Unicorn beacon Lausanne

Arjen K. Lenstra and Florian Standaert

EPFL IC LACAL, Station 14, CH-1015 Lausanne, Switzerland

Abstract. The *Unicorn beacon Lausanne* (`http://trx.epfl.ch/beacon`) generates a fresh, trustworthy, verifiable, 512-bit random number once every ten minutes. Anyone can influence, but not knowingly bias, beacon values by entering data on the website or by sending tweets with hashtag `#unicorn_beacon`. The beacon uses the unicorn protocol from [9] combined with a timed commitment similar to [5]. This note describes our implementation.
Keywords: random beacon, trustworthiness, verifiability, sloth, unicorn.

1 Introduction

In [9] the need for public randomness is discussed, several online services providing public randomness ("beacons") are reviewed, and a new public randomness generator, *unicorn*, is presented. Here we describe our implementation of the *Unicorn beacon Lausanne* which outputs once every ten minutes a new 512-bit random value on `http://trx.epfl.ch/beacon`. On top of being efficiently verifiable, the unicorn beacon has a number of other desirable properties not shared with other beacons; quoting [3]:

> The beacon is unpredictable even to an adversary who controls $n-1$ of the (n) participating parties. It has linear communication complexity and uses only two rounds. This stands in contrast to coin-tossing beacons which use verifiable secret sharing and are at best resistant to an adversary who controls a minority of the nodes [1,6,12]. These beacons also use super-linear communication and require multiple rounds of interaction.

The first part of this quote is of particular interest (and proved in [9]): as long as there is a party that contributes adequate data (on site or using a tweet with hashtag `#unicorn_beacon`), the beacon value to which the data contributed is trustworthy, irrespective of what anyone else may have done. In particular, the trustworthiness of the beacon value does not depend on the trustworthiness of the party that implemented or that operates the service:

- Anyone can efficiently verify that each beacon value has been properly generated based on the contributed data. This follows from the unicorn protocol from [9].
- A beacon value once committed to can in principle be calculated by anyone, independent of the beacon operator. Here we use a timed commitment similar to what was proposed in [5].

Because the operator must commit to a beacon value well before the beacon value can be known to the operator, the latter property (which is not present in the unicorn protocol) offers adequate protection against an operator that may want to suppress beacon values that it deems undesirable (for instance by claiming server failure).

Unicorn background. The need for a public randomness generator was triggered by our desire to generate parameters for elliptic curve cryptosystems in a truly random, trustworthy, and verifiable manner – an issue of some interest back in 2014 [2]. The resulting paper [8] (later published as [9]) introduces the slow-timed hash function *sloth*, which is used for the uncontestable random number generator *unicorn*, which in turn led to the aimed for generation of trustworthy random elliptic curve parameters usable in cryptosystems (*trx*). The security of the construction proposed in [8] (cf. [8, Section 4.2] for the applicable security notion) relies on the assumption that computing a square root modulo a prime modulus m requires a full modular exponentiation and thus at least $\log_2(m)$ squarings modulo m that are unparallelizable and that must be carried out sequentially. No relevant results have been published that undermine this security assumption, and the

construction from [8] remains unaffected (cf. above quote from [3]). The notion of a slow-timed hash function has recently been picked up and led to a number of promising follow-up results and applications [3,14,11,4].

Unicorn outline. The generation of each unicorn beacon value follows the approach described in [8, Section 4.1], with small modifications due to the inclusion of the timed commitment. With times measured in minutes, and for some start time t_0, let $t_i = t_0 + 10i$ for $i \in \mathbf{Z}_{>0}$. Generation of the i-th beacon value starts at time t_{i-1} with ten minutes of data gathering. This lasts until time t_i, at which moment the calculation of the i-th commitment value is started and the i-th sloth-calculation begins as soon as the i-th commitment values have been made available. As soon as the sloth-calculation finishes, which is after $t_i + 8$ but before t_{i+1}, the i-th beacon is published along with all values required for its fast verification.

Paper outline. Notation is defined in Section 2. The data gathering and commitments are described in Section 3, the sloth-calculation is described in Section 4, and details about the verification stage can be found in Section 5.

2 Notation

For any function g from D to D, for some domain D, we define $g^1 = g$ and $g^i = g^{i-1} \circ g$ for integers $i > 1$.

Let $\mathtt{H} = \{\mathtt{0}, \mathtt{1}, \ldots, \mathtt{9}, \mathtt{a}, \mathtt{b}, \ldots, \mathtt{f}\}$ denote the set of lower case hexadecimal characters. For any string s over $\mathtt{H}$ its integer value $\mathrm{int}(s)$ is defined in the usual manner; thus, for instance, $\mathrm{int}(\mathtt{a9f0}) = 10 \cdot 16^3 + 9 \cdot 16^2 + 15 \cdot 16^1 + 0 \cdot 16^0 = 43\,279$. Conversely, for any integer $i > 0$ we denote by $\mathrm{hex}(i)$ its lower case hexadecimal representation without leading hexadecimal zero, and by $\mathrm{hex}(0)$ we denote the hexadecimal zero $\mathtt{0}$.

By h we denote the cryptographic hash function SHA-512 [10]. It is assumed that hash values are strings of length 128 over $\mathtt{H}$, possibly with one or more leading hexadecimal zeros. A hash value x thus corresponds to a non-negative integer less than 2^{512}; we denote this integer by $\mathrm{int}(x)$. For any valid input y to h, we define $h^+(y)$ as $h^i(y)$ for the smallest integer $i > 0$ such that $\mathrm{int}(h^i(y)) \geq 2^{511}$, i.e, the smallest integer i such that the most significant hexadecimal digit of $h^i(y)$ is in $\{\mathtt{8}, \mathtt{9}, \mathtt{a}, \mathtt{b}, \ldots, \mathtt{f}\}$; under a reasonable assumption about the inputs and outputs of SHA-512 the expected value of i equals two.

Concatenation of strings over $\mathtt{H}$ is denoted by $||$. For any string s over $\mathtt{H}$ and non-negative integer i we define $s(i) = s||\mathtt{hex}(i)$ and, for any integer width $w > 0$,

$$h_{i,w}(s) = h(s(i)) \,||\, h(s(i+1)) \,||\, h(s(i+2)) \,||\, \ldots \,||\, h(s(i+w-1))$$

and

$$h^+_{i,w}(s) = h^+(s(i)) \,||\, h(s(i+1)) \,||\, h(s(i+2)) \,||\, \ldots \,||\, h(s(i+w-1));$$

note that $\mathrm{int}(h_{i,w}(s)) < 2^{512w}$ and $2^{512w-1} \leq \mathrm{int}(h^+_{i,w}(s)) < 2^{512w}$. Evaluation of either $h_{i,w}$ or $h^+_{i,w}$ requires w applications of h that can be performed in parallel, for $h^+_{i,w}$ possibly followed by on average a single application of h to complete the computation of $h^+(s(i))$.

Finally, for any string s over $\mathtt{H}$ and non-negative integer i we define $P_i(s)$ as the smallest prime number that is at least $\mathrm{int}(h^+_{i,2}(s))$ and $P(s)$ as the smallest prime number that is equal to 3 modulo 4 and that is at least $\mathrm{int}(h^+_{1,4}(s))$. Note that $P_i(s)$ is a 1024-bit prime, and that $P(s)$ is a 2048-bit prime that is equal to 3 modulo 4 (the possibility that $P_i(s) \geq 2^{1024}$ or $P(s) \geq 2^{2048}$ can safely be discarded). We chose 1024 to allow fast generation of $P_i(s)$ while making it unlikely that anyone can factor the product of two 1024-bit primes fast enough (to open the timed commitment early, cf. below).

3 Data gathering and commitments

To generate the i-th beacon value, the unicorn protocol first creates a seed. This seed is based on public data gathering during the time period $[t_{i-1}, t_i]$. In our implementation we use on site

messaging and twitter for the data collection: anyone can remotely contribute to the seed in a manner that is entirely under one's own control[1] by entering messages on the site `http://trx.epfl.ch/beacon` or by sending tweets with hashtag `#unicorn_beacon`. To avoid that no public data are contributed we include by default recent tweets containing the word "random". The messages and tweets received during the time period $[t_{i-1}, t_i]$ are both saved in the order in which they were received and concatenated in the file `seed.txt`.

Furthermore, at time t_i a picture is taken of a constantly changing view[2] and saved in the file `img.jpg`. A hexadecimal value S is then calculated as the hash of the concatenation of the hashes of the files `seed.txt` and `img.jpg`:

$$S = h(\ h(\texttt{seed.txt}) \ ||\ h(\texttt{img.jpg})\ )$$

and a hexadecimal commitment value c is calculated as the hash of S:

$$c \ = \ h(S).$$

Following the description of the unicorn protocol from [9], at this point the file `seed.txt` and the commitment c would be made available and the sloth-calculation would be applied to S. However, to allow for independent construction of the beacon value based on just the committed values, more is needed: additional commitment values have to be computed first, after which sloth will be applied to a slightly different value than just S.

Obviously, it is undesirable to make S, $h($`img.jpg`$)$, or `img.jpg` available along with `seed.txt`, because it would allow an independent sloth-calculation that may be completed before ours. Independent sloth-calculation should, however, eventually be possible to avoid operator misbehavior. Therefore, at least one of the three values (S, $h($`img.jpg`$)$, `img.jpg`) has to be made available before the sloth-calculation starts, but this must be done in such a way that accessing the value requires substantially more time than completing the sloth-calculation. We chose to do so by making available an encrypted version of the picture `img.jpg` that can, as the result of a calculation that requires an adequate amount of time, be decrypted using the committed values (and that allows fast decryption once the beacon value is made available along with its verification data). Our approach is similar to [5] and is described in the next paragraphs.

With the notation defined in Section 2, as soon as the picture `img.jpg` is taken we generate the 1024-bit primes $p = P_1(h(\texttt{img.jpg}))$ and $q = P_3(h(\texttt{img.jpg}))$ (which can be done in parallel) and compute the 2047- or 2048-bit RSA modulus $n = pq$.

Given $c = h(S)$, p, q, n and some positive integer ℓ, an integer value v is calculated as $v = \mathrm{int}(c)^e \bmod n$ for $e = 2^\ell \bmod (p-1)(q-1)$. This takes[3] $\log_2 \ell$ squarings modulo $(p-1)(q-1)$ followed by a full exponentiation modulo n. The same value v can also be calculated directly as $\mathrm{int}(c)^{2^\ell} \bmod n$ using ℓ consecutive (and unparallelizable) squarings modulo n if the factorization of n is not known. Given v, the picture `img.jpg` is now encrypted with AES-256, using the base 36 representation[4] of $v \bmod 2^{256}$ as a passphrase, which results in `img.jpg.enc`. The choice of ℓ is discussed at the end of this section.

Once `img.jpg.enc` has been calculated, the commitment values

$$c \quad \text{and} \quad \texttt{img.jpg.enc}$$

are made available, along with the values

$$n \quad \text{and} \quad \texttt{seed.txt};$$

[1] One may also make a contribution, not entirely under one's own control, by being at the right spot at time t_i; cf. next paragraph.

[2] A picture is taken of the parking lot opposite office INJ331 at the EPFL campus in Lausanne, Switzerland.

[3] It can be done about four times faster using Chinese remaindering, but that would require more software.

[4] Here we assume that each integer in $[0, 36)$ is represented by a unique character in $\{\texttt{0}, \texttt{1}, \ldots, \texttt{9}, \texttt{a}, \texttt{b}, \ldots, \texttt{z}\}$ in the canonical manner (thus with `a` through `z` having integer values 10 through 35). Our choice is arbitrary and does not affect the 256-bit security of the passphrase.

it may be assumed that this happens shortly after time t_i. Once these values are publicly available, the sloth-calculation is applied to $s = S||\text{hex}(n)$. This calculation is described in the next section. Anyone who contributed data to the i-th beacon value generation should be able to find those data in `seed.txt`.

Choice of ℓ. The value for ℓ must be chosen to offer protection against adversaries who want to get earlier access to the beacon value (by using just c and n to calculate v and by decrypting `img.jpg.enc` so that S required for the sloth-calculation follows) while still allowing a feasible recovery possibility. Adversaries with computational capabilities similar to ours are easy to deal with: their sloth-calculation time will be similar to ours too, so they cannot derive the beacon value faster anyhow, and any value for ℓ will suffice.

Adversaries that can do modular arithmetic f times faster than we can, must be forced to spend "our" sloth-calculation time[5] on the calculation of v, by using an ℓ-value that is at least f times the expected total number of 2048-bit modular multiplications and squarings required by the sloth-calculation. It follows that for parties that are comparable to us, the recovery time will be f times the time required for sloth, i.e., about $f/6$ hours. For instance, the rumored-to-be-developed 2 nanosecond 2048-bit modular multiplier [13], would lead to $f \approx 500$, would force us to use an ℓ-value on the order of 300 billion (cf. final paragraph of Section 4), and would thus lead to a recovery time between 3 and 4 days[6] for parties with computational capabilities similar to ours. We indeed chose to use $\ell = 3 \cdot 10^{11}$.

4 Sloth-calculation

In this section we mostly follow [9] and describe in detail how the various steps were implemented.

Given the seed $s = S||\text{hex}(n)$, let p' be the 2048-bit prime $P(s)$ and $w = \text{int}(h_{5,4}(s)) \bmod p'$ where w is regarded as an element of $\mathbf{Z}/p'\mathbf{Z}$ in the natural manner (and where the elements of $\mathbf{Z}/p'\mathbf{Z}$ are represented in the canonical manner by the non-negative integers less than p').

The permutation τ on $\mathbf{Z}/p'\mathbf{Z}$ is defined as $\rho \circ \sigma$ with ρ and σ the permutations on $\mathbf{Z}/p'\mathbf{Z}$ that are defined below.

- For a quadratic residue $y \in \mathbf{Z}/p'\mathbf{Z}$ let $\sqrt[+]{y}$ be the unique square root modulo p' of y that has an even representation as a non-negative integer less than p', and let $\sqrt[-]{y}$ be the unique square root modulo p' of y that has an odd representation as a non-negative integer less than p'. Then, for $x \in \mathbf{Z}/p'\mathbf{Z}$,

$$\rho(x) = \begin{cases} \sqrt[+]{x} & \text{if } x \text{ is a quadratic residue modulo } p' \\ \sqrt[-]{-x} & \text{otherwise.} \end{cases}$$

- Let $\oplus$ denote the bitwise xor-function, and let $m = 2^{1024} - 1$. Then

$$\sigma(x) = \begin{cases} x \oplus m & \text{if } x \oplus m < p' \\ x & \text{otherwise.} \end{cases}$$

For a positive integer λ (chosen as indicated in the next paragraph), the sloth-calculation now consists of computing $\tau^\lambda(w)$, after which the *beacon value* $h(\tau^\lambda(w))$ is published along with the *witness* $\tau^\lambda(w)$ and the picture `img.jpg`. It is assumed that these values are made public at least eight minutes after time t_i but before time t_{i+1}.

The sloth-calculation essentially consists of λ consecutive square root computations in $\mathbf{Z}/p'\mathbf{Z}$, each of which is done using a $\frac{p'+1}{4}$-th powering modulo p' (because $p' \equiv 3 \bmod 4$). As far as we currently know, this requires about $1.5\lambda \log_2(\frac{p'+1}{4}) \approx 3069\lambda$ squarings and multiplications modulo

[5] Actually, at least a fraction $\frac{f-1}{f}$ of our sloth-calculation time: to be on the safe side we disregard this factor $\frac{f-1}{f}$.

[6] Though this may be argued to be barely feasible, it is feasible, and may well turn out to be more feasible than the 2 nanosecond modular multiplier.

p' that have to be performed sequentially and that cannot be parallelized in any way. It follows that λ must be chosen in such a way that this sequence of modular operations takes at least eight minutes. For our implementation this led to the choice $\lambda = 155\,000$; note that $3069 \cdot 155\,000 \cdot 500$ is about 240 billion, which explains the ℓ-choice of 300 billion at the end of Section 3.

5 Verification

Assume that the i-th beacon generation process resulted (before time t_{i+1}) in beacon value b, witness w', and picture `img.jpg` . Furthermore, assume that the values c, `img.jpg.enc`, n and `seed.txt` that were committed to shortly after time t_i are available as well. The verification of the i-th beacon value consists of the following steps; verification fails if either step fails and is successful otherwise.

- If applicable, verify that one's contribution is among the messages or tweets in `seed.txt` (or that one indeed shows up on `img.jpg`).
- Verify that $h(w')$ equals b.
- Compute S as $h(h(\texttt{img.jpg})||h(\texttt{seed.txt}))$ and verify that $h(S)$ equals c.
- With $s = S||\mathrm{hex}(n)$, compute $p' = P(s)$ and $w = \mathrm{int}(h_{5,4}(s)) \bmod p'$.
- Apply the function $(\tau^\lambda)^{-1}$ to the witness w' and verify that the result equals w. Because $(\tau^\lambda)^{-1} = (\tau^{-1})^\lambda$, $\tau^{-1} = \sigma^{-1} \circ \rho^{-1}$, $\sigma^{-1} = \sigma$ and, for $x \in \mathbf{Z}/p'\mathbf{Z}$,

$$\rho^{-1}(x) = \begin{cases} x^2 & \text{if } x \text{ is represented by an even non-negative number less than } p' \\ -x^2 & \text{otherwise,} \end{cases}$$

  this requires essentially λ squarings in $\mathbf{Z}/p'\mathbf{Z}$, which can be done quickly.
- Optionally, the correctness of `img.jpg.enc` can now easily be verified too: derive (as described in Section 3) the proper passphrase based on c and the primes $P_1(h(\texttt{img.jpg}))$ and $P_3(h(\texttt{img.jpg}))$, encrypt `img.jpg` or decrypt `img.jpg.enc` and verify that the result equals `img.jpg.enc` or `img.jpg`, respectively.

6 Concluding remarks

Unlike other random beacons, our trust-model does not rely on a trusted operator or other trusted parties, but it relies exclusively on the trust that one can have in oneself. In this model, our beacon generates a random value that one can provably trust if one carries out the following two steps: contributing a value that one trusts – typing it in at the website suffices – and verifying that the outcome is correct. If these steps are taken, the trust that one can have in the resulting beacon value is not affected by anything done by anyone else, including the party that controls the website. Nevertheless, beacon values must be used with care – and obviously never for applications that require private randomness.

Server failure. As mentioned in Section 3, if for whatever reason a beacon value that should result from already committed values c, `img.jpg.enc`, n and `seed.txt` is never published, it can be calculated by decrypting `img.jpg.enc` as described in Section 3 (a computation that easily requires a few days), followed by the regular sloth-calculation as presented in Section 4.

Integrity of archived values. The current system does not guarantee the integrity of archived values (for instance, by using a variant of the classical system originally proposed in [7], currently referred to as "blockchains"). As is, after publication the operator can try to generate a possibly "better" beacon value and modify the archived values (which would again pass verification), by trying any number of minute changes in `seed.txt` or `img.jpg`. Because doing so would require about 8 minutes per attempt (but any number of attempts can be done in parallel), there is no reason to trust beacon values that are older than, say, 5 minutes; after that all trustworthiness-bets are off.

Contributions by Nourchene Ben Romdhane, Tristan Deloche, Patricia Egger, Novak Kaluderovic, and Benjamin Wesolowski are gratefully acknowledged.

References

1. Randao: A dao working as rng of Ethereum. Technical report, 2016.
2. D. J. Bernstein, T. Chou, C. Chuengsatiansup, A. Hülsing, T. Lange, R. Niederhagen, and C. van Vredendaal. How to manipulate curve standards: a white paper for the black hat. Cryptology ePrint Archive, Report 2014/571, 2014. `http://eprint.iacr.org/2014/571`.
3. D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. Verifiable delay functions. Cryptology ePrint Archive, Report 2018/601, 2018. `http://eprint.iacr.org/2018/601`.
4. D. Boneh, B. Bünz, and B. Fisch. A survey of two verifiable delay functions. Cryptology ePrint Archive, Report 2018/712, 2018. `http://eprint.iacr.org/2018/712`.
5. D. Boneh and M. Naor. Timed commitments. In M. Bellare, editor, *Advances in Cryptology, CRYPTO 2000*, volume 1880 of *Lecture Notes in Computer Science*, pages 236–254. Springer Berlin Heidelberg, 2000.
6. I. Cascudo and B. David. Scalable randomness attested by public entities. Cryptology ePrint Archive, Report 2017/216, 2017. `http://eprint.iacr.org/217/216`.
7. S. Haber and W. S. Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3:99–111, 1991.
8. A. K. Lenstra and B. Wesolowski. A random zoo: sloth, unicorn, and trx. Cryptology ePrint Archive, Report 2015/366, 2015. `http://eprint.iacr.org/2015/366`.
9. A. K. Lenstra and B. Wesolowski. Trustworthy public randomness with sloth, unicorn and trx. *International Journal of Applied Cryptography*, 3(4):330–343, 2017.
10. NIST. *Secure hash standard*. National Institute of Standards and Technology, Washington, 2002. URL: `http://csrc.nist.gov/publications/fips/`. Note: Federal Information Processing Standard 180-2.
11. K. Pietrzak. Simple verifiable delay functions. Cryptology ePrint Archive, Report 2018/627, 2018. `http://eprint.iacr.org/2018/627`.
12. E. Syta, P. S. Jovanovic, E. Kokoris Kogias, N. Gailly, L. Gasser, I. Khoffi, M. J. Fischer, and B. A. Ford. Scalable bias-resistant distributed randomness. *2017 IEEE Symposium On Security And Privacy (SP)*, pages 444–460, 2017.
13. B. Wesolowski, December 2018. Private communication.
14. B. Wesolowski. Efficient verifiable delay functions. Cryptology ePrint Archive, Report 2018/623, 2018. `http://eprint.iacr.org/2018/623`.