Trap Me If You Can

Million Dollar Curve

Thomas Baignères[1], Cécile Delerablée[1], Matthieu Finiasz[1], Louis Goubin[2], Tancrède Lepoint[1], and Matthieu Rivain[1⋆]

[1] CryptoExperts, Paris
[2] Laboratoire de Mathématiques de Versailles, UVSQ, CNRS, Université Paris-Saclay, 78035 Versailles, France
curves@cryptoexperts.com

March 1, 2016

Abstract. A longstanding problem in cryptography is the generation of publicly verifiable randomness. In particular, public verifiability allows to generate parameters for a cryptosystem in a way people can legitimately trust. There are many examples of standards using arbitrary constants which are now challenged and criticized for this reason, some of which even being suspected of containing a trap. Several sources of public entropy have already been proposed such as lotteries, stock market prices, the bitcoin blockchain, board games, or even Twitter and live webcams.

In this article, we propose a way of combining lotteries from several different countries which would require an adversary to manipulate several independent draws in order to introduce a trap in the generated cryptosystem. Each and every time a new source of public entropy is suggested, it receives its share of criticism for being "easy to manipulate". We do not expect our solution to be an exception on this aspect, and will gladly receive any suggestion allowing to increase the confidence in the cryptosystem parameters we generate.

Our method allows to build what we call a Publicly verifiable RNG, from which we extract a seed that is used to instantiate and initialize a Blum-Blum-Shub random generator. We then use the binary stream produced by this generator as an input to a filtering function which deterministically outputs secure and uniformly distributed parameters from uniform bitstreams.

We apply our methodology to the ECDH cryptosystem, and propose the *Million Dollar Curve* as an alternative to curves P-256 and Curve25519.

Keywords: Publicly verifiable RNG, lottery, trusted cryptosystem parameters, elliptic curve, Million Dollar Curve, decentralized beacon, NSA, Snowden.

⋆ With a Little Help from our Friends [3]

1 On the Need for Convincing Randomness Generation in Cryptography

Designing a secure cryptographic algorithm is a complex process. An even more complex process is to design a secure cryptographic algorithm that people legitimately trust as such. One reason stems from the fact that, when designing a cryptosystem, a cryptographer repeatedly has to make choices. Most of the time, a majority of these choices can be ruled out immediately for security reasons. But, in the end, one is often left with several "acceptable" options. In that situation, the cryptographer can essentially

- arbitrarily choose one of the options (e.g., the "best looking" one), or
- pick an option at random using a process that can be verified *a posteriori*.

Is arbitrariness acceptable? There is a long history of cryptographic algorithms and standards using arbitrary constants, sometimes referred to as "nothing up my sleeve numbers" [42] (NUMS). For example, all the hash functions belonging to the SHA family use, at some point, round constants. To the best of our knowledge, those constants never raised any serious concern among the cryptographic community, probably because introducing a non-obvious weakness (i.e., a trap) in a symmetric cryptography primitive by manipulating only a few constants appears to be extremely hard, if not unfeasible, as long as the primitive builds upon simple and sound security arguments.

Many asymmetric primitives also fall short when it comes to finding a justification for the choice of some of their inner constants. Unlike symmetric primitives, this fact has regularly been proven to be a serious threat for the security of the concerned primitive. Recent and infamous examples include Dual-EC-DRBG, one of the pseudo-random number generators defined in SP 800-90 [29].

The case of Dual-EC-DRBG. Like most PRNGs, Dual-EC-DRBG maintains an internal state s. This 256-bit state is updated by computing $s = x(sP)$, where P is a fixed point on the elliptic curve P-256 [43, Appendix J.5.3] specified in [29, Appendix A.1.1], and where $x(\cdot)$ maps a point to its first coordinate. From each state, one can derive up to 30 bytes of randomness, by computing $r = x(sQ)$ (discarding the 16 most significant bits of r), where Q is another fixed point on the curve, also specified in [29, Appendix A.1.1]. Apart from its poor performance and statistical properties [22,37], Shumow and Ferguson showed in [38] how simple it is for the designer to introduce a trap in Dual-EC-DRBG: since the order of P-256 is prime, Q is a generator, and there must exist some ℓ such that $P = \ell Q$. Computing ℓ from the sole knowledge of P and Q is hard, but the designer could have first generated Q and ℓ, and then computed $P = \ell Q$. Assume this is the case and consider the situation where Alice runs Dual-EC-DRBG and generates a public r. From r, anyone can recompute the y coordinate of sQ. From the knowledge of ℓ, the designer can also compute $\ell sQ = s\ell Q = sP$, from which he can deduce the next internal state of Alice's

Dual-EC-DRBG instance, and thus predict all future outputs.[3] One can note that the ℓ mentioned above *always* exists, the real question being whether the designer did, or did not draw the parameters in such a way that this value is actually *known* to him, both situations being indistinguishable for an outsider. As shown by Shumow and Ferguson [38] an obvious fix would be to generate a random point Q for each instance of the PRNG.

Unfortunately, the Dual-EC-DRBG case is not as specific as it might look. For example, Bernstein et al. show in [7] how easy it is for a malicious designer to manipulate his asymmetric design, by simply choosing a few constants without justification.

Rule of thumb. Of course, one should not conclude that cryptographic algorithms using similar constants are systematically insecure (certainly, some designers are honest) and we will not dispute the right to trust those algorithms. Yet, in the perspective of eventually obtaining a *legitimately trusted* cryptographic algorithm, we believe that one should rule out any cryptographic design involving arbitrary choices.

Randomness to the rescue. Many cryptographic standards have chosen to avoid unjustified choices. For example, X9.62-1998 [43, p.31] argues that "*In order to verify that a given elliptic curve was indeed generated at random, the defining parameters of the elliptic curve are defined to be outputs of the hash function SHA-1 [...]. The input (SEED) to SHA-1 then serves as proof (under the assumption that SHA-1 cannot be inverted) that the parameters were indeed generated at random*". The *proof* for the P-256 curve is given later in the document [43, p.117]:

```
SEED = C49D3608 86E70493 6A6678E1 139D26B7 819F7E90
```

Yet, the X9.62 standard does not explain how this particular seed was chosen. Concerning this issue, Schneier writes: "*I no longer trust the constants. I believe the NSA has manipulated them through their relationships with industry.*" [36] Bernstein et al. show in [7] that allowing unjustified seeds makes it possible for the designer to manipulate the standard almost as easily as in the previous case.[4]

Hopefully, not all standards follow the X9.62 example. For example, the Brainpool standard [1] complains that, although many standard proposals exist for elliptic curve cryptography, "*The choice of the seeds from which the curve parameters have been derived is not motivated leaving an essential part of the security analysis open*", further adding that "*The curves shall be generated in a pseudo-random manner using seeds that are generated in a systematic and comprehensive way*". Thereafter, the standard lists seven seeds, based on the decimals of the Euler number e. Those seeds are used to feed an algorithm, similar to the one specified in X9.62, which generates seven curves of various security

[3] Note that, for the sake of concision, we skipped a few technical details (see [10] for the full story of Dual-EC-DRBG).

[4] Also, we would like to emphasize that "undermining of cryptographic solutions and standards" is repudiated by the IACR (cf. the IACR Copenhagen Resolution).

levels. Yet, [7] criticizes this approach as well, arguing that the designer can still manipulate the choice of the seed (why e, and not π, $\sin(1)$, $\sqrt{2}$ or $(1+\sqrt{5})/2$?), the choice of the hash function in the algorithm turning the seed into concrete parameters (why SHA-1, and not SHA-xxx?), the way one extracts bits from the seed, etc. Similarly, Aumasson illustrated in [2] that the degrees of freedom in the process of going from NUMS choices to actual parameters (namely the encoding, the number itself, the hash functions, the decoding, etc.) gives a lot of leeway in how to select weak parameters.

Another pitfall suffered by the Brainpool standard has been recently discovered by the BADA55 Research Team. In their own words, "*none of the standard Brainpool curves below 512 bits were generated by the standard Brainpool curve-generation procedure*" [40]. It appears that, until this finding, no-one actively verified all of the Brainpool curve generation on the basis of the generation procedure specified in [1, Section 5]. This example shows how important it is for the honest designer to provide third-parties with simple means of checking the parameters generation. Although this can take many forms, the Brainpool example argues in favor of executable and readable *code* instead of an algorithmic description on paper.

One should not conclude that elliptic curve parameters should not be chosen at random, but rather that doing so is more complex than one might first think.

A way to generate a curve in a fully *transparent* way, so that it can be legitimely trusted, has recently been proposed [20], but no explicit proposal for an elliptic curve is made. This paper also provides a nice and complete list of security criteria for choosing elliptic curve parameters. The format of a certificate is then proposed to ease the computations of verifying elliptic curves parameters.

1.1 Our Contributions

In this work, we propose a hopefully convincing (and amusing) solution to the problem of generating publicly verifiable randomness in an unimpeachable fashion.

Our first contribution is to introduce the notion of Publicly verifiable RNG, that is a source of entropy that is not only unpredictable, unbiased, publicly observable but also archived (and therefore easily verifiable in the future). We then show how to leverage these strong security properties to (almost) fully automate the process of instantiating a trusted cryptosystem. In particular, we revisit the different steps of a cryptosystem instantiation by splitting it into: (1) a timestamped design that includes the full specification of the system, regardless of the seed it will use, (2) a timestamped standard that commits on the design, on a seed extractor that uses a well-specified Publicly verifiable RNG, and a time period in the future to instantiate the seed extractor, (3) the cryptosystem concrete instantiation using the aforementioned design and standardized procedure, and finally (4) the key generation in the cryptosystem by the final user. Distinguishing these steps allows to easily associate to each step the required competence; e.g. the design specification (1) is the only one requiring the intervention of a cryptographer, while *anyone* can instantiate the cryptosystem at Step (3).

Next, we describe how to build a Publicly verifiable RNG based on *several* national lottery results. National lotteries are randomness sources for which it is possible to measure the exact amount of entropy contained in each observation, and great care is usually taken on the randomness quality, which make them ideal entropy sources for a Publicly verifiable RNG. Then, we explain how to construct a seed extractor based thereon.

Last but not least, we provide a full-fledged example by instantiating *in the future*, using future national lottery results, an elliptic-curve suited for ECDH key exchanges named the *Million dollar curve* [18].

1.2 Related Work

A notion similar to our Publicly verifiable RNG was recently proposed by Bonneau, Clark and Goldfeder in [14], as a "decentralized beacon with no trusted parties". The notion of beacon in cryptography was first introduced by Rabin [32] in the context of contract signing: a trusted third party regularly emits randomly chosen integers. As of today, several proprietary randomness beacons are available on the Internet, such as the NIST Randomness Beacon or Random.org, but these beacons cannot be used as cryptographic sources of randomness.[5] To avoid trusted third parties, Bonneau et al. proposed a decentralized beacon based on bitcoin's blockchain [14], and provide precise security guarantees (namely that the beacon is manipulation-resistant against an attacker with a stake of less than ₿50 in the output). Our notion of Publicly verifiable RNG can be seen as a decentralized beacon with no trusted parties, which emits every time period an array of random values (possibly empty), and based on an archived source of randomness.[6] In this work we propose a construction of Publicly verifiable RNG based on national lotteries, and [14] could be trivially adapted to become a blockchain-based Publicly verifiable RNG, which is an interesting second possible Publicly verifiable RNG. Other attempts to produce randomness from publicly observable events have also been proposed in the literature, e.g. a beacon based on the stock market [15], or locally-verifiable randomness generation from cards, coins, dices, Boggle boards, sun stains or drawings [35,16,23,20]. However, it remained unclear (until [14]) how to construct a fully decentralized publicly verifiable beacon with no trusted third party.

Another interesting document is the RFC 2777 of the IETF [24]. To select as randomly as possible the nominations committee so that "no reasonable charges of bias or favoritism can be brought", this document proposes a method that commits on a design, and on 3 lottery draws that will happen in the future.[7]

[5] In particular, the NIST strongly emphasizes "*WARNING: DO NOT USE BEACON GENERATED VALUES AS SECRET CRYPTOGRAPHIC KEYS.*" on the beacon website, and Random.org states that "The numbers generated by RANDOM.ORG are buffered".

[6] In other words, one could say that a Publicly verifiable RNG produces an *unpredictable* common random string [13].

[7] The number of lottery draws is small on purpose to avoid unanticipated situations in case of cancellation, etc.; in our case we will rather take a large enough margin

From these draws, a hash will be derived (MD5 is suggested—which was already audacious in 2000), to obtain 128 bits of entropy which can then subsequently be used to select the nomination committee.

Finally, generating cryptographic parameters using "incontestable generation of random numbers" has already been suggested in the literature [23,28,20]. In [23], Mike Hamburg suggests to use games (such as Boggle boards or Bananagrams tiles) played during social events of cryptography conferences. Photographs and videos of the outputs together with handwritten hashes of the parameters generation algorithms allow to later verify the curves were randomly generated with these algorithms and random values. This interesting idea of using a source of randomness that can be influenced (manipulated?) by anyone participating in the initial stages of the seed generation is also at the core of Lenstra and Wesolowski's random number generator [28] (with tweets, pictures of a public place and a webcam that films the camera). In the latter paper, to deal with powerful adversaries who could try to manipulate the seed, they also use a slow hash function. However, the aforementioned elliptic curve service generations are centralized, and are not easily reproducible by everyone. In particular, after the period of time during which tweets are collected, the methodology requires the central authority to take a picture and to immediately share the hash of the tweets and the picture. The picture increases the final quantity of entropy and helps protecting against malicious tweets, the tweets protect against a malicious central authority (which could otherwise choose/alter a picture leading to a potentially unsafe curve). While this solution looks like it could be solving the problem, in practice people hardly ever participate with a tweet[8], leading to most curves being 100% centrally generated. In our solution, or in a solution based on Bitcoin's blockchain (as in [14]), the entropy source is fundamentally decentralized, leading to a solution that is easier to set up in a confidence-inspiring way.

1.3 Outline

First, motivated by a trustworthy generation of cryptosystem, we introduce the notion of Publicly verifiable RNG and revisit the process of instantiating a cryptosystem in Section 2. Then, we describe a Publicly verifiable RNG based on several national lottery draws in Section 3, and explain how to produce a random seed containing enough entropy from such a Publicly verifiable RNG in Section 4. Finally, we describe how to produce parameters for the Blum-Blum-Shub random number generator to spread the seed entropy. Each section will include a

to avoid being affected by such events. Also, it is emphasized that the last source (chronologically) should be especially strong and unbiased source of a large amount of the randomness; as we will extensively discuss in Section 4.4 and 4.5, we will completely departs from this paradigm in order to be secure against "*real* adversaries" with "enormous budget" and very strong adversarial motivation [34].

[8] One can check that none of the 128-bit security curves of January 2016 were influenced by any tweet.

concrete toy example, that will eventually produce to the *Two Cents Curve* in Section 6.2.

Last, but not least, we introduce the *Million Dollar Curve*, an Edwards curve to be generated in February 2016, using worldwide lottery draws, and we challenge any *real* adversary to trap it. Good luck!

2 Generating Legitimately Trusted Cryptosystems

2.1 State of the Art

Usually, everything starts with the publication of an article proposing a new cryptographic construction. Sometimes, the designers also propose some "secure" parameter sizes, based on their own analysis of the design. But then a lot of nice things turn bad out there [39], specific instances of this cryptosystem using these parameters generally get attacked, and parameter requirements are updated. After some time, things settle down, and the cryptographic community agrees on a set of criteria that *secure* parameters should verify. Eventually, a specific secure set of parameters gets standardized (or becomes a *de facto* standard) and people start using the resulting cryptosystem, each user generating his or her own key.

Generally, there are many parameter choices that will meet all the security criteria, so a choice will eventually have to be made. Keeping in mind that arbitrary (non verifiable) choices should systematically be avoided, there are only two options left:

- Define additional criteria so as to reduce the set of acceptable parameters to (essentially) one candidate. However, these criteria should not be arbitrary, and should follow sound and justifiable arguments, typically implementation aspects (see the example below).
- Draw one candidate at random, in a convincing way, avoiding the pitfalls of Section 1 to begin with.

Example. For the elliptic curve version of the Diffie-Hellman key exchange protocol, choosing parameters consists in specifying a curve and a group generator. Although there are many security criteria (underlying field size, group order, etc.[9]), the set of acceptable curves is huge. It is commonly agreed that Edwards curves [19] are well suited for cryptographic applications [9], so it appears natural to add this criterion. The equation of an Edwards curve over a prime field $\mathbb{F}_p$ of odd characteristic p is

$$x^2 + y^2 = 1 + dx^2y^2$$

where $d \in \mathbb{F}_p \setminus \{0, 1\}$. Remain to be chosen, the prime p and the scalar d. Curve25519 [4,5] is one example of *justified* parameter choices where $p = 2^{255} - 19$

[9] See [8,20] for a complete list of security criteria.

and $d = 121665/121666$ [9].[10] This curve verifies all the security criteria, and was specifically selected to offer extremely high performance on a prime field.

It is a well accepted principle that putting all your eggs in one basket is a bad idea, and cryptography is no exception. We therefore need convincing alternatives to Curve25519. Obviously, trying to verify the same criteria than Curve25519 will require to beat Curve25519 at its own game, which will certainly imply a considerable amount of work. Instead of focusing on raw performance on a prime field[11], another selection criteria could be used (such as code size, memory footprint, hyperelliptic curves, etc.), but this will allow to select, at best, a few alternatives. For all those situations where extreme performance or tiny code size are not mandatory, we believe there are easier ways to convince people. For this, we propose to automate the instantiation of the cryptosystem. As we will see in Section 2.3, we will require a particular way of generating randomness, through what we call a Publicly verifiable RNG.

2.2 The Need for a Publicly Verifiable RNG

A typical way of generating randomness in cryptography is to combine a hardware true random number generator (TRNG) with a cryptographically secure pseudo-random number generator (PRNG). When properly implemented, this design allows to output sequences of unbiased and independent bits at a very high output rate. However, the entropy of a TRNG generally comes from a small hardware module (oscillators, resistors, etc.) which make it impossible to prove that a specific seed was indeed generated by this TRNG. Even if the seed came with such a proof, it would be impossible to guarantee that this specific output was not cautiously selected among a huge set of outputs from this TRNG. As a consequence, using such a TRNG to generate the parameters of a cryptosystem would most probably lead to criticism similar to that raised in [7] about the Brainpool seeds.

The previous discussion leads us to the conclusion that we require a Publicly verifiable RNG, that is, a source of entropy which is

- very hard to manipulate/predict,
- publicly observable, i.e., such that *all* outcomes are public, and
- archived, so that anyone can pick a past date and get the corresponding outcome.

There are many such sources, such as stock market data[12], sports scores[13], weather data around the world, or national lottery results. All those examples

[10] Although Curve25519 was initially given in Montgomery form, it can be shown to be birationally equivalent over $\mathbb{F}_p$ to the Edwards form given here, see [9].

[11] For example, by departing from a prime field setting, curves such as the Four$\mathbb{Q}$ curve [17] or some genus-2 hyperelliptic curves [6] have shown to offer extremely high-performances.

[12] Under the "Perfect market" assumption.

[13] Excluding soccer results for safety.

produce randomness at predictable moments in time. Yet, among those examples, national lotteries are the only sources for which it is possible to measure the exact amount of entropy contained in each observation. For this reason (and some others), we will use them in Section 3 to implement a concrete Publicly verifiable RNG.

2.3 Automating the Instantiation of a Cryptosystem

Following the discussion of the two previous sections, we investigate how to (almost) fully automate the process of instantiating a cryptosystem by leveraging on the good properties offered by a Publicly verifiable RNG. Fig. 1 presents our vision of how the different steps of the instantiation of a cryptosystem should be organized, from its early design to its concrete instantiation/personalization for a user.

- The Designer is responsible for producing (at some time t_0) the complete Design, composed of
  - the full specification of the cryptographic algorithms composing the Cryptosystem, where there might still exist parameters left to be defined;
  - a list of Security criteria such that any instantiation (i.e., parameter choice) of the Cryptosystem meeting these criteria is secure with respect to the current state of cryptographic research;
  - the Parameter space, which is nothing more than the set of all parameters matching all the Security criteria;
  - a deterministic filtering function f, mapping any seed of appropriate length onto specific values of the Parameter space. This function should not introduce any distinguishable bias, i.e., a uniformly distributed random seed should produce parameters that are indistinguishable from uniformly distributed parameters in the Parameter space. In practice, this function has to be easy to compute for anyone, typically by publishing clear (and correct) source code for this function.
- The Design is then timestamped at time t_0. The resulting timestamp should allow Anybody, at any time $t \geq t_0$, to make sure that the Design did not suffer any modification since time t_0.
- The Standardizer chooses (at time $t_1 \geq t_0$) and commits (at time $t_2 \geq t_1$) on
  - a specific timestamped Design,
  - a specific Publicly verifiable RNG, and
  - a Seed Extractor, that is, a clear description of the way the entropy shall be extracted from the Publicly verifiable RNG and two times t_3 and t'_3. The time t_3 is the precise time at which the Seed Extractor starts to extract entropy and must be such that $t_3 > t_2$. The time t'_3 is a time after which enough entropy will have been collected by the Seed Extractor to generate a seed of appropriate length. Depending on the nature of the Publicly verifiable RNG, one might or might not be able to exactly predict the amount of entropy produced over a precise period of time, so

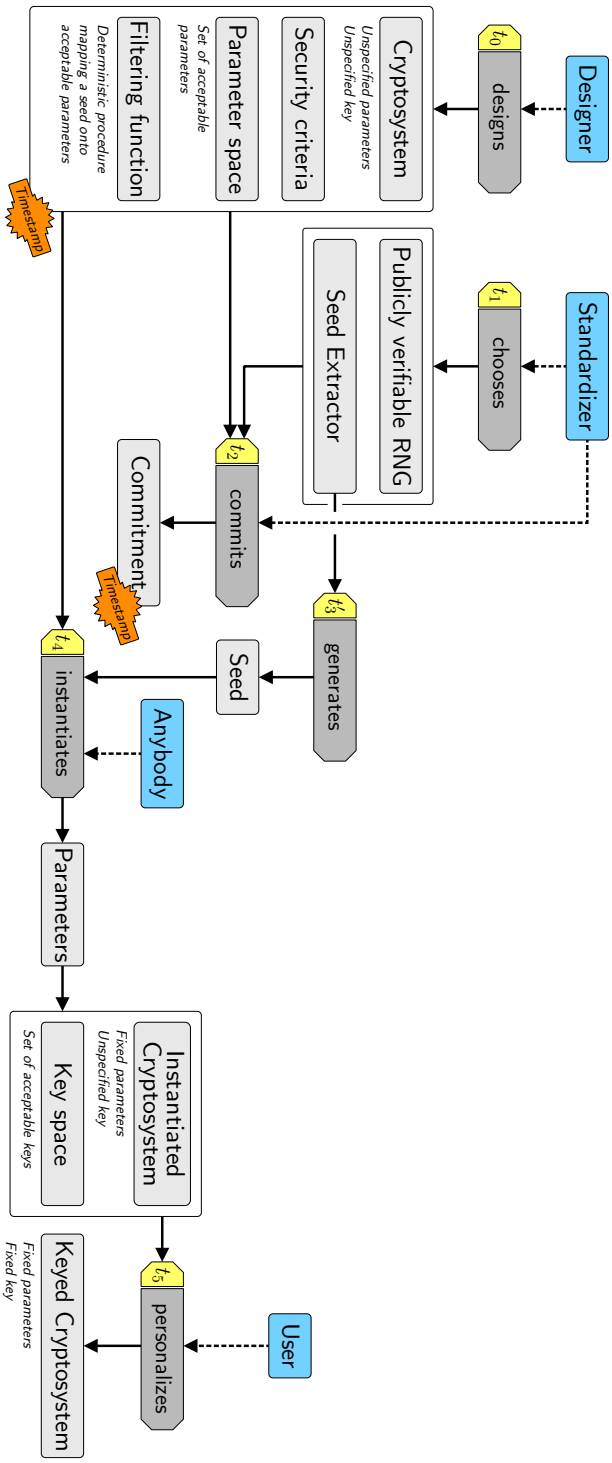

Fig. 1. How to properly randomly draw the parameters of a cryptosystem.

t'_3 should be picked with a safety margin, so that the probability of not gathering enough entropy between t_3 and t'_3 is negligible.

This Commitment is timestamped in such a way that Anybody, at any time $t \geq t_2$, can check that the Commitment was indeed produced at time $t_2 < t_3$.

- At any time $t_4 \geq t'_3$, Anybody can instantiate the timestamped Design into a unique Instantiated Cryptosystem and a Key space. Essentially, this step specifies the Parameters by computing $f(\mathsf{seed})$, where $f(\cdot)$ is the Filtering function specified in the Design and seed is the entropy extracted from the Publicly verifiable RNG between times t_3 and t'_3 with the Seed Extractor.
- At any time $t_5 \geq t_4$, any User can make use of the well defined Instantiated Cryptosystem (with fixed parameters, common to all users) and Key space. A User would typically uniformly draw a key in the Key space in order to generate his own Keyed Cryptosystem. This Personalization process is out of the scope of this article as we cannot control the way the User chooses his own key.

Leave (all) the hard work to the experts. Among the various steps we just suggested, producing the complete Design is the only one requiring expert knowledge (i.e., the intervention of a cryptographer). This is also why the Security criteria and the Filtering function are part of the Design: choosing them requires expert knowledge, and it is important that they can be reviewed by the cryptographic community, like the Cryptosystem itself. Of course, a strong expertise is also required to properly implement the final cryptosystem, but this step is outside the scope of this paper.

An automatable design. Selecting parameters should not require any hard work from the Standardizer. This is indeed the case in our process, since this selection only consists in choosing a Publicly verifiable RNG and a Seed Extractor with two times t_3 and t'_3. As we will see in Section 3, there are plenty of Publicly verifiable RNGs to choose from.

Designer vs. Standardizer. As we have seen in Section 1, it is generally a bad idea to let the Designer choose the parameters of his own Design. This situation occurred many times in the past, mainly because selecting secure parameters is a complex task. In our design, we dissociate the selection criteria from the selection itself. The Security criteria are complex and are thus left to the Designer, while the actual selection is simple and can be done by the Standardizer.

Using a real entropy source. Many of the concerns mentioned in Section 1 rise from the fact that both the seed and the Filtering function used for the parameters generation are chosen simultaneously. Because of this, there is no way to prove that the design of the Filtering function was not influenced by the outcome of the seed generation. The only "clean" way of choosing a Filtering function is to define it with no *a priori* knowledge of the seed. This also implies that the seed must contain enough entropy for the Standardizer to have no advantage at guessing

the outcome of the Filtering function (the actual parameters). In that situation, there is no way to introduce a trap in the parameters, even if the Standardizer colludes with the Designer. As a consequence, the Standardizer and the Designer can be the same person.

Protection against "0-day" attacks. Obviously, the Security criteria can only take *public* attacks into account. Yet, unpublished "0-day" attacks can exist (i.e., attacks that nobody else knows about), but chances are that these attacks only affect a small fraction of the Parameter space (see also [7]). In order to take a walk on the safe side [33], Anybody should make sure at instantiation that the Standardizer did not make too many commitments (ideally, a single one). We will discuss "0-day" attacks further in Section 4.4.

Nature of the timestamps. The *Timestamp* on the Design allows to freeze a Design in time, whereas the *Timestamp* on the Commitment allows to prove that this Design, the Publicly verifiable RNG, and the Seed Extractor and times t_3 and t'_3 were chosen before t_3. Because we assume that Publicly verifiable RNGs naturally generate, alongside the seed, a proof that this seed was generated at a specific moment in time, there is no need to add another *Timestamp*. The most natural way to timestamp the Design and the Commitment is, for example, to publish a hash of them in the day's newspapers.[14] This way, anyone may subsequently consult this "proof" at the public library archive and check that the *Timestamp*s are indeed older than t_3. In addition, this method makes it difficult/expensive for a malicious Standardizer to commit inconspicuously on many different Publicly verifiable RNGs, rendering "0-day" attacks less likely.

2.4 Parameters Validation Checklists

There are many things to validate before using parameters generated through our process. Some of these validations require expert knowledge and should be left to the cryptographic community, while some can be checked by just about anyone (and we encourage anyone to check them).

Members of the cryptographic community are in charge of validating any aspect of the Design, including:

- cryptanalysis of the Cryptosystem;
- ensuring that the Security criteria are exhaustive, i.e., sufficient to be safe against all known attacks against the Cryptosystem;
- verifying that the Parameter space matches the Security criteria;
- verifying that the Filtering function matches the Security criteria.

This validation should be an ongoing process, such that any advance on the cryptanalysis side results in an appropriate update of the Security criteria, Parameter space, and Filtering function. Note that any such update requires to

[14] In practice, online publishing on the IACR Cryptology ePrint Archive might be convincing enough.

timestamp the design again, but does not necessarily invalidate previously generated parameters if they are not subject to the new attack.

And now for something completely different [30], here is the complete parameters validation checklist for Anybody:

- ☐ Check the *Timestamp* on the Design and let t_0 be the *Timestamp*'s time.
- ☐ Check that the Standardizer did not commit on too many designs, Publicly verifiable RNGs, and times t_3 and t'_3 (only one such commitment per security level is required).
- ☐ Check the Commitment on the Design, the Publicly verifiable RNG, and times t_3 and t'_3.
- ☐ Check the *Timestamp* on the Commitment and let t_2 be the *Timestamp*'s time.
- ☐ Check that $t_0 < t_3$ and $t_2 < t_3$.
- ☐ Decide whether or not he wants to trust the selected Publicly verifiable RNG.
- ☐ Lookup the outcome of the Publicly verifiable RNG between times t_3 and t'_3 (available from its archive) and extracts the seed using the Seed Extractor.
- ☐ Apply the Filtering function to the seed, and check that the obtained Parameters match what was expected.

3 Lottery Winning Numbers: a Publicly Verifiable Entropy Source

There exist many publicly verifiable entropy sources (stock market data, sports scores, weather data around the world...), but most of them are very hard to analyze and contain strong correlations, making it difficult to estimate the *exact* amount of entropy effectively produced by these sources.

One noticeable exception are national lotteries, which already "solved" the problem of generating randomness in a trustworthy manner. Many of these lotteries even use mechanical drawing machines to perform the drawings in the presence of observers, sometimes with a live broadcast on national TV. Lotteries are also a very good source of entropy for the following reasons:

- the output is an unbiased selection of m numbers among n,
- the output rate is slow, but reliable (in many countries, there are 1 or 2 draws each week, since many years and for many years to come),
- the output is publicly verifiable at any time since most national lotteries maintain a public archive of past draws,
- the output is very hard to manipulate (otherwise we would have better things to do with our millions than writing this article).

One drawback of this entropy source is that it does not output bits, taking us cryptographers out of our comfort zone. The following sections will suggest a way to extract randomness from a single lottery draw, to define a Publicly verifiable RNG from a single-draw lottery (i.e., a lottery that performs at most one draw per day), and how to increase the entropy throughput by considering several lotteries at once.

3.1 Extracting Entropy from the Winning Numbers of a Lottery Draw

Consider a lottery draw that consists of m numbers in a set of n possible numbers. There are $\binom{n}{m}$ possible outcomes, which is $\log_2 \binom{n}{m}$ bits of entropy. But extracting exactly $t = \lfloor \log_2 \binom{n}{m} \rfloor$ independent unbiased bits from each draw is impossible. The best that can be done to generate t bits is the following:

- convert the draw to an integer x between 0 and $\binom{n}{m} - 1$,
- if $x < 2^t$, output the t bits of x, otherwise discard this draw.

This last step is required as it is the only way to have a uniformly distributed sequence of t bits, but it makes it impossible to predict the exact number of draws that will be necessary to obtain a given number of random bits.

However, as we will see in Section 4, it is somewhat useless to first filter the output (and lose some entropy) to obtain bits, and then filter these bits *again* to get parameters. Instead, we could keep the entropy in the form of integers and directly use these integers to generate parameters. This way, filtering only happens once in the end, when trying to match the Security criteria.

In practice, converting a set of winning numbers into an integer will be done with formula (1). Assume we draw m numbers $c_1 < c_2 < ... < c_m$, each between 1 and n, then a uniformly distributed index x between 0 and $\binom{n}{m} - 1$ can be computed as [27, Theorem L, p.360]:

$$x = \binom{c_1 - 1}{1} + \binom{c_2 - 1}{2} + \cdots + \binom{c_m - 1}{m}. \tag{1}$$

Note. The order in which the numbers are drawn could be taken into account to increase the amount of entropy extracted from each draw ($\log_2(m!)$ bits can be gained), but many lotteries only keep an archive of *sorted* draws. For this reason we only look at the unordered set of winning numbers. Also, the order in which the numbers are drawn is never used to determine the winners. For public verifiability it seems more natural to tie our entropy strictly to the "winning condition" and thus to ignore the drawing order.

3.2 Defining a Publicly Verifiable RNG from a Single-Draw Lottery

Given the above discussion, it is easy to define a proper but inefficient Publicly verifiable RNG, based on one single-draw lottery picking m numbers in a set of n possible numbers. This Publicly verifiable RNG produces daily entropy:

- If a draw was performed on the considered day, the entropy is extracted as described in the previous section, producing a random integer x in $[0, \binom{n}{m}-1]$;
- Otherwise, the Publicly verifiable RNG outputs `NaN` (which means that it produced no entropy on that day).[15]

[15] If the game rules change at some point, e.g. if m or n change, we simply consider the new set of rules as a different game. As a consequence, as soon as the new game rules are live, the Publicly verifiable RNG starts to output `NaN` forever.

National lotteries are certainly a very good verifiable source of entropy, but they also suffer from a very slow output rate. In France for example there are three Loto draws per week, each offering 20.86 bits of entropy, that is, 62.58 bits per week. Table 1 sums up the amount of entropy that can be easily collected each week from various lotteries around the world.

3.3 Combining Several Single-Draw Lotteries to Increase the Entropy Throughput

In order to increase the daily entropy produced by the aforementioned Publicly verifiable RNG, it seems natural to consider the draws from several (single-draw) lotteries. In that case, many draws may occur on the same day. In order to properly define the Publicly verifiable RNG, we must specify an order in which the lotteries should be considered. We suggest to associate a unique identifier `lottery_id` to each lottery. Once the lotteries are identified as above, one can easily specify the order (e.g., the alphabetical order).

Thus, a Publicly verifiable RNG based on ℓ well identified single-draw lotteries produces, on a given day, an array of ℓ values, where the i-th value corresponds to the output of a Publicly verifiable RNG based on the i-th lottery on the list, implemented as described in the previous section.

3.4 Including Multiple-Draw Lotteries

In practice, some lotteries perform several draws on the same day. However, these draws are always sequential and clearly identifiable. Thus, one can simply consider them as draws coming from distinct and uniquely identified single-draw lotteries. For example, if the lottery `lottery_id` performs two distinct draws on the same day, one can simply consider these draws as respectively coming from lotteries `lottery_id_1` and `lottery_id_2`. This approach enables to include multiple-draw lotteries in the Publicly verifiable RNG of the previous section.

3.5 Concrete Example

As an example, we consider the following list of lotteries, ordered alphabetically:

1. `fr_keno_1` (French Keno, noon draw)
2. `fr_keno_2` (French Keno, evening draw)
3. `us_powerball` (US Powerball)

On the Friday 4th of December 2015, the noon draw of the French Keno was

`2-4-7-9-12-13-16-21-23-30-31-32-36-39-42-49-52-57-64-68`

The index of this draw given by (1) is

$$x = \binom{2-1}{1} + \binom{4-1}{2} + \binom{7-1}{3} + \cdots + \binom{68-1}{20} = 64324389717285723$$

Table 1. List of lotteries around the world, with their corresponding weekly entropy.

Lottery name	m	n	Entropy	Draws per week	Entropy per week
Australian Monday Lotto	6	45	22.95	1	22.95
Australian OZ Lotto	7	45	25.43	1	25.43
Australian Powerball	6	40	21.87	1	21.87
Australian Saturday Lotto	6	45	22.95	1	22.95
Australian Wednesday Lotto	6	45	22.95	1	22.95
Belgian Lotto	6	45	22.95	2	45.90
Brasilian Dupla-Sena	6	50	23.92	2	47.84
Brasilian Lotofácil	15	25	21.64	3	64.92
Brasilian Mega-Sena	6	60	25.57	2	51.14
Brasilian Quina	5	80	24.51	6	147.06
Canadian Daily Keno (Midday Draw)	20	70	57.16	7	400.12
Canadian Daily Keno (Evening Draw)	20	70	57.16	7	400.12
Canadian Loto 649 (Main Draw)	6	49	23.73	2	47.46
Canadian Loto Max (Main Draw)	7	49	26.35	1	26.35
Canadian Lottario	6	45	22.95	1	22.95
Swiss Loto	6	42	22.32	2	44.64
German Euro Jackpot	5	50	21.01	1	21.01
German Keno	20	70	57.16	7	400.12
German Loto	6	49	23.73	2	47.46
Spanish Bonoloto	6	49	23.73	6	142.38
Spanish El Gordo	5	54	21.59	1	21.59
Spanish La Primitiva	6	49	23.73	2	47.46
European Euro Millions	5	50	21.01	2	42.02
French Keno (Noon)	20	70	57.16	7	400.12
French Keno (Evening)	20	70	57.16	7	400.12
French Loto	5	49	20.86	3	62.58
Italian SuperEnalotto	6	90	29.21	3	87.63
Mauritius Loto	6	40	21.87	1	21.87
Dutch Lotto (Standard 1st trecking draws)	6	45	22.95	1	22.95
New Zealand Keno (1st daily draw - 10AM)	20	80	61.61	7	431.27
New Zealand Keno (2nd daily draw - 1PM)	20	80	61.61	7	431.27
New Zealand Keno (3rd daily draw - 3PM)	20	80	61.61	7	431.27
New Zealand Keno (4th daily draw - 6PM)	20	80	61.61	7	431.27
New Zealand Lotto	6	40	21.87	2	43.74
UK Health Lottery (Saturday £1 draw)	5	50	21.01	1	21.01
US Hot Lotto	5	47	20.54	2	41.08
US Mega Millions	5	75	24.04	2	48.08
US NY Cash 4 Life	5	60	22.38	2	44.76
US NY Lotto	6	59	25.42	2	50.84
US Powerball	5	69	23.42	2	46.84
US Wild Card	5	33	17.85	2	35.70
Total					≈ 5189

Similarly, one can check that the evening draw[16] (i.e. from lottery `fr_keno_2`) leads to the index 55537728386360944 and that the US Powerball did not perform any draw on that day. Those results, together with those of the two following days, made the Publicly verifiable RNG produce the following outputs between the 4th and the 7th of December 2015:

$$\begin{bmatrix} 64324389717285723, & 55537728386360944, & \texttt{NaN} \end{bmatrix}$$
$$\begin{bmatrix} 103119038557241541, & 1139614140761531, & 9826130 \end{bmatrix}$$
$$\begin{bmatrix} 140625738347277372, & 155799364658105184, & \texttt{NaN} \end{bmatrix}$$
$$\begin{bmatrix} 94173221000906309, & 83857548427108173, & \texttt{NaN} \end{bmatrix}$$

4 Generating a Seed from a Publicly Verifiable RNG Based on Several Multiple-Draw Lotteries

In this section, we describe how to generate a random seed, containing at least k bits of entropy, using the Publicly verifiable RNG based on ℓ uniquely identified lotteries introduced in Section 3. Note that it suffices to draw the seed uniformly at random in $[0, L-1]$, where $L \geq 2^k$ (this exactly produces $\log_2(L) \geq k$ bits of entropy). Since k can be arbitrarily large, we need a method that uses several Publicly verifiable RNG outputs, i.e., a method combining several lottery draws.

4.1 Combining Several Lottery Draws

Each lottery has its own set of rules: the California lottery Powerball picks 5 numbers from 1 to 69, the French Loto picks 5 from 1 to 49, the Belgian Lotto picks 6 from 1 to 45, etc. Let us consider a ordered array of r lottery draws, where the ith draw is an unbiased selection of m_i numbers among n_i. We apply (1) on each draw in order to obtain an array of indices $[x_1, \ldots, x_r]$, where x_i is uniformly distributed in $[0, L_i - 1]$ for $L_i = \binom{n_i}{m_i}$. Considering the L_i's as a mixed radix numeral system [41], a unique representative of $[x_1, \ldots, x_r]$ in $[0, \prod_{i=1}^{r} L_i - 1]$ can be obtained by the following one to one mapping:

$$\begin{aligned} [0, L_1 - 1] \times \cdots \times [0, L_r - 1] &\longrightarrow [0, \prod_{i=1}^{r} L_i - 1] \\ (x_1, \ldots, x_r) &\longmapsto x_r + x_{r-1} \cdot L_r + x_{r-2} \cdot L_r L_{r-1} + \cdots + x_1 \cdot \prod_{i=2}^{r} L_i \,. \end{aligned} \tag{2}$$

As this is a one to one mapping, if the r inputs are indeed uniformly distributed over their domain, the output will also be uniformly distributed over its domain. Combining several lottery draws makes it possible to obtain uniformly distributed numbers of arbitrarily long size.

[16] `2-5-11-12-15-17-20-24-32-45-48-52-53-54-57-61-62-63-66-67`

Table 2. An ordered list of draws based on the Publicly verifiable RNG of Section 3.5.

r	Draw Id	m	n	L_i	Index
1	`2015-12-04_fr_keno_1`	20	70	$\binom{70}{20}$	64324389717285723
2	`2015-12-04_fr_keno_2`	20	70	$\binom{70}{20}$	55537728386360944
3	`2015-12-05_fr_keno_1`	20	70	$\binom{70}{20}$	103119038557241541
4	`2015-12-05_fr_keno_2`	20	70	$\binom{70}{20}$	1139614140761531
5	`2015-12-05_us_powerball`	5	69	$\binom{69}{5}$	9826130

4.2 Generating a Seed of at Least k Bits of Entropy using the Publicly Verifiable RNG

Since the Publicly verifiable RNG we consider is based on ℓ well specified lotteries, it is easy to evaluate the exact amount of entropy available during any time period. Given a starting date t_3 and the minimum amount k of entropy required, one can thus determine an exact list of r draws to extract from the Publicly verifiable RNG in order to achieve at least this amount of entropy. In order to use (2) to combine those draws, their order must be fully specified. Doing so will produce a uniformly distributed random integer in $[0, L-1]$, where $L = \prod_{i=1}^{r} L_i$, and $\log_2(L) \geq k$.

4.3 Concrete Example

We consider the (ordered) list of draws of Table 2 that one can extract from the Publicly verifiable RNG of the example given in Section 3.5. Those draws allow to collect more than $k = 252$ bits of entropy since $L = \binom{70}{20}^4 \binom{69}{5} \approx 2^{252.09}$. The random seed obtained by means of (2) in this case is

$$306691094761969777184326301362280350899838183369054660237074735075977250073 7.$$

4.4 The Last Draw Attack

The output of our algorithm is a set of cryptosystem parameters that might end up being used to secure millions of exchanges, it is thus necessary to assume that an adversary trying to manipulate this parameter selection process could have a lot of power[17]:

[17] Quoting Philip Rogaway [34]: "At this point, I think we would do well to put ourselves in the mindset of a *real* adversary, not a notional one: the well-funded intelligence agency, the profit-obsessed multinational, the drug cartel. You have an enormous budget. You control lots of infrastructure. You have teams of attorneys more than willing to interpret the law creatively. You have a huge portfolio of zero-days. You have a mountain of self-righteous conviction. Your aim is to *Collect it All, Exploit it All, Know it All.*"

– computational power to perform expensive computations in a short time,
– a way to influence certain lottery draws,
– some advanced cryptographic knowledge, giving him "0-day" attacks on certain classes of parameters satisfying all Security criteria.

Of course, even the most powerful organizations on the planet have some limitations, so we make the assumption that:

– it is impossible for the adversary to manipulate all lottery draws simultaneously,
– 0-day attacks only allow to break a fraction of the Parameter space, typically one set of parameters in a thousand, or in a million.[18]

Still, even with these assumptions, it could be possible for a powerful adversary to force the selection of weak parameters using what we call the *last draw attack*.

The seed extracted from the lottery draws is the only source of entropy in the parameter generation. The instantiation of the PRNG and the filtering functions are deterministic, so a given seed will always give the same parameters. Because of this, the last draw used to generate this seed plays a special role: after it, no more entropy will enter the system, so anyone able to manipulate this last draw can knowingly influence the parameter selection output. Just after the next to last draw is made public, an adversary can start computing the parameters that would be output by each possible value of the last draw. With a 0-day attack applying to a non-negligible fraction of sets of parameters, there is a high probability that some draw values yield a set of weak parameters. The adversary can then force the output of the last lottery draw and force the whole process to output parameters he can attack.

There are several solutions to overcome this attack, the best one will depend on the usage scenario:

– *Simultaneous lottery draws.* The simplest solution to avoid last draw attacks is to have multiple last draws happening exactly at the same time. This way, an adversary needs to manipulate all draws simultaneously to "choose" the output parameters. If we assume manipulating several lottery draws from different countries to be impossible for the adversary, this solution could work. Unfortunately all lottery draws from Table 1 seem to happen at distinct times, making it impossible to rely on this solution.
– *Using a slow function.* This idea was first presented in [28] but can be adapted to our lottery setting. If the filtering function or the seed extractor relies on a slow function that is guaranteed to take more than 10 minutes to compute (even on the most advanced hardware), then if the last two lottery draws occur less than 10 minutes apart, last draw attacks are impossible. However, if the last lottery draws occur several hours apart, or if resistance

[18] A similar adversary, named Jerry, was considered by Bernstein et al. in [7] and is assumed to have 0-day attack on a fraction of, say, $\approx 2^{-15}$ of the Parameter space.

to the collusion of the last few lotteries is needed, it might be necessary to use a function that takes several hours or several days to compute, which is not very practical. Also, depending on the effective drawing dates chosen by the Standardizer, one would have to modify the Filtering function choice accordingly, which is something we want to avoid!

Also, it is important that this slow function does not discard any entropy from the Publicly verifiable RNG, or does so in a "provably secure way". Otherwise, an attack on the slow function itself might allow to decrease the influence of the entropy from the last draws, possibly allowing the manipulation of the very first draws to be sufficient for an adversary.

– *Using a commitment.* When committing on the set of lottery draws that will constitute the seed, the standardizer could also commit on the hash of a random value (to be made part of the seed), keeping this random value hidden until after the last draw. As long as the standardizer and the organization operating the last lottery do not collude, the last draw attack is impossible. Of course, for the output parameter set to be convincing, it should be clear that the standardizer did not collude with the lottery organizations, and this approach is only valid when the standardizer is trusted. This is typically the case when a person wants to generate parameters for his personal use, or when a company generates parameters for internal use. This solution also suits situations where a client outsources the generation of a Cryptosystem parameters to a company: the client can himself commit on a nonce, and reveal it to the company once the full entropy has been extracted from the Publicly verifiable RNG.

– *Limiting the entropy of the last lottery draws.* This last solution relies on the assumption that a 0-day attack may affect a non-negligible part of the possible parameters, but not a large proportion. If the last lottery draw introduces only 1 bit of entropy, then the last draw attack can only try 2 sets of parameters, thus requiring a 0-day attack affecting one half of the possible parameters. Such an attack seems improbable. However, we must make sure that next-to-last-draw attacks are not possible: if the adversary can manipulate the next to last draw so that both possible outputs of the last draw lead to weak parameters it is also a problem, so we need to add a little more than a single bit of entropy after the last draw. The solution is thus to add (after the last draw) a series of, say, 20 or 30 lottery draws from each of which a single bit of entropy is extracted. We call those extra bits of entropy *lone bits*. If the 0-day attack affects a fraction $\frac{1}{N}$ of parameters, the adversary has to manipulate $\log N$ lone bits, and thus, $\log N$ lotteries (which we assumed was impossible).

In the following section, we present a way of properly integrating lone bits in the seed computation. This method is the one we will use in Section 6 to generate the *Million Dollar Curve*.

Table 3. An ordered list of 3 more draws based on the Publicly verifiable RNG of Section 3.5.

r	Draw Id	m	n	L_i	Index	Lone bit
6	`2015-12-06_fr_keno_1`	20	70	$\binom{70}{20}$	140625738347277372	0
7	`2015-12-06_fr_keno_2`	20	70	$\binom{70}{20}$	155799364658105184	0
8	`2015-12-07_fr_keno_1`	20	70	$\binom{70}{20}$	94173221000906309	1

4.5 Properly Integrating Lone Bits to the Seed

We first note that when using (2) to combine several lottery draws, the last draw influences the least significant bits of the seed. As we will see in the next section, because of the way we consume the seed to generate Blum-Blum-Shub parameters, some entropy may be lost among those least significant bits. This is a good thing for the last draw, since this decreases the power of a potential last draw attack. This is however a undesirable thing for the lone bits, since we need to maintain all their entropy to prevent a next-to-last draw attack. As a consequence, we suggest to add the lone bits in the most significant bits of the seed.

We start from a seed $s \in [0, L-1]$ computed using (2), to which we intend to integrate ℓ' lone bits coming from ℓ' additional sorted lottery draws. We first compute the ℓ' indexes of the lottery draws using (1) and let $b_1, \ldots, b_{\ell'}$ denote their respective least significant bit (i.e., the modulo 2 reduction of their respective index). We then integrate them to the seed as follows:

$$\mathsf{seed} = \mathsf{seed} + L \cdot \sum_{i=1}^{\ell'} b_i \cdot 2^{i-1}. \tag{3}$$

If we add the two draws of December 6 and the first draw of December 7 from the Publicly verifiable RNG given in Section 3.5 to the list given in Table 2, we can obtain three lone bits, as shown in Table 3. Integrating these lone bits to the seed of the example of Section 4.3 using (3), we obtain the following final seed value, containing more than $252 + 3 = 255$ bits of entropy:

339407779461999879064016020675985999424170260786738870240829151513778079802 89.

5 Entropy Spreading Using Blum Blum Shub

As we have seen in the previous sections, by combining easily accessible lottery draws from all over the world we reach an output rate of about 5,000 bits of entropy per week. This is enough entropy to generate almost any set of parameters as the Parameter space defined by the Security criteria will seldom contain more than 2^{2000} elements. Yet, we should distinguish between two kinds of Security criteria:

– "simple" Security criteria, which define a Parameter space in which it is easy to directly sample elements uniformly at random;
– "complex" Security criteria, which make direct uniform sampling impossible, forcing to iteratively query the Publicly verifiable RNG for parameters, discarding those that do not meet all the Security criteria.

A scheme with only "simple" Security criteria can be instantiated using the output of the Publicly verifiable RNG directly. However, for most cryptographic schemes, there will be at least a few "complex" Security criteria requiring to discard a huge proportion of this entropy. In that situation, it could take many weeks to generate appropriate parameters.

Instead of waiting so long, we propose to use this publicly verifiable entropy to instantiate a Blum Blum Shub [11,12] PRNG (abbreviated BBS from now on), and then use it to deterministically generate very long sequences of bits. Wasting bits outputted by the PRNG is then no longer an issue, but our use of BBS must ensure that if enough entropy is given for the instantiation, then the output parameters are uniformly distributed in the Parameter space or, at least, that the output distribution of parameters is computationally indistinguishable from a uniform distribution.

5.1 Uniformity of Parameters Output by the Blum Blum Shub PRNG

The BBS generator works as follow:

1. pick two strong strong primes[19] p and q and compute $N = pq$,
2. pick a random $s \in \mathbb{Z}_n^* \setminus \{1\}$ and compute an initial value $s_0 = s^2 \mod N$,
3. generate bits by iterating the function $s_i = s_{i-1}^2 \mod N$ and then output the least significant bit of s_i, for $i \geq 1$.

It is proven that the output of this generator is indistinguishable from a random sequence of bits as long as the quadratic residuosity assumption[20] holds. In our context, what matters is that parameters are *uniformly distributed* in the Parameter space. Now, suppose there is an algorithm $\mathcal{A}$ which, when given a sequence $rand$ of uniformly distributed independent bits, outputs a set of parameters uniformly distributed in the Parameter space. Now run algorithm $\mathcal{A}$ with a sequence BBS of bits output by BBS. If one is able to distinguish parameters output by $\mathcal{A}(rand)$ from parameters output by $\mathcal{A}(BBS)$, then this is also a distinguisher for the output of BBS, meaning it gives an advantage when deciding quadratic residuosity. As a consequence, for any p and q large enough

[19] A prime p is a *strong strong prime* if all three of $p, (p-1)/2$ and $(p-3)/4 = ((p-1)/2-1)/2$ are primes. We select p and q as strong strong primes in order to maximize the cycle length. Indeed, recall that the BBS generator is cyclic of order $\lambda(\lambda(N)) = \mathrm{lcm}(\phi(p-1), \phi(q-1))$ where λ is Carmichael's function and ϕ is Euler's totient function. In particular, when $p = 4p' + 3$ and $q = 4q' + 3$ are strong strong primes, the cycle is maximal and equal to $2p'q'$.

[20] That is, deciding if for a given $y < N$ there exists $z < N$ such that $y = z^2 \mod N$.

for the quadratic residuosity problem to be computationally unfeasible, BBS can be used to generate uniformly distributed parameters when s contains enough entropy.

Now if the distribution is uniform when p and q are unknown, this distribution does not change when p and q are revealed.[21] Therefore, the parameters are uniformly distributed, even if p and q are known.

Instantiating BBS using the publicly verifiable RNG. When using the Publicly verifiable RNG to directly instantiate a scheme, the amount of entropy to extract depends on the length of the parameters of that scheme. In the case the Publicly verifiable RNG is used to instantiate BBS, it is rather the hardness of the quadratic residuosity problem that drives the amount of required entropy. As we have seen, one can uniformly pick parameters from any fixed and large enough p and q and an initial value s_0 with enough entropy. So we could pick p and q for you, compute N and publish it. But this might look a little suspicious! Instead, we propose to also pick p and q from the Publicly verifiable RNG. To be conservative, we suggest to always consider at least 2048-bit strong strong primes and an s_0 of size $\log_2(N)$.

5.2 Inefficiently Choosing Random Parameters for BBS

In this section, we describe a deterministic but inefficient algorithm that takes as input a seed (e.g., generated as in Section 4) and outputs two strong strong primes p and q, and an appropriate initial value s_0 for BBS. When aiming at k-bit primes, seed should contain at least $4k$ bits of entropy. The algorithm is the following:

1. Let $p_{\text{start}} = \mathsf{seed} \bmod 2^k$ and let p be the smallest strong strong prime p greater than $2^{2048} + p_{start}$. Shift seed by k bits to the right.
2. Let $q_{\text{start}} = \mathsf{seed} \bmod 2^k$ and let q be the smallest strong strong prime p greater than $2^{2048} + q_{start}$. Shift seed by k bits to the right.
3. Let $s = \mathsf{seed} \bmod N$ where $N = pq$. While $s \notin \mathbb{Z}_N^* \setminus \{1\}$, replace s by $s + 1 \bmod N$. Let $s_0 = s^2 \bmod N$.

Obviously, choosing p and q as the next strong strong prime from a random starting point does not lead to uniformly distributed p and q. Indeed, strong strong primes are not equally spaced, so some values of p and q have a slightly higher probability of being picked. However, as any value of p and q will allow to pick parameters uniformly, we simply need the choice of p and q to retain as much entropy as possible, which is the case here.

[21] Quadratic residuosity becomes easy to decide when p and q are known, so when using BBS as a *cryptographically secure* PRNG, p and q should be kept secret.

5.3 Efficiently Choosing Random Parameters for BBS

Although we do not really care about the efficiency of the algorithm generating p and q (since it will essentially be used once), there exist well known methods for generating primes much more efficiently. We suggest to use an approach similar to the one suggested in [26, Section 7.2]. Since our only source of entropy is the seed, we cannot directly use the algorithms of [26] since the amount of entropy they require to generate primes is not known in advance. Yet, it is easy to adapt those algorithms to make them entropy-friendly.

In what follows, we describe how to generate strong strong prime *generators*, i.e., primes p' such that $p = 4p' + 3$ is a strong strong prime (and thus all three of $p', 2p' + 1$ and $2(2p' + 1) + 1 = 4p' + 3$ are prime).

We first denote by $\mathcal{P} = [p_1, p_2, \ldots, p_f]$ the list of the f first primes (starting with $p_1 = 2$), by Π be the product of those first primes, and let

$$\begin{aligned} \mathsf{CRT} : \mathbb{Z}_\Pi &\longrightarrow \mathbb{Z}_{p_1} \times \cdots \times \mathbb{Z}_{p_f} \\ c &\longmapsto (c \bmod p_1 \ , \ \ldots \ , \ c \bmod p_f) \end{aligned}$$

the CRT isomorphism. For $c \in \mathbb{Z}_\Pi$, we note that c, $2c+1$, and $2(2c+1)+1$ are invertible modulo Π (and thus, coprime with Π) if and only if $c_i = c \bmod p_i$, $2c_i+1$, and $2(2c_i+1)+1$ are invertible modulo p_i, for all $i = 1, \ldots, f$. We denote by $\mathcal{C}_i$ the list of all $c_i \in \mathbb{Z}_{p_i}$ such that c_i, $2c_i+1$, and $2(2c_i+1)+1$ are invertible modulo p_i. For example, $p_3 = 5$ and $\mathcal{C}_3 = [1, 4]$. Let C be the product of the cardinalities of the $\mathcal{C}_i$'s. Eventually, C will correspond to the number of strong strong prime generator candidates to test for primality.

Starting with a seed containing at least $2\log_2(C) + 2\log_2(\Pi)$ bits of entropy, the following algorithm provides an efficient way to choose two strong strong primes p and q of size close to k bits, and an initial value s_0 for BBS:

1. Determine f such that $\Pi/p_f \leq 2^{k-2} < \Pi$. Precompute the $\mathcal{C}_i$'s and C.
2. For $i = 1, \ldots, f$, let $\lambda_i = \mathsf{seed} \bmod \#\mathcal{C}_i$ and $\mathsf{seed} = (\mathsf{seed} - \lambda_i)/\#\mathcal{C}_i$.
3. Loop:
   3.1. Let $c = \mathsf{CRT}^{-1}(\mathcal{C}_1[\lambda_1], \ldots, \mathcal{C}_f[\lambda_f])$.
   3.2. If c is a strong strong prime generator of at least $k-2$ bits, set $p = 4c+3$ and break out of the loop.
   3.3. Set $i = 1$ and loop:
      – Replace λ_i by $(\lambda_i + 1) \bmod \#\mathcal{C}_i$. If $\lambda_i \neq 0$, break out of the loop, otherwise, replace i by $(i + 1) \bmod f$.
4. Repeat steps 2 to 4 to pick a second (independent) strong strong prime q and compute $N = pq$, the BBS modulus.
5. Let $s = \mathsf{seed} \bmod N$ where $N = pq$. While $s \notin \mathbb{Z}_N^* \setminus \{1\}$, replace s by $s + 1 \bmod N$. Let $s_0 = s^2 \bmod N$.

For the same reason as the inefficient algorithm of the previous section, this algorithm obviously produces *slightly* non-uniform BBS parameters. As we have seen, this is not an issue as this bias does not directly impact the uniformity of

the final parameter generation. However (again, like in the inefficient algorithm), this algorithm looses some of the seed entropy among its least significant bits. Indeed, flipping the least significant bit of the seed only influences the value of λ_3 (since, for primes 2 and 3, we have $\#\mathcal{C}_1 = \#\mathcal{C}_2 = 1$, and for prime 5, $\#\mathcal{C}_3 = 2$). Moreover, this λ_3 is the first one to be incremented at step 3.3 of the algorithm. As a consequence, in case the first candidate obtained from the seed is not a strong strong prime (which is highly likely), the final output is not influenced at all by the least significant bit of the seed.

The same applies, with lower probabilities, to the other least significant bits of the seed. This is the reason why we chose to integrate the last draw (of which we want to minimize the impact) to the least significant bits of the seed, and the lone bits to the most significant bits of the seed.

5.4 Concrete Example

Assume we want to generate BBS parameters with strong strong primes of 64 bits at least. The smallest list of the first primes leading to $\Pi > 2^{64-2}$ contains $f = 16$ primes and is

$$\mathcal{P} = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53].$$

In that case, $\Pi = 32589158477190044730 \approx 2^{64.82}$ whereas the list of the first 15 primes would lead to a product $\Pi' = \Pi/53 = 614889782588491410 \approx 2^{59.09}$. At this point, the algorithm precomputes $\mathcal{C}_1, \ldots, \mathcal{C}_{16}$, where $\mathcal{C}_i$ contains the list of $c_i \in \mathbb{Z}_{p_i}$ such that c_i, $2c_i + 1$, and $2(2c_i + 1) + 1$ are all invertible modulo p_i. Considering the first seven primes for example, we have

$$\mathcal{C}_1 = [1], \mathcal{C}_2 = [2], \mathcal{C}_3 = [1, 4], \mathcal{C}_4 = [2, 4, 5, 6], \mathcal{C}_5 = [1, 3, 4, 6, 7, 8, 9, 10],$$
$$\mathcal{C}_6 = [1, 2, 3, 4, 5, 7, 8, 10, 11, 12], \mathcal{C}_7 = [1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 13, 14, 15, 16].$$

Computing all the $\mathcal{C}_i$'s leads to $C = \prod_{i=1}^{16} \#\mathcal{C}_i = 237320116633600000$, which corresponds to the number of strong strong prime generator candidates. We therefore require from the seed that it contains at least $\log_2(C) \approx 58$ bits of entropy for drawing p, 58 more bits of entropy for q, and 128 bits for drawing s, for a total of 244 bits. Since the seed obtained in the example of Section 4.5 contains more than 255 bits of entropy, we can use it to proceed with the algorithm.

Since $\#\mathcal{C}_1 = \#\mathcal{C}_2 = 1$, then $\lambda_1 = \lambda_2 = 0$ for any seed. Since $\#\mathcal{C}_3 = 2$ and the seed we consider is odd, we have $\lambda_3 = 1$ and we must replace the seed by $(\mathsf{seed} - 1)/2$, which is

$$169703889730999939532008010337992999712085130393369435120414575756889039901 44.$$

Reducing this new seed modulo $\#\mathcal{C}_4 = 4$ gives $\lambda_4 = 0$. Proceeding similarly with the remaining first primes, one can easily check that

$$[\lambda_1, \lambda_2, \ldots, \lambda_{16}] = [0, 0, 1, 0, 0, 2, 3, 5, 4, 9, 22, 20, 22, 35, 33, 26]$$

and that the remaining seed is

$$1430168602124925022190969798008136551754487350727132144247 38.$$

These indexes lead to the first strong strong prime generator candidate:

$$\begin{aligned} c &= \mathsf{CRT}^{-1}(\mathcal{C}_1[0], \mathcal{C}_2[0], \mathcal{C}_3[1], \mathcal{C}_4[0], \mathcal{C}_5[0], \mathcal{C}_6[2], \mathcal{C}_7[3], \ldots, \mathcal{C}_{14}[35], \mathcal{C}_{15}[33], \mathcal{C}_{16}[26]) \\ &= \mathsf{CRT}^{-1}(1, 2, 4, 2, 1, 3, 4, 7, 6, 10, 25, 22, 24, 38, 36, 28) \\ &= 4200766960142310119\,. \end{aligned}$$

Since

$$\begin{aligned} c &= 26573 \times 541097 \times 292154699, \\ 2c+1 &= 5791 \times 6818837 \times 212762317, \text{ and} \\ 4c+3 &= 103 \times 1061 \times 3664103 \times 41963171 \end{aligned}$$

one can see that, as expected, neither c, $2c+1$, nor $4c+3$ is divisible by any of the 16 first primes. Yet, this first candidate obviously is not a strong strong prime generator and the indexes must be updated. Incrementing cyclicly the λ_i's leads to the second set of indexes

$$[\lambda_1, \lambda_2, \ldots, \lambda_{16}] = [0, 0, 0, 1, 0, 2, 3, 5, 4, 9, 22, 20, 22, 35, 33, 26],$$

from which we deduce a new strong strong generator candidate

$$c = 21892024419188334401 = 199 \times 683 \times 161069067292453.$$

Once again, this candidate is not a strong strong generator either, and we are required to iterate again.

After a few more bad candidates, we obtain the following 63rd set of indexes:

$$[\lambda_1, \lambda_2, \ldots, \lambda_{16}] = [0, 0, 1, 3, 7, 2, 3, 5, 4, 9, 22, 20, 22, 35, 33, 26]$$

leading to the first strong strong generator candidate $c = 7586653555175042039$ such that c, $2c+1$, and $4c+3$ are all primes. The algorithm thus chooses

$$p = 4c + 3 = 30346614220700168159.$$

Proceeding similarly with the remaining seed, one can obtain

$$q = 96890065747994265119$$

after 27 bad candidates, and the remaining seed is

$$60263268972388514258961239280717039213323 8.$$

Reducing this seed modulo $N = pq$ gives

$$s = 28144585210631061646454638820610503653 54$$

and thus

$$s_0 = s^2 \bmod N = 24587884803847069781202464964383777983 77.$$

Using the BBS generator with these N and s_0 yields the bit sequence

$$0100101000100111000001010111011110001000010100010100010111 01\ldots$$

6 Million Dollar Curve

In this section, we first act as a Designer and give:

- A Cryptosystem: the elliptic curve Diffie–Hellman (ECDH),
- An exhaustive list of Security criteria (which implicitly defines a Parameter space),
- A Filtering function given as an algorithm.

We then continue our concrete example from previous sections (where we acted as a Standardizer, and where we generated a seed) and illustrate how to use the seed previously obtained to instantiate a toy Cryptosystem which, of course, should *not* be used for cryptographic purposes. Indeed, the Publicly verifiable RNG we used relies on too few lottery draws, which were moreover chosen after their draw date.

6.1 The Designer Part

The cryptosystem. The Cryptosystem we consider is elliptic curve Diffie–Hellman (ECDH). Defining parameters for this cryptosystem consists in defining a specific elliptic curve and a base point on this curve.

The security criteria. This section provides an exhaustive list of the Security criteria that we want the Million Dollar Curve to meet. These criteria include all those listed on SafeCurves [8].

Underlying field. Recent advances (such as [25]) confirmed that elliptic curve over non prime fields might present a risk. We choose to restrict to prime fields. As we aim for a security level of 128 bits, we further restrict to 256-bit primes, i.e., primes in $[2^{255}, 2^{256})$. We shall denote by p this 256-bit prime and furthermore require that $p \equiv 3 \pmod 4$ (see the the discussion on twist security below).

Curve in Edwards form. Restricting to curves that can be written in Edwards form [19] is not a security criterion per se. However, as shown in [9], Edwards curves have nice properties that make them particularly well suited for cryptographic purposes, including an efficient complete addition formula. Restricting to these curves is unlikely to cause any security issue since about $1/4$ of all (isomorphism classes of) elliptic curves over non-binary finite field are birationally equivalent over the original field to an Edwards curve [9]. Given the finite field $\mathbb{F}_p$, an Edwards curve can be defined with a single parameter $d \in \mathbb{F}_p \setminus \{0, 1\}$, by the equation $x^2 + y^2 = 1 + dx^2y^2$. For the addition formula to be complete, d should furthermore not be a square in $\mathbb{F}_p$.

Point compression. Since we restrict to non-square d's, it is also possible to efficiently compress points exchanged during the Diffie–Hellman protocol by sending only the y coordinate of each point. This is shown in Theorem 1 in Appendix A. This is an interesting feature in some contexts where bandwidth or storage is constrained.

Point count. An Edwards curve always has a point of order 4, so the number of points $\#E(\mathbb{F}_p)$ is always a multiple of 4. The base point on the curve should generate a large subgroup of prime order. We thus impose that the number of points on the curve to be of the form $4q$, where q is a prime. The average complexity of the rho method (using negation) for computing discrete logarithms is $0.886\sqrt{q}$ group operations [8,21]. We impose the same restriction on q than SafeCurves, namely that $q > 2^{200}$. From the Hasse bound $|\#E(\mathbb{F}_p) - (p+1)| \leq 2\sqrt{p}$, one can see that this condition is always verified when p is of 256 bits.

Base point. The base point should generate a large prime order subgroup of the curve. We impose that its order is q.

Anomalous curve attack. We require $q \neq p$ to avoid attacks using additive transfer. This criterion is always verified for Edwards curve.[22]

Embedding degree. The ECDLP problem can be converted into a DLP problem in the finite field $\mathbb{F}_{p^m}$, where m is the embedding degree of the curve, i.e., the smallest integer m such that $p^m \equiv 1 \pmod{q}$. Thus ensuring that $m \geq 20$ should be enough. Like SafeCurves [8] and Brainpool [1], we choose the overkill criteria $m > (q-1)/100$.

CM field discriminants. Let $t = p + 1 - \#E(\mathbb{F}_p) = p + 1 - 4q$ be the trace of the curve and s^2 be the largest square dividing $t^2 - 4p$ then $(t^2 - 4p)/s^2$ is a square-free negative integer. Define D as $(t^2 - 4p)/s^2$ if $(t^2 - 4p)/s^2 \bmod 4 = 1$, otherwise as $4(t^2 - 4p)/s^2$. We require $|D| \geq 2^{100}$.

Twist security. We impose the same security criteria on the quadratic twist of the curve:

- Point count: the twist should have a number of points of the form $4q'$, where q' is prime. Note this cannot happen when $p \equiv 1 \pmod 4$, which is the reason why we restricted to primes such that $p \equiv 3 \pmod 4$.
- Anomalous curve attack: we require $q' \neq p$ to avoid attacks using additive transfer. Just as the $q \neq p$ criterion, this criterion is automatically verified.
- Embedding degree: the embedding degree of the twist should be at least $(q'-1)/100$.

The Filtering Function. Starting with the BBS parameters selected by means of the algorithm of Section 5.3, we use the following algorithm to output the parameters of the Cryptosystem that meets all the Security criteria:

1. Given a seed, use the algorithm from Section 5.3 to initialize a BBS generator.
2. Prime field selection loop:

[22] Let $\#E(\mathbb{F}_p) = c \cdot q$, where q is the large subgroup order and c is the cofactor ($c = 4$ in our case). From the Hasse bound, q lives in an interval of length $4\sqrt{p}/c$ around $(p+1)/c$. For $c \geq 2$, this implies that $q \neq p$.

2.1. Generate 253 bits $b_1, b_2 \dots b_{253}$ using BBS and let

$$p = 2^{255} + 3 + \sum_{i=1}^{253} b_i \cdot 2^{255-i}.$$

2.2. If p is prime, break free [31] of the loop.

3. Curve selection loop:

3.1. Generate 256 fresh bits $b_1, b_2 \dots b_{256}$ using BBS and let

$$d = \sum_{i=1}^{256} b_i \cdot 2^{256-i}.$$

3.2. If $d = 0$ or $d \geq p$, loop (i.e., go back to step 3.1).

3.3. If d is a square modulo p, loop.

3.4. Compute the cardinality $\#E$ of the Edwards curve E over $\mathbb{F}_p$ defined by d. If $q = \frac{\#E}{4}$ is not prime, loop.

3.5. Compute the cardinality[23] $\#E'$ of the twist E' of E. If $q' = \frac{\#E'}{4}$ is not prime, loop.

3.6. If $q = p$ or $q' = p$, loop.

3.7. Compute the embedding degree m of $E(\mathbb{F}_p)$. If $m \leq \lfloor \frac{q-1}{100} \rfloor$, loop.

3.8. Compute the embedding degree m' of the twist of $E(\mathbb{F}_p)$. If $m' \leq \lfloor \frac{q-1}{100} \rfloor$, loop.

3.9. Compute the CM field discriminant D. If $|D| < 2^{100}$, loop.

4. Base point selection loop:

4.1. Generate 256 bits $b_1, b_2 \dots b_{256}$ using BBS and let

$$y = \sum_{i=1}^{256} b_i \cdot 2^{256-i}.$$

If $y = 0$ or $y = 1$, loop.

4.2. Compute $u = \frac{1-y^2}{1-dy^2}$ in $\mathbb{F}_p$. If $u^{(p-1)/2} \bmod p = -1$, loop.

4.3. Let $x = u^{(p+1)/4} \bmod p$ and let $Q = 4(x, y)$. If $Q = (0, 1)$, loop.

5. return the order p of the underlying field, the parameter d of the Edwards curve $E(\mathbb{F}_p)$, and the base point Q of prime order $\#E/4$.

6.2 Concrete Toy Example: The Two Cents Curve

We continue the example of Section 5.4 where we obtained the following BBS parameters:

$$N = 294028544707265704129885749473092814592 1, \text{ and}$$
$$s_0 = 245878848038470697812024649643837779837 7.$$

The 253 first bits generated by BBS are

[23] Note that $\#E' = 2p + 2 - \#E$.

```
0100101000100111000001010111O...1101110111001010000011100111O111
```

which yields the following prime candidate
7466609239966277892672430668762951769396815596263590253267012635990274449251 1.

This candidate being divisible by 3, we need to iterate. After 27 iterations, we obtain:
$p = 869713485409456739044342874768237225350044461628038255366624397255489408443 51.$

After finding p, the internal state of BBS is

$$s_{27\times 253} = 247142955923429920842676648454562463301 1.$$

Proceeding with the algorithm we obtain the following candidates for d:

Candidate number	d	Step failed
1	83752311210909978...1882163619	3.4
2	60262627348904122...9394767349	3.4
3	60238149156736441...3400031922	3.3
4	69441333679866912...1816186410	3.4
5	949525674280108...3073909789	3.3
⋮	⋮	⋮
10	89860366267826151...2358172684	3.2
⋮	⋮	⋮
413	9474153613400913...3119639152	3.5
⋮	⋮	⋮
3397	65281261218558381...0956875702	None

After 3396 bad candidates, the algorithm generates the following candidate:
$d = 652812612185583810075307012196552865476704696384206077194674413142309568757 02.$

The cardinality of the curve is
869713485409456739044342874768237225349672240443310199178362180056284997120 52

which is of the form $4q$, where q is the following prime:
$q = 217428371352364184761085718692059306337418060110827549794590545014071249280 13$

The cardinality of the twist is
869713485409456739044342874768237225350416682812766311554886614454693819766 52

which is of the form $4q'$, where q' is the following prime:
$q' = 217428371352364184761085718692059306337604170703191577888721653613673454941 63$

The respective embedding degrees of the curve and of the twist are
217428371352364184761085718692059306337418060110827549794590545014071249280 12
and
217428371352364184761085718692059306337604170703191577888721653613673454941 62,

which are larger than $(q-1)/100$ and $(q'-1)/100$. The CM field discriminant is
-866249770150447795810416760018913006990782176508702270912532583681224150218 51

which is larger than 2^{100}. Finally, the algorithm finds the base point $Q = (x, y)$ where x is
$x = 466149447714993660886814217570950004803811877547530729902436671614343443728 07$
and y is
$y = 836027415504548531956304940821945713611917614238771142361458056056003145997206 3.$

6.3 The Million Dollar Curve

We plan to apply the *exact* same methodology as for the *Two Cents Curve* to generate the *Million Dollar Curve*. The only differences are:

- We will use *all* the lotteries listed in Table 1 to increase resistance to collusions.
- We will use primes of 2048 bits (or more) in BBS.
- We will properly follow the methodology of Section 2.3 and thus, commit on lottery draws *before* the drawing date.

Since the seed will need to contain more than 8000 bits of entropy, we expect to require about 2 weeks of lottery draws. We plan to start gathering entropy from a time $t_3 \approx$ 1st February 2016 in order to have a curve ready by Valentine's Day.

All the elements on which both the Designer and the Standardizer commit will be published on

https://cryptoexperts.github.io/million-dollar-curve/

This includes

- the latest version of this article (in case this version is not the final one), containing an up-to-date list of Security criteria,
- Python3 code allowing to turn an ordered list of lottery draws into a seed,
- a Python3 implementation of the Filtering function,
- the exact list of lottery draws that will be considered.

After time t_3', we will publish on the same site all the elements required to generate the Cryptosystem, allowing Anybody to verify our own results. This includes the list of all the draw outcomes between t_3 and t_3', so that a verifier does not have to download all these results from the lotteries archives, thus making verification much less painful. Of course, we encourage verifiers to actively check the content of this list (we expect a careful verifier to check at least a few draws of his choice).

7 Conclusion

In this article we presented a methodology to generate an elliptic curve in a way that can be trusted by the end-users. Each step of the generation is either justified by security or efficiency constraints, or randomized by the use of a non-manipulable entropy source, which in our case are national lotteries. In the end, introducing a trap in the curve itself requires to manipulate the draws of several independent lotteries around the world, which we assume is hard, even for the most powerful adversaries.

Of course, the same methodology could be used to generate curves with different properties (for example, curves suitable for pairings) by simply changing the Security criteria we use (see e.g. [20]), or to generate any sort of cryptographic parameters or constants in which it could otherwise be possible to introduce a

trap (e.g., symmetric cryptography round constants). The only requirement is to have a well defined set of Security criteria such that a random instance fulfilling these criteria is secure with respect to the current state of the art attacks.

Even though this paper will serve as a basis to generate the parameters of the Million Dollar Curve, its first purpose is to gather comments from the cryptographic community about the methodology itself. Our objective is for the Million Dollar Curve to inspire confidence to the greatest number. In that sense, we want to integrate comments we receive in the methodology before committing on the methodology and the Publicly verifiable RNG we will use. You can direct any comment you have to curves@cryptoexperts.com and we will gladly listen.

Once all the comments are integrated and the Security criteria are updated, we will commit on the methodology and the Publicly verifiable RNG (the exact list of lotteries and draw dates) we will use to generate the Million Dollar Curve. All the scripts and source code required to generate or verify the parameters will be made available on the official website:

https://cryptoexperts.github.io/million-dollar-curve/

Acknowledgments

The authors would like to thank Antoine Joux for his invaluable comments on previous versions of this work. We also thank Joseph Bonneau and Ivan Zuboff for their diligent comments.

Commitments for the Million Dollar Curve "MDCurve201601"

- **January 27, 2016**, commitment on the design of MDCurve201601:
  - the design is described in

    https://cryptoexperts.github.io/million-dollar-curve/specifications/mdcurve_201601/2016_01_27_million_dollar_curve.txt;
  - the SHA-256 sum of the file is

`e9dd4baf0d351b5a64c59ed6b1efd3108094b3585e17a0e5350fb200500058d9`.
- **January 29, 2016**, commitment on the seeding of MDCurve201601:
  - the seeding is described in

    https://cryptoexperts.github.io/million-dollar-curve/specifications/mdcurve_201601/2016_01_29_million_dollar_curve_seeding.txt;
  - the SHA-256 sum of the file is

`f8bdb5bd4957a2d65b567378bb32744d0d0573a77e4ef0247311a5a4b98744da`.

References

1. ECC Brainpool Arbeitsgruppe. ECC Brainpool Standard Curves and Curve Generation, October, 19 2005. http://www.ecc-brainpool.org/download/Domain-parameters.pdf.
2. Jean-Philippe Aumasson. Backdoors up my Sleeve. 8th issue of PoC||GTFO, June 2015. Source code available at https://github.com/veorq/numsgen.
3. The Beatles. With a Little Help from My Friends, 1967. https://en.wikipedia.org/wiki/With_a_Little_Help_from_My_Friends.
4. Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records, February 2006. http://cr.yp.to/ecdh.html.
5. Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer Berlin Heidelberg, 2006.
6. Daniel J. Bernstein. Elliptic vs. Hyperelliptic, Part I, September 2006. http://cr.yp.to/talks.html#2006.09.20.
7. Daniel J. Bernstein, Tung Chou, Chitchanok Chuengsatiansup, Andreas Hülsing, Eran Lambooij, Tanja Lange, Ruben Niederhagen, and Christine van Vredendaal. How to manipulate curve standards: a white paper for the black hat. IACR - Cryptology ePrint Archive, 2014. https://eprint.iacr.org/2014/571.
8. Daniel J. Bernstein and Tanja Lange. SafeCurves: choosing safe curves for elliptic-curve cryptography. Retrieved on the 6th of November, 2015. http://safecurves.cr.yp.to/.
9. Daniel J. Bernstein and Tanja Lange. Faster Addition and Doubling on Elliptic Curves. In Kaoru Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 29–50. Springer Berlin Heidelberg, 2007.
10. Daniel J. Bernstein, Tanja Lange, and Ruben Niederhagen. Dual EC: A Standardized Back Door. IACR - Cryptology ePrint Archive, July 2015. https://eprint.iacr.org/2015/767.
11. Lenore Blum, Manuel Blum, and Mike Shub. Comparison of Two Pseudo-Random Number Generators. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *Advances in Cryptology: Proceedings of CRYPTO '82*, pages 61–78, New York, 1983. Plenum Press.
12. Lenore Blum, Manuel Blum, and Mike Shub. A Simple Unpredictable Pseudo-Random Number Generator. *SIAM J. Comput.*, 15(2):364–383, May 1986.
13. Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications (extended abstract). In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 103–112. ACM, 1988.
14. Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. On Bitcoin as a public randomness source. IACR - Cryptology ePrint Archive, December 2015. https://eprint.iacr.org/2015/1015.
15. Jeremy Clark and Urs Hengartner. On the Use of Financial Data as a Random Beacon. In Douglas W. Jones, Jean-Jacques Quisquater, and Eric Rescorla, editors, *Electronic Voting Technology Workshop / Workshop on Trustworthy Elections, EVT/WOTE '10*, August 2010.
16. Arel Cordero, David Wagner, and David Dill. The role of dice in election audits. In *IAVoSS Workshop On Trustworthy Elections (WOTE 2006)*. USENIX Association, June 2006.

17. Craig Costello and Patrick Longa. Four$\mathbb{Q}$: Four-Dimensional Decompositions on a $\mathbb{Q}$-curve over the Mersenne Prime. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology - ASIACRYPT 2015, Part I*, volume 9452 of *Lecture Notes in Computer Science*, pages 214–235. Springer Berlin Heidelberg, 2015.
18. Clint Eastwood. Million Dollar Baby, 2004. http://www.imdb.com/title/tt0405159/.
19. Harold M. Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, 44:393–422, 2007.
20. Jean-Pierre Flori, Jérôme Plût, Jean-René Reinhard, and Martin Ekerå. Diversity and Transparency for ECC. NIST ECC Workshop 2015, July 2015. https://eprint.iacr.org/2015/659.
21. Steven D. Galbraith, Ping Wang, and Fangguo Zhang. Computing Elliptic Curve Discrete Logarithms with Improved Baby-step Giant-step Algorithm. IACR - Cryptology ePrint Archive, June 2015. https://eprint.iacr.org/2015/605.
22. Kristian Gjøsteen. Comments on Dual-EC-DRBG/NIST SP 800-90, Draft December 2005, March 16 2006. http://www.math.ntnu.no/~kristiag/drafts/dual-ec-drbg-comments.pdf.
23. Mike Hamburg. Any interest in random curves? Curves Mailing List, June 2014. https://moderncrypto.org/mail-archive/curves/2014/000216.html.
24. IETF. Publicly Verifiable Nomcom Random Selection, February 2000. RFC 2777.
25. Antoine Joux and Vanessa Vitse. Cover and Decomposition Index Calculus on Elliptic Curves Made Practical - Application to a Previously Unreachable Curve over $\mathbb{F}_{p^6}$. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology - EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 9–26. Springer Berlin Heidelberg, 2012.
26. Marc Joye, Pascal Paillier, and Serge Vaudenay. Efficient Generation of Prime Numbers. In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 340–354. Springer Berlin Heidelberg, 2000.
27. Donald E. Knuth. *The Art of Computer Programming*, volume 4A - Combinatorial Algorithms - Part 1. Addison Wesley, 2011. Fourth printing, August 2013.
28. Arjen K. Lenstra and Benjamin Wesolowski. A random zoo: sloth, unicorn, and trx. IACR - Cryptology ePrint Archive, April 2015. https://eprint.iacr.org/2015/366.
29. National Institute of Standards and Technology. Special Publication 800-90: Recommendation for random number generation using deterministic random bit generators, January 2012. First version June 2006, second version March 2007, http://csrc.nist.gov/publications/PubsSPs.html#800-90A.
30. Monty Python. Monty Python's Flying Circus, 1969 to 1974.
31. Queen. I Want to Break Free, 1984. https://en.wikipedia.org/wiki/I_Want_to_Break_Free.
32. Michael O. Rabin. Transaction Protection by Beacons. *Journal of Computer and System Sciences*, 27(2):256–267, 1983.
33. Lou Reed. Walk on the Wild Side, 1972.
34. Phillip Rogaway. The Moral Character of Cryptographic Work. IACR - Cryptology ePrint Archive, December 2015. https://eprint.iacr.org/2015/1162.
35. Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. Non-interactive zero-knowledge proof systems. In Carl Pomerance, editor, *Advances in Cryptology - CRYPTO '87*, volume 293 of *Lecture Notes in Computer Science*, pages 52–72. Springer, 1987.

36. Bruce Schneier. The NSA Is Breaking Most Encryption on the Internet, September 5 2013. See the comment posted on September 5, 2013 4:07 PM, https://www.schneier.com/blog/archives/2013/09/the_nsa_is_brea.html#c1675929.
37. Berry Schoenmakers and Andrey Sidorenko. Cryptanalysis of the Dual Elliptic Curve Pseudorandom Generator. IACR - Cryptology ePrint Archive, May 2006. https://eprint.iacr.org/2006/190.
38. Dan Shumow and Niels Ferguson. On the Possibility of a Back Door in the NIST SP800-90 Dual Ec Prng. CRYPTO 2007 Rump Session, August 2007. http://rump2007.cr.yp.to/15-shumow.pdf.
39. Cat Stevens. Wild World, 1970.
40. BADA55 Research Team. Brainpool curves. 2015.09.27 version of http://bada55.cr.yp.to/brainpool.html.
41. Wikipedia. Mixed-radix, December 2015. https://en.wikipedia.org/wiki/Mixed_radix.
42. Wikipedia. Nothing up my sleeve number, December 2015. https://en.wikipedia.org/wiki/Nothing_up_my_sleeve_number.
43. Accredited Standards Committee X9. American National Standard X9.62-1998, Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), 1998. Working Draft.

A Compressing Points on Edwards Curves

Theorem 1 (Adapted from Theorem 2.1 of [5]). *Let p be a prime number such that $p \geq 5$. Let $d \in \mathbb{F}_p$ such that d is not a square in $\mathbb{F}_p$. Let $E(K)$ be the elliptic curve defined by*

$$x^2 + y^2 = 1 + dx^2y^2$$

over a field $K \in \{\mathbb{F}_p, \mathbb{F}_{p^2}\}$. Define the projection

$$\begin{aligned} Y_0 \colon E(\mathbb{F}_{p^2}) &\longrightarrow \mathbb{F}_{p^2} \\ (x, y) &\longmapsto y. \end{aligned}$$

For any $q \in \mathbb{F}_p$ there are at most two distinct points Q_1 and Q_2 on $E(\mathbb{F}_{p^2})$ such that $Y_0(Q_1) = Y_0(Q_2) = q$. Moreover, for any integer n we have $Y_0(nQ_1) = Y_0(nQ_2)$, and this value lies in $\mathbb{F}_p$.

Proof. Let $d, q \in \mathbb{F}_p$ where d is not a square in $\mathbb{F}_p$. Note that necessarily $dq^2 - 1 \neq 0$: it trivially holds for $q = 0$, and when $q \neq 0$, an equality would imply that $d = (1/q)^2$ is a square.

Next, consider the equation $\alpha + q^2 = 1 + d\alpha q^2$ in $\mathbb{F}_p$. Since $dq^2 - 1 \neq 0$, the latter equation admits only one solution in $\mathbb{F}_p$, that is

$$\alpha = \frac{q^2 - 1}{dq^2 - 1}.$$

– Case 1: α is a square in $\mathbb{F}_p$. Let $r, -r \in \mathbb{F}_p$ be the two square roots of α. Since α is the only solution of $\alpha + q^2 = 1 + d\alpha q^2$ and since $\pm r$ are the only square roots of α, then $\pm r$ are the only solutions of $x^2 + q^2 = 1 + dx^2q^2$, and

$$\{Q \in E(\mathbb{F}_{p^2}) \ : \ Y_0(Q) = q\} = \{(r, q), (-r, q)\}.$$

Moreover, for any integer n,

$$Y_0(n(-r,q)) = Y_0(n(-(r,q))) = Y_0(-(n(r,q))) = Y_0(n(r,q)).$$

Finally, since $(r,q) \in E(\mathbb{F}_p)$, then $n(r,q) \in E(\mathbb{F}_p)$ and $Y_0(n(r,q)) \in \mathbb{F}_p$.

– Case 2: α is not a square in $\mathbb{F}_p$. Let $\delta \in \mathbb{F}_p$ be any non-square element and consider $\mathbb{F}_{p^2}$ as $\mathbb{F}_p[\sqrt{\delta}]$. Since α/δ is a square in $\mathbb{F}_p$, the two roots of α can be denoted $r\sqrt{\delta}$ and $-r\sqrt{\delta}$. Since α is the only solution of $\alpha + q^2 = 1 + d\alpha q^2$ and since $\pm r\sqrt{\delta} \in \mathbb{F}_{p^2}$ are the only square roots of α, then $\pm r\sqrt{\delta} \in \mathbb{F}_{p^2}$ are the only solutions of $x^2 + q^2 = 1 + dx^2q^2$, and

$$\{Q \in E(\mathbb{F}_{p^2}) \ : \ Y_0(Q) = q\} = \{(r\sqrt{\delta}, q), (-r\sqrt{\delta}, q)\}.$$

Moreover, for any integer n,

$$Y_0(n(-r\sqrt{\delta}, q)) = Y_0(n(-(r\sqrt{\delta}, q))) = Y_0(-(n(r\sqrt{\delta}, q))) = Y_0(n(r\sqrt{\delta}, q)).$$

Finally, it is easy to see that $\sqrt{\delta}\mathbb{F}_p \times \mathbb{F}_p$ with the Edwards addition law is a subgroup of $E(\mathbb{F}_{p^2})$. Therefore, $(r\sqrt{\delta}, q) \in \sqrt{\delta}\mathbb{F}_p \times \mathbb{F}_p$ implies that $n(r\sqrt{\delta}, q) \in \sqrt{\delta}\mathbb{F}_p \times \mathbb{F}_p$, an thus that $Y_0(n(r\sqrt{\delta}, q)) \in \mathbb{F}_p$. □

B Frequently Asked Questions

Q1. Is Million Dollar Curve nothing more than "yet another safe curve"?

Yes and no.

The Million Dollar Curve in itself is just another safe curve, but it is generated using the methodology proposed in this paper. Note that this methodology allows to generate parameters everyone can trust for new cryptosystem standards, but also in any other situation requiring verifiable randomness.

Q2. Is there anything wrong with Curve25519?

No.

We, at CryptoExperts, actually use Curve25519 and recommend it to our partners. Yet, we think that people should not rely on the same few safe curves that are currently out. Our methodology allows to easily produce safe alternatives.

Q3. Curve25519 vs. Million Dollar Curve

Curve25519 was designed to be as fast as possible, with no security compromise. This is both a strength and a potential weakness:

– a strength because it gives a valid argument that no trapdoor was introduced in the design,
– a potential weakness because Curve25519 uses a very specific prime field. As of now, no attack exploiting this specificity is known.

For applications where speed is paramount, Curve25519 is probably the best option. But for most applications, where losing a little on the efficiency side is "not a big deal", Million Dollar Curve is probably the safest choice.

See also the answer by Ruggero on Stack Exchange.

Q4. What if a government agency rigs a few lotteries?

This is not an issue.

Our parameter generation process was specifically designed to avoid this kind of problem. As explained in Section 4.4, in order to manipulate the parameter generation, an adversary has to rig all the last lottery draws. We mitigate this risk by relying on independent lotteries from various countries around the world.

Q5. Do you really think that national lotteries are unimpeachable?

Yes and no.

Most national lotteries follow a strict protocol which is specifically designed to avoid manipulations (certified drawing machines, supervision of a bailiff, legal process, etc.). It is pretty obvious that in the past some lotteries have been manipulated, but manipulations with financial motive are not a concern for us (at most, a few bits of entropy are lost). Besides, as explained in the previous answer, an adversary would have to manipulate all the last lottery draws.

Q6. How can I verify the parameters if a lottery archive is no longer available?

These archives are usually duplicated in several places over the Internet or sometimes printed in newspapers. If at a given time in the future the result of a drawing becomes completely unavailable, verification will become impossible.

However, the lotteries that have been selected are well established and will most certainly continue to exist for some time. During this period, everyone will be able to fully verify/re-generate the Million Dollar Curve, which in itself is a convincing argument for later use.

Q7. I have never heard about half of these lotteries, how do I even know they are real?

You can probably verify that lotteries from your own country actually exist: you might even have played them before hearing about Million Dollar Curve. Your friends in Mauritius, Canada, and New Zealand can certainly confirm the same thing.

Besides, lotteries are such popular games that "faking" one in a convincing way is probably impossible without being noticed at some point.

Q8. What if a lottery does not perform the expected draws during Phase IV of the generation process?

This situation is already taken into account in our generation process and the scripts we provide, so this is not a problem. The missing draws are simply ignored.