Colliding Heritage

```python
class MD5chnorr:

    def __init__(self):
        # while True:
        #     self.q = getPrime(128)
        #     self.p = 2*self.q + 1
        #     if isPrime(self.p):
        #         break
        self.p = 0x16dd987483c08aefa88f28147702e51eb
        self.q = (self.p - 1) // 2
        self.g = 3
        self.x = randbelow(self.q)
        self.y = pow(self.g, self.x, self.p)

    def H(self, msg):
        return bytes_to_long(md5(msg).digest()) % self.q

    def sign(self, msg):
        k = self.H(msg + long_to_bytes(self.x))
        r = pow(self.g, k, self.p) % self.q
        e = self.H(long_to_bytes(r) + msg)
        s = (k - self.x * e) % self.q
        return (s, e)

    def verify(self, msg, sig):
        s, e = sig
        if not (0 < s < self.q):
            return False
        if not (0 < e < self.q):
            return False
        rv = pow(self.g, s, self.p) * pow(self.y, e, self.p) % self.p % self.q
        ev = self.H(long_to_bytes(rv) + msg)
        return ev == e
```

Biased Heritage

```python
class SHA256chnorr:

    def __init__(self):
        # while True:
        #     self.q = getPrime(512)
        #     self.p = 2*self.q + 1
        #     if isPrime(self.p):
        #         break
        self.p = 0x184e26a581fca2893b2096528eb6103ac03f60b023e1284ebda3ab24ad9a9fe0e
        self.q = (self.p - 1) // 2
        self.g = 3
        self.x = randbelow(self.q)
        self.y = pow(self.g, self.x, self.p)

    def H(self, msg):
        return bytes_to_long(2 * sha256(msg).digest()) % self.q

    def sign(self, msg):
        k = self.H(msg + long_to_bytes(self.x))
        r = pow(self.g, k, self.p) % self.q
        e = self.H(long_to_bytes(r) + msg)
        s = (k - self.x * e) % self.q
        return (s, e)

    def verify(self, msg, sig):
        s, e = sig
        if not (0 < s < self.q):
            return False
        if not (0 < e < self.q):
            return False
        rv = pow(self.g, s, self.p) * pow(self.y, e, self.p) % self.p % self.q
        ev = self.H(long_to_bytes(rv) + msg)
        return ev == e
```

```python
if __name__ == '__main__':
    signal.alarm(30)
    main()
```

```python
class MD5chnorr:

    def __init__(self):
        # while True:
        #     self.q = getPrime(128)
        #     self.p = 2*self.q + 1
        #     if isPrime(self.p):
        #         break
        self.p = 0x16dd987483c08aefa88f28147702e51eb
        self.q = (self.p - 1) // 2
        self.g = 3
        self.x = randbelow(self.q)
        self.y = pow(self.g, self.x, self.p)

    def H(self, msg):
        return bytes_to_long(md5(msg).digest()) % self.q

    def sign(self, msg):
        k = self.H(msg + long_to_bytes(self.x))
        r = pow(self.g, k, self.p) % self.q
        e = self.H(long_to_bytes(r) + msg)
        s = (k - self.x * e) % self.q
        return (s, e)

    def verify(self, msg, sig):
        s, e = sig
        if not (0 < s < self.q):
            return False
        if not (0 < e < self.q):
            return False
        rv = pow(self.g, s, self.p) * pow(self.y, e, self.p) % self.p % self.q
        ev = self.H(long_to_bytes(rv) + msg)
        return ev == e
```

Schnorr signature

- $M \in \{0,1\}^*$, the set of finite bit strings
- $s, e, e_v \in \mathbb{Z}_q$, the set of congruence classes modulo q
- $x, k \in \mathbb{Z}_q^\times$, the multiplicative group of integers modulo q (for prime q, $\mathbb{Z}_q^\times = \mathbb{Z}_q \setminus \overline{0}_q$)
- $y, r, r_v \in G$.

Key generation [edit]

- Choose a private signing key, x, from the allowed set.
- The public verification key is $y = g^x$.

Signing [edit]

To sign a message, M:

- Choose a random k from the allowed set.
- Let $r = g^k$.
- Let $e = H(r \parallel M)$, where $\parallel$ denotes concatenation and r is represented as a bit string.
- Let $s = k - xe$.

The signature is the pair, (s, e).

Note that $s, e \in \mathbb{Z}_q$; if $q < 2^{160}$, then the signature representation can fit into 40 bytes.

Verifying [edit]

- Let $r_v = g^s y^e$
- Let $e_v = H(r_v \parallel M)$

If $e_v = e$ then the signature is verified.

Key leakage from nonce reuse [edit]

Just as with the closely related signature algorithms DSA, ECDSA, and ElGamal, reusing the secret nonce value k on two Schnorr signatures of different messages will allow observers to recover the private key.[2] In the case of Schnorr signatures, this simply requires subtracting s values:

$$s' - s = (k' - k) - x(e' - e).$$

Colliding Heritage

Biased Heritage

If $k' = k$ but $e' \neq e$ then x can be simply isolated. In fact, even slight biases in the value k or partial leakage of k can reveal the private key, after collecting sufficiently many signatures and solving the hidden number problem.[2]

Credit: https://en.wikipedia.org/wiki/Schnorr_signature

Colliding Heritage

$$\begin{cases} s_1 = \boxed{k} - x\boxed{e_1} \mod q \\ s_2 = \boxed{k} - x\boxed{e_2} \mod q \end{cases} \rightarrow s_1 - s_2 = xe_2 - xe_1 \mod q \rightarrow x = (s_1 - s_2) \cdot (e_2 - e_1) \mod q$$

md5 collision

```
def H(self, msg):
    return bytes_to_long(md5(msg).digest()) % self.q

def sign(self, msg):
    k = self.H(msg + long_to_bytes(self.x))
    r = pow(self.g, k, self.p) % self.q
    e = self.H(long_to_bytes(r) + msg)
    s = (k - self.x * e) % self.q
    return (s, e)
```

```
$ echo 'd131dd02c5e6eec4 693d9a0698aff95c 2fcab58712467eab 4004583eb8fb7f89
55ad340609f4b302 83e488832571415a 085125e8f7cdc99f d91dbdf280373c5b
d8823e3156348f5b ae6dacd436c919c6 dd53e2b487da03fd 02396306d248cda0
e99f33420f577ee8 ce54b67080a80d1e c69821bcb6a88393 96f9652b6ff72a70' | xxd -r -p | md5sum
79054025255fb1a26e4bc422aef54eb4  -
```

```
$ echo 'd131dd02c5e6eec4 693d9a0698aff95c 2fcab50712467eab 4004583eb8fb7f89
55ad340609f4b302 83e4888325f1415a 085125e8f7cdc99f d91dbd7280373c5b
d8823e3156348f5b ae6dacd436c919c6 dd53e23487da03fd 02396306d248cda0
e99f33420f577ee8 ce54b67080280d1e c69821bcb6a88393 96f965ab6ff72a70' | xxd -r -p | md5sum
79054025255fb1a26e4bc422aef54eb4  -
```

$ echo 'd131dd02c5e6eec4 693d9a0698aff95c 2fcab58712467eab 4004583eb8fb7f89
55ad340609f4b302 83e488832571415a 085125e8f7cdc99f d91dbdf280373c5b
d8823e3156348f5b ae6dacd436c919c6 dd53e2b487da03fd 02396306d248cda0
e99f33420f577ee8 ce54b67080a80d1e c69821bcb6a88393 96f9652b6ff72a70
0123456789abcdef 0123456789abcdef 0123456789abcdef 0123456789abcdef' | xxd -r -p | md5sum
5e1b7ce6b54d7313d856753d9d48f17c -

$ echo 'd131dd02c5e6eec4 693d9a0698aff95c 2fcab50712467eab 4004583eb8fb7f89
55ad340609f4b302 83e4888325f1415a 085125e8f7cdc99f d91dbd7280373c5b
d8823e3156348f5b ae6dacd436c919c6 dd53e23487da03fd 02396306d248cda0
e99f33420f577ee8 ce54b67080280d1e c69821bcb6a88393 96f965ab6ff72a70
0123456789abcdef 0123456789abcdef 0123456789abcdef 0123456789abcdef' | xxd -r -p | md5sum
5e1b7ce6b54d7313d856753d9d48f17c -

Biased Heritage?

SHA256 is collision-resistant

```python
def H(self, msg):
    return bytes_to_long(2 * sha256(msg).digest()) % self.q

def sign(self, msg):
    k = self.H(msg + long_to_bytes(self.x))
    r = pow(self.g, k, self.p) % self.q
    e = self.H(long_to_bytes(r) + msg)
    s = (k - self.x * e) % self.q
    return (s, e)
```

Potential ECDSA disasters: Biased nonce

[Boneh Venkatesan 96], [Howgrave-Graham Smart 2001], [Nguyen Shparlinski 2003]

Potential pitfall #3

k must be generated uniformly at random,
or we can use many signatures to compute the private key d.

$$\left.\begin{aligned} k_1 - s_1^{-1} r_1 d - s_1^{-1} h_1 &\equiv 0 \bmod n \\ k_2 - s_2^{-1} r_2 d - s_2^{-1} h_2 &\equiv 0 \bmod n \\ &\vdots \\ k_m - s_m^{-1} r_m d - s_m^{-1} h_m &\equiv 0 \bmod n \end{aligned}\right\} \to \text{lattice attacks} \to d$$

If the k_i are *small*, system of equations likely has unique solution and lattice techniques can find d.

Credit: Biased Nonce Sense: Lattice attacks against weak ECDSA signatures in the wild, Nadia Heninger et al.

Formulating ECDSA as a hidden number problem

[Howgrave-Graham Smart 2001], [Nguyen Shparlinski 2003]

We have a system of equations in unknowns $k_1, \ldots, k_m, d$:

$$s_1 = k_1 - xe_1 \mod q$$

$$s_2 = k_2 - xe_2 \mod q$$

$$a_i = s_i$$
$$t_i = e_i$$
$$d = x$$
$$n = q$$

$$
\begin{aligned}
k_1 - t_1 d - a_1 &\equiv 0 \bmod n \\
k_2 - t_2 d - a_2 &\equiv 0 \bmod n \\
&\vdots \\
k_m - t_m d - a_m &\equiv 0 \bmod n
\end{aligned}
$$

We assume the k_i are small.

This is an instance of the *hidden number problem* [Boneh Venkatesan 96].

Credit: Biased Nonce Sense: Lattice attacks against weak ECDSA signatures in the wild, Nadia Heninger et al.

Formulating ECDSA as a hidden number problem

[Howgrave-Graham Smart 2001], [Nguyen Shparlinski 2003]

We have a system of equations in unknowns $k_1, \ldots, k_m, d$:

$$
\begin{aligned}
k_1 - t_1 d - a_1 &\equiv 0 \bmod n \\
k_2 - t_2 d - a_2 &\equiv 0 \bmod n \\
&\vdots \\
k_m - t_m d - a_m &\equiv 0 \bmod n
\end{aligned}
$$

$$k' = k \,||\, k = 2^{256} k + k = (2^{256}+1)k$$

$$
\begin{aligned}
s_i &= k_i' - x \cdot e_i \mod q \\
s_i &= (2^{256}+1)k_i - x \cdot e_i \mod q \\
M s_i &= k_i - x \cdot M e_i \mod q
\end{aligned}
$$

$$M = (2^{256}+1)^{-1} \mod q$$

$\boxed{\text{We assume the } k_i \text{ are small.}}$

This is an instance of the *hidden number problem* [Boneh Venkatesan 96].

Credit: Biased Nonce Sense: Lattice attacks against weak ECDSA signatures in the wild, Nadia Heninger et al.

Solving the hidden number problem with CVP

Input:

$$k_1 - t_1 d - a_1 \equiv 0 \bmod n$$
$$\vdots$$
$$k_m - t_m d - a_m \equiv 0 \bmod n$$

in unknowns $k_1, \ldots, k_m, d$, where $|k_i| < B$.

Construct the lattice basis

$$M = \begin{bmatrix} n & & & & \\ & n & & & \\ & & \ddots & & \\ & & & n & \\ t_1 & t_2 & \ldots & t_m & \end{bmatrix}$$

Solve CVP with target vector $v_t = (a_1, a_2, \ldots, a_m)$.

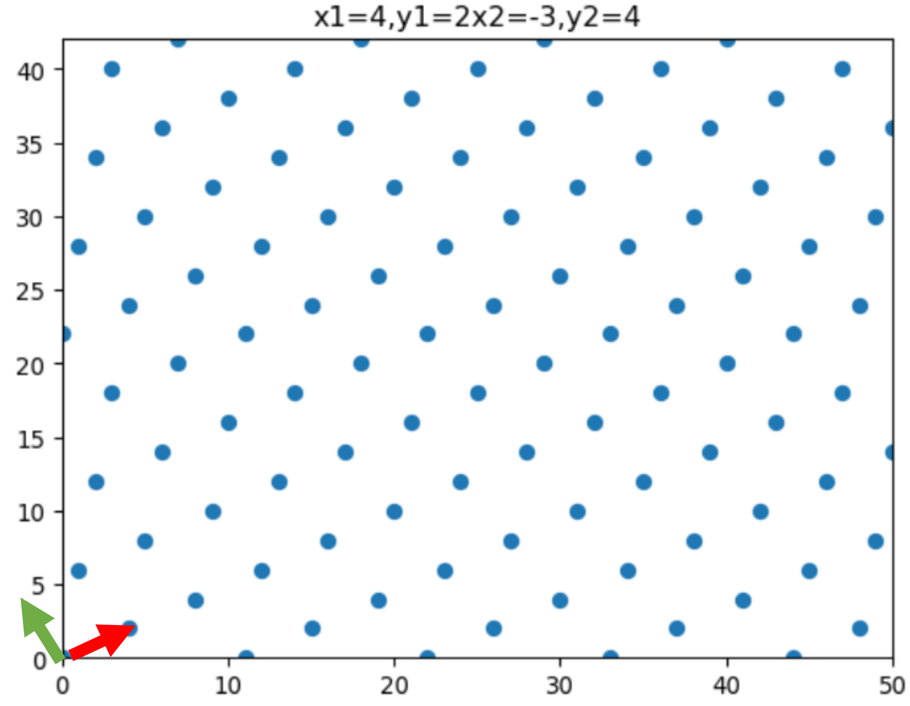
$v_k = (k_1, k_2, \ldots, k_m)$ will be the distance.

Credit: Biased Nonce Sense: Lattice attacks against weak ECDSA signatures in the wild, Nadia Heninger et al.

What is Lattice

- Integer Lattice: Z-linear combination of n independent vectors $b_i \in Z^n$.
- Basis $B = \{b_i\}$
- $L = \sum (z_i * b_i)$

- Example:
  - $n = 2$, $b_1 = [4, 2]$, $b_2 = [-3, 4]$.
  - Lattice: $L = \{z_1 b_1 + z_2 b_2\}$



Credit: Jiewen Yao

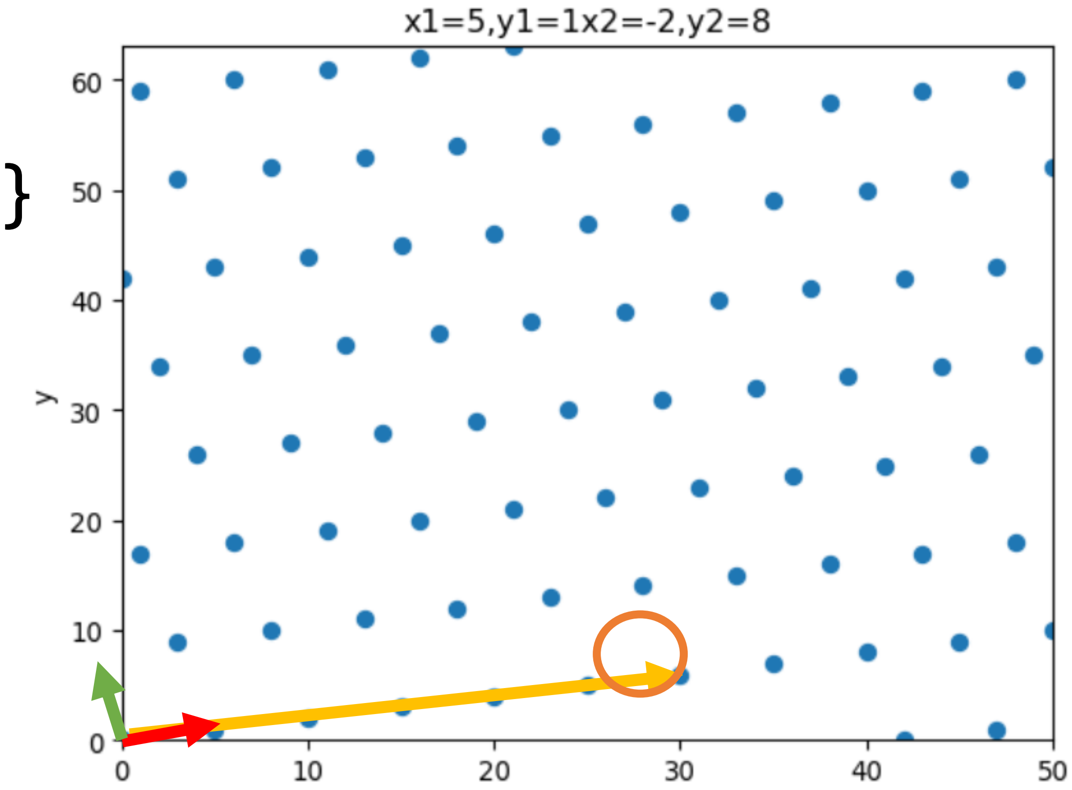
Hard Problem (CVP)

- Closest Vector Problem (CVP)
- Example: Lattice: L = $\{z_1b_1+z_2b_2\}$
  - n = 2, $\mathbf{b_1} = \mathbf{[5, 1]}$, $\mathbf{b_2} = \mathbf{[-2, 8]}$.
  - Find **CVP for [27, 8]**:

$$\begin{cases} 5 z_1 - 2 z_2 = 27 \\ 1 z_1 + 8 z_2 = 8 \end{cases}$$

$$\begin{cases} z_1 = 5.52 \\ z_2 = 0.309 \end{cases}$$

  - $(z_1,z_2) = (6, 0)$, **CVP = [30, 6]**



- NOTE: Only valid if the angle between 2 points is around 90º.

Credit: Jiewen Yao

Hard Problem (CVP)

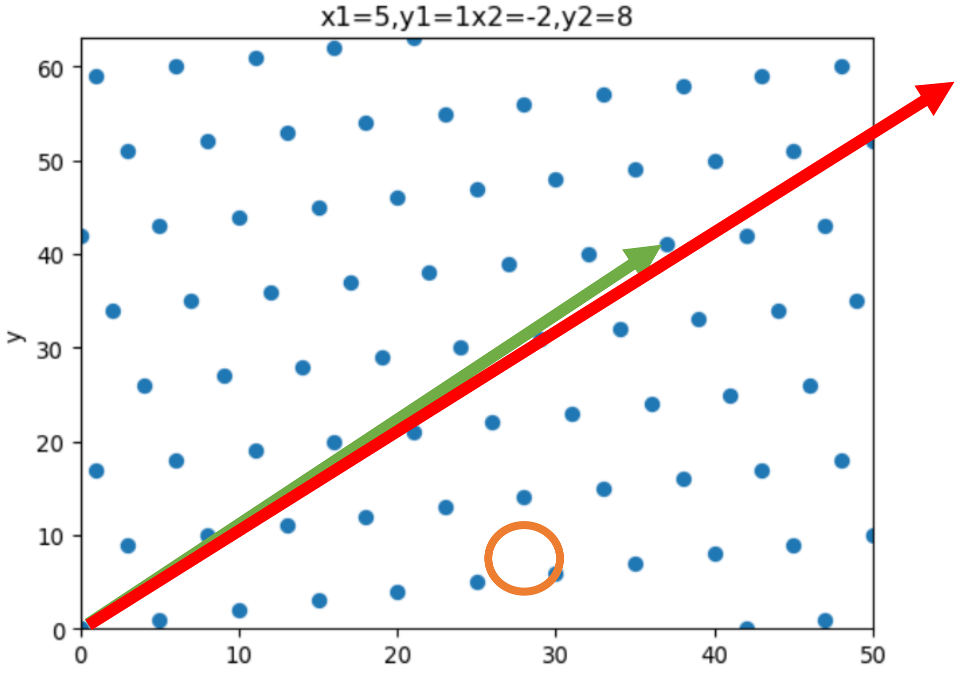
- What if the angle between 2 points is around 0º ?

- Example: Lattice: L = $\{z_1b_1+z_2b_2\}$
  - n = 2, b_1 = [37, 41], b_2 = [103, 113].
  - (Same Lattice, different basis)
  - Find **CVP for [27, 8]:**

$$\begin{cases} 37\,z_1 + 103\,z_2 = 27 \\ 41\,z_1 + 113\,z_2 = 8 \end{cases}$$

$$\begin{cases} z_1 = -53.023 \\ z_2 = 19.309 \end{cases}$$

  - (z_1,z_2) = (-53, 19), answer = [-4, -26] (wrong)



Credit: Jiewen Yao

Solving the hidden number problem with CVP embedding

Input:

$$k_1 - t_1 d - a_1 \equiv 0 \bmod n$$
$$\vdots$$
$$k_m - t_m d - a_m \equiv 0 \bmod n$$

in unknowns $k_1, \ldots, k_m, d$, where $|k_i| < B$.

LLL, BKZ implementations better than CVP implementations.

Construct the lattice basis

$$M = \begin{bmatrix} n & & & & & \\ & n & & & & \\ & & \ddots & & & \\ & & & n & & \\ t_1 & t_2 & \ldots & t_m & B/n & \\ a_1 & a_2 & \ldots & a_m & & B \end{bmatrix} \longrightarrow \begin{pmatrix} q & 0 & 0 & 0 \\ 0 & q & 0 & 0 \\ t_1 & t_2 & R/q & 0 \\ a_1 & a_2 & 0 & R \end{pmatrix}$$

$v_k = (k_1, k_2, \ldots, k_m, Bd/n, B)$ is a short vector in this lattice.

$$\left(\mathrm{k}_1, \mathrm{k}_2, \frac{Rx}{q}, R\right)$$

Credit: Biased Nonce Sense: Lattice attacks against weak ECDSA signatures in the wild, Nadia Heninger et al.