A random zoo: sloth, un$\overset{\text{i}}{\vee}$corn, and trx

Arjen K. Lenstra and Benjamin Wesolowski

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

Abstract. Many applications require trustworthy generation of public random numbers. It is shown how this can be achieved using a hash function that is timed to be as slow as desired (*sloth*), while the correctness of the resulting hash can be verified quickly. It is shown how *sloth* can be used for uncontestable random number generation (*unicorn*), and how *unicorn* can be used for a new trustworthy random elliptic curves service (*trx*) and random-sample voting.
Keywords: slow-timed hash, uncontestable random numbers, you and i, trustworthy random elliptic curves service, random-sample voting, animal farm
Version: 1.0, April 22, 2015

1 Introduction

There are many situations where large interests depend on random choices. Obvious examples are national lotteries and sporting events schedules, but it also plays a role in governance. Sortition was the cornerstore of Athenian democracy, where both the βουλη (the legislative council) and the ‘ηλιαία (the supreme court) consisted of a random sample of citizens. Even to the present day sortition-based democracy is advocated by some as a fair and simple alternative to elected assemblies.

The required random choices must be made in such a way that no one can knowingly bias the choices to anyone's advantage or disadvantage, and such that everyone affected, directly or indirectly, can be assured that foul play is impossible. Such assurance is, to some extent, meant to be provided by live broadcasts of lotteries and draws for sports events. But multiple scenarios are conceivable to influence the outcomes, like any skilled prestidigitator can fool entire crowds while publicly tossing a coin or rolling a die.

In this paper potential solutions to this problem are discussed and a new one is proposed. The new approach relies on two simple observations. The first is that even though casual observation of events and human behavior may, on a short time scale, offer little surprises or variation, at the bit level *any* even *very briefly* observed physical scene provides large amounts of entropy (including for cryptographic applications, and no matter how carefully the scene may have been orchestrated). Secondly, in many applications there is no time pressure: quite on the contrary, if one wants to turn the randomness selection into an entertaining public event it must be considered an advantage – also to cater to the expectations of sponsors and advertisors – if the proceedings are stretched a bit.

Another application of uncontestable generation of random numbers without time pressure is the seeding of the generation of standardized parameter choices for elliptic curve cryptography (ECC). Although there are many elliptic curve parameters that would be suitable for ECC, there is only a small set of elliptic curve parameters that are recommended or standardized for general use [4]. Using one of these curves implies trusting the way it was generated: [3] elaborates why this could be problematic even if care seems to be taken to avoid trust issues. Furthermore, any of these curves may already have been cryptanalyzed – for reasons unbeknownst to a new user, for instance because a worthwhile target uses it –

and quite extensively so if the curve has been around for a while already. Unlike the trust issue, possibly long term prior exposure to cryptanalysis does not seem to be a concern that is often expressed. Nevertheless, there may be users that would prefer to always use parameters that are as fresh as possible or to use their own personalized parameters. This is not an option yet: due to the current state of the art in elliptic curve point counting, generating good random elliptic curve parameters is a tedious process whereas parameters that can be quickly generated (using the complex multiplication method) are frowned upon – albeit for unknown reasons.

Classical methods that provide incorruptible public randomness and their disadvantages are discussed in Section 2. Using a new slow-timed hash function, *sloth*, described in Section 3 (and pronounced "slow-th"), a new approach to public randomness selection, *unicorn*, is proposed in Section 4: *unicorn* results in a high entropy random seed that can be influenced by anyone participating in the initial stages of its generation without any party being able to manipulate the outcome to its advantage, the correct generation of which can be verified by everyone, once the seed has been made public. A cryptographic application of the method from Section 4 is generating parameters for ECC. A service resulting from this application, *trx*, is currently being implemented and will produce a slow but constant stream of trustworthy random elliptic curve parameters at various security levels. *Trx* is described in Section 5. An application of the method from Section 4 to democratic voting procedures is sketched in Section 6. The possibilities of using the newly proposed methods for other applications (including cryptographic elections, other cryptographic standards and a random beacon) are briefly mentioned in the final section, Section 7.

Notation. The integer $k \in \{128, 192, 256\}$ denotes a security level. With k clear from the context, the cryptographic hash function h denotes the $2k$-bit version of the secure hash algorithm SHA-2. The function h is regarded as a function from $\mathcal{A}^*$ to $\mathcal{H}^{k/2}$, where $\mathcal{A}^*$ is the set of strings over some alphabet $\mathcal{A}$ and where $\mathcal{H} \subset \mathcal{A}$ is the set $\{\mathtt{0}, \mathtt{1}, \ldots, \mathtt{9}, \mathtt{a}, \mathtt{b}, \ldots, \mathtt{f}\}$ of hexadecimal characters. The secure hash function SHA-2 may be replaced by any other suitable cryptographic hash function one sees fit. It is assumed that the hash function satisfies the usual security requirement that it takes effort on the order of at least 2^k to find a (chosen) pre-image or collision. The constructions presented in this paper may fail if either of the security assumptions does not hold. If this happens in practice, the cryptographic hash function used should be considered to be broken, which would be a surprising side-result.

The function $\mathtt{int} : \mathcal{H}^* \to \mathbf{Z}_{\geq 0}$ maps $x \in \mathcal{H}^*$ in the canonical manner to the non-negative integer with hexadecimal representation x. Even though $\mathtt{int}$ is not injective (due to leading zeros), $\mathtt{hex}(n) = \mathtt{int}^{-1}(n) \in \mathcal{H}^*$ for $n \in \mathbf{Z}_{>0}$ is defined as the hexadecimal representation of n, without leading $\mathtt{0}$-characters, and $\mathtt{hex}(0) = \mathtt{0}$.

For a prime p the finite field with p elements is denoted by $\mathbf{F}_p$. The multiplicative group of $\mathbf{F}_p$ is denoted by $\mathbf{F}_p^\times$. For $x \in \mathbf{F}_p$ the notation $\widehat{x}$ refers to the canonical lift to the set $\{0, 1, \ldots, p-1\}$ of least non-negative residues modulo p.

For any function $f : D \to R$ with $R \subseteq D$ and $\ell \in \mathbf{Z}_{\geq 0}$ the customary notation f^ℓ is used for ℓ-fold iteration of f:

$$f^\ell(x) = \underbrace{f(f(\cdots f(}_{\ell}x)\cdots)) \text{ for any } x \in D.$$

2 Incorruptible public randomness

Consider the problem where a set G of people wants to agree on a (pseudo)random number s in $\{0,1\}^n$ for some $n \in \mathbf{Z}_{>0}$. They do not trust each other and they do not want any individual to be able to tamper with s in any meaningful way, i.e., being able to force s away from a uniform distribution. Because there is no way to guarantee that any party is incorruptible, an independent third party is not an option, certainly not one that (with a clever slight of hands) flips coins or rolls dice. Complex transparent machines with balls flying around in seemingly total chaos, as commonly used for national lotteries, are easy to fool [23,5]. The situation gets even worse when the winning numbers are generated by a computer [21]. See also [10]. In this section a number of tempting approaches are discussed. They all appear to be flawed both from a security and a usability perspective, at least when the number of participants becomes large.

A naive approach would be to let each $g \in G$ independently choose an $s_g \in \{0,1\}^n$, and to set $s = \bigoplus_g s_g$, where "$\oplus$" denotes exclusive-or. As long as at least a single s_g is chosen uniformly and independently from all others, the resulting s will be uniformly distributed, no matter how the others collude or otherwise fail to follow the rules. But the independence relies on the unrealistic assumption that all choices are perfectly synchronized: the party $l \in G$ that last reveals its choice can target any value v for s by selecting $s_l = v \oplus \left(\bigoplus_{g \neq l} s_g\right)$.

A common way to get around this problem uses commitments, resulting in a two-round protocol. First each party secretly chooses its s_g and publishes a commitment c_g to it; c_g could for instance be $h(h(s_g)||\mathrm{id}_g)$, with h as in Section 1, and id_g a unique identifier for party g. Once all parties have received all commitments, the s_g-values are revealed, the commitments checked, and the value $s = \bigoplus_g s_g$ is calculated as usual. This clearly obviates the possibility for anyone to target a specific value or to bias the result: as long as one participant is honest (and not hacked by the others), the resulting s will be unbiased, irrespective of any colluding group of dishonest other parties.

In theory, using commitments works for any finite number of participants, assuming one is honest and not hacked. In practice, and in particular if the number of participants is large, a number of them may be expected to drop out between the two rounds, either due to technical problems or maliciously, and never reveal a value they have committed to. This results in a denial of service attack: the protocol will never finish because not all committed values are revealed. The attack can be countered by setting a time limit and to compute s as the exclusive-or of just those s_g-values that have been received on time. However, this may make it possible for a malicious party g to influence the protocol in a meaningful way, by deciding, right before the time limit, whether or not to reveal its s_g depending on whether it prefers s or $s \oplus s_g$, where s is the exclusive-or of all values received but not including s_g. Such an attempt may fail if another malicious party independently tries to do the same, but may get worse if m parties collude, allowing them to choose the best among 2^m possible outcomes. A single miscreant controlling m fake participants makes this scenario even worse.

In the next section a slow-timed hash function is described that could be used to resolve this problem in applications where there is no need for an immediate result. The resulting protocol has only one round, and anybody can participate without prior notice. Also, unlike what was presented in this section, it is easy for anyone to take part in the process without need for any special software or technical skills; indeed, it can be as easy as tweeting.

3 *Sloth*: slow-timed hash

In this section *sloth* is presented, a slow-timed hash function that satisfies the two following design criteria: given any $\omega > 0$

- it must be possible to choose the parameters in such a way that computing *sloth* takes wall-clock time at least ω seconds, irrespective of the amount of computer resources available;
- the wall-clock time required to verify that the resulting hash is correct must be modest compared to ω, the computation time required.

3.1 A trivial iterative design

It is not hard to design a function that meets the above requirements. Given a security level k and corresponding cryptographic hash function h as in Section 1, computing the ℓ-fold iteration h^ℓ of h is inherently sequential and does not allow parallelization beyond a small constant number of cores (i.e., not depending on ℓ). Thus, $\ell \in \mathbf{Z}_{>0}$ can be determined such that for any $s \in \mathcal{A}^*$ the computation of $h^\ell(s)$ takes wall-clock time at least ω, on this same number of cores. Although verification of the result requires the same amount of computing, for any suitably chosen n with $2 \leq n \leq \ell$ the wall-clock time for the verification can be reduced by a factor of n using n-fold parallelization on n times as many resources, assuming the n checkpoints $h^{\lfloor i\ell/n \rfloor}(s)$ for $i = 1, 2, \ldots, n$ are kept during the calculation of $h^\ell(s)$. To be able to guarantee a specified wall-clock time, usage of an ASIC-resistant hash function (combined with a regular one) could be considered. Another approach is pursued below.

3.2 Using modular square roots

More interesting solutions require functions that are, unlike cryptographic hash functions, easier to verify than to compute. Obvious candidates are the commonly used cryptographic trapdoor functions. For instance, computing the hash could require the factorization of a large integer or computation of a discrete logarithm in a suitably chosen group, while a witness (containing a factor or the discrete logarithm, respectively) would be provided along with the resulting hash to allow fast verification. But wall-clock times for both these hard problems decrease mostly linearly with the amount of parallel resources available, making proper parameter selection cumbersome if not outright impossible.

Another idea is to use polynomial factorization over finite fields, the simplest case of which is modular square root extraction: given some prime p, calculating a modular square root takes, to the best of current knowledge, at least $\log_2(p) - 2$ unparallelizable modular squarings, whereas a single modular squaring suffices to verify the result. Given $\omega > 0$, it thus suffices to take the smallest p for which the sequence of squarings modulo p require at least ω seconds. For a wall-clock time of about ten minutes on a single core (running at, say, 2.3GHz), p is going to be on the order of hundreds of thousands of bits long, implying that for the computation of the hash the amount of available parallelism (or special purpose architectures) would become more of an issue. The approach sketched below offers more effective assurance against parallelization by combining a still reasonably small p with the earlier iterative idea: the computation is stuck at ℓ necessarily sequential modular square root calculations, each of which is necessarily sequential as well, while the verification, already at least $\log_2(p) - 2$ times faster than the computation, can be sped up by another factor of n by remembering n checkpoints (as inherent in iterative approaches). The size of p can then be set to match the relatively modest verification wall-clock time one settles for (under mild restrictions with respect to the desired security level, cf. below); see also the discussion on wall-clock time guarantee below.

Let $p \equiv 3 \bmod 4$ be a prime number. It follows that precisely one of x and $-x$ is a square for any $x \in \mathbf{F}_p^\times$, and a square root can be calculated by raising the square to the $\frac{p+1}{4}$-th power. If y is a square root of a square $x \in \mathbf{F}_p^\times$, then y and $-y$ are the only two square roots of x. Observing that the canonical lifts $\widehat{y}$ of y and $\widehat{-y} = p - \widehat{y}$ of $-y$ have different parities, define $\sqrt[+]{x}$ as the unique square root of x with even lift and $\sqrt[-]{x}$ as the unique one with odd lift. This leads to the following permutation on $\mathbf{F}_p^\times$:

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

with inverse

$$\rho^{-1}(y) = \begin{cases} +y^2 & \text{if } \widehat{y} \text{ is even;} \\ -y^2 & \text{if } \widehat{y} \text{ is odd.} \end{cases}$$

Simply iterating the permutation ρ allows a shortcut in the computation of ρ^ℓ, as shown in Section 3.4. This can be avoided by adding a layer of unstructured confusion in the following manner.

Let σ be an permutation on $\mathbf{F}_p^\times$ such that both σ and σ^{-1} are easy to compute. Given σ define $\tau = \rho \circ \sigma$ and use τ^ℓ for some appropriately chosen ℓ as a slow-to-compute function with easily computable verification function $(\tau^\ell)^{-1} = (\tau^{-1})^\ell = (\sigma^{-1} \circ \rho^{-1})^\ell$. Note that, given ω, the value of ℓ will mostly depend on the size of p. The resulting slow hash function is described below. In Section 3.4 it is shown how σ may be chosen so that undesirable shortcuts are avoided that would allow computation of τ^ℓ faster than by sequential ℓ-fold iteration of τ, while maintaining easy verification.

3.3 *Sloth*

Let k be a security level, h a corresponding hash function (Section 1), $p \equiv 3 \bmod 4$ a prime such that $p \geq 2^{2k}$, and τ as in Section 3.2. Compared to regular hash functions from $\mathcal{A}^*$ to $\mathcal{H}^{k/2}$ (cf. Section 1), the slow-timed hash function *sloth* produces two additional outputs. In the first place a *witness* is provided that allows fast verification of the resulting hash value. Furthermore, to enable disclosure of its input value only at a later point in time (it may be undesirable if others simultaneously run *sloth* on the same input) while avoiding the possibility of selecting a particular input from a number of inputs (as any number of copies of *sloth* could be run in parallel), a commitment to the input will be output right away, before *sloth* embarks on its long, wall-clock time ω consuming iteration. In applications where a commitment is not relevant, this first component of the output can be discarded. Thus, *sloth* maps elements of $\mathcal{A}^*$ to $(\mathcal{H}^{k/2})^2 \times \mathbf{F}_p^\times$, where the first component in $\mathcal{H}^{k/2}$ is the commitment, the second component in $\mathcal{H}^{k/2}$ is the resulting hash, and the $\mathbf{F}_p^\times$-part is the witness that allows fast verification.

Sloth is defined below. The time-security of *sloth* (i.e., shortcuts during its calculation are impossible; see also the Appendix for a precise notion of security) relies on the *slowness assumption* implicitly made in Section 3.2 that extraction of modular square roots cannot be done faster than using a modular exponentiation – in similarity with hardness assumptions for cryptographically relevant problems. It would require a new idea to prove that the slowness assumption is incorrect, in which case *sloth* may fall back on modular roots of more complex polynomials.

```
Sloth. Let $s \in \mathcal{A}^*$ be the input.
  Let $u \leftarrow h(s)$.
  Return $h(u)$ as the first component of the output and continue to the next step.
  Let $w_0 \in \mathbf{F}_p$ be such that $\widehat{w}_0 = \mathtt{int}(u)$ (note that $0 \leq \mathtt{int}(u) < 2^{2k} \leq p$).
  For $i = 1, 2, \ldots, \ell$ in succession do the following:
    Let $w_i \leftarrow \tau(w_{i-1})$.
  Return $h(\mathtt{hex}(\widehat{w}_\ell))$ and $w_\ell$ as the second and third components of the output and quit.
```

The output $(c, g, w) \in (\mathcal{H}^{k/2})^2 \times \mathbf{F}_p^\times$ of *sloth* with input s thus consists of the commitment c, the hash g, and the witness w. It can be verified as follows.

```
Sloth-verification. Let $(s, c, g, w) \in \mathcal{A}^* \times (\mathcal{H}^{k/2})^2 \times \mathbf{F}_p^\times$ be the input.
  Let $u \leftarrow h(s)$.
  If $c$ is not equal to $h(u)$ then return “false” and quit.
  If $h(\mathtt{hex}(\widehat{w}))$ is not equal to $g$ then return “false” and quit.
  For $i = \ell, \ell - 1, \ldots, 1$ in succession do the following:
    Replace $w$ by $\tau^{-1}(w)$.
  If $\widehat{w}$ equals $\mathtt{int}(u)$ then return “true” and quit.
  Return “false” and quit.
```

3.4 Choices for the permutation σ

In the random oracle model it can be proved that *sloth* is indeed inherently sequential and no information about the outcome can be guessed with non-negligible advantage in less time than that required to sequentially compute $\Omega(\ell \log(p))$ multiplications in $\mathbf{F}_p$, for $p \to \infty$. The proof (cf. Appendix) relies on the assumptions that σ is a random permutation and that computing a square root of a random square in $\mathbf{F}_p^\times$ requires an exponentiation in $\mathbf{F}_p$ (cf. the above slowness assumption); this is made more precise in the Appendix. The second assumption does not seem to be unreasonable given the current state of the art of modular square root extraction. Concerning the former assumption, as argued in the present section good time-security can still be obtained without it, and it seems that simple choices for σ suffice.

Omitting σ. It is first shown that, as mentioned above and assuming that $p \equiv 3 \bmod 4$, the computation of ρ^ℓ indeed allows a shortcut. Thus, omitting σ is not an option. For $s \in \mathcal{A}^*$, let $w \in \mathbf{F}_p$ be such that $\widehat{w} = \mathtt{int}(h(s))$, as in the definition of *sloth*. With σ equal to the identity function on $\mathbf{F}_p$, the iteration computes $\tau^\ell(w) = \rho^\ell(w)$. It follows that $(\rho^\ell(w))^{2^\ell} = z$ where z is the unique square among w and $-w$. Because, as is easily seen, $\pm\rho^\ell(w)$ are the only two roots of $X^{2^\ell} - z$ in $\mathbf{F}_p$, it suffices to determine one of those two roots and to use the fast verification to decide if it or its negative equals $\rho^\ell(w)$. With $e = \frac{p+1}{4}$, a square root of $z \in \mathbf{F}_p^\times$ is given by z^e and, iterating this for i, a root of $X^{2^i} - z$ is given by z^{e^i}. Thus, with $j = e^\ell \bmod (p-1)$, a root of $X^{2^\ell} - z$ is found by computing z^j at the cost of an exponentiation in $\mathbf{Z}/(p-1)\mathbf{Z}$ and an exponentiation in $\mathbf{F}_p$.

Swapping neighbors. The problem of omitting σ lies in the fact that $\rho^\ell(w)$ is the root of a simple, explicitly given polynomial. Consider the permutation σ on $\mathbf{F}_p^\times$ with $\sigma = \sigma^{-1}$ that swaps neighbors:

$$\sigma(x) = \begin{cases} x+1 & \text{if } \widehat{x} \text{ is odd;} \\ x-1 & \text{if } \widehat{x} \text{ is even.} \end{cases}$$

With w as above, it is not clear how to express $(\rho \circ \sigma)^i(w) = \tau^i(w)$ as the root of a polynomial. For instance, if w is a square, then $\rho(w) = \sqrt[+]{w}$ is even so that $\tau(w) = \sqrt[+]{w} - 1$, which is a root of $(X+1)^2 - w$ and which has an odd lift. But the latter does not give any information about the quadratic residuosity of the root $\sqrt[+]{w} - 1$. Therefore, without computing that quadratic residuosity, it is only known that $\tau^2(w)$ is a root of one of $((X+1)^2+1)^2 - w$ and $((X+1)^2-1)^2 - w$. In general, $\tau^i(w)$ is a root of precisely one of the 2^i polynomials $(\ldots((X \pm 1)^2 \pm 1)^2 \ldots \pm 1)^2 - w$, but there does not seem to be an efficient way to predict which of the 2^ℓ polynomials has $\tau^\ell(w)$ as root. Of course, all 2^ℓ polynomials could be tested in parallel, but even finding the roots of a single one seems to require at least the same amount of unparallelizable time as the successive square root extractions in *sloth*, as all polynomials are dense and of degree 2^ℓ over $\mathbf{F}_p$. Therefore, using a permutation σ that simply swaps neighbors seems to be time-secure enough. Nevertheless, it still preserves some algebraic structure, which could lead to unforeseen adversarial strategies.

Using binary permutations. As mentioned above and shown in the Appendix, time-security of *sloth* can be proven – in the cryptographic sense of the word – if σ is a random permutation of $\mathbf{F}_p^\times$. Keyed block ciphers are commonly used to emulate allegedly good pseudo-random permutations of $\{0,1\}^n$ for $n \in \mathbf{Z}_{>0}$. It is shown how they can be used to define permutations of $\mathbf{F}_p^\times$.

Given $\mathbf{F}_p^\times$, select an integer $n \geq \log_2(p)$. Identifying the set $\{0,1\}^n$ with the set of integers $\{0,1,\ldots,2^n-1\}$, a permutation $\varsigma : \{0,1\}^n \to \{0,1\}^n$ can be regarded as a permutation of $\{0,1,\ldots,2^n-1\}$. With the map $\pi : \mathbf{Z} \to \mathbf{F}_p$ that maps $a \in \mathbf{Z}$ to the $y \in \mathbf{F}_p$ for which $\widehat{y} \equiv a \bmod p$, and the map $\iota : \mathbf{F}_p \to \{0,1,\ldots,p-1\}$ that maps $x \in \mathbf{F}_p$ to $\widehat{x}$, this induces a map $\widetilde{\sigma} = \pi \circ \varsigma \circ \iota : \mathbf{F}_p \to \mathbf{F}_p$. Unfortunately, $\widetilde{\sigma}$ is not necessarily a permutation on $\mathbf{F}_p^\times$, because there may be elements $x \in \mathbf{F}_p^\times$ for which $\varsigma(\widehat{x}) \notin \iota(\mathbf{F}_p^\times) = \{1,2,\ldots,p-1\}$.

However, a permutation of $\mathbf{F}_p^\times$ can be obtained if, depending on the input x, the permutation ς is performed as often as required until $\widehat{x}$ is mapped to $\{1,2,\ldots,p-1\}$:

$$\sigma(x) = \pi(\varsigma^v(\widehat{x})), \text{ with } v > 0 \text{ minimal such that } \varsigma^v(\widehat{x}) \in \{1,2,\ldots,p-1\}.$$

For a uniformly random $x \in \mathbf{F}_p^\times$, the probability that $\varsigma(\widehat{x}) \notin \{1,2,\ldots,p-1\}$ is at most

$$\frac{\#\{0,p,p+1,p+2,\ldots,2^n-1\}}{\#\{1,2,\ldots,p-1\}} = \frac{2^n-p+1}{p-1},$$

which can be made negligibly small by selecting p as $2^n - \epsilon$ for a small positive integer ϵ. As a result, for such primes a permutation of $\mathbf{F}_p^\times$ is obtained that will in practice be as efficient as the underlying block cipher. The fact that for rare elements of $\mathbf{F}_p^\times$ the computation takes more time cannot be exploited to speed up the calculation of τ – indeed, it only further slows it down. The additional overhead for the verifier is negligible.

Given that the analysis is generic in ς it can be improved for specific instances, and it allows great flexibility in the design of ς, as long as p is close to 2^n: binary operations such as exclusive-or, shifts, and bit(s) swapping may all be used. Because such choices do not allow a meaningful algebraic interpretation in $\mathbf{F}_p$ shortcuts in the calculation of *sloth*, as discussed above, are avoided. If the binary operations are restricted to the least significant bits (in a similarly fast but more liberal manner than just swapping neighbors) the prime p does not have to be chosen close to 2^n.

Choices for the prime p. To conclude this section, Table 1 lists the verification times depending on the choice of the size of p, for two practically relevant permutations: swapping neighbors and block cipher based. In all cases, the number of iterations ℓ is chosen such that the computation of *sloth* takes ten minutes. When σ is a simple binary operation the number of iterations and the verification times are similar to those for swapping neighbors.

Table 1. For *swapping neighbors* random n-bit primes are used. For *AES-128 block cipher*, the primes are of the form $2^n - \epsilon$ and the largest possible (with ϵ as listed), $\varsigma =$ AES-128, and $\sigma = \pi \circ \varsigma^v \circ \iota$ for an input-dependent v as explained in the text (which, in the experiments, always turned out to be equal to one). The columns *iterations* list the number of iterations of the permutation needed to obtain a runtime of ten minutes for the computation of *sloth*. The columns *verification time* list the time in seconds to verify the resulting value, to be compared to the 600 seconds of the generation (and scales linearly with other computation wall-clock times that may be used). The computations where performed on a 2.3 GHz Intel Core i7, using the libraries GMP for multiprecision numbers and OpenSSL for AES. No attempts were made at optimization, as they equally affect computation and verification times.

	swapping neighbors			**AES-128 block cipher**	
n	iterations	verification time	ϵ	iterations	verification time
256	31’000’000	7.7 seconds	189	20’000’000	177 seconds
512	6’750’000	2.7 seconds	569	5’400’000	97 seconds
1024	1’100’000	0.84 seconds	105	1’000’000	37 seconds
2048	155’000	0.33 seconds	1557	150’000	11 seconds
4096	23’600	0.15 seconds	2549	23’000	3.8 seconds

Wall-clock time guarantee. The wall-clock time required for a calculation consisting of ℓ necessarily sequential steps, each consisting of at least $\log_2(p)$ necessarily sequential modular multiplications, can easily be measured for a single core running at a certain speed and using any standard software package. This should give some information about the shortest possible wall-clock time using the fastest conceivable software on the optimal number of fastest possible cores that could be employed given the value $\log_2(p)$ at hand. The simplest speed-up is overclocking: clock-speeds close to 9GHz have been reported (cf. [9]). For parameter choices as reported in Table 1 it would lead to a wall-clock time guarantee of about three minutes. The multi-core approaches and runtime figures presented in [2,13] suggest that for the lower range of $\log_2(p)$-values no further speed-ups have been obtained. Similarly, a shortest possible wall-clock time using special purpose hardware can be derived, but published results (cf. [14,15,19]) seem to suggest that the software results listed in Table 1 are again hard to beat. For all time-security arguments below, a conservative lower bound estimate of two minutes will be used. But in the practical, down-to-earth implementation of *sloth* the originally aimed for wall-clock time of ten minutes on a single standard core as above will have to be dealt with.

4 Unicorn: uncontestable random numbers

In this section *unicorn* is described, one of many conceivable scenarios how *sloth* may be used to generate uncontestable random numbers: everyone may contribute inputs to *unicorn* to influence its result while no one will be able to knowingly bias the result one way or another, everyone can quickly verify that the resulting random numbers have been generated according to the *unicorn* protocol, and all participants can check correct inclusion of their contribution. However, anyone who wants to use the outcome of a particular execution of *unicorn* without taking part in it, will have to trust that at least one participant followed the rules: as is the case for current lotteries and live-broadcast drawings, outsiders have no choice but to believe the integrity of the outcome. Unlike lotteries, however, anyone has the opportunity to take part in *unicorn* even without being physically present. The only requirement is to be on time. The formal security notion of the incorruptibility of *unicorn* is described in the Appendix, and proved in the case where the permutation σ used by *sloth* is a random permutation.

Given a security level k and corresponding hash function h (cf. Section 1), each execution of *unicorn* proceeds on a time line from t_{-2} to t_2 (with $t_i < t_j$ if $i < j$) as described below.

Unicorn

time t_{-2}: It is publicly announced that public data gathering will take place during the time interval $[t_{-1}, t_0)$. This announcement will be made on a public website, along with instructions how data may be contributed. For instance, contributors could be invited to send a tweet with a specified hashtag.

time t_{-1}: Data reception starts: all data received will be concatenated, in the order in which it arrives, to form the public part $s_0 \in \mathcal{A}^*$ of the input to *sloth*.

time t_0: Data reception stops, the resulting s_0 is published on the website right away, and *sloth* is applied to the concatenation $s = s_0||s_1$ of the public data s_0 and the independently generated part s_1. The first component $c = h(h(s))$ (the commitment to the input) of the output of *sloth* is published on the website as soon as it becomes available (i.e., almost instantaneously), say at time $t_0 + \gamma$ for a very small positive γ. Although γ in principle depends on the size of s_0, it can be expected to be a fraction of a second because in practical circumstances $h(s)$ can be computed on-the-fly while contributions to s_0 are still arriving. Even in the worst case γ will be on the order of seconds at most, because it takes only about a second to hash 100MB of data (which will first have to be downloaded from the server).

time $t_1 \geq t_0 + \gamma + \omega$: The second component g (the hash, i.e., the uncontestable random number) and the third component w (the witness) of the output of *sloth* are published on the website, along with the independently generated input s_1.

time t_2: At this point in time, everyone interested should have been able to perform the *sloth* verification step for (s, c, g, w).

Choosing $\boldsymbol{t_{-1}}$. The value that needs to be picked with most care is t_{-1}. If all public contributions to s_0 are sent and received immediately after t_{-1}, the final s_0 may be known shortly after t_{-1} as well. If $t_0 - t_{-1} > \omega$, with ω the targeted computation time for *sloth*, the party in control of selecting s_1 has the opportunity to finish before time t_0 parallel computations of *sloth* for many distinct possibilities for the value of s_1. A particular s_1 choice can then be committed to at time t_0, with the biased corresponding output of *sloth* (already known at time t_0) publicly revealed only at time t_1.

It follows that $t_0 - t_{-1}$ must be small compared to the fastest conceivable wall-clock time required for the computation of *sloth*. Given the considerations at the end of Section 3.4 and a targeted ten minute wall-clock time for the computation of *sloth*, the point in time t_{-1} should not be more than two minutes before t_0. Or, if that is not an option, participants may be encouraged to submit their contributions close to the deadline t_0. Indeed, that is what smart participants will do anyway, as submitting at time $t < t_0$ implies one has to trust that *sloth* must take wall-clock time more than $t_0 + \gamma - t$ to compute. Combined with the observable, very small γ (as $t_0 + \gamma$ is the point in time that the commitment value is published) the time-window for surreptitious *sloth* computations can be made infeasibly short.

Generating s_1. The value $s_1 \in \mathcal{A}^*$ complementing the public input s_0 is included in *unicorn* for the following reasons:

1. to increase the entropy, if s_0 (which may be lacking altogether) has no or little entropy;
2. to replace (or complement) the s_0-contribution (e.g., the tweet) for people who are physically present, in a way that is instantaneous and guaranteed to be included;
3. to add a salt-value that is not shared before time t_1, so other parties cannot perform *sloth* before t_1;
4. to include elements that **no one** can control (e.g., the weather; cf. the *unicorn* application in Section 5).

At this point, it should be emphasized that a party who contributed to s_0, e.g., via a tweet, does not need to trust that s_1 was generated honestly; as proven in the Appendix, the incorruptness of the outcome can be deduced by any participant from their own (honest) contribution to s_0. The *raison d'être* of s_1 are the four points above, and adding it does not weaken the trust a participating party can have in the outcome.

The value s_1 can be independently generated in the following manner. It is assumed that the computing device on which *unicorn* is executed has a digital camera with an unobstructed view of a public area where the goings-on can be monitored (and participated in) by anyone who desires to do so. At time t_0 the camera will take a picture (or, alternatively, a short video clip, possibly including a sound bite), resulting in a `jpg`-file (or other applicable format) s_1. For reasons set forth below, this set-up may be complemented by a full-time webcam connected to an independent other computer, the live output-stream of which is made available on the same website as above.

Given the uniqueness of the camera's point of view, the type of scene captured, and the alleged properties of the hash function h, the value $h(s)$ and the second component g of the output of *sloth* may be assumed to behave as random elements of $\mathcal{H}^{k/2}$ (and thus as random integers in $\{0, 1, \ldots, 2^{2k} - 1\}$), by any party including the party C that executes *unicorn*, and for any value of t_0. So far this assumption has not been contradicted by [16]. Extensive experiments, also at night-time and with or without s_0-values (or s_0-values identical to previously used ones), show that the $h(s)$-values resulting from consecutively taken pictures s_1 are uncorrelated (cf. [16]). This lack of correlation between the $h(s)$-values may be accredited to the cryptographic hash function h and the necessary difference between the `jpg`-encapsulations (among others caused by even the tiniest time difference) even though the actual `jpg`-payloads of consecutive pictures are obviously strongly correlated to the human eye. However, despite this strong perceptual correlation, the bit-level difference between any pair of distinct `jpg`-payloads was found to be large compared to any regular cryptographic security level, both before and after the compression used to generate the payload, and for any reasonable camera resolution. It is a subject of further investigation to determine how this difference behaves as a function of the temporal closeness of consecutive pictures. In any case, given *any* number of past s_1-values it is infeasible – to anyone – to predict a future one.

Once the picture s_1 is published, parties that have monitored the scene at time t_0 can attest to its correctness, for instance by pointing out that the picture indeed shows them waving at the camera – parties who cannot be present may prefer to contribute a tweet to s_0 instead. Independently it can be checked that there are no discrepancies with what appeared on the webcam's live-stream around time t_0. For additional assurance the scene captured by the camera could include a screen that is constantly refreshed with the latest information from popular news websites or live TV-broadcasts, thereby independently fixing s_1 in time.

With s_1 fixed in time, and its commitment immediately published, there is no opportunity for the party C that is in control of the system to select s_1 in a meaningful manner from any number of alternatives, because of the slowness of *sloth*. Without *sloth* biased results are conceivable: it suffices for C to have access to enough computational resources to simultaneously test many alternatives in a short amount of (wall-clock) time.

Earlier independent work that collects entropy in a similar fashion can be found in [22,8]; see also [18].

Formatting and restricting s_0. For reference and cosmetic reasons begin and end markers may be used for the public part s_0 of the input: before time t_{-1} data collection could be initiated with the unix command

```
date "+[start %Y%m%d:%H%M%S" > s0,
```

terminating it at time t_0 with

```
date "+%Y%m%d:%H%M%S stop]" >> s0.
```

An invitation to contribute data may trigger attempts to cause trouble, so the data received will have to be filtered and may have to be restricted to strings over a subset of $\mathcal{A}$.

Number of participants. The number of messages from participants that an actual implementation can handle depends on the physical set-up, the amount of hardware employed, and, if applicable (cf. potential usage of tweets), on the resources of the social media provider(s) involved. The current record number of tweets per second suggests at most ten million messages per minute, which sounds challenging to deal with. But given the very limited public concern about the issues raised in this paper ([3] may be the most prominent one, and only concerns a niche market) there is no reason to be concerned that a single desktop computer would not be able to handle the load. On the other hand, one never knows what sudden enthusiasm there may be to *really* play the lottery, to make one's mark on the FIFA pool selection process, or for the rise of Athenian democracy 2.0.

5 *Trx*: trustworthy random elliptic curves service

This section proposes *trx*, a service that provides a stream of trustworthy random elliptic curve parameters suitable for cryptographic applications. It uses a mild adaptation of *unicorn* and is currently being implemented. Thus, compared to other methods that have been used to generate elliptic curve parameters, *trx* introduces a new way to deal with the trust issue: everyone can influence and verify the choices made by *trx*, but no one (including party C that controls the set-up) can knowingly affect the choices to anyone's advantage or disadvantage. Because the resulting parameters cannot be predicted or effectively manipulated, the possibility is prevented of prior cryptanalysis or of targeting malicious choices. Along with each parameter choice, *trx* provides information that allows any party to ascertain that the resulting parameters were calculated in a deterministic manner based on the random number produced by *unicorn*. *Trx* does not enable the user community to fully exploit the wealth of suitable random curves in a fully personalized manner, the possibly preferred method mentioned in Section 1: that would require substantially faster point counting. But *trx* does away with the fixed small set of elliptic curve parameters currently used, and it allows usage of parameters that are frequently refreshed and that cannot have been scrutinized before.

The above summary presents *trx* from a high level point of view. The remainder of this section describes two parts in more detail: the physical set-up of the *unicorn*-variant used, and the deterministic computation of elliptic curve parameters as a function of a seed value. Many more or less equivalent methods exist for the latter computation, none of which are particularly interesting and one of which can be found below – for documentation purposes only, and with further details provided when *trx* comes online. The two main steps are followed by a brief description of the planned long term operation of *trx* and a short discussion on various trust issues.

***Trx* set-up.** The currently envisioned set-up consists of a computing device D at a physically secure location. D has a single digital camera with an unobstructed view of a public, outside area that is sufficiently large and busy: the view comprises a parking lot with working street lights, a relatively busy road, a glittering lake, and a mountainous part of a foreign country, all across the street from C's third floor office and all with snow patterns, cloud formations, and other occurrences that are beyond anyone's control.

Two options are under consideration for the communication with D. The first, straightforward one would use a regular TLS connection to a nearby web-connected server that instantaneously posts data received from D (t_i-values, *sloth* commitments, other *sloth* results along with their s-values, and of course everything related to the resulting elliptic curve parameters) and transmits *unicorn*'s s_0-contributions (cf. Section 4) to D, everything at the shortest possible delay (on the order of a fraction of a second).

The other option under consideration would entirely shield D from any attempts at interference with its operations, by only allowing outgoing communications from D via a data diode. This implies that *unicorn* must rely exclusively on the value s_1 as generated by D, as there is no way for contributions made to s_0 to reach D. For a rather arcane activity such as elliptic curve parameter generation which, furthermore, runs continuously, little interest from the public at large may be expected, so excluding participation from parties who cannot be physically present is probably not a serious issue. But it puts a heavier burden on party C running the system to present convincing evidence that the picture taken at time t_0 is not replaced by a possibly manipulated one. Pre-announcing the moments in time that the pictures will be taken (the points in time t_0 for each execution of *unicorn*) along with a webcam running on the webserver capturing the same scene (from a slightly different angle) goes a long way to address this problem: this allows validation of the scene of the picture by the time the picture and the resulting set of parameters get published. It also allows any interested party to visit the public area captured by the camera at the scheduled time to add – and later check the presence of – a personalized touch to the parameter generation process (which does not require physical presence if s_0 is used, as in the other set-up). As mentioned in Section 4 an additional confidence-inspiring measure would be to include in the scene captured by the camera a screen with constantly updated independent live information. The amount of trouble to be invested to address this concern should be commensurate with the perceived practical importance of *trx*. As a general piece of advice to prospective users of the resulting parameters: if you don't like the picture, don't use the resulting parameters and wait for a next batch.

Elliptic curve parameter generation. Let k be a security level and h the corresponding hash function, as in Section 1. As an example of elliptic curve parameter generation consider *random twist secure curves.* In this case a triple $(\delta, \alpha, \beta) \in (\mathcal{H}^{k/2})^3$ is called *acceptable* if

- the number $q = \texttt{int}(\delta)$ is a $2k$-bit prime;
- the pair $(a, b) \in (\mathbf{F}_q)^2$, where the lifts $\widetilde{a}, \widetilde{b} \in \{0, 1, \ldots, q-1\}$ of a and b are modulo q equal to $\texttt{int}(\alpha)$ and $\texttt{int}(\beta)$, respectively, defines an elliptic curve E over $\mathbf{F}_q$ such that the order of the group of points of E over $\mathbf{F}_q$ and the order of the group of points of the quadratic twist of E over $\mathbf{F}_q$ are both prime.

This definition is just an example of one of many different sets of criteria that can be imposed. Refer to [4] for a good overview of additional or different requirements (bounding the embedding degree from below, other types of curves such as Edwards curves, etc.). Information specifying an appropriate base point can trivially be added.

A method is described that given an input value $g \in \mathcal{H}^{k/2}$ (as produced by *unicorn*) deterministically determines an acceptable triple in $(\mathcal{H}^{k/2})^3$, and that can trivially be changed to cater to any other definition one sees fit for a triple to be acceptable. Let $F = \{\delta, \alpha, \beta\}$ be a set of iterated hash functions from $\mathbf{Z}_{\geq 0}$ to $\mathcal{H}^{k/2}$ with $\delta(0) = h(g||\texttt{p})$, $\alpha(0) = h(g||\texttt{a})$, $\beta(0) = h(g||\texttt{b})$ (with "$||$" denoting concatenation), and $f(i) = h(f(i-1))$ for all $f \in F$ and $i \in \mathbf{Z}_{>0}$. Let $I(-1) = -1$ and $I(j) = \min\{i : i > I(j-1),\ \texttt{int}(\delta(i)) \geq 2^{2k-1},\ \texttt{int}(\delta(i)) \text{ is prime}\}$.

Random twist secure elliptic curve parameter generation

```
For $i = 0, 1, 2, \ldots$ in succession do the following:
  For $j = 0, 1, 2, \ldots, i$ in succession do the following:
    For $v = 0, 1, 2, \ldots, i - j$ in succession do the following:
      If the triple $(\delta(I(j)), \alpha(v), \beta(i - j - v))$ is acceptable:
        return the triple along with the values of $i$, $j$, and $v$ and quit.
```

Checking acceptability of triples can be done using a standard software package (such as MAGMA or Sage) or using one's own software. For each triple that was found not to be acceptable, a small amount of data may be provided that would facilitate a check that the acceptable triple as produced is indeed the first one (given the chosen enumeration). Experiments are underway to decide on the most efficient approach. On a single core running at 2.3GHz parameter generation times vary from less than two hours for $k = 128$ to about a week for $k = 256$.

If parameter generation (as above, or of simple modifications that cater to other requirements) fails due to a cycling hash then there is obviously cause for celebration, but h should be replaced – not just for the purposes of the present paper – by a hash function for which the security assumptions have a higher chance to be correct. So far all iterated hashes generated as above passed the tests from [16].

Operation of *trx*. The implementation of *trx* that is currently underway will run twelve independent processes, one on each of twelve cores of a desktop computer: for each security level $k \in \{128, 192, 256\}$ four different types of twist secure elliptic curves may be generated (with q, a, and b as above): either a random q or q chosen in a fixed set of $2k$-bit primes that allow fast modular arithmetic (pseudo-Mersenne primes), both either with random a and b or with fixed a (i.e., $\widehat{a} = q - 3$) and random b. Depending on feedback that may be received, either of these possibilities can be replaced by others that are felt to be more useful or desirable; or a wider variety of parameter choices may be offered (though less efficiently if using the same hardware) by alternating between different generation processes.

Each of the twelve processes operates in the same manner, independent of each other and each with their own unique process-identifier (or hashtag). Given some pre-announced future moment in time t_{-2}, and assuming public data are accepted, the following steps are performed in immediate succession:

1. The future point in time t_{-2} along with t_{-1} a few minutes later than t_{-2} and t_0 two minutes later than t_{-1} are transmitted to the server and publicly announced (for the process-specific identifier or hashtag).
2. Assign an adequate initial value to s_0 and wait until time t_{-1}.
3. During the time interval $[t_{-1}, t_0)$ concatenate to s_0 all process-relevant data received from the server and start calculation of $h(s_0)$.
4. At point in time t_0 take a picture resulting in a `jpg`-file s_1, transmit $h(s_1)$, s_0 as received (which may be a proper subset of the data sent by the server, and possibly with an appropriate termination appendage) and $h(s_0)$ to the server for immediate publication on the website, and start the calculation of *sloth* (calibrated to take ten minutes). The first component of the output of *sloth*, the commitment value c, is immediately (i.e., at time $t_0 + \gamma$ for a very small positive γ) transmitted to and published by the server.
5. At a point in time about ten minutes after t_0, the calculation of *sloth* is completed and the applicable elliptic curve parameter generation process is started using as input the second component g of the output of *sloth*.
6. Define t_1 as the point in time that the elliptic curve parameter generation process is finished. At this point in time, transmit the picture s_1, the second and third components of the output of *sloth* (the hash g and the witness w), and the output of the elliptic curve parameter generation process to the server for immediate publication.
7. Replace t_{-2} by t_1 plus a few minutes and return to Step 1.

The scenario where no public data are accepted follows in a straightforward manner.

Trust issues. As argued above, the only advantage that the party C in control of the system gets is a headstart cryptanalyzing the elliptic curve parameters resulting from the calls to *sloth*. When C plays fair and publishes newly generated parameters without delay, this cryptanalytic advantage is on the order of seconds. From the output of the parameter generation process and additional data that may be provided along with it (as suggested above) it can easily be inferred how long the computation should have taken, so the possibilities for cheating are limited. C could, surreptitiously, have access to more computational resources and perform parameter generation much faster. On average, this buys C at most a week headstart for the cryptanalysis for the highest security level $k = 256$, and less than two hours for $k = 128$: either way, the advantage is insignificant compared to the alleged security provided.

Although the correctness of each resulting parameter set can quickly, independently, and conveniently be checked, it can hardly be expected that all users will consistently do so. Once *trx* is established (if ever) as a trustworthy service, the parameters produced by it could become trusted by default, at which point party C could decide to sneak in a manipulated curve, take advantage of it and leave with the profits, never to be heard of again. Obviously, this would tarnish C's reputation. It may help if C is a party that can reasonably be expected to be concerned about reputational loss – unfortunately, this is something one never knows in advance.

6 A tool for democracy

Randomness can play a crucial role in various models of governance. The first known democracy in the world, in the Greek city-state of Athens, distributed the power to assemblies of randomly selected citizens. In today's world, the benefits of sortition-based democracy are defended by some as a fairer alternative to elected assemblies. Without going as far as a full Athenian-like democracy, more familiar modern-day models of governance can make use of random-sample voting: instead of consulting the full population for elections or referenda, one can randomly select a small, yet statistically significant, sample of voters. Such a system is advocated as leading to a better quality voting at a far lower cost [6]. In Switzerland, the population regularly votes for diverse elections, referenda, and popular initiatives. For reasons of cost, these so-called *votations* are organized four times per year, each time about multiple topics. Being solicited so often about so many questions sometimes far from their everyday life, voters can feel overwhelmed and unable to develop an informed opinion for each of them. In such a system, replacing the full population by random samples would have multiple advantages: while remaining statistically representative as long as the sample has an appropriate size, the costs would be dramatically reduced. At the same time, each voter would be requested much more rarely and about a single question at a time, allowing for a more important involvement.

Whether it be for legislative assemblies, small samples of voters, or juries for public policy, the random sampling must be conducted in a trustworthy, incorruptible and verifiable manner. Advocating for setting up a Parliament of randomly sampled citizens in Northern Ireland, John Garry wrote (cf. [12]):

> There are three crucial ingredients for a high quality democracy: a very large hat, a pen and lots of small bits of paper. Write the name of each citizen in the land on a bit of paper, put all the bits of paper in the hat, close your eyes and pluck out 500 names from the hat. Write to each of the 500 saying: "Congratulations, you have been picked as one of the 500 people who will run the country for the next five years."

Of course a giant hat and millions of pieces of paper are metaphorical and do not constitute a practical setup: a feasible, fair and incorruptible procedure needs to be defined. Designing such a method is a delicate task which [10] tries to address. As in *trx*, it boils down to two main components: a public random number generator, and a deterministic procedure which when fed a number outputs a sample of citizens (respectively, an elliptic curve), in a completely unambiguous and verifiable manner. This second component needs to be precisely defined and published ahead of the random number generation. It should also be unbiased: no particular set of voters should be advantaged as long as the random seed is uniform. Straightforward ways to achieve this could use cryptographic hash functions and reductions modulo the size of the pool of potential voters, and an unambiguous ordering of the list of potential voters. An example of a precise procedure can be found in [10, Paragraph 4], and for discussions on how to design this component in order to render vote buying or other ways to influence the voters impractical, see [6].

In the following, the random number generation is addressed. It is critical that the random number is generated in an incorruptible manner. If the party conducting the generation is simply asked to provide a number without any justification that it is "random", it would be trivial for them to provide a "random-looking" number carefully cooked to result in a biased sample of voters. Classical methods as described in [10, Paragraph 3] are not that trivial to fool, yet are still subject to corruption or other forms of manipulation. The author of [10] suggests to select different sources of randomness in advance (examples include government run lotteries, the daily balance in the US Treasury, or sporting events), and to combine the outcomes via a cryptographic hash function. Besides a few technical issues that need to be addressed (enough entropy has to be gathered, and the format of all the data needs to be canonicalized) matters of greater concern arise: various strategies are conceivable to manipulate those sources. Great care needs to be taken to audit each of them, and even if various presumably independent parties are involved, not every skeptical citizen can be given the chance to personally make sure everything went right and no form of manipulation was going on.

This trust issue can be addressed by *unicorn*. A similar setup as used for *trx* would allow any citizen to contribute very easily to the random number generation, by simply publishing a tweet including a specified hashtag during an announced time interval. Then, by checking if their tweets appear in the input of *sloth*, and running the fast *sloth* verification, they can make sure the outcome has not been manipulated. The very concerned citizen will make sure to tweet high entropy, unpredictable data, as close as possible to the time t_0. This process transfers the power from the auditing authorities or media, to the hands of any person willing to get involved.

7 Conclusion

It was shown how high entropy public random values can be generated in a verifiable and trustworthy manner. Applications were presented to parameter selection for elliptic curve cryptosystems and to democratic random sampling.

In the same vein as *trx*, *unicorn* could be used to generate constants for other kinds of cryptographic standards. In [20], it is described how to design the constants of the S-boxes in some block ciphers to hide a trapdoor. Similarly, [1] exposes a way to weaken SHA-1 and find collisions by simply tweaking its round constants.

Public randomness has also found applications in the context of cryptographic elections. For auditing via random selection, or the generation of random challenges in cryptographic election systems, the source of randomness has to be unpredictable and incorruptible. As an example, in at least two cases, the random generation was based on financial data (in [11] and [7]). Entrusting *unicorn* with the random number generation would allow anyone to verify the outcome irrespective of one's faith in the unmalleability of published financial data.

Unicorn could also be considered as a building block for a secure random beacon. A random beacon is an online service that makes available fresh random numbers at regular intervals. Unlike the current NIST randomness beacon [17], the use of *unicorn* would provide a way to verify that the claimed random values are indeed fresh, and have not been cooked in advance by the service provider, without the need to be physically present to check that their complex quantum source of entropy is really doing its job. A service that provides allegedly random numbers whose trustworthiness cannot be verified is frowned upon, in particular if the service comes with precise ways how to use the random numbers along with an imprecise description of how **not** to use them.

As a final note of warning, it could be tempting to use the live events capturing method to collect entropy (cf. generation of s_1 in Section 4) for usage in private key selection or nonce generation on, say, smartphones. In principle this is possible, but it would require the assumption that it is possible to compute hashes or to make recordings, pictures, or videos that are guaranteed not to be shared with other parties. One does not have to be overly paranoid to suspect that such a guarantee cannot be given, in particular if smartphones are involved.

Acknowledgement. The authors thank Rob Granger for pointing out [3] and for many useful discussions and comments on draft versions of this paper, and Ian Goldberg for his insightful feedback.

References

1. A. Albertini, J.-P. Aumasson, M. Eichlseder, F. Mendel, and M. Schläffer. Malicious hashing: Eve's variant of SHA-1. In A. Joux and A. Youssef, editors, *Selected Areas in Cryptography – SAC 2014*, volume 8781 of *Lecture Notes in Computer Science*, pages 1–19. Springer International Publishing, 2014.
2. S. Baktir and E. Savas. Highly-parallel Montgomery multiplication for multi-core general-purpose microprocessors. In E. Gelenbe and R. Lent, editors, *Computer and Information Sciences III*, pages 467–476. Springer London, 2013.
3. 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`.
4. D. J. Bernstein and T. Lange. Safecurves: choosing safe curves for elliptic-curve cryptography. `http://safecurves.cr.yp.to`, accessed 2 September 2014.
5. Businesspundit. Biggest lottery scandals. `http://www.businesspundit.com/5-of-the-biggest-lottery-scandals`, 2012.
6. D. Chaum. Random-sample elections: Far lower cost, better quality and more democratic, 2012.
7. D. Chaum, R. Carback, J. Clark, A. Essex, S. Popoveniuc, R. L. Rivest, P. Y. A. Ryan, E. Shen, and A. T. Sherman. Scantegrity ii: End-to-end verifiability for optical scan election systems using invisible ink confirmation codes. In *USENIX/ACCURATE Electronic Voting Technology Workshop (EVT)*, 2008.
8. I.-T. Chen. Random numbers generated from audio and video sources. *Mathematical problems in engineering*, 2013:7, 2013.
9. CPU-Z OC world records. `http://valid.canardpc.com/records.php`, 2015.
10. D. Eastlake 3rd. Publicly verifiable nominations committee (NomCom) random selection, 6 2004. RFC 3797.
11. A. Essex, J. Clark, R. T. Carback, and S. Popoveniuc. Punchscan in practice: an e2e election case study. In *IAVoSS Workshop on Trustworthy Elections (WOTE)*, 2007.
12. J. Garry. Randomocracy in Northern Ireland. `http://sluggerotoole.com/2015/03/21/randomocracy-in-northern-ireland/`, 2015.
13. P. Giorgi, L. Imbert, and T. Izard. Parallel modular multiplication on multi-core processors. In *Computer Arithmetic (ARITH), 2013 21st IEEE Symposium on*, pages 135–142, April 2013.
14. M. Huang, K. Gaj, and T. El-Ghazawi. New hardware architectures for montgomery modular multiplication algorithm. *Computers, IEEE Transactions on*, 60(7):923–936, July 2011.
15. M. E. Kaihara and N. Takagi. A hardware algorithm for modular multiplication/division. *Computers, IEEE Transactions on*, 54(1):12–21, 2005.
16. G. Marsaglia. The Marsaglia random number cdrom including the diehard battery of tests of randomness, 1995. `http://www.stat.fsu.edu/pub/diehard/`.
17. NIST randomness beacon. `https://beacon.nist.gov`, 2011.
18. L. C. Noll, R. G. Mende, and S. Sisodiya. Method for seeding a pseudo-random number generator with a cryptographic hash of a digitization of a chaotic system, March 1998. U.S. patent number 5,732,138 A.
19. S. Ors, L. Batina, B. Preneel, and J. Vandewalle. Hardware implementation of a montgomery modular multiplier in a systolic array. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, April 2003.
20. V. Rijmen and B. Preneel. A family of trapdoor ciphers. In E. Biham, editor, *Fast Software Encryption*, volume 1267 of *Lecture Notes in Computer Science*, pages 139–148. Springer Berlin Heidelberg, 1997.
21. D. Simmons. US lottery security boss charged with fixing draw. `http://www.bbc.com/news/technology-32301117`, 2015.
22. J.-M. Tsai, I.-T. Chen, and T. Jengnan. Random numbers generated from white noise of webcam. In *International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, pages 214–217. IEEE, 2009.
23. Wikipedia. Pennsylvania lottery scandal. `http://en.wikipedia.org/wiki/1980_Pennsylvania_Lottery_scandal`, 1980.

A Incorruptibility of *unicorn* in the random oracle model

In this appendix, the incorruptibility of *unicorn* is discussed, and analysed in the case where the permutation σ used in *sloth* is a random permutation of $\mathbf{F}_p^\times$. The proof relies on the assumption that square root extraction cannot be done faster than an exponentiation in $\mathbf{F}_p$. Let $S_p \subset \mathbf{F}_p^\times$ denote the set of non-zero squares in the field $\mathbf{F}_p$.

Definition 1. *Given the length $\varDelta > 0$ of a time interval, a probability ϵ, and a bound c on computational resources, the $(\varDelta, \epsilon, c)$-$\mathtt{sqrt}(\mathbf{F}_p)$ assumption is the following: any party with resources bounded by c that gets a uniformly chosen $\alpha \in S_p$ at time t, succeeds with probability at most ϵ to compute $\sqrt{\alpha}$ at or before time $t + \varDelta$.*

Remark 1. The bound c deals with the computational speed, parallelism, and the amount of available precomputed data. From now on, it will be omitted, considering that it corresponds to any set of state-of-the-art computing devices worth a polynomial amount of money. All parties are supposed to be bounded by c, and the assumption in Definition 1 will be referred to as the $(\varDelta, \epsilon)$-$\mathtt{sqrt}(\mathbf{F}_p)$ assumption. Based on the best of current knowledge about square root extraction in finite fields, the assumption seems reasonable for $\varDelta = \varOmega(\log(p)M(p))$, where $M(p)$ is the time complexity of multiplication in $\mathbf{F}_p$, and $\epsilon = \widetilde{O}(p^{-1/2})$ (the latter because square root extraction can be very fast for $\alpha = \beta^2$ with $\hat{\beta}$ bounded by $p^{1/2}$ times a factor polynomial in $\log(p)$). It should be stressed that this is just an assumption: $(p-2)$-nd-powering modulo a prime p, for instance, can be done much faster than in time $\varOmega(\log(p)M(p))$.

Remark 2. Under the $(\varDelta, \epsilon)$-$\mathtt{sqrt}(\mathbf{F}_p)$ assumption, if there is some λ such that α follows a distribution $\mu : S_p \to [0, \lambda/|S_p|]$ as opposed to the uniform distribution on S_p, the success probability is at most $\lambda\epsilon$. Indeed,

$$\Pr(\text{success}) = \sum_{\beta \in S_p} \mu(\beta) \Pr[\text{success}|\alpha = \beta] \le \frac{\lambda}{|S_p|} \sum_{\beta \in S_p} \Pr[\text{success}|\alpha = \beta] \le \lambda\epsilon.$$

The incorruptibility of *unicorn* is measured as a bound on the probability of winning the *unicorn corruption game*, a game consisting in compelling *unicorn* to produce an output (the presumably random number) with a targeted property. This targeted property is encoded by a map $b : \mathcal{H}^{k/2} \to \{0,1\}$. If g denotes the outcome of *unicorn*, the goal is to have $b(g) = 1$. In *unicorn*, the input of *sloth* is the concatenation of the contributions of any number of individual participants. The only part of the input that a particular participant can trust is its own contribution, which can be chosen by the participant to be uniformly distributed in $\mathcal{H}^{k/2}$ (which easily fits into the at most 140 characters allowed in a tweet), and could be extracted from the concatenation via a map $f : \mathcal{A}^* \to \mathcal{H}^{k/2}$. The concatenated data s is committed to at time t_0 when fed into *sloth*. As long as the commitment is not broken (which would imply finding a collision in h), the *unicorn*-generated random number at time t_1 will be the hash value resulting from *sloth* applied to input s. Then the incorruptibility of *unicorn* reduces to the difficulty of finding suitable data $s \in \mathcal{A}^*$ to commit to at time t_0 when the last honest contribution x is published at time $t_0 - \delta$. Therefore the game goes as follows: at the start of the game a time limit δ is given along with a uniformly random input $x \in \mathcal{H}^{k/2}$. The game is won if within time δ a value $y \in \mathcal{A}^*$ is produced such that $f(s) = x$, and $b(g) = 1$, where g is the hash value resulting from *sloth* applied to input s. It is worth emphasizing that this security model not only deals with dishonest contributors, but also with any attempt of the party centralising the process (generating the value s_1, committing to the data and running *sloth*) to bias the outcome.

Proposition 1 (*Unicorn* incorruptibility in the random oracle model). *Let Δ and ϵ be such that the (Δ, ϵ)-$\mathtt{sqrt}(\mathbf{F}_p)$ assumption holds. If $\sigma : \mathbf{F}_p^\times \to \mathbf{F}_p^\times$ is a random permutation, $h : \mathcal{A}^* \to \mathcal{H}^{k/2}$ from Section 1 is a random function, and the number of queries to the σ- and h-oracles is limited to $q > 0$ (including precomputations), then the probability* $\Pr_{\mathrm{w}}$ *to win the* unicorn corruption game *with time limit $\delta < \ell\Delta$ for a target property $b : \mathcal{H}^{k/2} \to \{0, 1\}$ is at most $q/2^{2k-1} + \epsilon\ell q(p-1)/(p-1-q) + q/(p-1-q) + |b^{-1}(\{1\})|/2^{2k}$.*

Remark 3. In other words, by contributing to *unicorn* within the time interval $(t_0-\ell\Delta+\gamma, t_0]$, one can make sure the output is not corrupted (unless a collision has been found in h). The first three terms in the bound on the probability are all negligible when q is polynomial in the security level k, and $\epsilon = \widetilde{O}(p^{-1/2})$. The last term is the probability of a uniformly chosen $g \in \mathcal{H}^{k/2}$ to satisfy $b(g) = 1$.

Remark 4. The slowness of computing *sloth* can easily be deduced from this result. In order to find a valid output for sloth on a random input, there is no faster strategy than computing ℓ square roots sequentially.

Proof. Let $\mathcal{Q}_t$ (respectively, $\mathcal{R}_t$) be the set of queries to the σ-oracle (respectively, h-oracle) done before point in time t since the start of the unicorn corruption game (with $\mathcal{Q}_0$ and $\mathcal{R}_0$ the precomputed queries), with $\alpha \in \mathcal{Q}_t$ denoting that the value of $\sigma(\alpha)$ was queried before time t. Suppose that on input $x \in \mathcal{H}^{k/2}$, a string $s \in \mathcal{A}^*$ is output. Let $w \in \mathbf{F}_p^\times$ be the corresponding *sloth* witness, and $g = h(\mathtt{hex}(\hat{w}))$ be the hash. Winning the game implies that s is output at or before time δ, that $b(g) = 1$, and that $f(s) = x$. Then,

$$\Pr_{\mathrm{w}} \leq \Pr[b(g) = 1 | f(s) = x].$$

By abuse of notation, all the following probabilities will be conditional in $f(s) = x$. Applying the law of total probability,

$$\Pr_{\mathrm{w}} \leq \Pr[b(g) = 1] \leq \Pr[\mathtt{hex}(\hat{w}) \in \mathcal{R}_\delta] + \Pr[b(g) = 1 | \mathtt{hex}(\hat{w}) \notin \mathcal{R}_\delta].$$

If $\mathtt{hex}(\hat{w}) \notin \mathcal{R}_\delta$, then $g = h(\mathtt{hex}(\hat{w}))$ is uniformly distributed in $\mathcal{H}^{k/2}$ so the second term is $|b^{-1}(\{1\})|/2^{2k}$. For the first term, the law of total probability yields

$$\Pr[\mathtt{hex}(\hat{w}) \in \mathcal{R}_\delta] \leq \Pr[w \in \mathcal{Q}_\delta] + \Pr[\mathtt{hex}(\hat{w}) \in \mathcal{R}_\delta | w \notin \mathcal{Q}_\delta].$$

In the second term, as $w \notin \mathcal{Q}_\delta$, the witness w is uniformly distributed among the elements of $\mathbf{F}_p^\times$ not previously chosen by the σ-oracle, so $\Pr[\mathtt{hex}(\hat{w}) \in \mathcal{R}_\delta | w \notin \mathcal{Q}_\delta] \leq q/(p-1-q)$. With w_i as in the definition of *sloth*, the first term is split into

$$\Pr[w \in \mathcal{Q}_\delta] \leq \Pr[w_{\ell-1} \in \mathcal{Q}_{\delta-\Delta}] + \Pr[w \in \mathcal{Q}_\delta | w_{\ell-1} \notin \mathcal{Q}_{\delta-\Delta}].$$

Considering the second term, it follows from $w_{\ell-1} \notin \mathcal{Q}_{\delta-\Delta}$ that there is time at most Δ to compute $w_\ell = \tau(w_{\ell-1}) = \rho(\sigma(w_{\ell-1}))$, where $\sigma(w_{\ell-1})$ is uniformly chosen among the elements of $\mathbf{F}_p^\times$ not previously chosen by the σ-oracle. The (Δ, ϵ)-$\mathtt{sqrt}(\mathbf{F}_p)$ assumption combined with Remark 2 then implies the bound $\Pr[w \in \mathcal{Q}_\delta | w_{\ell-1} \notin \mathcal{Q}_{\delta-\Delta}] \leq \epsilon q(p-1)/(p-1-q)$.

The term $\Pr[w_{\ell-1} \in \mathcal{Q}_{\delta-\Delta}]$ is handled by induction, computing $\Pr[w_{\ell-j} \in \mathcal{Q}_{\delta-j\Delta}]$ for each j from 1 to $\ell-1$. The same reasoning as in the previous paragraph leads to

$$\Pr[w_{\ell-j} \in \mathcal{Q}_{\delta-j\Delta}] \leq \Pr[w_{\ell-(j+1)} \in \mathcal{Q}_{\delta-(j+1)\Delta}] + \Pr[w_{\ell-j} \in \mathcal{Q}_{\delta-j\Delta} | w_{\ell-(j+1)} \notin \mathcal{Q}_{\delta-(j+1)\Delta}],$$

and

$$\Pr[w_{\ell-j} \in \mathcal{Q}_{\delta-j\Delta} | w_{\ell-(j+1)} \notin \mathcal{Q}_{\delta-(j+1)\Delta}] \leq \epsilon q_j \frac{p-1}{p-1-q_j} \leq \epsilon q \frac{p-1}{p-1-q},$$

where q_j is the number of oracle queries done before time $\delta - j\Delta$. Inductively,

$$\Pr[w \in \mathcal{Q}_\delta] \leq \Pr[w_0 \in \mathcal{Q}_{\delta-\ell\Delta}] + \epsilon \ell q \frac{p-1}{p-1-q}.$$

It remains to bound $\Pr[w_0 \in \mathcal{Q}_{\delta-\ell\Delta}]$. Since $\delta < \ell\Delta$, the queries of $\mathcal{Q}_{\delta-\ell\Delta}$ were done during the precomputation phase, before the input x was revealed. Then,

$$\Pr[w_0 \in \mathcal{Q}_{\delta-\ell\Delta}] \leq \Pr[w_0 \in \mathcal{Q}_0] \leq \Pr[s \in \mathcal{R}_0] + \Pr[w_0 \in \mathcal{Q}_0 | s \notin \mathcal{R}_0].$$

If $s \notin \mathcal{R}_0$ at the start of the game, $h(s)$ is uniformly distributed over $\mathcal{H}^{k/2}$ because h is assumed to be a random function, so w_0 is uniformly distributed among the 2^{2k} corresponding elements of the field (recall that $2^{2k} < p$). Hence $\Pr[w_0 \in \mathcal{Q}_0 | s \notin \mathcal{R}_0] \leq q/2^{2k}$. It is only for the remaining quantity $\Pr[s \in \mathcal{R}_0]$ that the distribution of the input x comes into play. Recall that all the probabilities where conditional in $f(s) = x$ (if it is false, the game is lost), and that x is uniformly distributed over $\mathcal{H}^{k/2}$. Therefore,

$$\Pr[s \in \mathcal{R}_0] \leq \Pr[f(s) \in f(\mathcal{R}_0)] \leq q/2^{2k},$$

which concludes the proof.