A Note on Randomness Extraction

This version of the document is dated 2020-09-06.

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)]) and two-universal hash functions with a fixed but randomly chosen seed (Frauchiger et al., 2013)[(2)]. 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.

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)[(3)]:

1. Flip a coin twice (whose bias 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 from (Morina et al. 2019)[4] extends this to loaded dice. Based on personal communication by K. Łatuszyński, perhaps this works for any distribution of random numbers, not just loaded dice, as 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 die twice (whose bias is unknown), call the results X and Y, respectively.
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.

Pae (2005)[5] and (Pae and Loui 2006)[6] 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 bias.[7] Among others, von Neumann's extractor and the one by Peres (1992)[8] are extracting functions. The Peres extractor takes a list of bits (zeros and ones with the same bias) as input and is described as follows:

1. Create two empty lists named U and V. Then, while two or more bits remain:
    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. Run this algorithm recursively, with the bits placed in U.
3. Run this algorithm recursively, with 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)[9].

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)[10], 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:

- 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 bias of 0.75, there should be three extractor streams (for instance, three extractor trees that implement the Zhou and Bruck algorithm).
- 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, their output can serve as input to an extraction algorithm.

- Peres (1992)[8] warns that if a program takes enough biased bits 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 biased 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.
- 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)[5], (Peres 1992)[8]): $\ln(2) / ((\lambda - 1) * \ln(1 - \lambda) - \lambda * \ln(\lambda))$, where λ is the bias 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 is strongly biased towards heads or tails.

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] Frauchiger, D., Renner, R., Troyer, M., "True randomness from realistic quantum devices", 2013.
- [3] von Neumann, J., "Various techniques used in connection with random digits", 1951.
- [4] Morina, G., Łatuszyński, K., et al., "**From the Bernoulli Factory to a Dice Enterprise via Perfect Sampling of Markov Chains**", arXiv:1912.09229v1 [math.PR], 2019.
- [5] Pae, S., "Random number generation using a biased source", dissertation, University of Illinois at Urbana-Champaign, 2005.
- [6] 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.
- [7] 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.
- [8] Peres, Y., "Iterating von Neumann's procedure for extracting random bits", Annals of Statistics 1992,20,1, p. 590-597.
- [9] Zhou, H. and Bruck, J., "**Streaming algorithms for optimal generation of random bits**", arXiv:1209.0730 [cs.IT], 2012.
- [10] S. Pae, "**Binarization Trees and Random Number Generation**", arXiv:1602.06058v2 [cs.DS].

4 License