secp256k1, ECDSA and Schnorr

January 21, 2026

0.0.1 Point, Curve and Scalar classes to define Elliptic Curve of the form $y^2 = x^3+ax+b$

[1]:
```
from __future__ import annotations
from dataclasses import dataclass
from typing import Optional
```

[2]:
```
@dataclass(frozen=True)
class Scalar:
    """
    A scalar modulo `mod`.
    Use:
      - Scalar(x, p) for field elements (F_p)
      - Scalar(k, n) for group scalars (mod curve order)
    """
    value: int
    mod: int

    def __post_init__(self):
        object.__setattr__(self, "value", self.value % self.mod)

    def __int__(self) -> int:
        return self.value

    def inv(self) -> "Scalar":
        if self.value == 0:
            raise ZeroDivisionError("inverse of 0")
        # mod is prime for secp256k1 field p; for group order n also prime
        return Scalar(pow(self.value, self.mod - 2, self.mod), self.mod)

    def __neg__(self) -> "Scalar":
        return Scalar(-self.value, self.mod)

    def _coerce(self, other) -> "Scalar":
        if isinstance(other, Scalar):
            if other.mod != self.mod:
                raise ValueError(f"Mod mismatch: {self.mod} vs {other.mod}")
            return other
        if isinstance(other, int):
```

```
            return Scalar(other, self.mod)
        return NotImplemented

    def __add__(self, other) -> "Scalar":
        other = self._coerce(other)
        if other is NotImplemented:
            return NotImplemented
        return Scalar(self.value + other.value, self.mod)

    def __sub__(self, other) -> "Scalar":
        other = self._coerce(other)
        if other is NotImplemented:
            return NotImplemented
        return Scalar(self.value - other.value, self.mod)

    def __mul__(self, other) -> "Scalar":
        other = self._coerce(other)
        if other is NotImplemented:
            return NotImplemented
        return Scalar(self.value * other.value, self.mod)

    def __truediv__(self, other) -> "Scalar":
        other = self._coerce(other)
        if other is NotImplemented:
            return NotImplemented
        return self * other.inv()

    def __pow__(self, e: int) -> "Scalar":
        return Scalar(pow(self.value, e, self.mod), self.mod)
```

[3]:
```
@dataclass(frozen=True)
class Curve:
    p: int
    a: int
    b: int
    n: Optional[int] = None  # group order (optional but useful)

    def F(self, x: int) -> Scalar:
        return Scalar(x, self.p)

    def is_on_curve(self, P: "Point") -> bool:
        if P.is_infinity():
            return True
        x = self.F(P.x)
        y = self.F(P.y)
        return (y * y - (x * x * x + self.F(self.a) * x + self.F(self.b))).
 ↪value == 0
```

[4]:
```python
@dataclass(frozen=True)
class Point:
    curve: Curve
    x: Optional[int] = None
    y: Optional[int] = None

    @staticmethod
    def infinity(curve: Curve) -> "Point":
        return Point(curve, None, None)

    def is_infinity(self) -> bool:
        return self.x is None and self.y is None

    def __post_init__(self):
        if self.is_infinity():
            return
        if not (0 <= self.x < self.curve.p and 0 <= self.y < self.curve.p):
            raise ValueError("Point coordinates out of field range")
        if not self.curve.is_on_curve(self):
            raise ValueError("Point is not on the curve")

    def __neg__(self) -> "Point":
        if self.is_infinity():
            return self
        return Point(self.curve, self.x, (-self.y) % self.curve.p)

    def __add__(self, Q: "Point") -> "Point":
        P = self
        if P.curve != Q.curve:
            raise ValueError("Cannot add points from different curves")

        if P.is_infinity():
            return Q
        if Q.is_infinity():
            return P

        p = P.curve.p
        x1, y1 = P.x, P.y
        x2, y2 = Q.x, Q.y

        # P + (-P) = infinity
        if x1 == x2 and (y1 + y2) % p == 0:
            return Point.infinity(P.curve)

        F = P.curve.F

        if P != Q:
```

```
            lam = (F(y2) - F(y1)) / (F(x2) - F(x1))
        else:
            # tangent slope (3x^2 + a) / (2y)
            lam = (F(3) * F(x1) * F(x1) + F(P.curve.a)) / (F(2) * F(y1))

        x3 = lam * lam - F(x1) - F(x2)
        y3 = lam * (F(x1) - x3) - F(y1)

        return Point(P.curve, int(x3), int(y3))

    def __sub__(self, Q: "Point") -> "Point":
        return self + (-Q)

    def __rmul__(self, k: int) -> "Point":
        # scalar multiplication: k * P
        if not isinstance(k, int):
            return NotImplemented

        if self.is_infinity():
            return self

        # If curve order known, reduce k mod n for convenience
        if self.curve.n is not None:
            k %= self.curve.n

        if k == 0:
            return Point.infinity(self.curve)
        if k < 0:
            return (-k) * (-self)

        R = Point.infinity(self.curve)
        addend = self

        while k:
            if k & 1:
                R = R + addend
            addend = addend + addend
            k >>= 1

        return R
```

0.0.2 Defining the SECP256K1 Curve

Parameters are defined here: https://www.secg.org/sec2-v2.pdf

[5]:
```
SECP256K1_P = 2**256 - 2**32 - 977
SECP256K1_A = 0
```

```
SECP256K1_B = 7
SECP256K1_N = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141

# Generator (base point)
SECP256K1_GX =␣
↪55066263022277343669578718895168534326250603453777594175500187360389116729240
SECP256K1_GY =␣
↪32670510020758816978083085130507043184471273380659243275938904335757337482424

secp = Curve(p=SECP256K1_P, a=SECP256K1_A, b=SECP256K1_B, n=SECP256K1_N)
G = Point(secp, SECP256K1_GX, SECP256K1_GY)
INF = Point.infinity(secp)

assert secp.is_on_curve(G)
assert (SECP256K1_N * G).is_infinity()
assert (1 * G) == G
assert (2 * G) == (G + G)
assert (G + (-G)).is_infinity()
```

0.0.3 Generating a Public/Private Keypair

[6]:
```
import secrets

def random_private_key(curve: Curve) -> int:
    while True:
        d = secrets.randbelow(curve.n)
        # Uniform in [1, n-1]
        if 1 <= d < curve.n:
            return d

def public_key_from_private(d: int, G: Point) -> Point:
    if not (1 <= d < G.curve.n):
        raise ValueError("Invalid private key range")
    return d * G

d = random_private_key(secp)
print("Private Key:", hex(d))

P = public_key_from_private(d, G)
print(f"Public Key: ({hex(P.x)},{hex(P.y)})")
```

```
Private Key: 0x66db07ed5f81441c4c6a975cdebe9b128d1a9b02005e28084bb1050215c22b99
Public Key: (0xe8979a44449b78b9c7d159562bb658605b22d9a1455f4395d60b846f63c998d0,
0x278de61d7c39b12f1b10075dd568cc0fed6cf5cff5a00850d6ee9a8060384fda)
```

[7]:
```
# Serialization Format for Public Keys (SEC1, not a BIP)
x = P.x.to_bytes(32, "big")
```

```
y = P.y.to_bytes(32, "big")

uncompressed = b"\x04" + x + y
print("Uncompressed Public Key: 0x" + uncompressed.hex())

compressed_prefix = b"\x03" if (P.y & 1) else b"\x02"
compressed = compressed_prefix + x
print("Compressed Public Key: 0x" + compressed.hex())
```

```
Uncompressed Public Key: 0x04e8979a44449b78b9c7d159562bb658605b22d9a1455f4395d60
b846f63c998d0278de61d7c39b12f1b10075dd568cc0fed6cf5cff5a00850d6ee9a8060384fda
Compressed Public Key:
0x02e8979a44449b78b9c7d159562bb658605b22d9a1455f4395d60b846f63c998d0
```

0.0.4 Elliptic Curve Digital Signature Algorithm (ECDSA)

ECDSA Signing Algorithm $s = k^{-1}(z + rd) \bmod n$

- s = Signature's s-point
- k = Deterministic Nonce
  - Can't be reused or Private Key d leaks
  - Deterministic digital signature RFC6979 used by Bitcoin to protect us from this, where $k = fn(d, z)$
- z = Message to be signed (hashed via sigops)
- r = Signature's r-part, x-coordinate of $R = kG$
- d = Private Key

The Public Key $P = dG$

The actual signature is (r, s)

[8]:
```
import hmac
import hashlib
from typing import Tuple

def bits2int(b: bytes, qlen: int) -> int:
    """RFC6979 bits2int: take leftmost qlen bits of b, interpret as int."""
    i = int.from_bytes(b, "big")
    blen = len(b) * 8
    if blen > qlen:
        i >>= (blen - qlen)
    return i

def int2octets(x: int, rolen: int) -> bytes:
    """RFC6979 int2octets: encode int x as rolen bytes big-endian."""
    return x.to_bytes(rolen, "big")

def bits2octets(b: bytes, q: int, rolen: int) -> bytes:
    """RFC6979 bits2octets."""
```

```
    z1 = bits2int(b, q.bit_length())
    z2 = z1 % q
    return int2octets(z2, rolen)

def rfc6979_generate_nonce(d: int, z: int, n: int) -> int:
    """
    Deterministically generate ECDSA nonce k using RFC6979 + HMAC-SHA256.

    Inputs:
      d: private key (1..n-1)
      z: message hash as integer (usually 256-bit, e.g. sha256(msg))
      n: curve order

    Output:
      k in [1..n-1]
    """
    if not (1 <= d < n):
        raise ValueError("Invalid private key d")
    qlen = n.bit_length()
    holen = 32  # SHA256 output length in bytes
    rolen = (qlen + 7) // 8

    # Convert inputs
    bx = int2octets(d, rolen)
    bz = bits2octets(z.to_bytes(32, "big"), n, rolen)

    # Step B
    V = b"\x01" * holen
    # Step C
    K = b"\x00" * holen

    def hmac_sha256(key: bytes, data: bytes) -> bytes:
        return hmac.new(key, data, hashlib.sha256).digest()

    # Step D
    K = hmac_sha256(K, V + b"\x00" + bx + bz)
    # Step E
    V = hmac_sha256(K, V)
    # Step F
    K = hmac_sha256(K, V + b"\x01" + bx + bz)
    # Step G
    V = hmac_sha256(K, V)

    # Step H
    while True:
        T = b""
        while len(T) < rolen:
```

```
            V = hmac_sha256(K, V)
            T += V

        k = bits2int(T, qlen)
        if 1 <= k < n:
            return k

        K = hmac_sha256(K, V + b"\x00")
        V = hmac_sha256(K, V)
```

[9]:
```
def ecdsa_sign(z: int, d: int, G: Point) -> Tuple[int, int]:
    """
    Sign a message hash interpreted as integer z.
    Returns (r, s) as integers in [1, n-1].
    """
    n = G.curve.n

    if n is None:
        raise ValueError("Curve must have order n")
    if not (1 <= d < n):
        raise ValueError("Invalid private key d")

    z = z % n # message should also be mod n

    while True:
        # Random Nonce (fine for learning but not production because what if␣
 ↪nonce reused because of randomness?)
        # k = secrets.randbelow(n)
        # RFC6979 Deterministic Nonce
        k = rfc6979_generate_nonce(d, z, n)

        R = k * G
        if R.is_infinity(): continue

        r = R.x % n
        if r == 0: continue

        k_inv = Scalar(k, n).inv().value
        s = (k_inv * (z + r * d)) % n
        if s == 0: continue

        # This is NOT a point on the curve, just a tuple of signature values
        return (r, s)
```

[10]:
```
import hashlib

msg = b"Satoshi Nakamoto is everywhere"
```

```
z = int.from_bytes(hashlib.sha256(msg).digest(), "big")

ecdsa_sign(z, d, G)
```

[10]:
```
(100152184108366984890303104940864049914791486390339255436185776490785798557454,
 113500030669257307105211891168802504449007931918099992694904885078096013136713)
```

[11]:
```
# As we are using deterministic nonce, every sign will be exactly the same
assert ecdsa_sign(1, d, G) == ecdsa_sign(1, d, G)
```

ECDSA Verification Algorithm

1. Verify $r \in [1, n-1]$, $s \in [1, n-1]$
2. $w = s^{-1} \bmod n$ (exists because $s \neq 0$
3. $u_1 = zw \bmod n$ $u_2 = rw \bmod n$
4. $X = u_1 G + u_2 P$ (reject if $X = \mathcal{O}$, point at infinity)
5. $v = X_x \bmod n$

The signature is valid iff $v = r$

Proof why ECDSA verfication works Motivation: Reconstruct kG without knowing k

$s = k^{-1}(z+rd) \bmod n \Rightarrow ks = z+rd \bmod n \Rightarrow k = s^{-1}(z+rd) \bmod n \Rightarrow k = w(z+rd) \bmod n \Rightarrow k = zw+rdw \bmod n \Rightarrow k = u_1 + u_2 d \bmod n \Rightarrow kG = (u_1 + u_2 d)G \bmod n \Rightarrow kG = u_1 G + u_2 (dG) \bmod n \Rightarrow kG = u_1 G + u_2 P \bmod n$

Verifier computes the RHS of this equation. If you recall, from the signing steps, $R = kG$. We match the x-coordinate to check the validity of the signature.

$R_x = X_x$

[12]:
```
def ecdsa_verify(z: int, sig: Tuple[int, int], P: Point, G: Point) -> bool:
    """
    Verify ECDSA signature (r,s) for message hash integer z against public key␣
 ↪P.
    """
    r, s = sig
    n = G.curve.n
    if n is None:
        raise ValueError("Curve must have order n")

    if P.is_infinity() or (P.curve != G.curve):
        return False
    if not (1 <= r < n and 1 <= s < n):
        return False

    z = z % n # message should be mod n
```

```
    s_inv = Scalar(s, n).inv().value

    u1 = (z * s_inv) % n
    u2 = (r * s_inv) % n

    X = (u1 * G) + (u2 * P)
    if X.is_infinity():
        return False

    return (X.x % n) == r
```

[13]:
```
z = 1
ecdsa_verify(z, ecdsa_sign(z, d, G), P, G)
```

[13]:
```
True
```

ECDSA Nonce Reuse (Issue #1) If one use the same k again for another ECDSA signature, then private key d is leaked.

$s = k^{-1}(z + rd) \bmod n \Rightarrow sk = z + rd \bmod n$

Now, suppose you sign s_1 and s_2 with the same nonce k.

$s_1 k = z_1 + rd \bmod n \; s_2 k = z_2 + rd \bmod n$

Subtract:

$(s_1 - s_2)k = (z_1 - z_2) \bmod n \implies k = (z_1 - z_2)(s_1 - s_2)^{-1} \bmod n$

And once k is known,

Private Key $d = (sk - z)r^{-1} \bmod n$

ECDSA Signature Malleability (Issue #2) A valid ECDSA signature (r, s) can be transformed into another valid one:

$(r, s) = (r, n - s)$

Both, verify.

[14]:
```
def malleable_execution(z):
    sig = ecdsa_sign(z, d, G)
    r, s = sig

    n = secp.n

    # Low s
    s1 = s
    if s1 > n // 2:
        s1 = n - s1
    sig_low_s = (r, s1)
```

```
    # High s
    s2 = s
    if s2 <= n // 2:
        s2 = n - s2
    sig_high_s = (r, s2)

    print("Malleable signatures for same 'z' off low/high s, z =", z)
    print(hex(sig_low_s[0]), hex(sig_low_s[1]), "lo")
    print(hex(sig_high_s[0]), hex(sig_high_s[1]), "hi")

    if sig == sig_low_s: print("Original: low s")
    else: print("Original: high s")

    return (sig_low_s, sig_high_s)

malleable_execution(secp.n // 2);
```

```
Malleable signatures for same 'z' off low/high s, z =
57896044618658097711785492504343953926418782139537452191302581570759080747168
0xfaff26648e90600eda1dbfd7ce2f0012a15e982614784d578e3c646533a7ae44
0x296b54b53b9940fe94bcb47231135d9b55668595861431e3935b48fec62faf78 lo
0xfaff26648e90600eda1dbfd7ce2f0012a15e982614784d578e3c646533a7ae44
0xd694ab4ac466bf016b434b8dceeca2636548575129346e582c77158e0a0691c9 hi
Original: low s
```

[15]:
```
z = 1

# Generating both the low-s and high-s signatures for the same message
sig_low_s, sig_high_s = malleable_execution(z)

# Low & High s - both verifies
print("Low-s Signature Verification:", ecdsa_verify(z, sig_low_s, P, G))
print("High-s Signature Verification:", ecdsa_verify(z, sig_high_s, P, G))
```

```
Malleable signatures for same 'z' off low/high s, z = 1
0x972e74ee34136e8f15bbdaeff1522501fc15536271872a0a6c011a4fd1f98bdb
0x7f11c7a36d4fb7752c1b6b3e20e28ee8b5a8caf1f4a237ce24b34805cfcf8b28 lo
0x972e74ee34136e8f15bbdaeff1522501fc15536271872a0a6c011a4fd1f98bdb
0x80ee385c92b0488ad3e494c1df1d7116050611f4baa6686d9b1f16870066b619 hi
Original: low s
Low-s Signature Verification: True
High-s Signature Verification: True
```

0.0.5 Reasons Bitcoin moved away from ECDSA (towards Schnorr / Taproot)

1. Transaction ID Malleability: ***(pre- SegWit)*** Before SegWit, the transaction ID was computed as: $txid = sha256d(\text{serialized transaction})$

and the serialization included `scriptSig`, which contains the signature bytes. So any mutation to signature bytes could change the txid even though the spend remained valid.

a) DER encoding malleability (non-canonical encodings)

- The same mathematical ECDSA signature (r, s) could be encoded in multiple valid (but non-canonical) ways in DER
- This allowed third parties to mutate signature bytes, and therby mutate txid

It was fixed by BIP66 (Strict DER encoding) (consensus rule).

b) s-value malleability (high-s / low-s) For any valid signature (r,s), this is also valid: (r, n - s)

This produces a different signature that still verifies, again allowing txid mutation pre-SegWit.

Mitigated by enforcing low-S signatures as policy/standardness (often discussed in context of BIP62), and later enforced more strictly for SegWit spends.

c) SegWit solved it structurally SegWit moved signatures into the witness, so: - txid no longer commits to signatures - malleability no longer breaks txid-based chaining

Fixed by SegWit (BIP141) at the protocol level.

2. Nonce Fragility / Leaking Private Key (discussed above)

3. No clean key aggregation - not Linear like Taproot

4. Weird signature malleability vectors

5. No batch verification possible like Schnorr

0.0.6 Schnorr Signature

Schnorr Signing Algorithm

1. $P = dG \bmod n$ If P_y is odd, flip $d = n - d$
2. Generate nonce k deterministically. $k = fn(d, m)$ by using BIP340 tagged hashes (and optional aux randomness).
3. $R = kG \bmod n$ If R_y is odd, flip nonce $k = n - k$
4. Compute Challenge e $e = H(R_x, P_x, m) \bmod n$
5. Signature $s = k + ed \bmod n$

The signature is (R_x, s).

- Public Key is stored as 32 byte `x`-coordinate of P, because `y`-coordinate can be determined because it's even.
- BIP340 Serialization: 64 byte => rx(32) || s(32)

[16]:
```
import hashlib
from typing import Tuple
```

[17]:
```python
import hashlib
from enum import Enum

class BIP340Tag(str, Enum):
    AUX = "BIP0340/aux"
    NONCE = "BIP0340/nonce"
    CHALLENGE = "BIP0340/challenge"

def tagged_hash(tag: BIP340Tag, msg: bytes) -> bytes:
    """
    BIP340 tagged hash:
      SHA256(SHA256(tag) || SHA256(tag) || msg)
    """
    tag_hash = hashlib.sha256(tag.value.encode("ascii")).digest()
    return hashlib.sha256(tag_hash + tag_hash + msg).digest()
```

[18]:
```python
from typing import Optional, Tuple

def schnorr_sign(z: int, d: int, G: Point, aux: Optional[bytes] = None) ->␣
 ↪Tuple[int, int]:
    """
    BIP340-style Schnorr sign on secp256k1.

    Inputs:
      z: message (expects 32-byte message in BIP340; here we encode int -> 32␣
 ↪bytes)
      d: private key integer (1..n-1)
      G: generator point

    Output:
      (rx, s) where:
        rx: x-coordinate of R (int, 0..p-1)
        s : scalar (int, 0..n-1)
    """
    n = G.curve.n
    if n is None:
        raise ValueError("Curve must have order n")
    if not (1 <= d < n):
        raise ValueError("Invalid private key d")

    # message as 32 bytes (BIP340 expects 32-byte msg)
    m = (z % (1 << 256)).to_bytes(32, "big")

    def enforce_even_y_when_multiplied(k: int, G: Point) -> Tuple[int, Point]:
        """
        Compute R = k*G.
        If R has odd y, negate both: k <- n-k, R <- -R so that Ry becomes even.
```

```
        """
        R = k * G
        if R.is_infinity():
            raise RuntimeError("Invalid point: infinity")
        if (R.y & 1) == 1:  # odd y
            return (n - k) % n, -R
        return k % n, R

    # 1) Public key and "even-y" convention:
    #    P = dG. If Py is odd, use d' = n-d and P' = -P (same x, even y).
    d_eff, P = enforce_even_y_when_multiplied(d, G)
    px = P.x.to_bytes(32, "big")

    d_eff_bytes = d_eff.to_bytes(32, "big")  # 0..n-1 fits in 32 bytes

    # 2) Deterministic nonce k with tagged hash and optional aux randomness
    if aux is None:
        aux = b"\x00" * 32
    if len(aux) != 32:
        raise ValueError("aux must be exactly 32 bytes")

    aux_hash = tagged_hash(BIP340Tag.AUX, aux)
    t = bytes(a ^ b for a, b in zip(d_eff_bytes, aux_hash))

    k0 = int.from_bytes(tagged_hash(BIP340Tag.NONCE, t + px + m), "big") % n
    if k0 == 0:
        raise RuntimeError("Unlucky: derived k is 0; choose different aux or␣
↪message")

    # 3) R = kG, enforce even-y for R
    k, R = enforce_even_y_when_multiplied(k0, G)
    rx_bytes = R.x.to_bytes(32, "big")

    # 4) Challenge e = H(rx || px || m) mod n
    e = int.from_bytes(tagged_hash(BIP340Tag.CHALLENGE, rx_bytes + px + m),␣
↪"big") % n

    # 5) s = k + e*d_eff mod n
    s = (k + e * d_eff) % n

    # Signature is (rx, s) with rx as the x-coordinate integer
    return (R.x, s)
```

[19]:
```
schnorr_sign(1, d, G)
```

[19]:
```
(39836057919412435847140014597710287468357009319972349163940291975844716737627,
 43971663946341611215525666085042447717984519607815484791784896323587200557401)
```

[20]:
```
z = 1
assert schnorr_sign(z, d, G) == schnorr_sign(z, d, G)

aux = secrets.token_bytes(32)
assert schnorr_sign(z, d, G, aux) == schnorr_sign(z, d, G, aux)

aux2 = secrets.token_bytes(32)
assert schnorr_sign(z, d, G, aux) != schnorr_sign(z, d, G, aux2)
```

Schnorr Verification Algorithm **Given:** - Curve generator point G, group order n, field prime p - Message m (32 bytes) — in our code we map your `z:int` to $m = \text{bytes32}(z)$ - X-only public key $p_x \in [0, p-1]$ (32-byte x-coordinate) - Signature $\sigma = (r_x, s)$ where: - $r_x \in [0, p-1]$ is the x-coordinate of the nonce point R - $s \in [0, n-1]$

---

1. Verify $r_x \in [0, p-1]$, $s \in [0, n-1]$
2. **Lift the x-only public key**

   Compute $P = \text{lift\_x}(p_x)$, where lift_x returns the unique curve point with x-coordinate p_x and **even** y-coordinate:

   - Compute $y^2 \equiv p_x^3 + 7 \pmod{p}$
   - Find $y = \sqrt{p_x^3 + 7} \pmod{p}$ (reject if no square root exists)
   - If y is odd, set $y \leftarrow p - y$
   - Output $P = (p_x, y)$

   Reject if P does not exist.
3. **Compute the challenge**

   $e = H_{\text{challenge}}(\text{bytes32}(r_x) \,||\, \text{bytes32}(p_x) \,||\, m) \bmod n$

   where:

   $H_{\text{challenge}}(x) = \text{tagged\_hash}(\texttt{"BIP0340/challenge"}, x)$
4. **Reconstruct R from the verification equation**

   Schnorr verification equation:

   $sG \stackrel{?}{=} R + eP$

   Rearranged to compute R':

   $R' = sG - eP$

   Reject if $R' = \mathcal{O}$ (point at infinity).
5. **Check canonical conditions**

   The signature is valid iff:

   - R'_y is even
   - $R'_x = r_x$

Proof why Schnorr Verification works From signing, $s = k + ed \bmod n$

Multiplying G on both sides and substituting $P = dG$ and $R = kG$,

$\implies sG = (k + ed)G \implies sG = kG + e(dG) \implies sG = R + eP \implies R = sG - eP$
$\implies R = sG + (n - e)P$

So $R'_x = r_x$ and (by construction in signing) R'_y is even.

[21]:
```
from typing import Optional, Tuple

def sqrt_mod_p_secp256k1(a: int, p: int) -> Optional[int]:
    """
    secp256k1 prime satisfies p % 4 == 3, so sqrt can be computed by:
      y = a^((p+1)//4) mod p
    Returns y if it exists, else None.
    """
    a %= p
    if a == 0:
        return 0
    y = pow(a, (p + 1) // 4, p)
    if (y * y) % p != a:
        return None
    return y

def lift_x(curve: Curve, x: int) -> Optional[Point]:
    """
    BIP340 lift_x: given x, return the curve point (x,y) with EVEN y, if it␣
 ↪exists.
    """
    p = curve.p
    if not (0 <= x < p):
        return None

    # y^2 = x^3 + 7 mod p  (secp256k1: a=0, b=7)
    rhs = (pow(x, 3, p) + curve.b) % p
    y = sqrt_mod_p_secp256k1(rhs, p)
    if y is None:
        return None

    # choose even y
    if y & 1:
        y = p - y

    try:
        return Point(curve, x, y)
    except ValueError:
        return None
```

[22]:
```python
def schnorr_verify(z: int, px: int, sig: Tuple[int, int], G: Point) -> bool:
    """
    Verify BIP340-style Schnorr signature for message `z` (int mapped to 32␣
 ↪bytes),
    x-only public key `px`, and signature (rx, s).
    """
    n = G.curve.n
    p = G.curve.p
    if n is None:
        raise ValueError("Curve must have order n")

    rx, s = sig

    # 1) Range checks
    if not (0 <= rx < p):
        return False
    if not (0 <= s < n):
        return False

    # 2) Lift x-only pubkey to even-y point
    P = lift_x(G.curve, px)
    if P is None:
        return False

    # m = bytes32(z)
    m = (z % (1 << 256)).to_bytes(32, "big")
    rx_bytes = rx.to_bytes(32, "big")
    px_bytes = px.to_bytes(32, "big")

    # 3) Challenge e
    e = int.from_bytes(
        tagged_hash(BIP340Tag.CHALLENGE, rx_bytes + px_bytes + m),
        "big",
    ) % n

    # 4) R' = sG - eP = sG + (n-e)P
    R = (s * G) + ((n - e) * P)

    if R.is_infinity():
        return False

    # 5) Check canonical conditions
    if (R.y & 1) == 1:      # must be even-y
        return False
    if R.x != rx:
        return False
```

```
    return True
```

[23]:
```
z = 1
schnorr_verify(z, P.x, schnorr_sign(z, d, G), G)
```

[23]: True

Signatures with different aux values verify

[24]:
```
z = 1
aux1 = secrets.token_bytes(32)
aux2 = secrets.token_bytes(32)

sig1 = schnorr_sign(z, d, G, aux1)
sig2 = schnorr_sign(z, d, G, aux2)
assert sig1 != sig2

assert schnorr_verify(z, P.x, sig1, G)
assert schnorr_verify(z, P.x, sig2, G)
```

Check ECDSA-style low/high-s malleability does NOT apply to Schnorr (BIP340 style)

[25]:
```
d = random_private_key(secp)
P = public_key_from_private(d, G)

z = 1
sig = rx, s = schnorr_sign(z, d, G)

print("verify(sig):", schnorr_verify(z, P.x, sig, G))

# Attempt ECDSA-style malleation: (rx, n-s)
n = secp.n
sig_flip = (rx, (n - s) % n)

print("verify(sig_flip):", schnorr_verify(z, P.x, sig_flip, G))
```

```
verify(sig): True
verify(sig_flip): False
```

0.0.7 Footnotes

- **Two algebra "worlds" exist:**
  - Field arithmetic in $\mathbb{F}_p$ (mod p): used for point addition formulas; modular inverses exist for all non-zero elements
  - Group arithmetic on the curve $E(\mathbb{F}_p)$: point addition and scalar multiplication
- **A point does not have a "modular inverse".** The inverse defined for a point is its **group inverse**: $-P = (x, -y \bmod p)$ So that, $P + (-P) = \mathcal{O}$, point at infinity

- **Division by a point is not defined.** Expressions like P^{-1} or Q/P do not make sense in elliptic curve group operations
- **Scalar inverses exist modulo n:** $a^{-1} \bmod n$ exists iff $a \neq 0$. This is why ECDSA requires modular inverses (of k and s)
- **ECDSA nonce fragility is catastrophic:** Random nonce k is okay for learning, but unsafe in production because:
  - If the same k is reused even once, the private key d can be recovered
  - Biased or partially leaked nonces can also leak d (lattice/HNP style attacks)
- **RFC6979 exists to remove RNG dependence for ECDSA.**
  It makes nonce generation deterministic: $k = f(d, z)$ reducing risk from weak system randomness
- **ECDSA "low-s / high-s" malleability:**
  If (r, s) is valid, then $(r, n - s)$ is also valid. This is called **ECDSA signature malleability / s-value malleability**
- **DER signature encoding was not invented by a BIP**
  Bitcoin inherited ASN.1 DER ECDSA encoding conventions (OpenSSL/X9.62-style)
- **BIP66 enforced strict DER encoding (consensus)**
  Multiple byte encodings that parsed to the same (r, s) used to be accepted; strict DER made the encoding canonical
- **Transaction malleability was a third-party mutation issue.**
  Before SegWit, the signature bytes lived in `scriptSig` $\rightarrow$ changing valid encodings could change the `txid` without needing the user to re-sign
- **SegWit was not only a "malleability fix"**, but it structurally fixed txid malleability by moving signatures into witness, enabling:
  - stable `txid`s for chained protocols (e.g. Lightning)
  - effective block capacity increase (weight)
  - a cleaner upgrade path (witness versions)
- **Schnorr signatures (BIP340) remove a lot of encoding pain:**
  - fixed 64-byte signatures (r_x, s)
  - no DER encoding required
  - no ECDSA-style low/high-s malleability in the same way
- **BIP340 uses x-only pubkeys + even-y convention:**
  - Public keys are represented by $x(P)$ only.
  - The full point is recovered via lift_x, choosing the **even** y solution. This makes signatures canonical and avoids ambiguity.
- **Tagged hashes are fixed constants in BIP340** for domain separation:
  - `BIP0340/aux`
  - `BIP0340/nonce`
  - `BIP0340/challenge`

- **`aux` randomness is signer-only and not needed for verification.**
  It is optional 32-byte CSPRNG input to harden nonce derivation against side-channels; the verifier never receives or checks it.
- **Why Schnorr enables MuSig-like constructions:**
  Schnorr has linear structure: $s = k + ed \pmod{n}$ which is the algebraic foundation of multisignatures and key aggregation — but real MuSig requires nonce coordination and rogue-key protections (so it can't be done by just "adding signatures" naively).