A Note on Randomness Extraction

This version of the document is dated 2021-04-13.

Peter Occil

Randomness extraction (also known as *unbiasing*, *debiasing*, *deskewing*, *whitening*, or *entropy extraction*) is a set of techniques for generating unbiased random bits from biased sources. This note covers some useful extraction techniques.

1 In Information Security

In information security, randomness extraction serves to generate a seed, password, encryption key, or other secret value from hard-to-predict nondeterministic sources.

Randomness extraction for information security is discussed in NIST SP 800-90B sec. 3.1.5.1, and RFC 4086 sec. 4.2 and 5.2. Possible choices of such extractors include keyed cryptographic hash functions (see, e.g., (Cliff et al., 2009)[(1)]; (Coretti et al., 2019)[(2)]) and two-universal hash functions with a fixed but randomly chosen seed (Frauchiger et al., 2013)[(3)]. In information security applications:

- Unkeyed hash functions and other unkeyed extraction functions should not be used by themselves in randomness extraction.
- Lossless compression should not be used as a randomness extractor.
- Where possible, there should be two or more independent nondeterministic sources from which to apply randomness extraction (McInnes and Pinkas 1990)[(4)].

Some papers also refer to two-source extractors and resilient functions (especially the works by E. Chattopadhyay and D. Zuckerman), but there are few if any real implementations of these extraction techniques.

> **Example:** The Cliff reference reviewed the use of HMAC (hash-based message authentication code) algorithms, and implies that one way to generate a seed is as follows:
>
> 1. Gather data with at least 512 bits of entropy.
> 2. Run HMAC-SHA-512 with that data to generate a 512-bit HMAC.
> 3. Take the first 170 (or fewer) bits as the seed (512 divided by 3, rounded down).

2 Outside of Information Security

Outside of information security, randomness extraction serves the purpose of recycling randomly generated numbers or, more generally, to transform those numbers from one form to another while preserving their randomness. This can be done, for example, to reduce calls to a pseudorandom number generator (PRNG) or to generate a new seed for such a PRNG.

Perhaps the most familiar example of randomness extraction is the one by von Neumann (1951)[(5)], which works if "independence of successive [coin] tosses is assumed"[(6)]:

1. Flip a coin twice (whose probability of heads is unknown).
2. If the coin lands heads then tails, return heads. If it lands tails then heads, return tails. If neither is the case, go to step 1.

An algorithm found in (Morina et al. 2019)[7] (called **Algorithm M** in this note) extends this to loaded dice. According to personal communication with K. Łatuszyński, the key "is to find two non overlapping events of the same probability" via "symmetric events {X_1 < X_2} and {X_2 < X_1} that have the same probability".

1. Throw a (loaded) die, call the result X. Throw the die again, call the result Y.
2. If X is less than Y, return 0. If X is greater than Y, return 1. If neither is the case, go to step 1.

Algorithm M in fact works in a surprisingly broad range of cases; for more, see the **appendix**.

Pae (2005)[8] and (Pae and Loui 2006)[9] characterize *extracting functions*. Informally, an *extracting function* is a function that maps a fixed number of digits to a variable number of bits such that, whenever the input has a given number of ones, twos, etc., every output bit-string of a given length is as likely to occur as every other output bit-string of that length, regardless of the input's probability of zero or one.[10] Among others, von Neumann's extractor and the one by Peres (1992)[11] are extracting functions. The Peres extractor takes a list of bits (zeros and ones generated from a "coin" with a given probability of heads) as input and is described as follows:

1. Create two empty lists named U and V. Then, while two or more bits remain in the input:
    1. If the next two bits are 0/0, append 0 to U and 0 to V.
    2. Otherwise, if those bits are 0/1, append 1 to U, then write a 0.
    3. Otherwise, if those bits are 1/0, append 1 to U, then write a 1.
    4. Otherwise, if those bits are 1/1, append 0 to U and 1 to V.
2. If U is not empty, do a separate (recursive) run of this algorithm, reading from the bits placed in U.
3. If V is not empty, do a separate (recursive) run of this algorithm, reading from the bits placed in V.

A streaming algorithm, which builds something like an "extractor tree", is another example of a randomness extractor (Zhou and Bruck 2012)[12].

I maintain **source code of this extractor and the Peres extractor**, which also includes additional notes on randomness extraction.

Pae's "entropy-preserving" binarization (Pae 2020)[13], given below, is meant to be used in other extractor algorithms such as the ones mentioned above. It assumes the number of possible values, n, is known. However, it is obviously not efficient if n is a large number.

1. Let f be a number in the interval $[0, n)$ that was previously randomly generated. If f is greater than 0, output a 1 (and go to step 2).
2. If f is less than $n - 1$, output a 0 x times, where x is $(n - 1) - f$.

Some additional notes:

1. Different kinds of random numbers should not be mixed in the same extractor stream. For example, if one source outputs random 6-sided die results, another source outputs random sums of rolling 2 six-sided dice, and a third source outputs

coin flips with a probability of heads of 0.75, there should be three extractor streams (for instance, three extractor trees that implement the Zhou and Bruck algorithm).

2. Hash functions, such as those mentioned in my **examples of high-quality PRNGs**, also serve to produce random-behaving numbers from a variable number of bits. In general, they can't be extracting functions; however, as long as their output has more bits than used to produce it, that output can serve as input to an extraction algorithm.
3. Peres (1992)**(11)** warns that if a program takes enough input bits (such as flips of a coin with unknown probability of heads) so that the extracting function outputs *m* bits with them, those *m* bits will not be uniformly distributed. Instead, the extracting function should be passed blocks of input bits, one block at a time (where each block should have a fixed length of at least 2 bits), until *m* bits or more are generated by the extractor this way.
4. Extractors that maintain state, such as the Zhou and Bruck extractor tree, should be used only on sources whose distribution does not change significantly over time. Dividing the source into blocks, as in the previous note, and assigning one extractor instance to each block, can improve robustness for sources whose distribution can change over time.
5. The lower bound on the average number of coin flips needed to turn a biased coin into an unbiased coin is as follows (and is a special case of the *entropy bound*; see, e.g., (Pae 2005)**(8)**, (Peres 1992)**(11)**): ln(2) / ((λ − 1) * ln(1 − λ) − λ * ln(λ)), where λ is the probability of heads of the input coin and ranges from 0 for always tails to 1 for always heads. According to this formula, a growing number of coin flips is needed if the input coin strongly leans towards heads or tails. (For certain values of λ, Kozen (2014)**(14)** showed a tighter lower bound of this kind, but this bound is non-trivial and assumes λ is known.)

Devroye and Gravel (2020)**(15)** suggest a special randomness extractor to reduce the number of random bits needed to produce a batch of samples by a sampling algorithm. The extractor works based on the probability that the algorithm consumes X random bits to produce a specific output Y. Since the algorithm seems not to be well developed, I discuss this extractor in detail elsewhere, in "**Miscellaneous Notes on Randomization**".

3 Notes

- (1) Cliff, Y., Boyd, C., Gonzalez Nieto, J. "How to Extract and Expand Randomness: A Summary and Explanation of Existing Results", 2009.
- (2) Coretti, S., Dodis, Y., et al., "Seedless Fruit is the Sweetest: Random Number Generation, Revisited", 2019.
- (3) Frauchiger, D., Renner, R., Troyer, M., "True randomness from realistic quantum devices", 2013.
- (4) McInnes, J. L., & Pinkas, B. (1990, August). On the impossibility of private key cryptography with weakly random keys. In Conference on the Theory and Application of Cryptography (pp. 421-435).
- (5) von Neumann, J., "Various techniques used in connection with random digits", 1951.
- (6) However, this method and Peres's extractor also works if the coin tosses are *exchangeable*, which roughly means that changing the order of the tosses doesn't change their overall probability of heads (Peres 1992).
- (7) Morina, G., Łatuszyński, K., et al., "**From the Bernoulli Factory to a Dice Enterprise via Perfect Sampling of Markov Chains**", arXiv:1912.09229 [math.PR], 2019.
- (8) Pae, S., "Random number generation using a biased source", dissertation, University of Illinois at Urbana-Champaign, 2005.
- (9) Pae, S., Loui, M.C., "Randomizing functions: Simulation of discrete probability distribution using a source of unknown distribution", *IEEE Transactions on Information Theory* 52(11), November 2006.

- [10] It follows from this definition that an extracting function must map an all-X string (such as an all-zeros string) to the empty string, since there is only one empty string but more than one string of any other length. Thus, no reversible function can be extracting, and a function that never returns an empty string (including nearly all hash functions) can't be extracting, either.
- [11] Peres, Y., "**Iterating von Neumann's procedure for extracting random bits**", Annals of Statistics 1992,20,1, p. 590-597.
- [12] Zhou, H. and Bruck, J., "**Streaming algorithms for optimal generation of random bits**", arXiv:1209.0730 [cs.IT], 2012.
- [13] S. Pae, "**Binarization Trees and Random Number Generation**", arXiv:1602.06058v2 [cs.DS].
- [14] Kozen, D., **"Optimal Coin Flipping"**, 2014.
- [15] Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020.
- [16] Montes Gutiérrez, I., "Comparison of alternatives under uncertainty and imprecision", doctoral thesis, Universidad de Oviedo, 2014.
- [17] De Schuymer, Bart, Hans De Meyer, and Bernard De Baets. "A fuzzy approach to stochastic dominance of random variables", in *International Fuzzy Systems Association World Congress* 2003.
- [18] Camion, Paul, "Unbiased die rolling with a biased die", North Carolina State University. Dept. of Statistics, 1974.

4 Appendix

4.1 On Algorithm M

Algorithm M works regardless of what numbers X and Y can take on and with what probability, and even if the "dice" for X and Y are loaded differently, as long as the chance that the first "die" shows a number less than the second "die" is the same as the chance that the first "die" shows a greater number, and as long as each *pair* of throws is independent of any other.

More formally, $P(X < Y)$ must be equal to $P(X > Y)$. This relationship is equivalent to *statistical indifference* (Montes Gutiérrez 2014)**(16)**, (De Schuymer et al. 2003)**(17)**. This relationship works even if X and Y are dependent on each other but independent of everything else; this is easy to see if we treat X and Y as a single random "vector" $[X, Y]$. This is shown by the following two propositions. In the propositions below, a random variable is *non-degenerate* if it does not take on a single value with probability 1.

Proposition 1. *Let X and Y be real-valued non-degenerate random variables. Then Algorithm M outputs 0 or 1 with equal probability if and only if X and Y are statistically indifferent.*

Proof. For any X and Y there are only three mutually exclusive possibilities, $X>Y$, $Y>X$, and $X=Y$. Because both random variables are nondegenerate, $P(X>Y)$ or $P(Y>X)$ or both are nonzero, and $P(X=Y) < 1$. For the algorithm to return 0, X must be less than Y, and for it to return 1, X must be greater than Y.

For the "only if" part: For the algorithm to return 0 or 1 with equal probability, it must be that $P(X>Y) = P(Y>X)$. But this necessarily means that $P(X>Y)$ and $P(Y>X)$ are both 1/2 or less. And if we assign half of the remainder (the remainder being $P(X=Y)$) to each probability, we get—

- $P(X>Y) + P(X=Y)/2 = 1/2$, and
- $P(Y>X) + P(X=Y)/2 = 1/2$,

and thus, X and Y must be statistically indifferent by definition (see below).

For the "if" part: If X and Y are statistically indifferent, this means that $\alpha = P(X>Y) + P(X=Y)/2$ and $\beta = P(Y>X) + P(X=Y)/2$ are equal and $\alpha = \beta = 1/2$. Since both α and β are equal and $P(X=Y)$ in α and β are also equal, this must mean that $P(X>Y) = P(Y>X)$. It thus follows that for X and Y, the algorithm will return 0 or 1 with equal probability. □

Proposition 2. *Let X and Y be real-valued non-degenerate random variables that are independent, identically distributed, and defined on the same probability space. Then X and Y are statistically indifferent.*

Proof. By definition, X and Y are statistically indifferent if and only if X is statistically preferred to Y and vice versa (that is, $P(X>Y) + P(X=Y)/2 >= P(Y>X) + P(Y=X)/2$) (De Schuymer et al. 2003)**(17)**. Because both random variables are nondegenerate, $P(X>Y)$ or $P(Y>X)$ or both are nonzero, and $P(X=Y) < 1$. Moreover, because both random variables are identically distributed, their distribution functions F_X and F_Y are the same, and therefore their values and expectations for any given z (e.g., $F_X(z)$ and $E[F_X(z)]$, respectively) are the same.

If we look at Theorem 3.12 in (Montes Gutiérrez 2014)**(16)**, we see that we can replace—

- the left hand side of Equation 3.5 with $0 - 0$, since it's a difference of expectations of the same distribution function and random variable, and
- the right hand side with $(1/2) * 0$, since the difference of $P(X =Y)$ and $P(X = X')$ is taken and $P(X =Y)$ is equivalent to $P(X = X')$, which is equivalent because X, X' and Y are identically distributed by the hypotheses of this proposition and Theorem 3.12.

As a result, Equation 3.5 becomes $0 >= 0$, which is true and thus establishes that X is statistically preferred to Y (by Theorem 3.12). It thus trivially follows that Y is likewise statistically preferred to X once we replace the roles of both variables, since both variables are identically distributed. As a result, X and Y are found to be statistically indifferent and the proposition is proved. □

Here are some of the many examples where this algorithm works:

- Set X and Y to two independent Gaussian random numbers with a mean of 0 but a different standard deviation. Or...
- Set X and Y to two independent uniform(0, 1) random numbers. Or...
- Set X and Y to two independent uniform(0, 1) random numbers, then set Y to $(X+Y)/2$.

See also a procedure given as a remark near the end of a paper by Camion (1974)**(18)**.

5 License