A Practical Analysis of Dual Elliptic Curve Deterministic Random Bit Generator

Tim Johns

Friday, December 26th, 2026

1 Introduction

A reliable source of randomness is an often overlooked requirement of any modern cryptographic system. We use random bits to generate private keys, nonces, salts, and protocol challenges. As such, if an attacker can predict a system's "random" values, they can often reconstruct secrets that were never meant to leave the machine. For this reason, standards bodies publish vetted designs for *deterministic random bit generators* (DRBGs). A DRBG takes a short, high-entropy *seed*—obtained from an operating system's entropy sources—and expands it into a long stream of pseudorandom bits that are computationally indistinguishable from truly random ones. In practice, the seed is the scarce resource: it is costly to collect high-quality unpredictability from the physical world, while it is efficient to use a cryptographic construction to *stretch* that unpredictability into as many bits as an application needs. The resulting output can then be used anywhere a protocol requires random-looking values, such as key generation, nonces, salts, and challenges.

DUAL_EC_DRBG (Dual Elliptic Curve Deterministic Random Bit Generator) is one such design. It was standardized by NIST as part of the SP 800-90 family of DRBG recommendations and, for years, appeared alongside other approved constructions such as Hash_DRBG, HMAC_DRBG, and CTR_DRBG. Yet DUAL_EC_DRBG quickly stood out for two reasons: it was unusually slow compared to alternatives, and it relied on two fixed elliptic-curve points (P, Q) whose origins were not transparently justified in the public documentation.

The controversy begins with a simple (and alarming) observation: if those public points were chosen with a secret relationship—specifically, if there exists a secret scalar d such that $P = dQ$—then someone who knows d can use a small amount of generator output to recover the internal state and predict future outputs. In other words, the points (P, Q) can function like a public key, while d behaves like the corresponding private key: publicly harmless-looking parameters that quietly grant a powerful advantage to whoever generated them. This possibility was publicly discussed as early as 2007, when Dan Shumow and Niels Ferguson presented an argument showing how such a relationship could enable state recovery.

For several years, this remained an unsettling *possibility* rather than a confirmed backdoor. The issue exploded into public view in 2013 amid reporting on leaked intelligence documents, and it intensified when Reuters reported that RSA Security had accepted a secret \$10 million deal to make DUAL_EC_DRBG the default in its widely used BSAFE cryptographic library—effectively amplifying the impact of any weakness in real systems. In response to the loss of public confidence, NIST advised users to transition away from DUAL_EC_DRBG and ultimately removed it from its

DRBG recommendations.

This paper explains, at a practical level, how the suspected trapdoor mechanism works. We first review the mathematical background (finite fields, elliptic curves, and point operations), then walk through a concrete implementation of DUAL_EC_DRBG, and finally show how an attacker who knows the hidden relationship between P and Q can clone the generator from observed output. Much of the implementation and attack flow is adapted from Lesson 3.2 of *Hacking Cryptography* by Kamran Khan and Bill Cox, with modifications for clarity and for a Python-based demonstration.

2 Background Concepts

2.1 Very large numbers

On most computers, the maximum value of an integer type is limited by the bit-width of the processor. So, for example, on a 32-bit machine, the largest unsigned integer we could store is $2^{32} - 1$, or 4,294,967,295.

In order to work with numbers larger than 4,294,967,295 (which we will need to in order to impliment DUAL_EC_DRBG), we must use a software library. Rather than performing mathematical operations directly on the hardware, as is possible when we are working with numbers that correspond to the 32 or 64-bit word size of the processor, we must use software packages that can perform these calculations.

2.1.1 Hexadecimal representation

When working with cryptographic algorithms, we often represent large integers in *hexadecimal* (base 16) rather than in decimal (base 10). This is not because the math is done in base 16—internally, the computer still stores the value as bits (base 2), but because hexadecimal is a compact, human-readable way to write binary data.

Hexadecimal uses sixteen symbols:

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F,$$

where $A = 10$ through $F = 15$. The key convenience is that one hexadecimal digit corresponds to exactly four bits (a *nibble*). Therefore:

$$1 \text{ hex digit} = 4 \text{ bits}, \qquad 2 \text{ hex digits} = 8 \text{ bits} = 1 \text{ byte}.$$

So a sequence of bytes can be written cleanly as pairs of hex digits. For example, the four bytes

```
DE AD BE EF
```

correspond to the 32-bit binary string

$$11011110\ 10101101\ 10111110\ 11101111.$$

This matters in practice because cryptographic values—keys, nonces, internal states, and elliptic-curve coordinates—are usually specified and displayed as fixed-length byte strings. For instance, a 256-bit value is 32 bytes long, and is commonly written as 64 hexadecimal characters. In DUAL_EC_DRBG, we will repeatedly interpret byte strings as integers (and vice versa), so you will often see operations described in terms of “take the leftmost 30 bytes,” “discard 2 bytes,” or “guess the missing 2 bytes”—which is especially easy to visualize when those bytes are shown in hex.

2.2 Elliptic Curves

Elliptic curves are a foundational mathematical element of many cryptographic algorithms. An elliptic curve is a set of points defined by the equation:

$$y^2 \equiv x^3 + ax + b \pmod{p}$$

The " $\pmod{p}$" indicates that we are working in the finite field F_p (often written $\mathbb{F}_p$), meaning that all arithmetic within the field is done modulo p and the coordinates x and y are elements of the set F_p. What is p, you ask? p is simply a large prime number. In practice, when using elliptic curve cryptography (ECC), the p used is a public parameter of the curve that the communicating parties agree to use (so each curve has a fixed p—we don't need to pick one). Given all these attributes, we can define an elliptic curve as a set that contains all the points (x, y) in a finite field that satisfy this "$y^2 \equiv x^3 + ax + b$" function.

But what exactly is a "finite field?". Well, put in simple terms, a finite field is a set of numbers where you can add, subtract, multiply, divide (except by zero), and the resulting sum, product or quotient always lands within the field. Here's a concrete example. Suppose I would like to use ECC, and I pick a curve that has an extremely small p, like $p = 7$. Our finite field, F_p, is as follows:

$$F_p = \{0, 1, 2, \ldots, p-1\} \to F_7 = \{0, 1, 2, 3, 4, 5, 6\}$$

Essentially, you can think of our finite field set like a miniature number system that only allows for the existence of numbers already contained in the field (in this case 0 through 6), and abides by three rules:

(1.) **Closure:** If you add or multiply two allowed numbers, the result is still an allowed number. The reason our finite field is able to meet this rule is because we are using modular arithmetic, rather than traditional arithmetic. That means that our sums, products, quotients, wrap around at p, so our result never exceeds p

Ex: Suppose, in our finite field F_7, we want to multiply two "allowed" numbers, 3 and 5. We multiply, and wrap at 7:

$$\begin{aligned} 3 \cdot 5 &\equiv 15 \pmod{7} \\ 15 - 7 &= 8 \\ 8 - 7 &= 1 \\ \Rightarrow \quad 3 \cdot 5 &\equiv 1 \pmod{7} \end{aligned}$$

We can work with larger numbers, like 32, for example, but 32 is not a real number within our finite field. Any number greater than p (in this example 7), does not exist. It is just a non-canonical representative of an allowed number. So, for example, 32 is just a representation of 4:

$$\begin{aligned} 32 - 7 - 7 - 7 - 7 &= 4 \\ \Rightarrow \quad 32 &\equiv 4 \pmod{7} \end{aligned}$$

If we did $32 \cdot 86$ within the finite field, our answer is allowed, because we have

$$\begin{aligned} 32 \cdot 86 &= 2752 \\ 2752 &= 7 \cdot 393 + 1 \\ \Rightarrow \quad 32 \cdot 86 &\equiv 1 \pmod{7} \end{aligned}$$

and 1 is an allowed number.

(2.) **Normal algebraic rules hold, and these identities still hold:**

- $a + b = b + a$
- $(a + b) + c = a + (b + c)$
- $a(b + c) = ab + ac$

(3.) **Division works;** For division to be possible, it means that for every non-zero number, a, such that $a \neq 0$, there exists another whole number in the field, b, such that $a \cdot b = 1$ after reducing mod p. This allows us to divide numbers in our finite field because division is multiplication by an inverse.

Ex: If $a = 3$, then $b = 5$, since

$$\begin{aligned} 3 \cdot 5 &\equiv 15 \pmod 7 \\ 15 - 7 &= 8 \\ 8 - 7 &= 1 \\ \Rightarrow \quad 3 \cdot 5 &\equiv 1 \pmod 7 \end{aligned}$$

Now that we understand the concept of a finite field a bit better, we can return to the definition of an elliptic curve. We can define an elliptic curve over F_p in set-builder notation like this:

$$E(F_p) = \{(x, y) \in F_p^2 : y^2 \equiv x^3 + ax + b \text{ (mod } p)\} \cup \{O\}$$

When translated to natural language, we now arrive at a precise, actionable definition of what an elliptic curve is:

An elliptic curve is a set (or collection) of points, which has the following properties:

- **Each point is a pair of numbers** (x, y)**.**
- x **and** y **are each existing numbers in the field** $\{0, 1, 2, \ldots, p - 1\}$**.**
- **A pair is on the curve if, when you plug** x **and** y **into the elliptic curve equation,** $y^2 \equiv x^3 + ax + b \pmod p$**, and they match.** (i.e., $y^3 \pmod p$ is equal to $x^3 + ax + b \pmod p$).

Note, you also have a point at infinity in the set. This is required because we need an identity element (like zero in regular addition) to form a group under point addition. On an elliptic curve, $P + O = P$ for any point, just like how $a + 0 = a$ in regular addition. This is why we have $\cup\{O\}$ as part of our set-builder definition of an elliptic curve.

Now that we have more of a grasp on what elliptic curves are, we can discuss more about the operations that are used in elliptic curve cryptography: **point addition, and scalar multiplication.**

2.3 Operations on Elliptic curves

Suppose we have two points: P and Q, on an elliptic curve. When we perform point addition, we want the sum point, R, to also lie on the elliptic curve. As such, we cannot just use ordinary addition to add the x and y coordinates of P and Q, as the sum would fall outside the curve (not be a part of our finite field). Instead, we must use **point arithmetic**. Conceptually, to perform addition on two points on a curve, we draw a line through those two points. This line intersects the curve, of course, at these points P and Q. However, it will also intersect our curve at a third point, R. This third point is our sum. You do not need to understand the full derivation in order to understand DUAL_EC_DRBG, but it is mostly basic algebra, and is useful for understanding point addition. Here it is:

Derivation (Point Addition Formula). Let $E(F_p)$ be an elliptic curve of the form

$$y^2 \equiv x^3 + ax + b \pmod{p},$$

and let

$$P = (p_x, p_y), \qquad Q = (q_x, q_y)$$

be two points on $E(F_p)$ with $p_x \not\equiv q_x \pmod{p}$. Define $R = P + Q = (r_x, r_y)$.

First, define the slope λ of the line through P and Q:

$$\lambda = \frac{q_y - p_y}{q_x - p_x} \text{ (mod } p).$$

Since division is multiplication by an inverse, this is:

$$\lambda \equiv (q_y - p_y)\,(q_x - p_x)^{-1} \pmod{p}.$$

Multiplying by $(q_x - p_x)$ to cancel the inverse, we get:

$$\lambda(q_x - p_x) \equiv (q_y - p_y) \pmod{p}.$$

Add p_y to get slope-intercept form:

$$\lambda(q_x - p_x) + p_y \equiv q_y \pmod{p}, \quad \text{or} \quad q_y \equiv \lambda(q_x - p_x) + p_y \pmod{p}.$$

Now, we have our line. To find the third point where it intersects the curve, we substitute the line into

$$y^2 \equiv x^3 + ax + b \pmod{p},$$

giving

$$(\lambda(q_x - p_x) + p_y)^2 \equiv x^3 + ax + b \pmod{p}.$$

This produces a cubic polynomial in x whose roots (counting multiplicity) correspond to the x-coordinates of the intersection points. Since $x = p_x$ and $x = q_x$ are two roots, the remaining root is $x = r_x$. Using polynomial root identities / Vieta's formulas, we obtain

$$p_x + q_x + r_x \equiv \lambda^2 \pmod{p},$$

so

$$r_x \equiv \lambda^2 - p_x - q_x \pmod{p}.$$

To find r_y, plug r_x into the line equation:

$$y \equiv \lambda(r_x - p_x) + p_y \pmod{p}.$$

Finally, reflect across the x-axis $(x, y) \to (x, -y)$ to get:

$$r_y \equiv -(\lambda(r_x - p_x) + p_y) \pmod{p},$$

or equivalently

$$r_y \equiv \lambda(p_x - r_x) - p_y \pmod{p}.$$

Therefore,

$$\boxed{\begin{aligned} P + Q = R, \qquad (p_x, p_y) + (q_x, q_y) = (r_x, r_y),\\ \lambda \equiv (q_y - p_y)(q_x - p_x)^{-1} \pmod{p},\\ r_x \equiv \lambda^2 - p_x - q_x \pmod{p},\\ r_y \equiv \lambda(p_x - r_x) - p_y \pmod{p}. \end{aligned}}$$

Based off of that proof, we do not need anything new to perform scalar multiplication, as it is just repeated addition. To add the same point to itself, we just slightly alter the formula since we use a tangent line, rather than the slope.

Scalar multiplication (repeated addition):

$$\boxed{kP = \underbrace{P + P + \cdots + P}_{k \text{ times}}}$$

Point addition to add a point to itself ($P = Q$):

$$\boxed{\begin{aligned} \lambda &\equiv (p_x^2 + a)(2p_y)^{-1} \pmod{p}\\ r_x &\equiv \lambda^2 - 2p_x \pmod{p}\\ r_y &\equiv \lambda(p_x - r_x) - p_y \pmod{p} \end{aligned}}$$

Now, we have a basic understanding of large numbers, elliptic curves, what it means for points P and Q to exist on a curve, and some of the point operations we can perform on P and Q. We will now show how these basic ingredients are used to implement DUAL_EC_DRBG!

3 How Does DUAL_EC_DRBG Work?

To understand how the DUAL_EC_DRBG works, I am going to walk you through it's implimentation in the Python programming language. The code I am using came from the sample file included with the textbook *Hacking Cryptography: Write, Break, and Fix Real-world Implementations.* It was originally written in the programming language Go, by Kamran Khan and Bill Cox. This makes sense, as Go is optimized for performing operations on very large numbers, and is thus much faster than python. I am more familiar with python (as is much of the academic world), however, so I used Gemini 3.0 Pro to port the code to Python.

3.1 Dependencies

This program utilizies the following dependencies:

```
import unittest
import secrets
import sys
import time
import binascii
from ecdsa import NIST256p, ellipticcurve
```

Brief explanation of each import:

- `unittest`: Provides a built-in testing framework for defining and running test cases.
- `secrets`: Generates cryptographically secure random values (e.g., random scalars).
- `sys`: Used for low-level interaction with the interpreter (e.g., writing progress updates to stdout and flushing).
- `time`: Used for time-based values (e.g., seeding with the current Unix time).
- `binascii`: Utilities for binary/ASCII conversions (commonly used for hex/byte conversions).
- `ecdsa.NIST256p, ecdsa.ellipticcurve`: Supplies the NIST P-256 curve parameters and elliptic-curve point operations.

3.2 Initialize Points on the Curve

DUAL_EC_DRBG depends on two points, P and Q. In the official implementation of this RNG, NIST specifies standard 32-bit values for P and Q in order to standardize the behavior of the algorithm. We begin by adding these constant coordinates for each point into our program. *These are very large numbers, so we must store them as base-16 encoded hexadecimal strings while defining them. They will be converted to integers while in use by the program.*

```
# ==========================================
# PART 1: The Basic DRBG Implementation
# ==========================================

# Constants defining the curve points P and Q
PX_HEX = "6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296"
PY_HEX = "4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5"
QX_HEX = "c97445f45cdef9f0d3e05e1e585fc297235b82b5be8ff3efca67c59852018192"
QY_HEX = "b28ef557ba31dfcbdd21ac46e2a91e3c304f44cb87058ada2cb815151e610046"
```

Below these constants, we also define a "Point" class, which allows us to initialize a Point data type, that contains two fields: an x coordinate and a y coordinate. Our hexadecimal strings are converted to an integer and stored in self.x and self.y respectively. We are able to store such large numbers as ints because Python's int type is actually an arbitrary-precision (a "bignum"), meaning it automatically grows to as many bits as needed. So a 256-bit value like those P-256 coordinates fits comfortably.

```
class Point:
    def __init__(self, x, y):
        self.x = int(x, 16) if isinstance(x, str) else int(x)
        self.y = int(y, 16) if isinstance(y, str) else int(y)

    def __eq__(self, other):
        return isinstance(other, Point) and self.x == other.x and self.y == other.y
```

3.3 How Random Number Generation Works

Before looking at any code, we need to establish how DUAL_EC_DRBG actually generates random streams of bits. Let's begin by looking at a few main components of the algorithm:

- *seed* – The initial secret value used to initialize the generator's internal state.
- $state_t$ – The internal state at a given point in time, t. In our program, it is a bignum, which we will call n.
- $g_P(x)$ – An internal function that advances the state, n, at each point in time, t.
- $g_Q(x)$ – An internal function that transforms the state, n, at each point in time, t, turning it into the output.

Here is a diagram of the flow:

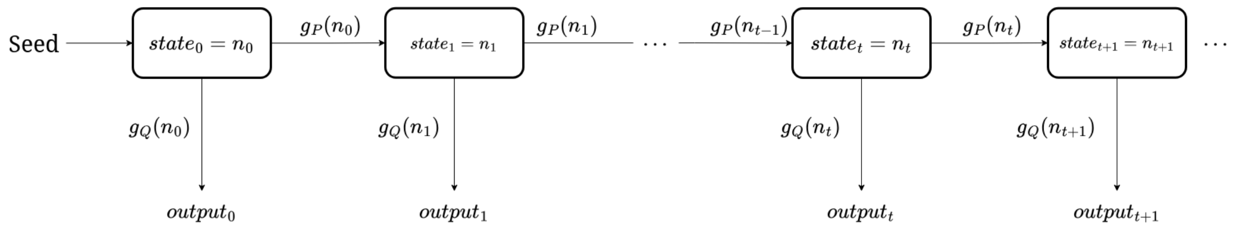

Figure 1: State diagram showing how g_P advances the internal state and g_Q produces the output.

These internal functions, $g_P(x)$ and $g_Q(x)$, rely on scalar multiplication of the current state at a given time (which is a bignum, n) with P and Q respectively. If you recall from our discussion of point arithmetic, the product of scalar multiplication is another point on the curve, since scalar multiplication is just repeated point addition.

$$kP = P'$$

However, we don't need this whole point, P'. Instead, we use two helper functions:

- $X(P')$ – This helper function just discards the y coordinate and returns the x coordinate, p'_x.
- $T(x)$ – This function returns the 30 least significant bytes of x (it truncates whatever input it receives to 30 bytes).

We can now define our internal functions $g_P(x)$ and $g_Q(x)$ as follows:

$g_P(n) = X(nP)$ – The x coordinate of the point that is the product of scalar multiplication between the current state, n, and point P.

$g_Q(n) = T\big(X(nQ)\big)$ – The least significant 30-bytes of the x coordinate of the point that is the product of scalar multiplication between the current state, n, and point Q.

3.4 Implementing DUAL_EC_DRBG in Code

Now, we have all the components to implement DUAL_EC_DRBG in code. Below our Point class, we will create a class for a DualEC random bit generator type:

```
class DualEcDrbg:
    def __init__(self, p, q):
        if not p or not q:
            raise ValueError("Invalid points P or Q")

        self.state = None
        self.curve = NIST256p.curve
        self.p_point = ellipticcurve.Point(self.curve, p.x, p.y)
        self.q_point = ellipticcurve.Point(self.curve, q.x, q.y)

    def seed(self, seed_int):
        self.state = seed_int
```

The constructor takes two points, P and Q as arguments. The initial state field is left empty. We set the curve field to the NIST provided NIST256p curve. *Note: "NIST256p.curve" is an object we get from the ecdsa library. It contains curve parameters like the field prime, p, coefficients (a, b) etc.* These lines:

```
        self.p_point = ellipticcurve.Point(self.curve, p.x, p.y)
        self.q_point = ellipticcurve.Point(self.curve, q.x, q.y)
```

construct an ecdsa.ellipticcurve.Point object on the NIST256p curve (stored in self.curve), for both points, P and Q. Next, we define a very simple generate() function that progresses the state of our generator:

```
def generate(self):
    if self.state is None:
        raise ValueError("DRBG not seeded")

    # Helper function: X(P') = returns the x-coordinate of point P'
    def X(point):
        return point.x()

    # Helper function: T(x) = returns the 30 least significant bytes of x
    def T(x_int):
        x_bytes = x_int.to_bytes(32, byteorder='big')  # P-256 x is 32 bytes
        return x_bytes[-30:]                           # truncate to 30 LSB bytes

    # Step 1: Update state using g_P(n) = X(nP)
    nP = self.p_point * self.state
    self.state = X(nP)
```

```
        # Step 2: Produce output using g_Q(n) = T(X(nQ))
        nQ = self.q_point * self.state
        output = T(X(nQ))

        return output
```

As you can see, this function acts exactly the same as the functions we discussed in the previous section.

If we wanted to use our implementation, we could do so in a main function like this:

```
import secrets
from full_exploit import DualEcDrbg, Point, PX_HEX, PY_HEX, QX_HEX, QY_HEX
from ecdsa import NIST256p

# 1) Define the fixed curve points P and Q (from hex constants)
P = Point(PX_HEX, PY_HEX)
Q = Point(QX_HEX, QY_HEX)

# 2) Create the DRBG instance
drbg = DualEcDrbg(P, Q)

# 3) Seed it (best: OS CSPRNG via secrets)
seed_int = secrets.randbelow(NIST256p.order - 1) + 1
drbg.seed(seed_int)

# 4) Generate output bytes (30 bytes each call)
out = drbg.generate()
print(out.hex())          # hex string
print(len(out))           # 30

# Generate multiple blocks if you need more bytes
stream = b"".join(drbg.generate() for _ in range(10))  # 300 bytes
```

Note the use of the secrets module to generate a seed. Here, we choose a random nonzero integer in the valid scalar range for the P-256 group (i.e., a good "seed/state" value to use in scalar multiplication). This integer is generated by the operating system's CSPRNG, which is seeded from real entropy sources on the machine. At the end of the day, the initial seed of any cryptographic system must come from a real-world non-deterministic source. We use PRNG's to make it easier to produce random byte streams needed for seeding cryptographic values like keys and salts.

4 How can DUAL_EC_DRBG be Exploited?

DUAL_EC_DRBG can be exploited if there is a secret relationship between the generator points, P and Q. Suppose that P and Q are related such that $P = dQ$, for some scalar d that is known to us. To demonstrate how this works, consider this segment of the generator:

Note, I've substituted $g_P(x)$ and $g_Q(x)$ for the helper functions that they implement.

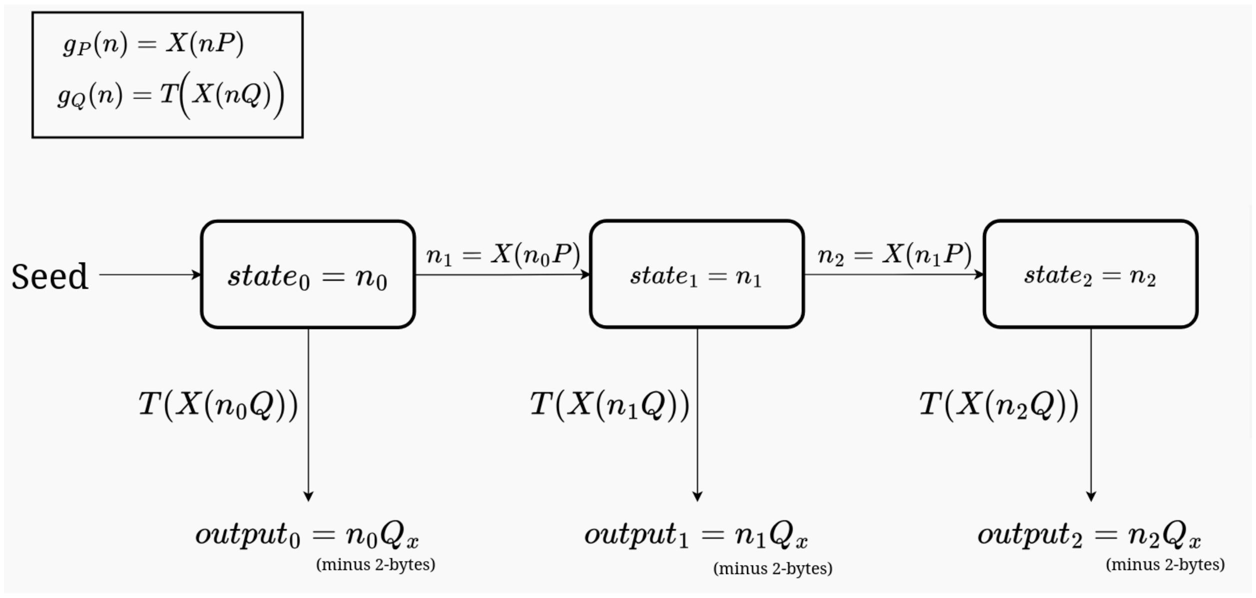

Figure 2: State diagram showing how g_P advances the internal state and g_Q produces the output, with g_P and g_Q replaced by their helper functions.

Now, suppose we are an observer of the generator at time $t = 0$. We cannot see the internal state of the generator, but we can see the output, which at it's core, is n_0Q. In reality, it is the least significant 30 bytes of n_0Q's x-coordinate (we'll write this as $(n_0Q)_x$), but we are going to ignore this fact for now.

Given that we are assuming that $P = dQ$ for a known d, we are able to figure out what n_0P is!

$$\text{If } Qd = P, \text{ then } (n_0Q)d = n_0P.$$

With n_0P, we can predict the next state, and thus the next output! Of course, we aren't quite there yet due to the transformations that the $X(x)$ and $T(x)$ functions apply. However, as we'll see, both are reversible.

The easier of the two to reverse is function $X(x)$, which takes a point and discards the y-coordinate. Re-calculating this lost coordinate is trivial, however, if you recall this last component of our definition of an elliptic curve:

"**A pair is on the curve if, when you plug x and y into the elliptic curve equation,** $y^2 \equiv x^3 + ax + b \pmod{p}$**, and they match.** (i.e., $y^3 \pmod{p}$ is equal to $x^3 + ax + b \pmod{p}$)."

Well, given that a, b and p are all known constants that are provided by NIST, we simply plug $(n_0Q)_x$, a and b into $\sqrt{x^3 + ax + b \pmod{p}}$ to determine $(n_0Q)_y$. With both $(n_0Q)_x$ and $(n_0Q)_y$, we have n_0Q, since $n_0Q = ((n_0Q)_x, (n_0Q)_y)$. As was established, with d and n_0Q, we can recover n_0P. However, we have yet to reverse the function $T(x)$. Since our output was $T(X(n_0Q))$, we need to reverse $T(x)$ to recover $X(n_0Q)$, which then allows us to recover n_0Q

While there is no algorithm that can recover the original discarded bytes from our internal value of n_0Q, we aren't in too bad shape, since only two bytes were discarded! (Our points are 32-bytes long if you recall, and $T(x)$ returns the least significant 30 bytes. To undo $T(x)$, we can guess the missing **two bytes** by trying all 16-bit values:

$$0000_{16}, 0001_{16}, \ldots, FFFF_{16}.$$

Furthermore, we can streamline this process by using our equation for the y coordinate to check our guess! If our current guess for the missing two bytes satisfies the equation:

$$y = \sqrt{x^3 + ax + b \pmod{p}}$$

Then they may be our missing bytes! This is the case because, while every guess will generate a value when you plug into the right side of the equation, only a few will have a valid square root! Of these guesses that have a valid square root, we simply then need to generate our n_0P value and compare $T(X((X(n_0P))Q))$ to the next output (recall that $X(n_0P) = n_1$). This will confirm that we've found the actual missing bytes in n_0Q_x!

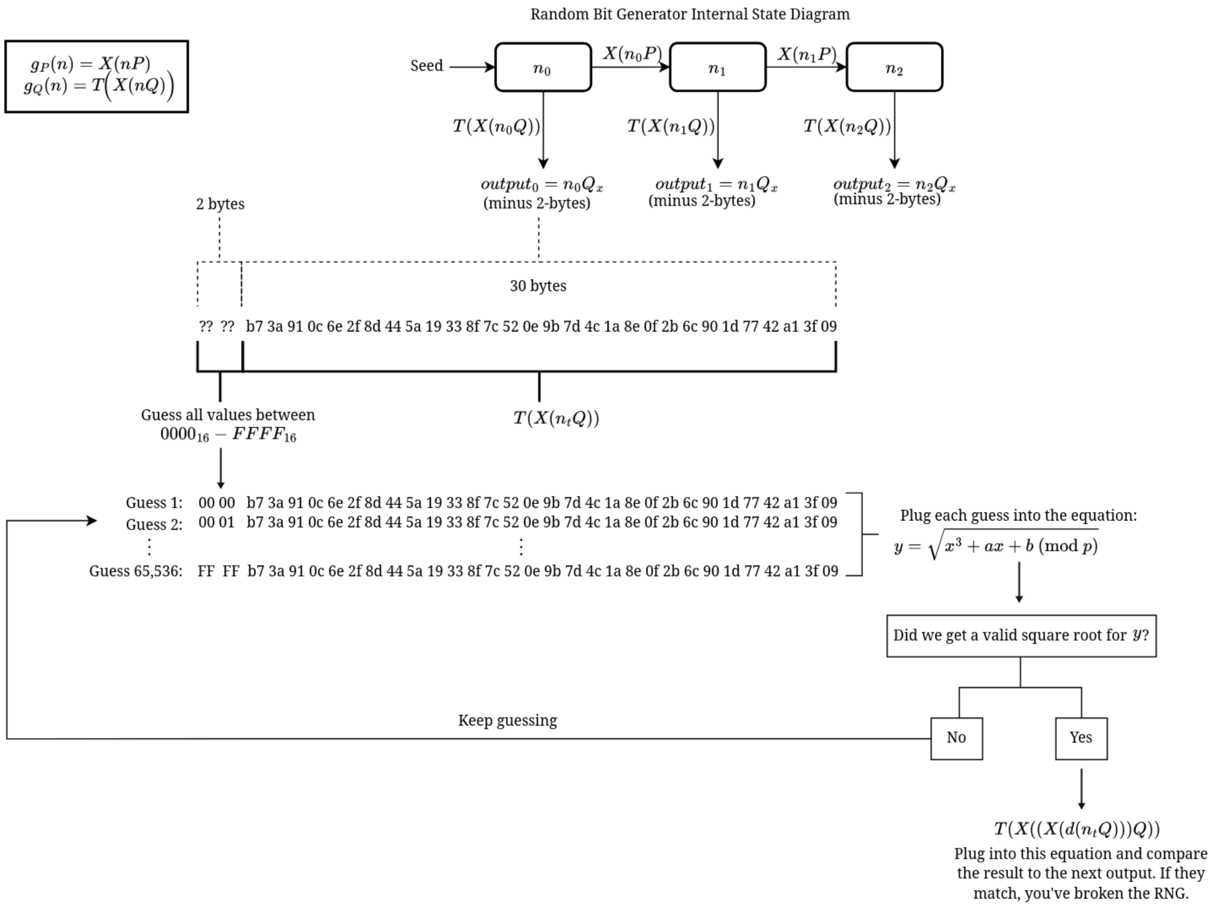
Random Bit Generator Internal State Diagram
$g_P(n) = X(nP)$
$g_Q(n) = T\big(X(nQ)\big)$
Seed
n_0
n_1
n_2
$X(n_0P)$
$X(n_1P)$
$T(X(n_0Q))$
$T(X(n_1Q))$
$T(X(n_2Q))$
$output_0 = n_0Q_x$ (minus 2-bytes)
$output_1 = n_1Q_x$ (minus 2-bytes)
$output_2 = n_2Q_x$ (minus 2-bytes)
2 bytes
30 bytes
?? ?? b7 3a 91 0c 6e 2f 8d 44 5a 19 33 8f 7c 52 0e 9b 7d 4c 1a 8e 0f 2b 6c 90 1d 77 42 a1 3f 09
Guess all values between $0000_{16} - FFFF_{16}$
$T(X(n_tQ))$
Guess 1: 00 00 b7 3a 91 0c 6e 2f 8d 44 5a 19 33 8f 7c 52 0e 9b 7d 4c 1a 8e 0f 2b 6c 90 1d 77 42 a1 3f 09
Guess 2: 00 01 b7 3a 91 0c 6e 2f 8d 44 5a 19 33 8f 7c 52 0e 9b 7d 4c 1a 8e 0f 2b 6c 90 1d 77 42 a1 3f 09
Guess 65,536: FF FF b7 3a 91 0c 6e 2f 8d 44 5a 19 33 8f 7c 52 0e 9b 7d 4c 1a 8e 0f 2b 6c 90 1d 77 42 a1 3f 09
Plug each guess into the equation:
$y = \sqrt{x^3 + ax + b \pmod{p}}$
Did we get a valid square root for y?
No
Yes
Keep guessing
$T(X((X(d(n_tQ)))Q))$
Plug into this equation and compare the result to the next output. If they match, you've broken the RNG.

Figure 3: Diagram showing the process for reversing $T(x)$ and $X(x)$ to find n_tQ_x, for a given state time, t.

5 How can DUAL_EC_DRBG be Backdoored?

As we established in the previous section, we want a secret value d such that

$$P = dQ.$$

This is easier said than done. In standard elliptic curves (like P-256), the point P is fixed (the **base point**), so we cannot choose P freely. Instead, we choose d first and then build a matching Q.

1. Pick a random nonzero value for d (so it has a modular inverse).
2. Compute the modular inverse of d and call it e:

   $$e \equiv d^{-1}.$$

3. Multiply both sides of $P = dQ$ by e:

   $$eP = edQ.$$

   Since $ed \equiv 1$, this simplifies to:

   $$eP = Q.$$

   So we define:

   $$Q = eP.$$

Now, returning to our code, we are ready to exploit our implementation of DUAL_EC_DRBG. We begin by defining some constants, and then creating a function that picks a d value, gnerates P and Q with it, and returns all three:

```
CURVE = NIST256p.curve
p_curve = CURVE.p()
a_curve = CURVE.a()
b_curve = CURVE.b()
n_order = NIST256p.order

def generate_backdoor_constants():
    """Generates P, Q, and the secret backdoor scalar d."""
    # 1. Generate random d
    d = secrets.randbelow(n_order - 1) + 1

    # 2. Calculate e = d^(-1) mod n
    e = pow(d, -1, n_order)

    # 3. P is the standard generator
    G = NIST256p.generator
    px, py = G.x(), G.y()

    # 4. Calculate Q = e * P
    Q_point = G * e
    qx, qy = Q_point.x(), Q_point.y()
    P_obj = Point(px, py)
    Q_obj = Point(qx, qy)

    return P_obj, Q_obj, d
```

Note: n_ order is the modulus used for scalar inverses and scalar relationships between points. p_ curve is still the modulus used for our elliptic curve equation calculation.

We also create a function used to check if the n_tQ_x 32-byte value we are guessing is correct by trying to solve our elliptic curve equation:

```
def calculate_y_coordinate(x):
    """Solves for y in y^2 = x^3 - 3x + b (mod p)."""
    rhs = (pow(x, 3, p_curve) + (a_curve * x) + b_curve) % p_curve
    # P-256 prime is 3 mod 4, so we use this sqrt formula
  ‘  y = pow(rhs, (p_curve + 1) // 4, p_curve)

    if pow(y, 2, p_curve) != rhs:
        return None
    if not CURVE.contains_point(x, y):
        return None
    return y
```

Now, with these functions out of the way, we can create our function that attempts to exploit an actual drbg by observing two consecutive outputs, and then cloning it.

```
def clone_dual_ec_drbg(drbg, p_const, q_const, d):
    """Attacks the DRBG instance to recover the internal state."""
    observed = drbg.generate()
    check = drbg.generate()

    print(f"Target check value: {check.hex()}")
```

We begin the function by accepting a deterministic random bit generator, drbg, P, Q, and d. We then grab the 30-byte output at time $t = n$. This is the output we are going to use to find n_tQ_x. We call the generate() function on the drbg object again, so that it advances the state. That output is the 30-byte output from time $t = n + 1$. We store this in "check", as this is the value we will be checking against to determine whether we have successfully cloned the drbg.

Next, we just convert our custom backdoor-ed Q Point object to an ecdsa Point object, which defines useful operations like scalar multiplication (*) and addition (+).

```
Q_ecdsa = ellipticcurve.Point(CURVE, q_const.x, q_const.y)
```

Next, we begin brute force-guessing the missing 2-bytes. For each value from 0000 to $FFFF$, we concatenate those guessed bytes with the observed 30-bytes we got from the output, storing the full value in "guess_bytes." It is then converted to an integer and passed into the function that checks the value against our elliptic curve equation. If the value of y is returned as "None", meaning it did not have a square root, we just continue guessing.

```
 # Brute force the missing top 2 bytes (0x0000 to 0xFFFF)
    for i in range(0xFFFF + 1):
        guess_bytes = i.to_bytes(2, 'big') + observed
        x = int.from_bytes(guess_bytes, 'big')

        y = calculate_y_coordinate(x)
        if y is None:
            continue
```

Otherwise, we have found a valid point on the elliptic curve. We then create a new point, R, with the x and y values that *might* correspond to $n_t Q_x$ and $n_t Q_y$

```
try:
            R = ellipticcurve.Point(CURVE, x, y)

            # The backdoor magic: next_s = d * R
            next_s_point = R * d
            next_s = next_s_point.x()

            # Verify if this state produces the 'check' value
            next_o_point = Q_ecdsa * next_s
            next_o_bytes = next_o_point.x().to_bytes(32, 'big')
            next_o_truncated = next_o_bytes[-30:]

            # Simple progress indicator
            if i % 1000 == 0:
                sys.stdout.write(f"Scanning... guess: {i:04X}\r")
                sys.stdout.flush()

            if next_o_truncated == check:
                print(f"\nSuccess! Found match with guess {i:04X}")
                cloned_drbg = DualEcDrbg(p_const, q_const)
                cloned_drbg.seed(next_s)
                return cloned_drbg

        except Exception:
            continue
```

Next, we use the backdoor! We multiply R (which might be $n_t Q$), by d this value, stored in next_s_point, might be $n_t P$. On the next line, we grab the x-coordinate, which effectively computes $X(n_t P)$. This makes next_s_point $= n_{t+1}$ (a.k.a. the next internal state). From here, over the next three lines, we multiply the next internal state we calculated, n_{t+1}, and multiply it by Q, thus computing $X(n_{t+1}Q)$. That is converted to bytes, then truncated to 30-bytes, effectively computing $T(X(n_{t+1}Q))$. We have now calculated what *SHOULD* be the output at $t = n + 1$. We check if the output we generated is equal to the check we calculated at the beginning, and if it is, we were succesful! All that is left is to create a new drbg using the same P and Q we generated, and seed it with the next state. Otherwise, if the next state we calculated didn't match the one we measured, we just return to the beginning of the loop and keep guessing. If we never find a match, the operation fails and we return None.

Here is the full function:

```
def clone_dual_ec_drbg(drbg, p_const, q_const, d):
"""Attacks the DRBG instance to recover the internal state."""
observed = drbg.generate()
check = drbg.generate()

print(f"Target check value: {check.hex()}")

Q_ecdsa = ellipticcurve.Point(CURVE, q_const.x, q_const.y)

# Brute force the missing top 2 bytes (0x0000 to 0xFFFF)
for i in range(0xFFFF + 1):
    guess_bytes = i.to_bytes(2, 'big') + observed
    x = int.from_bytes(guess_bytes, 'big')

    y = calculate_y_coordinate(x)
    if y is None:
        continue

    try:
        R = ellipticcurve.Point(CURVE, x, y)

        # The backdoor magic: next_s = d * R
        next_s_point = R * d
        next_s = next_s_point.x()

        # Verify if this state produces the 'check' value
        next_o_point = Q_ecdsa * next_s
        next_o_bytes = next_o_point.x().to_bytes(32, 'big')
        next_o_truncated = next_o_bytes[-30:]

        # Simple progress indicator
        if i % 1000 == 0:
            sys.stdout.write(f"Scanning... guess: {i:04X}\r")
            sys.stdout.flush()

        if next_o_truncated == check:
            print(f"\nSuccess! Found match with guess {i:04X}")
            cloned_drbg = DualEcDrbg(p_const, q_const)
            cloned_drbg.seed(next_s)
            return cloned_drbg

    except Exception:
        continue

print("\nFailed to clone.")
return None
```