUM_TRIALS , NUM_SUCCESSES - 1) ) ;

```

```
int [] significantCutoffs = new int [ NUM_TRIALS + 1];
significantCutoffs [0] = 0;
for ( int observedTrials = 1; observedTrials <= NUM_TRIALS ; observedTrials ++) {
boolean foundCutoff = false ;
for ( int sucTrials = significantCutoffs [ observedTrials - 1]; ! foundCutoff
&& sucTrials <= observedTrials ; sucTrials ++) {
foundCutoff = ( BigDecimal . ONE . subtract ( binomCDF ( observedTrials , sucTrials - 1) )
. compareTo ( targetP ) <= 0) ;
if ( foundCutoff )
significantCutoffs [ observedTrials ] = sucTrials ;
}
if (! foundCutoff )
significantCutoffs [ observedTrials ] = observedTrials ;
}

BigDecimal [] lastRow = { BigDecimal . ONE , BigDecimal . ZERO };
for ( int N = 1; N <= NUM_TRIALS ; N ++) {
lastRow = genNthRowOfPascalWithCutoffs (N , lastRow , significantCutoffs ) ;
}

BigDecimal total = BigDecimal . ZERO ;
for ( BigDecimal b : lastRow ) {
total = total . add ( b ) . setScale ( SCALE , RoundingMode . CEILING ) ;
}

return BigDecimal . ONE . subtract ( total ) . doubleValue () ;
}

static BigDecimal nCr (int n , int r ) {
if ( r > n / 2)
r = n - r ;
```

```

BigDecimal answer = BigDecimal . ONE . setScale ( SCALE , RoundingMode . CEILING ) ;
for ( int i = 1; i <= r ; i ++) {
answer = answer . multiply (new BigDecimal ( n - r + i ) . setScale ( SCALE , RoundingMode . CEILING ) ) ;
answer = answer . divide (new BigDecimal ( i ) . setScale ( SCALE , RoundingMode . CEILING ) , RoundingMode . CEILING ) ;
}
return answer ;
}

static BigDecimal binomialPDF (int n , int k ) {
return nCr (n , k ) . multiply ( P_SUCCESS . pow ( k ) ) . multiply ( P_FAIL . pow ( n - k ) ) . setScale ( SCALE , RoundingMode . CEILING ) ;
}
```

```

static BigDecimal binomCDF (int n , int maxSuccessesInc ) {
BigDecimal answer = BigDecimal . ZERO . setScale ( SCALE , RoundingMode . CEILING ) ;
for ( int i = 0; i <= maxSuccessesInc ; i ++) {
answer = answer . add ( binomialPDF (n , i ) ) . setScale ( SCALE , RoundingMode . CEILING ) ;
}
return answer ;
}

static BigDecimal [] genNthRowOfPascalWithCutoffs ( int N , BigDecimal [] lastRow , int [] significantCutoffs ) {
BigDecimal [] result = new BigDecimal [ N + 2];
result [0] = lastRow [0]. multiply ( P_FAIL ) . setScale ( SCALE , RoundingMode . CEILING ) ;
for ( int i = 1; i < N + 2; i ++) {
if ( i <= significantCutoffs [ N ]) {
result [ i ] = lastRow [ i - 1]. multiply ( P_SUCCESS ) . add ( lastRow [ i ]. multiply ( P_FAIL ) ) . setScale ( SCALE ,
RoundingMode . CEILING ) ;
} else {
result [ i ] = BigDecimal . ZERO ;
}
}
return result ;
}
}
```