A publicly verifiable protocol for random number generation

João Penna[1], **Jeroen van de Graaf**[1]

[1]Universidade Federal de Minas Gerais

{joaopenna, jvdg}@dcc.ufmg.br

Abstract. *Chance plays an essential role in many decision procedures such as lotteries, draws etc. As such procedures are moving on-line, several web services offering randomness have appeared over the last few years. NIST's randomness beacon, which publishes a sequence of 512 random bytes every minute, unfortunately lacks transparency: the beacon does not eliminate the possibility of an insider attack who knows the outcomes beforehand. We propose an improvement of NIST's beacon which is publicly verifiable and fully transparent: any outsider who did not witness the bit generation in person but has internet access can convince himself that the beacon acted honestly, provided he can be sure that fresh, independent random bits were contributed to the seed value. Our proposal is based on a novel cryptographic assumption: the existence of functions that are slow to compute even on the fastest supercomputers.*

1. Introduction

1.1. The NIST Randomness Beacon

Since 2011 there exists a public service of randomness in the United States provided by the National Institute of Standards and Technology (NIST) called the Randomness Beacon [Fischer et al. 2011]. The concept of a beacon as a randomness source was originally proposed by [Rabin 1983]. Every 60 seconds, NIST produces a sequence of 512 random bytes and publishes on the internet on their website. Each sequence also includes a timestamp, a digital signature, and the hash of the previous value. This way one can verify that it was generated by NIST, and it is also possible to detect tampering and retroactive changes on values.

But can one really trust NIST's randomness beacon? Perhaps the bits could be doctored through some insider attack. Or perhaps they were generated a year ago and are only being published now, meaning that an insider could know beforehand what values would come up at specific times. This could be just paranoia, but NIST has been infiltrated before by the NSA, as Edward Snowden revealed. And since there exist protocols which can eliminate these doubts, why not implement them?

1.2. Objective of this paper

The main objective of this paper is to present an alternative to NIST's randomness beacon that provides optimal verifiability. NIST's beacon uses a very sophisticated, quantum-based randomness source [NIST 2016], but it is not verifiable. It might appear to have high entropy, but it is not feasible to physically inspect it, meaning one has to trust the output values to be random and not being influenced on particular occasions. Furthermore, it is not possible to assert the freshness of the values generated. Insiders might have access to the random values long before they are published.

We approach this issue in the following way. We imagine a service in which any person or entity can contribute in a verifiable way to the random seed value z, for instance by hashing all the contributions. The output of the beacon will be $F(z)$, where F is a function with special properties, to be discussed below.

If you have a sequence z of $N = 4096$ random bits and apply a permutation F over $\{0,1\}^N$ on z, then the result $F(z)$ still will be random. Usually in cryptography we require F to be easy to compute and hard to invert. But if the input z is composed of many contributions, the person who provides the last contribution could try to influence the final outcome by choosing a contribution resulting in an $F(z)$ with special properties.

To thwart this attack we impose a different requirement on F: that F is **slow** to compute. In particular, in the scenarios outlined below we assume it takes at least 1 minute to compute $F(x)$, even with the fastest supercomputers available today. However, it would be very inefficient if would take hours to compute $F(x)$ on an ordinary workstation with a fast processor.

Finding a suitable F is not an easy endeavor, but our efforts were greatly simplified by the realization that the function Sloth introduced by Lenstra and Wesolowsk [Lenstra and Wesolowski 2015] is perfectly suited for this task. Using their techniques it is possible to define a function which takes 5 minutes to compute on an ordinary computer, while taking at least 1 minute on the best supercomputers available.

The goal of their paper is different, however: they present a transparent and verifiable way to produce the parameters of elliptic curves to be used in standards, thus avoiding any suspicion of tampering. Our contribution is to extend their techniques in order to provide a service equivalent to the NIST randomness beacon, but which provides verifiability.

Another approach is taken in the recent paper of [Syta et al. 2017], which contains excellent references to related work. Their goal is to develop a beacon scalable to many hundreds of parties connected to the internet, and is available as an on-line service at https://pulsar.dedis.ch. Their cryptographic assumptions are different from ours.

This paper is structured as follows: the next section provides additional background on the need for verifiable public randomness. Section 3 discusses a two-party setting based on the coin-flipping by telephone problem, which is extended to a multi-party setting in Section 4. By then we can define the properties needed for the function $F()$, and in Section 5 we show that Sloth from [Lenstra and Wesolowski 2015] addresses these needs. In Section 6 we specify our protocol for a new randomness beacon equivalent to NISTs but providing verifiability, succeeded by a brief discussion about the protocol's security properties.

2. Background

Suppose you need to make a random choice. Perhaps you could use a die to decide. But the die could be loaded; how could you know if the roll had no bias at all? How do you know if the lottery tickets are really randomly drawn? How could you tell if a physical random bit generator worked correctly without there being a way of influencing the result on particular occasions? It may be difficult to answer these questions. Take for instance the 1980 Pennsylvania Lottery Scandal, in which balls with different

weights were used to greatly increase the chance that certain numbers would be chosen [Togyer 1999]. One other lottery had its draw corrupted in the form of bribed children and marked balls [BBC 1999]. Random numbers generated by computer can also be tampered with [Simmons 2015].

2.1. An example: how to implement a verifiable random draw

Fortunately, in the digital world, cryptography can offer solutions. Suppose you teach a crypto class, and you have to make a random list of the order in which the students will present their final project. One transparent way to do this is as follows: you make a list with the name of the students and you concatenate it with the same, randomly chosen number r; you compute SHA256(name+r); and sort the list by the hash value obtained. Draws are possible but very unlikely; in fact, a draw implies a collision in SHA256 which would imply a nice publication.

Now a student could argue that you manipulated the order by testing various random numbers and reveal only the one corresponding to the order you liked. To avoid this argument, you agree to use an externally generated r to be published at some exact moment in the future, like one obtained from NIST's randomness beacon. Now a very paranoid student can argue that these bits may be random, but that you have an insider working at NIST who manipulated those bits for you. This is the problem we are trying to solve here.

The case of the algorithm which distributes court cases to the Supreme Judges of the *Supremo Tribunal Federal* is not essentially different. It is a well-known theoretical result that a randomized algorithm $R(x)$ can be modeled as a deterministic algorithm $D(x, r)$, where x is the same input and r is randomly chosen bit string of appropriate size.

The random numbers produced by NIST are of length 2048 bits, which already is a lot. However, if this would be deemed insufficient, one could use this r as the seed of a pseudo-random bit generator, like Blum-Blum-Shub, to produce an unlimited supply of bits which no poly-time algorithm can distinguish from truly random bits.

So the question is, where do these initial random bits r come from? What we need is a randomness service that is publicly verifiable on the internet, meaning that it is verifiable even without being present in person to witness the generation process.

3. Coin Flipping by Telephone

To explain our idea let us first consider two parties A and B who want to produce a random sequence $c \in S$, where $S = \{0,1\}^{2048}$. Suppose for now we dispose of a function $F : S \to S$ that takes 5 minutes on A's computer and which cannot be parallelized; how we obtain such a function will be discussed in Section 5. Then A and B can execute the following protocol:

1. A chooses a random bit string $a_0 \in S$, sends it to B and start computing $F(a_0)$.
2. B chooses a $b_0 \in S$ at random and sends it to A
3. The outcome $c \in S$ is defined as $c = F(a_0) \oplus b_0$, where $\oplus$ denotes bitwise XOR.

So after 5 minutes they have the outcome. But what if B has a faster computer than A? Then he could also compute $F(a_0)$ quickly, and choose a value for b_0 to obtain

a c that he likes. Well, B's computer may be faster, but by an appropriate choice of F and doing some hardware estimates (discussed in Section 5 and the paper cited), one can conclude that B cannot be more than five times faster. So A accepts Step 2 only if B sends it within 1 minute of her having sent the message in Step 1, otherwise she aborts. This time constraint guarantees that B cannot know $F(a_0)$ and therefore cannot bias the value of c.

Observe that in the traditional protocol of Coin-flipping by Telephone [Blum 1983], A will send a string commitment to a_0 in Step 1. At some later stage she unveils a_0 by opening the commitment; in that case the final result of the coin flip is defined as $a_0 \oplus b_0$. In the traditional case there are cryptographic restrictions on the commitment, which must be hiding and binding.

Note that in our protocol the function F could be invertible. *Our cryptographic assumption is that F takes at least one minute to compute even by the fastest supercomputers on the planet, while taking 5 minutes on an off-the-shelf PC with a fast processor.* Another difference is that in the traditional protocol A can wait as long as she wants to open the commitment, whereas is our protocol there is a time window in which she must accept or reject B's message in Step 2. Otherwise she is vulnerable to an attack of Bob.

The advantages of this approach may not be too clear in a two-party setting, but becomes more obvious when there are passive observers who have a stake in the outcome.

4. A multi-party setting with a beacon

We now change the setting: A and B rely on a trusted randomness beacon Z. As a very naive first approach, suppose that Z chooses an initial seed z and computes $H(z)$, where H is some very fast hash function like SHA256 modified to have a 2048-bit output.

If A is suspicious of Z secretly colluding with B to bias the outcome, this approach cannot remove this suspicion, because Z could try thousands of values for z, compute $H(z)$ for each and choose the value that suites B best. One way to allay A's suspicion is to allow her to contribute to the random seed with her value a. Say that the result is $H(z, a)$. The problem we now face is this: who is last, Z or A? Suppose that Z is last. Then, given a, he can still apply the same attack as before, searching for some suitable z. But if A is last to provide her value, she could mount this attack. Using a very fast function H, it is not possible to eliminate this possible attack by the party who is last.

However, substituting H for a function F which is deliberately slow in terms of wall clock time, the situation changes. Assume that $F(z, a)$ is a very slow function and enforce that any contribution must be submitted inside a small time frame before the computation starts. If at least one of z, a makes an honest contribution not known to the others previously, it would be impossible to compute F for many input values and choose another contribution that would generate a specific output.

5. Choosing a function that takes a long time to compute

Choosing the right function F is a subtle process but is possible. A simple solution is defining $F(n)$ as n consecutive iterations of a function G:

$$F(x) = \underbrace{G(G(G(G(x)...)))}_{n} = G^{(n)}(x)$$

Since this calculation is inherently sequential, n can be appropriately chosen to ensure it requires 5 minutes of wall clock time to compute.

One possible candidate for G would be Argon2 [Biryukov et al. 2016], a memory-hard function for password hashing. It has very interesting features, including that it is optimized for the x86 architecture, this way specialized hardware such as a FPGA board, an ASIC circuit or a GPU board do not gain a big advantage at runtime. Unfortunately, the downside is that the verification would take the same amount of time to compute. A more elegant approach is presented below, using a function for which the inverse is easy to compute: given $F(z)$ it is fast (a few seconds) to invert F and compute z.

5.1. Sloth

Sloth is a function proposed by [Lenstra and Wesolowski 2015] which takes a long time to compute, but its result can be verified very quickly. In other words, calculating G is slow, but calculating its inverse G^{-1} is quick.

Sloth is based on computing the square root of a number modulo a prime p. More specifically, for a prime p the finite group of p elements is denoted by Z_p, and its multiplicative group is denoted by Z_p^*, where we identify the elements with the integer between 1 and p. Let p be a large prime number congruent to 3 modulo 4. It follows that for any x in Z_p^* precisely one of x and $-x$ is a square, and a square root x can be computed by calculating $x^{\frac{p+1}{4}}$. It is generally believed that this computation cannot be done faster than using $log_2(p) - 2$ unparallelizable modular squarings, but computing the inverse, for verification, involves just a single modular squaring.

Recall that x has two square roots: if y is a root then so is $p - y$; consequently one of them is even and the other odd. To determine which of the two square roots will be used as the result of the computation, the following convention is used: If x is a quadratic residue, take its even root; if not take the odd root of $-x$, which is a square. Whether x is a quadratic residue can be checked using the Legendre symbol: if $x^{\frac{p-1}{2}} = 1$, then it is a quadratic residue, and if $x^{\frac{p-1}{2}} = -1$ it is not. Note that this turns $G(x)$ into a permutation on Z_p^*.

To avoid algebraic shortcuts in computing $G^{(n)}$, each iteration also contains a additional permutation. By using a block cipher such as AES (Advanced Encryption Standard), attacks that would explore the algebraic structure of the calculation are avoided. Note that when using this kind of permutation there is a possibility that it makes the number bigger than the prime p chosen. Depending on the choice of p, this chance is extremely small, but if it does happen then executing another permutation until the result generates a number smaller than p solves the issue.

The result of this calculation is an output that results in high quality random bits. The output is also used to calculate the inverse function, allowing one to check whether the result corresponds to the computation of the inputs given, thus verifying if the whole process was done correctly. The output is deterministic and will always yield the same result for the same inputs and parameters, but in our protocol the output cannot be predicted during the first minute.

6. A Publicly Verifiable Randomness Beacon

In this section we present a novel construction of a randomness beacon with the following properties:

- The beacon produces 2048 bits every minute. These bits will be signed digitally.
- Any party who wishes can contribute a (random) string which is used as input to the initial value z_i for the process that starts at minute i.
- Every minute a process Z_i is started which computes $F(z_i)$, which takes 5 minutes to complete.
- To this end we have 5 parallel processors, T_1 to T_5, which work in pipelined fashion: at minute i processor $T_{(i \bmod 5)}$ yields the result of the process started 5 minutes earlier, $G(z_{i-5})$, and starts computing $G(z_i)$.

A diagram illustrating this scheme is shown in Figure 1. As previously explained, if a party is suspicious that the generation might be corrupted, submitting an honest contribution prevents other parties from forcing a specific result. While this makes the output unpredictable by the start of Z's computation, note that the output is deterministic and always the same if the same inputs are given, thus enabling anyone to verify afterwards if the computation was done correctly. Another positive side effect of an honest contribution is that one can be assured that the generated value is fresh. Of course, all of this relies on the fact that there exist a function that guaranteedly takes a long time to compute.

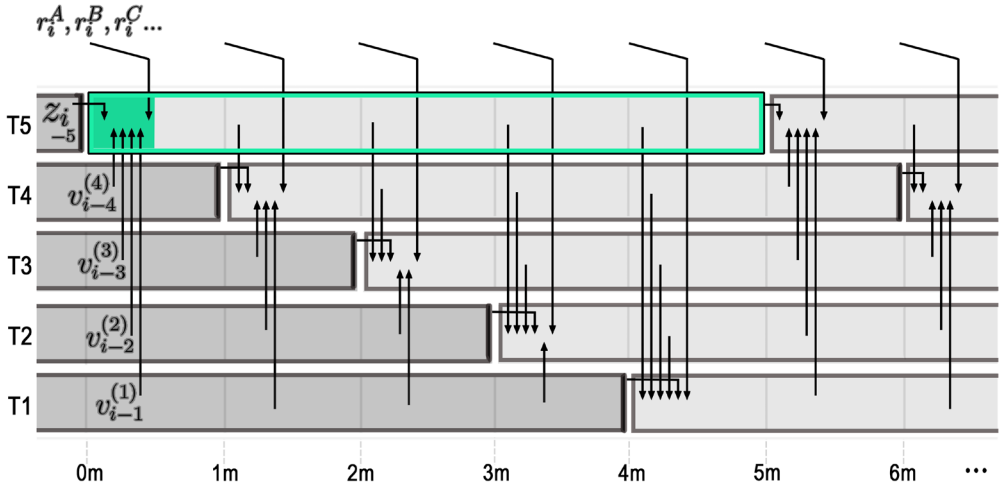

Figure 1. At the start of every minute, about one second is spent synchronizing the inputs from the values contributed from outsiders, intermediate values from the four parallel process, and the final value from the process that just ended

We now describe the actions of Z which implements a beacon with these properties, assuming that all data it publishes is authenticated. At the beginning of minute i:

1. Z collects all random external contributions $r_i^A, r_i^B, r_i^C, \ldots$ submitted by outsiders A, B, C, ... for minute i;
2. Z receives the intermediate values v of the four parallel processes run on the other cores, and the final value of the process that just ended running on the same core. Let v_i^j be the intermediate value of computing $F(z_i)$ after $j = 1, 2, 3, 4$ minutes.

Observe that superscripts represent indexes, not exponents. Then the values received by Z can be represented as $v_{i-1}^{(1)}, v_{i-2}^{(2)}, v_{i-3}^{(3)}, v_{i-4}^{(4)}, z_{i-5}$;
3. Z produces his own seed value s_i;
4. Z uses some publicly known hash function H to reduce all these values to 2048 bits: $z_i = H(r_i^A, r_i^B, r_i^C ..., v_{i-1}^{(1)}, v_{i-2}^{(2)}, v_{i-3}^{(3)}, v_{i-4}^{(4)}, z_{i-5}, s_i)$;
5. Z publishes $BC(z_i)$ where BC is some unconditionally hiding commitment scheme;
6. Z dispatches a new process to compute $F(z_i)$ om processor $T_{(i \bmod 5)}$;
7. After $j = 1, 2, 3, 4$ minutes Z publishes the intermediate values $v_i^1, v_i^2, v_i^3, v_i^4$, resp.
8. After 5 minutes Z publishes $F(z_i)$, opens $BC(z_i)$ and terminates.

7. Informal security analysis

Let us assume that the authority that runs the new service conspires with a three-letter agency which can run F up to 5 times faster. The security of the protocol, that is, the freshness of the output $F(z_i)$ depends on the presence of random external contributions $r_i^A, r_i^B, r_i^C, ...$ submitted by outsiders.

If no random external contribution is submitted, this protocol offers no advantage. Z can just compute all random values a week ahead in time, selectively reveal these values to insiders, and publish them exactly at the right moment. Nothing is gained.

If A is suspicious of Z secretly colluding with B to bias the outcome, what she could do is to provide some random contribution r_i^A to z_i, because then she will be sure that $F(z_i)$ is fresh and random. Even a malicious Z cannot impose any bias on $F(z_i)$ because $F(x)$ takes at least one minute to compute, and the bit commitment binds Z to z_i. A's timing is essential here: she must submit r_i^A less then one minute before the computation of $F(z_i)$ starts.

Note that, even though generating 4096 random bits for A is easy, the system would already be secure if r_i^A had an entropy of about a 100 bits. Because, for Z's attack to work it would have to compute $F(H(..., s_i))$ for 2^{100} different values s_i, and this is clearly infeasible.

In fact, the freshness propagates to $F(z_{i+1})$ too. In case Z disposes of a supercomputer, he can compute the intermediate value v_i^1 in $60/5 = 12$ seconds. Now, even supposing that no new contributions $r_{i+1}^A, r_{i+1}^B, r_{i+1}^C ...$ are submitted in the last 48 seconds before dispatching the computation of $F(z_{i+1})$, still Z will need at least a full 60 seconds to compute $F(z_{i+1})$, leading to a total of 72 seconds. But he is forced to dispatch the computation of $F(z_{i+1})$ after 60 seconds, and a 12 second delay will certainly be noticeable, even on the internet. The freshness does not propagate to $F(z_{i+2})$. A malicious Z can compute v_i^2 in 24 seconds, leaving it $60 + 36 = 96$ seconds to compute a suitable $z_{i+2} = H(..., s_{i+2})$, again assuming no fresh contributions $r_{i+2}^A, r_{i+2}^B, r_{i+2}^C ....$

For an outsider B, who did not contribute to z_i, the situation is more complicated. He can only be sure that Z is honest if verifiably fresh randomness has been submitted. In fact, what we see is that in this adversarial scenario, the freshness of the results $F(z_i)$ hinges on the freshness and independence of the external contributions $r_{i+1}^A, r_{i+1}^B, r_{i+1}^C ....$

We believe that this can be made practically feasible, as follows. It would be sufficient if neutral entities such as INMETRO, universities or other organizations (like

the Electronic Frontier Foundation) submit every minute 2048 random bits to Z for B to believe the bits were really random. This belief would even be retro-actively, i.e. applying to bits generated in the past.

It would be even better if some of these trusted entities would run their own version of the beacon Z, cross-feeding each other with fresh randomness by broadcasting there outputs to all the others to be included to the seed for the next input.

8. Conclusion

This paper proposes a novel protocol which offers a randomness beacon on the internet which provides transparency and verifiability, even to persons not physically present to witness the generation. The idea of receiving public contributions to influence random number generation allows any person to verify that the beacon outputs are indeed fresh, and prevents insiders from unduly biasing these outputs. By pipelining parallel instances of the beacon we achieve the desired throughput, publishing fresh randomness every minute. Any person can audit the generator for correctness by verifying that the output published indeed is obtained from the initial seed.

References

BBC (1999). Italy hit by lottery scandal. `http://news.bbc.co.uk/1/hi/world/europe/256206.stm`.

Biryukov, A., Dinu, D., and Khovratovich, D. (2016). Argon2: New generation of memory-hard functions for password hashing and other applications. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016*, pages 292–302.

Blum, M. (1983). Coin Flipping by Telephone a Protocol for Solving Impossible Problems. *SIGACT News*, 15(1):23–27.

Fischer, M. J., Iorga, M., and Peralta, R. (2011). A public randomness service. In *SECRYPT 2011*, pages 434–438.

Lenstra, A. K. and Wesolowski, B. (2015). A random zoo: sloth, unicorn, and trx. *IACR Cryptology ePrint Archive*, 2015:366.

NIST (2016). Truly random numbers – but not by chance. `https://www.nist.gov/news-events/news/2012/10/truly-random-numbers-not-chance`.

Rabin, M. O. (1983). Transaction protection by beacons. *J. Comput. Syst. Sci.*, 27(2):256–267.

Simmons, D. (2015). Us lottery security boss charged with fixing draw. `http://www.bbc.com/news/technology-32301117`.

Syta, E., Jovanovic, P., Kokoris-Kogias, E., Gailly, N., Gasser, L., Khoffi, I., Fischer, M. J., and Ford, B. (2017). Scalable bias-resistant distributed randomness. In *IEEE Symposium on Security and Privacy*, pages 444–460.

Togyer, J. (1999). Putting in the fix. `http://www.tubecityonline.com/history/perry.html`.

