Using Cryptol to explore elliptic curve-based random-number generators

MFES — Formal Methods for Software Engineering Cohesive Project

Ana Paula Carvalho[1] and Catarina Correia[2]

[1] Minho University, Portugal
`pg25335@alunos.uminho.pt`
[2] Minho University, Portugal
`pg19643@alunos.uminho.pt`

Abstract. Internet security is an issue that recent news have given special focus to. To tackle how this works behind the curtain, this project aims at exploring the Dual_EC_DRNG, an algorithm standardized by NIST as a cryptographically secure pseudo random number generator. This algorithm is known to be insecure, which means that it is possible to recover its internal state.
The task at hand will be performed with Cryptol, a language developed by Galois for cryptographic purposes. In order to provide high assurance programming, Cryptol allows the verification of properties using SMT solvers and automated random testing.

1 Introduction

Modern cryptography stands on the idea that keys used to encrypt data can be made public while keys used to decrypt them should be kept private. Although different, the two parts of this key pair are mathematically linked. Public-key algorithms rely on the mathematics of integer factorization, discrete logarithm problem and elliptic curves, problems that currently admit no efficient solution in some domains. The strength of the public-key algorithms lies in the fact that it is computationally infeasible for a proper generated private key to be determined from its corresponding public key. Algorithms having this characteristic – *easy* in one direction, *hard* in the other – are known as *trapdoor functions*.

The most well known algorithms based on this premise are the RSA algorithm (based on prime factorization) and the Diffie-Hellman key-exchange (based on the discrete logarithm problem) introduced in 1976 [1]. In 1985, an approach to public-key cryptography based on the algebraic structure of *elliptic curves* [2] was proposed. These systems were not widely used until the early 2000s.

Around 2006, the American *National Institute of Standards and Technology* (NIST) published a specification for four cryptographically secure pseudo-random number generators (or deterministic random bit generators, DRBG), under the code *NIST SP 800-90* [3]. Such four algorithms present distinctive features: *Hash DRBG*, based on hash functions; *HMAC DRBG*, hash-based message authentication code; *CTR DRBG*, based on block ciphers and the *Dual Elliptic Curve DRBG* (Dual_EC_DRBG), founded on the *elliptic curve discrete logarithm problem* (ECDLP).

The main advantage of elliptic curves is that of providing smaller keys for the same levels of security. To compare the costs of breaking the cryptographic primitives presented before, there is the concept of *Global Security*, introduced recently[4]. The authors compute how much energy is needed to break a cryptographic algorithm and compare that with how much water the same energy could boil. By this measure, breaking a 228-bit RSA key requires less energy than it takes to boil a teaspoon of water. Comparatively, breaking a 228-bit EC key requires enough energy to boil all the water on earth. For this level of security with RSA one would need a key with 2,380 bits.

Capitalizing on these improvements, several discrete logarithm-based protocols have been adapted to elliptic curves. In [5] there is a list of companies that use validated instances of algorithms in SP 800-90. The bad news is, after NIST released the standard, some studies were published stating that the algorithm has a backdoor [6]. More recently, news about memos of the former NSA Edward Snowden disclose that the backdoor might have been introduced by the NSA themselves [7]. Later advising against the use of such algorithm, only two months ago NIST decided to remove the algorithm from the recommendation [8].

This report aims at using the Cryptol language to implement elliptic curve-based random number generators with a backdoor (imitating NIST's Dual_EC_DRNG) and to show how the internal state can be recovered after observing a few bits of output. This goal was extended to cover proofs about some fundamental properties of elliptic curves arithmetic with the Cryptol toolset.

Contribution. This work resulted in: a Cryptol implementation of the Dual_EC_DRNG for multiple curves as well as tests on the arithmetic; computation of the backdoor and guidance on how to recover the internal state; verification proofs about elliptic curve properties over small fields (such as the addition law); and report of running issues with the development version of Cryptol.

Report structure. The remainder of this report is structured as follows: section 2 introduces the theory of elliptic curves; section 3 analyzes how the Dual_EC_DRNG operates as well as the mathematics of backtracking; section 4 explores a way to tackle the issue with Cryptol; section 5 aims at providing assurance about the implementation; and section 6 discusses the findings and proposes follow-up work.

2 Elliptic Curves

An elliptic curve (**EC**) is the set of points (x, y) that satisfy the *Weierstrass equation*, of the form:

$$y^2 = x^3 + ax + b \tag{1}$$

It is useful to add a *point at infinity* to this set of points, denoted by $\mathcal{O}$ [9]. The *infinity point* does not exist in the XY-plane: a line intersects $\mathcal{O}$ iff it is a vertical line. An EC is non-singular, which geometrically means that the graph has no cusps, self-intersections or isolated points (see Fig. 1). Algebraically, it expresses that a and b satisfy the condition (2), thus not allowing the curve to have multiple roots.

$$-16(4a^3 + 27b^2) \neq 0 \tag{2}$$

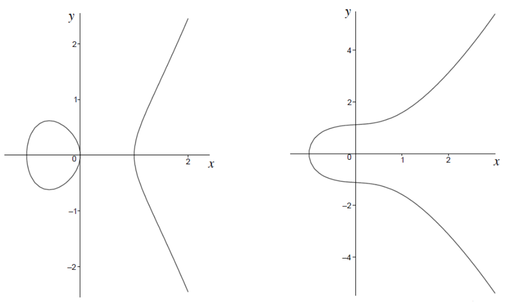

Fig. 1: Examples of EC basic forms. On the left is the graph of the equation $y^2 = x^3x$ and on the right the equation $y^2 = x^3 + \frac{1}{4}x + \frac{5}{4}$ both over $\mathbb{R}$. Source: [10].

ECs can defined over real numbers $\mathbb{R}$ (see Fig. 1), complex numbers $\mathbb{C}$, rational numbers $\mathbb{Q}$, finite fields over a prime p $\mathbb{F}_p(= \mathbb{Z}_p)$ or finite fields $\mathbb{F}_q$, where $q = p^k$ with $k \geq 1$.

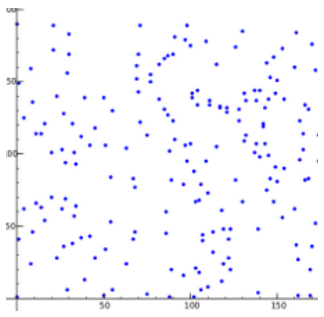

Fig. 2: Example of a elliptic curve over a finite field. Source: [11].

2.1 Group Law

Let E be an elliptic curve defined over the field $\mathbb{K}$. A particular feature of elliptic curves is that an addition operation can be defined between its points, denoted by $P \oplus R$, with P, R points over the EC. This addition operation is not the common addition. On the other hand, it combines two points producing a third one in a manner analogous to addition that will be made clear in a moment.

The most natural way to describe the addition law on elliptic curves is to use geometric representation. Let $P, Q, R \in E$ be three distinct points $P = (x_1, y_1)$, $Q = (x_2, y_2)$ and $R = (x_3, y_3)$ on an elliptic curve E. R is said to be the sum of P and Q, written $R = P \oplus Q$, if it fits in the following geometric construction. First, a line through P and Q is drawn. It is a characteristic of EC that every line through two points on an EC intersects a third one on the same EC. Then R is chosen to be the reflection of this point relative to the X-axis (see the left hand side of Fig. 3). Doubling a point P is the addition of P to itself, $P \oplus P = R$. This "limit situation" corresponds to drawing a tangent line to the curve on P (considered to intersect the point P twice) and proceed as before[3].

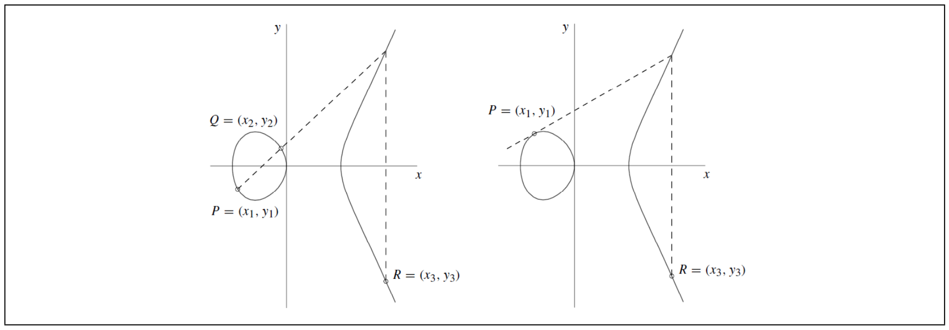

Fig. 3: Geometric addition (on the left) of two different points $P, Q \in E$ ($P \oplus Q = R, R \in E$,) and doubling (on the right) of $P \in E$ ($P \oplus P = R, R \in E$). Source: [10].

A potential problem with this addition law arises if a point is added to its reflection about the X-axis, its symmetric point $P' = (x_1, -y_1)$. If a line is drawn between this two points the result will be a vertical line, so it intersects E in only two points. Without a third point to complete the sum, it was necessary to assume that this line goes through the point $\mathcal{O}$, which makes $P \oplus P' = \mathcal{O}$.

The next step is to figure out how to add $\mathcal{O}$ to another point P over the EC. As previously agreed, the line that connects P and $\mathcal{O}$ is a vertical line since $\mathcal{O}$ can be seen as the point that lies on all vertical lines. Consequently, the vertical line drawn intersects the three points $\mathcal{O}$, P and P'. Therefore, the third point intersected by the line is P'. Its reflection is P itself so $P \oplus \mathcal{O} = P$.

[3] This line will intersect the elliptic curve on another point and R will be its X-reflection, see Fig. 3 on the right hand side.

Theorem 1. *Properties of the addition law on an elliptic curve E:*

- $P \oplus \mathcal{O} = \mathcal{O} \oplus P = P, \forall P \in E$ *[Identity]*
- $P \oplus (-P) = \mathcal{O}, \forall P \in E$ *[Inverse]*
- $(P \oplus Q) \oplus R = P \oplus (Q \oplus R), \forall P, Q, R \in E$ *[Associative]*
- $P \oplus Q = Q \oplus P, \forall P, Q \in E$ *[Commutative]*

This makes the addition law on the elliptic curve into an *abelian group.* Note that it is possible to define the scalar multiplication as the addition of a point n times to itself in a way that $R = n * P$.

Throughout this report, acronym "EC" will implicitly be assumed to denote an elliptic curve defined over a finite field $\mathbb{F}_p$ for a prime p.

2.2 Elliptic Curve Cryptography

The first practical public-key cryptosystem was the famous RSA cryptosystem that bases its security on the difficulty of factoring large numbers. Later on, Diffie and Hellman described a key-exchange algorithm whose security relies on the discrete logarithm problem (DLP) in a finite field $\mathbb{F}_p$ followed by ElGamal that created a public key cryptosystem based on the same underlying problem. Koblitz [12] and Miller [13] suggested the use of an EC instead of the finite field $\mathbb{F}_p$, under the assumption that the discrete logarithm problem in the elliptic curve group (ELDCP) $E(\mathbb{F}_p)$ is harder to solve than the discrete logarithm problem in the multiplicative group. Thus, the elliptic curves arithmetic can be applied in well known algorithms like DiffieHellman key exchange, which involves no more than replacing the DLP for the finite field $\mathbb{F}_p$ with the DLP for an EC and like the ElGamal public key cryptosystem.

Public key cryptosystems Public key cryptosystems rely on what are known as one-way trapdoor functions. These functions have the feature of being easily computable while its inverse is very hard to compute, unless one possesses extra important information to solve it. It is not clear that one-way trapdoor functions exist and it is still an open problem to prove their existence [14]. Yet, a number of hard mathematical problems have been proposed for one-way trapdoor functions, including the discrete logarithm problem.

Discrete logarithm problems. A discrete logarithm is an integer k solving the equation $b^k = g$, where b and g are elements of a group. The solution of discrete logarithm problems (DLP) in different groups may exhibit different levels of difficulty [15]. The DLP in $\mathbb{F}_p$ with addition has a linear-time solution, while the best known general algorithm to solve the DLP in $\mathbb{F}_p^*$ with multiplication is sub-exponential.

The elliptic curve discrete logarithm problem (ECDLP). The DLP for elliptic curves is believed to be even more difficult than that for $\mathbb{F}_p^*$. In particular, if the EC group is chosen carefully and has N elements, then the best known algorithm to solve the DLP requires $\mathcal{O}(\sqrt{N})$ steps. Thus it currently takes exponential time to solve the ECDLP.

Coordinate Systems A system can be represented with respect to several coordinate systems., e.g., affine or projective coordinates. Equation 1 expresses the curve in *affine coordinates*. In the projective system, a curve can be expressed in Jacobian coordinates as 3. The point (x1 : y1 : z1) on E corresponds to the affine point $(x1/z1^2\ , y1/z1^3\ )$ when $z1 \neq 0$ and to the point at infinity $P = (1 : 1 : 0)$. The symmetric of $(x1 : y1 : z1)$ is $(x1 : -y1 : z1)$.

$$y^2 = x^3 + axz^4 + bz^6 \tag{3}$$

3 Building and breaking NIST's Dual_EC_DRNG

The Dual_EC_DRNG is based on the ECDLP described earlier: given points P and Q on an elliptic curve of order n, find a point Q such that $P = e * Q$ [3]. The algorithm uses a seed to initiate the generation of pseudorandom strings by performing scalar multiplications on two points in an EC group. In [3], NIST also claims that backtracking resistance is inherent in the algorithm even if the internal state is compromised, i.e., the knowledge of a seed s_{i+1} does not allow an adversary to determine s_i. It is also stated that inverting the direction of the line implies solving the ECDLP for that specific curve.

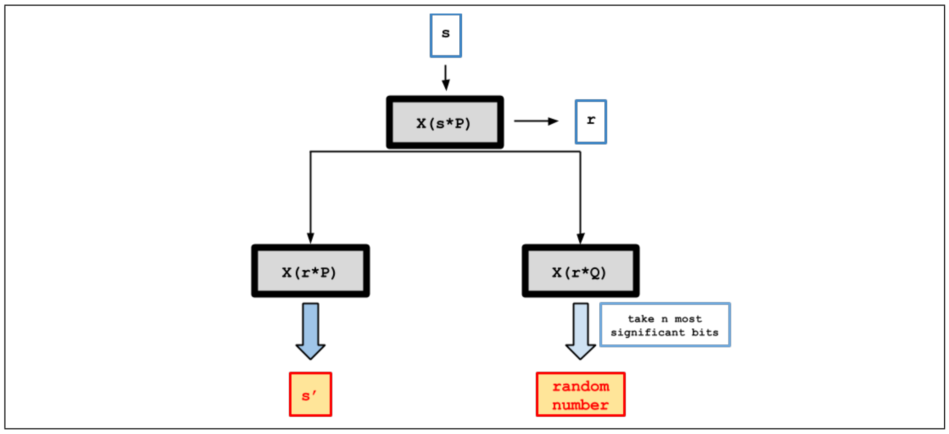

Fig. 4: Dual_EC_DRNG workflow. The function X extracts the x-coordinate of the point. $*$ represents the elliptic curve scalar multiplication. The first step is to multiply the point **P** by the initial seed **s**. Extracting the x-coordinate will result in **r**. This **r** is going to be used in two ways : to generate the seed s' for the next iteration (left branch) and to reproduce the random number (right branch). Note that some high order bits are extracted from the output number, so the length of the generated number is less than the seed length.

The Dual_EC_DRNG is initialised with two points (P and Q) on some EC and a random integer seed s. The NIST standard suggests the use of certain points, allegedly random and secure, but no explanation is given on how they were obtained. The seed represents the hidden state of the algorithm, therefore it shall be kept secret at all times.

$$\begin{gathered} r = X(s * P) \\ s' = X(r * P) \\ full\_nr = X(r * Q) \\ random\_nr = extract\_bits(full\_nr) \end{gathered}$$

In short, the algorithm comprises the generation of a random point in the curve $(r*Q)$, followed by the extraction of high order bits from its X-coordinate. Refer to Fig. 4 for a simplified illustration[4]. The authors of [16] prove that the raw point $r * Q$ generated is indeed random. However, the second part (the bit extraction) is not cryptographically sound. In fact, there is evidence that breaking this second part can be done with an ordinary computer [17]. From now on, $r * Q = A$.

3.1 The backdoor

Being the discovery of [16] and [17] an issue by itself, a bigger one was meanwhile found by two Microsoft researchers[6]. They state that, if we know the point A generated (and we can do so via brute force, with at least 0.5 probability[17]) and we know the relationship between P and Q then we can find out the seed for the next iteration and uncover the next number in the random stream.

The relation between Q and P is supposed to be a secret. However, these points were suggested by the standard and no explanation was given about how they were obtained. One cannot be sure whether or not the designer of the algorithm knew some relationship between them (if any).

Suppose the designer knows such relationship. It is known that the new seed is the $X-$coordinate of $r * P$. Replacing P with the relation $P = e * Q$ we get $r * e * Q$; finally applying commutativity, we get $e * r * Q$, which, after finding A, is easy to compute.

$$r * P = r * (e * Q) = e * A \tag{4}$$

In summary, we find all possibilities for A which are 2^n (with n the number of bits extracted), multiply A e-times by itself and the resulting points' X-coordinates are all the options for the following seed. Once we know what the next seed can be, we try them all with the algorithm and rule out the ones that do not yield the next number generated. Experiments made in [6] estimate that only 256 bits are needed to exactly determine the seed in a $p - 256$ curve[5].

The next section illustrates this process with a small curve.

[4] Note that there are more details to this algorithm than those explained. This project attempts at specifying and explaining the basic mechanisms and therefore only the most important features are considered.

[5] NIST routines are also labeled by their bit sizes, so the $p - 256$ curve means that the prime that defines the field is 256 bits wide.

3.2 Toy-Example (1 block)

Let $y^2 = x^3 - 3x + 7$ [6] be a curve defined over $\mathbb{F}_{23}^*$ and $P = (22, 3)$, $Q = (14, 15)$ be two points on that curve. We also need an input seed and to know the secret, $P = 5 * Q$.

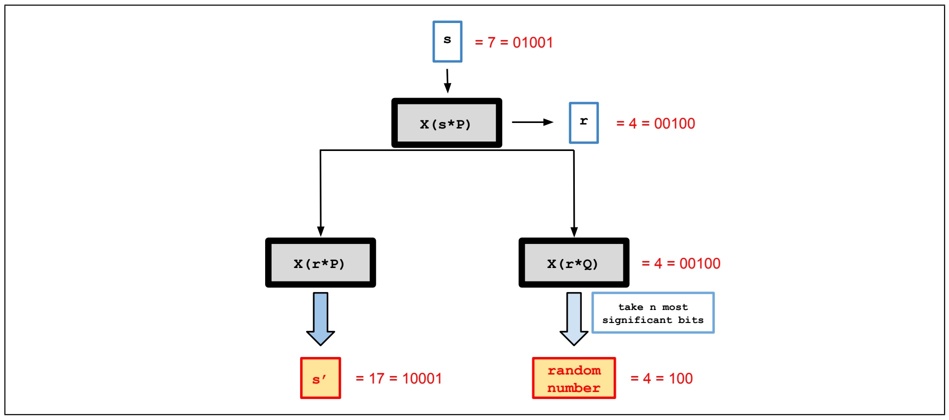

Fig. 5: As this is computed over field 23 (5 bits), every number will be represented with 5 bits and take the 2 most significant bits. $4_{10} = 00100_2$ taking the 2 leftmost bits we still get 4 so the number returned is 4, but with only 3 bits.

Figure 5 illustrates the generation of the random number, which in this case is 4. Following is a step by step explanation of the backdoor, using Figure 6 as reference.

Step 1: This is were the algorithm is firstly insecure. This means that the output number can be efficiently distinguished from the sequence of uniformly distributed random bits[17]. Moreover, the authors of [17] add that we can guess that number with 0.5 probability and this can be increased by increasing the size of the bit stream. After getting the number 4, it is possible to compute all possible values for the full number, knowing it droped the two most significant bits. From the four options, we can remove two of them: one is over 23 thus it is not on the field; and the other is not a conceivable value for a X-coordinate on the curve.

Step 2: In this step is computed the $Y-$coordinate that corresponds to some x in the curve, by square root calculation.

Step 3: This step is were the backdoor relies on. The relationship between P and Q should be kept secret. However, if it is known, we can multiply the A values by this relation, which is 5 in this example. This allows us to go from the random number calculation to the seed calculation, which should be impossible.

[6] This curve was chosen bearing in mind the elliptic curve properties and implementation details. See Appendix **??** for the table of point additions. The curve points and point additions were computed using the implementation of EC arithmetic in Cryptol (see section 4).

Step 4: Here we just take the $X-$coordinate for the next seed.

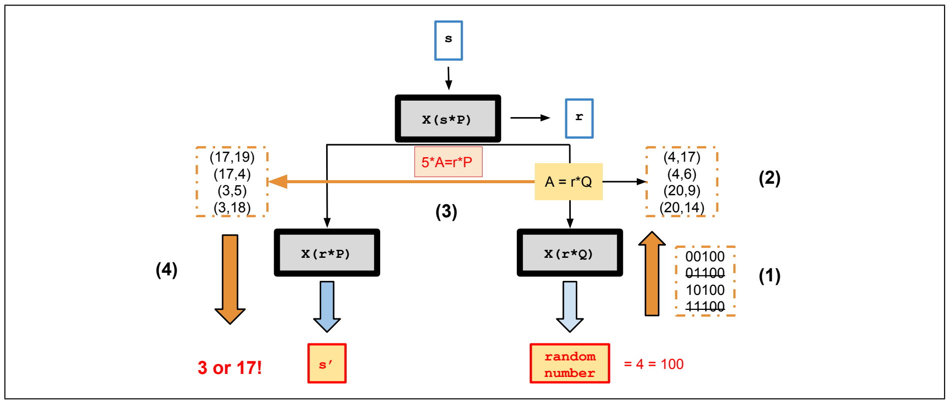

Fig. 6: Step by step calculation of the backdoor.

Extension for two blocks Let us say that a stream of two blocks and the number $34_{10} = 100010_2$ is generated. It is easy to see that one can divide the number, take the first half and compute all the steps that we did before. After that, it is known that the seed that generated the other half is either 17 or 3. Next, one run the algorithm with both seeds. One of them will generate the second half of the stream and the internal state is disclosed, because now it is possible to predict the following "random" numbers on the stream.

This experiment produces probability values similar to the ones in [17]. It started with four possible points and ended up with only two options for the next seed. Increasing the number of blocks also lowered the options and for this curve it was enough to be 100% sure.

4 Implementation

The language used Cryptol version 2.0, which has just launched. Cryptol is a domain-specific language for cryptographic algorithms.

The implementation comprises three layers of complexity: arithmetic in finite fields, arithmetic in the elliptic curve group and the building and breaking of the Dual_EC_DRNG. Most of the finite field and elliptic curve arithmetic were already implemented for a previous project (in a previous version of Cryptol). To complete the required arithmetic toolset, there were some operations left to be implemented such as scalar multiplication in the elliptic curves group, exponentiation and modulo square root in the finite fields. A study on the implementation choices was also conducted.

The most economic coordinate system to compute elliptic curve arithmetic is the projective system presented in section 2. Every NIST curve has the property that $a = -3$ for optimal security and efficiency [15]. As a result, all the routines are implemented from [18] bearing this two issues in mind.

4.1 Finite Field and Elliptic Curve Arithmetic

Scalar multiplication. Considering that it is the operation in focus on the Dual_EC_DRNG (see section 3), scalar multiplication is of utter importance. The algorithm was written using Montgomery's ladder, according to recommendation from [19], Algorithm 13.35. At each step, one addition and one doubling are performed, which makes this method interesting for side-channel attacks.

Modulo square root. The modulo square root function was implemented specifically for a p-256 field, as described in [18] Routine 3.2.10. This operation makes use of exponentiation which was also implemented as suggested in [20].

Other extra functions were defined but, considering they are not as important, they will be referred to when needed in the following subsection. All the operations are polymorphic over their bit size.

Testing The routines described in [18] come with calculation examples of arithmetic operations for various curve sizes. The examples of curves $p - 256$ and $p - 192$ were used in order to test addition, subtraction, doubling and scalar multiplication.

4.2 Reproducing the Dual_EC_DRNG

```
dualec256: CurveOps [256] [256] -> [256]
        -> ProjectivePoint [256] -> ProjectivePoint [256]
        -> InternalState [240] [256]

dualec256 c s p q = {rnr=join( drop`{16} xs), seed=s1}
        where
        xs= (split t:[256][1]Bit)
        t=affx(smul r q)
        r=affx(smul s p)
        s1= affx (smul r p)
        smul=ec_smul1 c
        affx = ec_affx c.field
```

Listing 1.1. Cryptol implementation of the Dual_EC_DRNG for a curve defined over a field of a prime 256-bit wide. *smul* is the scalar multiplication function, *affx* transforms to affine and takes the x coordinate.

The code in 1.1 takes the least significant 16 bits and returns the other 240 bits. For a more algorithmic explanation, see section 3.

```
p256 = ((2^^256)-(2^^224)+ (2^^192)+(2^^96)-1):[256]

pf256 = prime_field p256

b256= 0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b

a = 3:[256]

q256= 0xffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc632551

pg256=prime_field q256

curve256 = mk_curve(pf256,pg256,b256,a)
```

Listing 1.2. Definition of a NIST curve P-256[18]. *p256* is the prime field order; *q256* is the order of the elliptic curve group; *curve256* is the curve $y^2 = x^3 - 3x + b256$ over $\mathbb{F}_p256$

After defining all field and elliptic curve parameters as explained in 1.2, it is time to compute a seed, choose two points in the curve and feed all the data to the algorithm.

```
//Point S
sx= 0xde2444bebc8d36e682edd27e0f271508617519b3221a8fa0b77cab3989da97c9
sy= 0xc093ae7ff36e5380fc01a5aad1e66659702de80f53cec576b6350b243042a256

S= nzAffinePoint(sx ,sy )
Sproj = ec_projectify(pf256,S)


// Point R
rx= 0x51d08d5f2d4278882946d88d83c97d11e62becc3cfc18bedacc89ba34eeca03f
ry= 0x75ee68eb8bf626aa5b673ab51f6e744e06f8fcf8a6c0cf3035beca956a7b41d5

R=nzAffinePoint(rx ,ry)
Rproj=ec_projectify(pf256,R)

seed = random 123 : [256]
```

Listing 1.3. NIST curve P-256 points[18] and seed generation. *ec_projectify* converts the affine coordinates into projective (jacobian); *random* function takes an entropy input value and the size of the required random number.

Everything is set at this point. Assuming that the file (e.g., dualec_p256.cry) is loaded into Cryptol prelude, run:

```
Main> dualec256 curve256.ops seed Rproj Sproj

{fullnr =
    0x9e0ce7fcba977fa33d36a3f6ae2d4044fe41b350d0017e7babede64ac7a82698,
 rnr =
    0xe7fcba977fa33d36a3f6ae2d4044fe41b350d0017e7babede64ac7a82698,
 seed =
    0x2ad5fae9dd6b6402cdf9aa00194521190f124b292974fc6b2b43c706d87b5e45}
```

Listing 1.4. The generation of a random number in Cryptol. Note that we only pass the curve operations as argument (there is no need for other curve parameters for this). The return value is a record of the type {fullnr= raw number, rnr= the real output, seed= seed for the next iteration}. We should only know the rnr value but, for experimenting/testing purposes, it is useful to also save the rest.

4.3 Finding the next seed

For convenience, this example will be concluded with the data from curve used in section 3. Consider that *curve23* is defined the same way *curve256* is, and points P, Pproj, Q and Qproj are defined as S, Sproj, R and Rproj are (file dualec_p23).

```
Main> dualec5 curve23.ops seed5 projP5 projQ5
{fullnr = 4, rnr = 4, seed = 17}
```

Listing 1.5. The generation of a random number in Cryptol for curve $y^2 = x^3 - 3x + 7$.

What is known, at this point, is the output number of the last section, the rnr field. It is also known that the 2 most significant bits were taken from this number, therefore all possible combinations for the full number can be computed. This is one of the Dual_EC_DRNG flaws, as described in section 3.

```
Main> find_x5 23 ite1.rnr
[4, 12, 20, 0]
Main> :set base=2
Main> find_x5 23 ite1.rnr
[0b00100, 0b01100, 0b10100, 0b00000]
```

Listing 1.6. *find_x5 p nr* is a function that computes all possible values of a#nr, such that the resulting number is always smaller than p. You can see this in the example, there is no option 0b11100 because the number is bigger than 23 and so it is not the full number for sure. The function replaces these numbers with zeros. Note that Cryptol allows for multiple number bases visualizations what makes it easier.

Reasoning as in section 3 The values in listing 1.9 refer to possible x-coordinates of $r * Q$. In order to find the full point, $x^3 - 3x + 7$ has to be calculated its square roots found. For the $p - 256$ curve, there is a function that computes the square root (it is polymorphic on bit size), but for the small curve the way is to find y is via brute force.

```
Main> find_points5 curve23 xlist [1..23-1]
[{x = 4, y = 6}, {x = 4, y = 17},
 {x = 20, y = 9}, {x = 20, y = 14}]
```

Listing 1.7. *find_points5 curve xlist ylist* is a function that takes all the possible values of x and y and returns the pairs that are on the curve. Note that *xlist* is the result of the previous computation and [0..23-1] are the possible values of y on the prime field.

The points computed correspond to $r * Q$. It is known that $e = 5$ so taking the next step back means to multiply every point by 5:

```
Main> [(ec_affinify(pf23,ec_smul1 curve23.ops 5 p)).x | p<-projpontslist
    ]
[17, 17, 3, 3 ]
```

Listing 1.8. *find_points5 curve xlist ylist* is a function that takes all the possible values of x and y and returns the pairs that are on the curve. Note that *xlist* is the result of the previous computation and [0..23-1] are the possible values of y on the field.

This shows that seed for the next iteration is either 17 or 3! The initial list of possible values was reduced to 50% by removing elements with mathematical arguments. This measure is consistent with the probabilities calculated in [17]. The way to know for sure, as seen in section 3, is to compute two blocks instead of just one. The initial computed value with two blocks was 34. Proceeding as explained in section 3 one runs the Dual_EC_DRNG twice: once feeding it the seed 17 and then feeding it the seed 3.

```
Main> dualec5 curve23.ops 3 projP5 projQ5
{fullnr = 17, rnr = 1, seed = 19}
Main> dualec5 curve23.ops 17 projP5 projQ5
{fullnr = 18, rnr = 2, seed = 20}
```

Listing 1.9. The generation of a random number in Cryptol for seed=3 and seed=17

The result with seed 17 was the next number on the stream, so now the internal state of the algorithm is disclosed and there is no obstacle to predict following values.

Recommended NIST curve example. There are examples of this same mechanism described here using curves recommended by NIST. To discover the raw random number $r{*}Q$, it took 3h with an implementation in the C language on an ordinary computer[17]. Our attempt at this resulted in over 30h and not even half was computed. This may be because Cryptol is not as efficient as C and it is still in an early stage. Due to this, we did not present a real life example but instead a small one that is easy to follow and understand. There are no implications on the mechanism or the breakability of the algorithm whatsoever.

5 Proofs

The Cryptol toolset includes three main commands that explicitly state correctness properties. The *:prove* command constructs rigorous formal proofs, using SAT and SMT solvers. By default, Cryptol uses CVC4 SMT solver but it also supports others like Z3 or SRI's Yices. The underlying technique used by Cryptol is complete, considering it will always either prove or find a counterexample. Nonetheless, the proof process can take a larger amount of time or it can run out of memory. The *:sat* command checks satisfiability, i.e., displays some satisfying solutions of the property, if there are any. The *:check* command is inspired by Haskell's quick-check library. It performs random tests on the property in order to give a quick feedback about bugs. If only few values are provided, this command can also prove/disprove a property.

Cryptol properties can be polymorphic and they can hold at some, all, or none of its monomorphic instances. Therefore, no polymorphic properties can be proved in Cryptol. Since most of the functions are polymorphic, properties will be polymorphic as well. One fix for this is to give the property a type annotation to make it monomorphic.

5.1 Proving Group Law properties

One of the most important assumptions of this work is that all elliptic curve operations work properly. Taking into account that the addition of points is the most important operation, the following proofs will focus on the assurance of group law properties in the Cryptol implementation. The properties will be tested for the curve23 defined in listing (1.5)

Comments on proving in Cryptol. At the time of writing, Cryptol had a known issue with the conversion of 1-bit vectors to SMTLIB format, resulting in type errors. This is a drawback since all the elliptic curve arithmetic relies on bit-vector operations. It turns out that for the small curve in demonstration, the :check command is capable of covering all possible cases in a reasonable amount of time.

```
ec_full_add : {fv, gv}
    CurveOps fv gv -> ProjectivePoint fv-> AffinePoint fv
    -> ProjectivePoint fv
```

Listing 1.10. Type of the Cryptol implementation of the addition operation in the elliptic curve.

The listing 1.10 shows the point addition in the curve. It is important to note that the first point is represented in the projective system and the second point is represented in the affine system. The sum of the two will be a projective point.

Identity. Checking the identity first resulted in errors because the add function only accounted for the infinity on the left side (projective) and therefore there was no notion of infinity in affine coordinates. The infinity in projective coordinates is any point such that $z = 0$. A verification was added in the ec_full_add for the infinity in affine {x=0,y=0} as well as a second representation for the infinity in the projective system ({x=0,y=0,z=1}), that would arise from the use of ec_projectify on {x=0,y=0}.

```
point_add_ident: {fv, gv} (Cmp fv) => Curve fv gv -> AffinePoint fv ->
    Bit
point_add_ident c xa = (point_eq f r1p xp) && (point_eq f r2p xp) && (
    point_eq f r1p r2p)
  where id = {x=f.field_zero,y=f.field_zero}
        idp = ec_projectify(c.field, id)
        xp = ec_projectify(c.field, xa)
        r1p = add idp xa
        r2p = add xp id
        add=c.ops.add
        f=c.field

point_add_ident_II: {fv, gv} (Cmp fv) => Curve fv gv -> AffinePoint fv
    -> Bit
point_add_ident_II c xa = (point_eq f r1p xp) && (point_eq f r2p xp) &&
    (point_eq f r1p r2p)
  where id = {x=f.field_unit,y=f.field_unit,z=f.field_zero}
        ida = ec_affinify(f, id)
        xp = ec_projectify(f, xa)
        r1p = add id xa
        r2p = add xp ida
        add=c.ops.add
        f=c.field
        in_curve = point_in_curve c
        vals = point_vals f
```

Listing 1.11. Identity properties: for the affine and projective infinity representations. See Theorem 1.

After the previous error was corrected, both properties were proved by exhaustive testing.

```
Main> :check point_add_ident curve23
Using exhaustive testing.
passed 1024 tests.
QED

Main> :check point_add_ident_II curve23
Using exhaustive testing.
passed 1024 tests.
QED
```

Listing 1.12. Checks for the identity properties. Each one of them took less than a minute.

Commutativity This property was a great asset to figure out the conditions that needed to be guaranteed on the initial values: all coordinates have to be on the field that defines the curve and the input points have to belong to the curve. Without it, :check was finding counterexamples that did not matter to the evaluation.

```
point_add_is_comm: {fv, gv} (Cmp fv) => Curve fv gv -> AffinePoint fv ->
      AffinePoint fv -> Bit
point_add_is_comm c xa ya = if ((vals xa) && (vals ya) && (in_curve xa)
    && (in_curve ya)) //points in curve
                              then point_eq f r1p r2p
                              else True
    where
    xp = ec_projectify(f, xa)
    yp = ec_projectify(f, ya)
    // let r1p = x + y
    r1p = add xp ya
    // let r2p = y + x
    r2p = add yp xa
    f=c.field
    add= c.ops.add
    in_curve = point_in_curve c
    vals = point_vals f
```

Listing 1.13. Commutativity property. See Theorem 1.

After the conditional structure was added, the property was proved by exhaustive testing. These findings tell us that we have to be carefull with the values we pass to the functions.

```
Main> :check point_add_is_comm curve23
Using exhaustive testing.
passed 1048576 tests.
QED
```

Listing 1.14. Check for the commutativity property. The full execution took less than five minutes.

Associativity With the experience from the other properties, there were no counterexamples displayed when checking associativity. Considering the fact that this property takes three points (which means more test cases), the execution took a long time and it was not possible to cover all values.

```
point_add_is_comm: {fv, gv} (Cmp fv) => Curve fv gv -> AffinePoint fv ->
      AffinePoint fv -> Bit
point_add_is_comm c xa ya = if ((vals xa) && (vals ya) && (in_curve xa)
    && (in_curve ya)) //points in curve
                            then point_eq f r1p r2p
                            else True
    where
    xp = ec_projectify(f, xa)
    yp = ec_projectify(f, ya)
    // let r1p = x + y
    r1p = add xp ya
    // let r2p = y + x
    r2p = add yp xa
    f=c.field
    add= c.ops.add
    in_curve = point_in_curve c
    vals = point_vals f
```

Listing 1.15. Associativity property. See Theorem 1.

```
Main> :check point_add_is_comm curve23
Using random testing.
passed 10800000 tests.
Coverage: 1.01\% (10800000 of 2^^30 values)
```

Listing 1.16. Check for the associativity property. It took a couple of hours to prove 1%

6 Conclusions and Future Work

This project accomplished to show that the Dual_EC_DRNG is not secure through a mathematical explanation and a guided recover of the internal state.

Cryptol is a great tool for cryptographic applications mainly because of its polymorphism over word sizes. Its also has the ability to use the specifications to check and prove properties, without the need to move to another platform. However, the version used could benefit from some improvements. There are some issues that prevented us from taking the next step as, for instance, the SMTLIB translation bug. Nevertheless, the inability to use the :prove command was overcome by the use of exhaustive testing on a small-sized curve.

The results accomplished here can be complemented by using the :prove command to formally prove the properties and do it for bigger curve sizes. It would also be interesting to compare the advantages of proving with Cryptol to other well known theorem provers like Coq or EasyCrypt. The elliptic curve suit developed in this project could assist the implementation of other cryptographic primitives such as the elliptic curve DiffieHellman (ECDH) key agreement scheme.

All the goals for this project were accomplished and extra value was added. In the light of these findings, the authors support NIST's decision of removing the algorithm from the standard and point out the importance of being secure in the digital world.

Acknowledgements. This work was carried out in the context of a proposal from Galois for the Cohesive Project of the Formal Methods of Software Engineering course.

We would like to thank Mr. Joe Hendrix for proposing this project and for his availability throughout; Prof. José Nuno Oliveira for his counseling and exhaustive meetings, even on short notice; Prof. Maria João Frade for helping us with the proofs and pointing us in the right direction; Prof. José Bacelar Almeida for advising us and taking interest in our work; DI technicians for their ability to solve every problem with our machines; and all MFES team, colleagues and professors, for being there every milestone listening, criticizing and helping us grow this project.

We gratefully acknowledge usage of Wikipedia and gladly blame its contributors for any errors in our data.

References

1. Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Transactions on Information Theory **22**(6) (1976) 644–654
2. Miller, V.S.: Use of elliptic curves in cryptography. In: CRYPTO. (1985) 417–426
3. Barker, E.B., Kelsey, J.M.: Sp 800-90a. recommendation for random number generation using deterministic random bit generators. Technical report, Gaithersburg, MD, United States (2012)
4. Lenstra, A.K., Kleinjung, T., Thomé, E.: Universal security - from bits and mips to pools, lakes - and beyond. In: Number Theory and Cryptography. (2013) 121–124
5. NIST: Drbg validation list (2014)
6. Shumow, D., Ferguson, N.: On the possibility of a back door in the nist sp800-90 dual ec prng. Crypto 2007 rump session (2007)
7. Perlroth, N., Larson, J., Shane, S.: N.s.a. able to foil basic safeguards of privacy on web. New York Times (September 2013)
8. : Nist removes cryptography algorithm from random number generator recommendations (April 2014)
9. Rosen, K.H.: Discrete Mathematics and Its Applications. 2nd edn. McGraw-Hill Higher Education (2002)
10. Hankerson, D., Menezes, A.J., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2003)
11. Dams, J.: An introduction to elliptic curve cryptography
12. : Elliptic curve cryptosystems. Mathematics of Computation **48**(177) (1987) 203–209

13. Miller, V.S.: Use of elliptic curves in cryptography. In: Lecture Notes in Computer Sciences; 218 on Advances in cryptology—CRYPTO 85, New York, NY, USA, Springer-Verlag New York, Inc. (1986) 417–426
14. Silverman, J.: The Arithmetic of Elliptic Curves. Graduate Texts in Mathematics. Springer (2009)
15. Hoffstein, J., Pipher, J., Silverman, J.: An introduction to mathematical cryptography. (2008)
16. Brown, D.R.L., Gjøsteen, K.: A security analysis of the nist sp 800-90 elliptic curve random number generator. IACR Cryptology ePrint Archive **2007** (2007) 48
17. Schoenmakers, B., Sidorenko, A.: Cryptanalysis of the dual elliptic curve pseudorandom generator. IACR Cryptology ePrint Archive **2006** (2006) 190
18. NSA: Mathematical routines for the nist prime elliptic curves (2010)
19. Cohen, H., Frey, G., Avanzi, R., Doche, C., Lange, T., Nguyen, K., Vercauteren, F.: Handbook of Elliptic and Hyperelliptic Curve Cryptography. 2nd edn. Chapman & Hall/CRC (2012)
20. Gordon, D.M.: A Survey of Fast Exponentiation Methods. Journal of Algorithms **27**(1) (1998) 129–146