tardíGrados: 3.14159265 3v6 [q-fin.TR] 23 Nov 2021

Ataque a una lista de direcciones de *Bitcoin*

Albert Zotkin
El autor gana sus bicoins de forma legítima

Resumen: En esta breve nota voy a distribuir todas las claves privadas de *Bitcoin*, en una cuadrícula tridimensional, donde cada una de esas llaves privadas se asumirá ocupando un nodo en esa cuadrícula. Y presentaré un breve código escrito en node para atacar contra una lista direcciones. La probabilidad de tener éxito en este ataque es equivalente a tirar 256 veces una moneda al aire y que salgan 256 caras (o cruces) seguidas, es decir, casi nula, por no decir imposible.

Palabras claves y frases: universo, tridimensional, llave privada, hash, llave pública, address, dominio,Bitcoin, curva elíptica, Secp256k1, punto, operación, grupo, número primo, node.js, wif, bitcore-lib

1 Dominio de la curva elíptica `secp256k1`

Bitcoin usa, para la generación de sus llaves públicas, una criptografía basada en una curva eliptica muy específica: la curva eliptica llamada **secp256k1**, que tiene la forma siguiente:

$$y^2 = x^3 + 7 \tag{1}$$

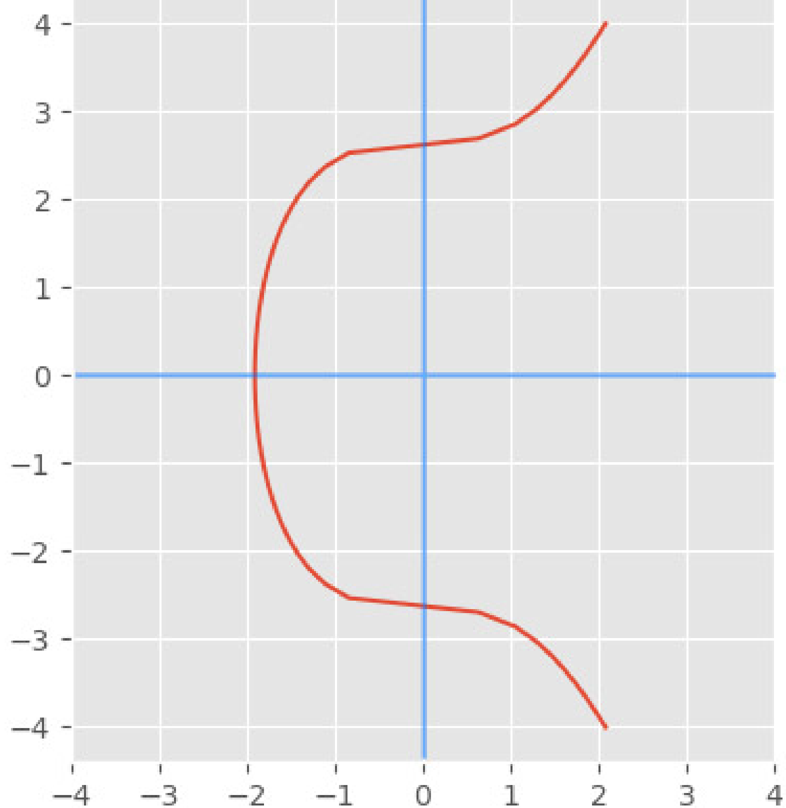

Figura 1: Esta sería la gráfica de la curva eliptica **secp256k1**, $y2 = x3 + 7$, en el dominio de los números reales. Pero, como **secp256k1**

ha sido definida sobre un campo en p de numero enteros positivos, su grapdica aparecerá como una conjunto de puntos dispersos, nada que se le parezca a esta. Lineas de Balmer tanto de absorcion como de emision.

pero, para facilitar el cálculo e impedir que operaciones definidas sobre puntos de la curva puedan ser fácilmente revertidas, han acordado que los valores que puede tomar la variable x (dominio de la función **secp256k1**) deben ser números enteros positvos, hasta cierto valor máximo, a partir del cual la función completaría ciclos (volviendo a repetir los mismos valores x). Eso se consigue eficientemente haciendo que el valor máxino que puede tomar la x sea un número entero positivo menor que cierto número primo p. Así, la curva **secp256k1** quedaría *cuarteada* de esta forma:

$$y^2 \mod p = x^3 + 7 \mod p \tag{2}$$

donde $\mod$ es el operador módulo p. Ese número primo p expresado en base decimal es::

$$p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1 = 2^{256} - 2^{32} - 977 =$$
$$115792089237316195423570985008687907853269984665640564039457584007908834671663$$

y expresado en base hexadecimal es:

$$p = FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F$$

2 Definamos una operación en el dominio de `secp256k1`

Llamaremos suma + de dos puntos, P y Q, de la curva elíptica **secp256k1**, a otro punto R, perteneciente también a la curva, y que será obtenido de la siguiente forma:

- Trazamos una recta que pase por P y Q.
- Buscaremos en qué punto, $R' = (a, b)$, esa recta corta a la curva (se eligen dichas curvas elípticas porque siempre aseguran que existe al menos un tercer punto de intersección)
- Elegimos el punto $R = (a, -b)$, que es verticalmente simétrico a R', como resultado de la operación suma

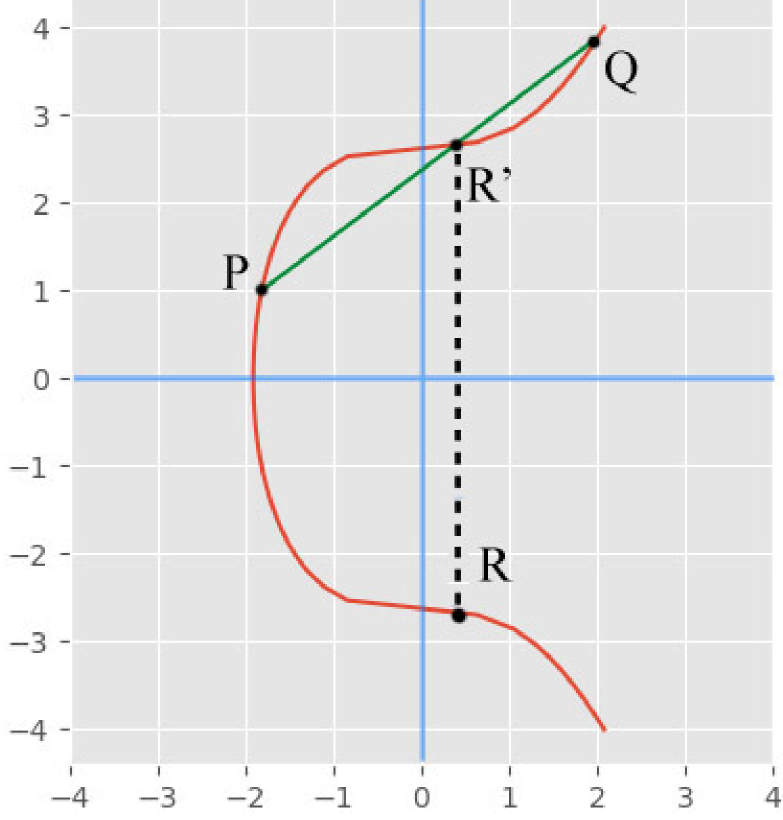

Figura 2: $R = P + Q$. Esa operación +, definida para los puntos de la curva forma un grupo.

Cuando un punto P se sume a sí mismo, la recta será la tangente a dicho punto. En *Bitcoin* se suma siempre un mismo punto, llamado punto base o punto origen, G. Y sus coordenadas son:

$$G = (x_0, y_0)$$

$$x_0 = 55066263022277343669578718895168534326250603453777594175500187360389116729240$$

$$y_0 = 32670510020758816978083085130507043184471273380659243275938904335757337482424$$

si sumamos G k veces, obtendremos el punto

$$R = \underbrace{G \ldots G}_{\text{k}} = kG$$

Y ese punto R sería nuestra **llave pública** correspondiente a la **llave privada** k. La criptografía basaada en curvas elítpticas asegura que sea muy fácil calcular R dado k, pero sea muy dificil, o casi imposible, dado R averiguar k en un tiempo razonable. Para complicar aún más las cosas, a cualquiera que intente un ataque a esta criptografía, en *Bitcoin* lo que se publica, no es la llave pública en sí misma, sino un **hash** de ella, (una *huella digital*), que también es una `función unidireccional`.

3 El orden N del grupo definido en `secp256k1`

El conjunto de todos los puntos del primer ciclo de la curva **secp256k1**, junto con la operación + ya definida, forman un grupo. El orden N de dicho grupo será pues el número de elementos que contiene. Dicho orden no es el número primo p que define el dominio de la curva, sino un número entero positivo menor que él, pero muy próximo. Ese número N en base decimal es:

$$N = 115792089237316195423570985008687907852837564279074904382605163141518161494337$$

y expresado en base hexadecimal es:

$$N = FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141$$

Esto quiere decir que cualquier operación para calcular una llave pública ha de usar valores de k dentro del intervalo cerrado [0,N], o sea, si intentamos $R = kG$, con $k \leq 0$, o con $k \geq N$, cometeremos un error, y el algoritmo de calculo así nos lo dirá. Una llave privada de *Bitcoin* es pues un simple número entero positivo mayor que cero y menor que N.

4 Todas las llaves Privadas de *Bitcoin*

Escribamos un pequeño programa en node para indexar todas y cada una de las llaves privadas de *Bitcoin*. Para ello calcularemos el **WIF** (Wallet Import Format) de cada uno de los números enteros k entre 0 y N, y para ello usaremos nuestra libreria de node.js **bitcore-lib**:

```
1.  const bitcore = require('bitcore-lib');
2.  const fs = require('fs');
3.
4.  function WIF(num){
5.          //Primero nos aseguramos de que num sea una cadena, no un número ni un objeto.
6.          if(typeof num !='string') num =''+num;
7.
8.          //Construimos nuestro Big Number desde num en base 10 (decimal)
9.          var bn = bitcore.crypto.BN.fromString(num,10);
10.
11.         //Definimos la unidad en Big Number
12.         const unit = bitcore.crypto.BN.fromNumber(1);
13.
14.         //Adquirimos el orden N del grupo (el número máximo)
15.         const N = bitcore.crypto.Point.getN();
16.
17.         //filtramos para que no haya error de dominio
18.         if(bn.lt(unit) ) {
19.                 console.log({error:'Integer must be positive and greater than 0'});
20.         return null;}
21.         if(!bn.lt(N)) {
22.                 console.log({error:'Integer must be less than N'});
23.         return null;}
24.
25.         //construimos dos versiones de llaves privadas en Bitcoin, para el mismo número num
26.         //una que dará una llave pública sin comprimir, y la otra la dará comprimida.
27.         var o  ={bn: bn, compressed:false,network:'livenet'};
28.         var o2 ={bn: bn, compressed:true,network:'livenet'};
29.         var privateKey   = new bitcore.PrivateKey.fromObject(o);
30.         var privateKey2  = new bitcore.PrivateKey.fromObject(o2);
31.
32.         //la funcion retorna dos WIF (Wallet Import Format) por cada num
33.
34.         return  [
35.                                                 privateKey.toWIF().toString('hex'),
36.                                                 privateKey2.toWIF().toString('hex')
37.                                         ];
38.
39. }
40.
41.  // extraemos una muestra para num entre 1000 y 1100
42.  // Guardamos nuestro muestreo en el archivo 'wifs.txt'
43.
44. var wifs ='';
45. for(var i=1000;i<1100;i++)
46. wifs += JSON.stringify(WIF(i))+'\r\n';
```

```
47.   fs.writeFileSync('wifs.txt', wifs);
48.
```

Como las llaves privadas de *Bitcoin* son simples números enteros positivos, aunque, eso sí, bastante grandes, ya que pueden llegar a tener hasta 78 digitos decimales o hasta 64 digitos hexadecimales, mi idea era distribuir esos números en tres intervalos según su tamaño, y de esa forma, cada llave privada podrá ser expresada mediante un trio de números enteros positivos, pero mucho más pequeños que el original. Pongamos el mejor ejemplo posible. Consideremos el número $L = N - 1$, donde N es el orden del grupo, que es el límite superior que no pueden igualar o sobrepasar las llaves privadas. Ese número L será:

$$L = N - 1 =$$
$$115792089237316195423570985008687907852837564279074904382605163141518161494336$$

y si ahora lo dividimos en tres partes iguales, cada una tendrá 26 digitos enteros.

$$L_1 = 11579208923731619542357098$$
$$L_2 = 50086879078528375642790749$$
$$L_3 = 04382605163141518161494336$$

es decir, que

$$L = L_1 \times 10^{2\times 26} + L_2 \times 10^{26} + L_3$$

¿Por qué hago esto?. En nuestra cuadrícula 3D las llaves privadas son ahora puntos distribuidos al azar por toda ella. Cuando, mediante cualquier algoritmo, intentamos atacar una llave privada desde su direccion pública (*address*), lo que estamos haciendo es aplicar la fuerza bruta, generando números al azar (que serian las llaves privadas), y calculando sus direcciones públicas. Ese ataque no sería fructifero. Pero, si se produjera alguna coincidencia, diriamos que se ha producido una *colisión*. Supongamos que tenemos una lista de direcciones de *Bitcoin*, de las que sabemos que tienen suculentos balances que podriamos substraer. Escribimos nuestro algoritmo para comparar las direcciones de llaves privadas generadas al azar, con las direcciones de esa lista. La probabilidad de que se produjera una colisión es muy pequeña, casi cero. La probabilidad de que un **neutrino** procedente del Sol colisionara con la Tierra sería más alta que la que tendría lugar en nuestra cuadrícula 3D.

5 Ataque a una lista de direcciones *Bitcoin*

Escribamos un pequeño programa para *atacar* una lista de direcciones de *Bitcoin*, en la que vemos que está llena de monedas, y las queremos mover hacia nuestras billeteras. Evidentemente, la criptografía de *Bitcoin* no nos pone las cosas fáciles. De hecho, es casi seguro que nunca tendremos éxito con este ataque. De todas formas aqui presento el código en node:

```
1.   /*
2.           Librerias [módulos] de NODEjs requeridas:
3.   */
4.   const bitcore = require('bitcore-lib');
5.   const crypto = require('crypto');
6.   const {readFileSync, writeFileSync} = require('fs');
7.
8.   /*
```

```
9.          Supongamos que tenemos una lista [list] de 20
10.         direcciones de Bitcoin, con suculentos balances
11.         que queremos atacar.
12. */
13.
14. const list =[
15.                 "1MSqdotjLkiEogM6uuB53WxsQwsKW4njS1",
16.                 "15fZAwqxktZHke83DTWPauQXFZGkfzcbmN",
17.                 "12SEP6zcuo9MrsAYr6ZQ6J6j1P9bYtMPp4",
18.                 "1CcvnQRG8RCJfL3trmpvDfjYTHSBZQKfbc",
19.                 "1AR7dhm6xzXaofhDocFBbbqfK57sipzkFE",
20.                 "178WXHXBD9JtT5oM9fiLAVJyg5kSuAT4WL",
21.                 "1EgUFvZ4w2QdXsrqHiYM9MGWSQRaphVEMm",
22.                 "1Pm8oDsMESc4bE1TYrTu85Tp7oZX59Jj2y",
23.                 "1MBfZUyeGr4pmSsSRG9jJmNNRNcFpbYirC",
24.                 "1PApCLv7PVWDQo7yAZh2rjip94FfahLeeL",
25.                 "1PvLmB3o7TNxRfqY1GvoVrdMeiDw4e92jM",
26.                 "15Aip35rp93Dh873KTPfrd8RBwdZMq9wXR",
27.                 "1BP39doLVRJNhzG1A1H5GmPptXKBnAAygH",
28.                 "1BQkxDS9dVaKMWH6zYyEv6ukfWRFNaRKGc",
29.                 "1L9nDbSiGCZFKoqhJ4QkuRz5LxhgjxWMNV",
30.                 "13cc1YC1bhsVenfEpabyT3mWMmC2xFMfHd",
31.                 "1JGgLtrxj3fQhRkxcD5rJdZU9zPYxvj4er",
32.                 "1L4pUsVkE1kVbhHhPFaTttjSKLdwXjsefK",
33.                 "154uXCpaR6cXg5qxAKYKAHX4VV3zwgaWfs",
34.                 "17WbYvDkRZo79Nk29VKhjGwJMhb7r5kGhs",
35. ]
36.
37. var     privateKey = null,
38.         address    = null,
39.         rnd        = false,
40.         match      = false;
41.
42.     console.log('Buscando coincidencias ...');
43.
44.     //  Creamos un bucle sin fin, del que no sale
45.     //  hasta que no encuentre un match [match == true].
46.                 while(!match){
47.                          //     Generamos un buffer de 32 bytes de valores aleatorios.
48.                                 var bytes = crypto.randomBytes(32);
49.                         //      Creamos un Big Number tipo BN desde esos bytes.
50.                                 var bn = bitcore.crypto.BN.fromBuffer(bytes);
51.                         //      Generamos un valor booleano aleatorio, que usaremos
52.                         //      para especificar si la llave pública será compromida o no.
53.                                 rnd = (Math.floor(Math.random()*2+1)%2)==0;
54.                         //      Generamos una llave privada aleatoria en la
55.                         //      red Bitcoin [livenet]
56.                                 privateKey = new bitcore.PrivateKey(
57.                                         {
58.                                                 bn:bn,
59.                                                 compressed:rnd,
60.                                                 network:'livenet'
61.                                         }
62.                                 );
63.                                 //      Confeccionamos la dirección publica [address]
64.                                 //      desde la llave privada.
65.                                 address = privateKey.toAddress().toString();
66.                                 //      Cotejamos si esa dirección está en la lista [list]
67.                                 match = (list.indexOf(address)+1)
68.                 }
69.
70.                 //      Creamos el objeto json con los resultados, si hemos salido
71.                 //      con écito [math==true] del bucle While.
72.                 //      Para salir del bucle sin fin cuando queramos, pero sin éxito,
73.                 //      [no se imprimirán resultados], tendremos que pulsar
74.                 //      simultaneamento las teclas [Ctrl+c].
75.                 var json =
76.                                 {
77.                                         match:match,
78.                                         compressed:rnd,
```

```
79.                                         privateKey:privateKey.toString(),
80.                                         wif:privateKey.toWIF().toString(),
81.                                         address
82.                                 };
83.
84.                 //      Imprimimos en consola el objeto JSON resultante,
85.                 //      y también guardamos el resultado en un archivo,
86.                 //      que llamaremos "exito.json".
87.                 console.log(json);
88.                 writeFileSync('exito.json',JSON.stringify(json,null,4));
```

¿Cuánto éxito tendremos con nuestro algoritmo de ataque?. Practicamento cero. Acertar esa loteria sería como lanzar al aire repetidas veces una moneda (no trucada, se entiende) y que saliera cara (o cruz) 256 veces seguidas. Es claro que la probablidad de acertar no es técnicamente cero, pero ese gracioso evento está tan próximo a cero que debemos perder toda esperanza.

En nuestro sencillo algoritmo de ataque, hemos extraido la dirección pública [*address*] (no confundir con la llave pública) directamente de la llave privada. Pero, hay que aclarar que esa dirección pública es sólo uno de los cuatro tipos que vienen implimentados en la librería *bitcore-lib*. En concreto, ese tipo de dirección que hemos extraido es la que viene por defecto, y es la **p2pkh** (pubkeyhash), los otros 3 tipos de direcciones son: **p2sh** (scripthash), **p2wpkh** (witnesspubkeyhash) y **bech32**. Esto quiere decir que, si quisieramos mejorar nuestro algoritmo de ataque, deberiamos extraer todos esos tipos cuatro de direcciones de la llave privada aleatoria que generamos, y cojetarlos contra las direcciones de la lista [list], ya que es muy probable que en esa lista existan diferentes tipos de direcciones.

6 ¿Cómo obtenemos los diferentes tipos de direcciones [*addresses*] usando la libería *Bitcore-lib*

1. P2PKH (***P****ay* ***T****o* ***P****ublic* ***K****ey* ***H****ash*): Este tipo de dirección es el más fácil de extraer de la llave privada usando la librería *Bitcore-lib*, ya que es el tipo que viene por defecto:

```
var p2pkh = privateKey.toAddress('livenet','pubkeyhash').toString();
```

o simplemente:

```
var p2pkh = privateKey.toAddress().toString();
```

2. P2SH (***P****ay* ***T****o* ***S****cript* ***H****ash*):
Este formato sólo es para llaves públicas comprimidas.

```
var p2sh = privateKey.toAddress('livenet','scripthash').toString();
```

3. P2WPKH (***P****ay* ***T****o* ***W****itness* ***P****ublic* **K**ey} **H**ash}):
Este formato tampoco puede usarse para llaves publicas sin comprimir.

```
var p2wpkh = privateKey2.toAddress('livenet','witnesspubkeyhash').toString();
```

4. BECH32:
Bech32 es un tipo de formato para direcciones que está especificado en el documento **BIP 0173**. Sin embargo, cuando pretendemos usar la librería *libcore-lib* para expressar la dirección en este formato, surgen algunos problemas. La librería *libcore-lib* usa sus propias funciones para codificar

y decodificar en este formato bech32, pero usa la librería externa **bech32**. Si aún así estamos empeñados en usar la *libcore-lib* para direcciones en formato bench32, tenemos que hacer algunas enmiendas en ella. Primero, hay que implenentar la función **toWords**, en el módulo `lib/encoding/bech32.js`: lo abrimos, y escribimos la siguiente implementación:

```
var toWords = function(data){
        return bech32.toWords(data);
        }
```

y al final, en `module.exports` añadimos en nuevo elemento `toWords:toWords`. Con lo que quedará así: `module.exports = {decode: decode, encode: encode, toWords:toWords};`Guardamos los cambios en el módulo.

Ahora vamos al módulo index.js, inicial de la libreria, y en el apartado encoding añadimos la siguiente linea:

```
bitcore.encoding.Bech32=require('./lib/encoding/bech32');
```

Guardamos los cambios, y la libreria *libcore-lib* ya estará lista para confeccionar directamente direcciones en formato bech32 .

```
var hash = bitcore.crypto.Hash.sha256ripemd160(publicKey.toBuffer());
var b = bitcore.encoding.Bech32.toWords(hash);
var bech32_p = bitcore.encoding.Bech32.encode("bc",0, b);
```

AUTORES

Albert Zotkin
Departmento de Matemáticas de la Universidad de Alicante, DMUA
Carr. de San Vicente del Raspeig, s/n, 03690 San Vicente del Raspeig, Alicante
https:tardigrados.wordpress.com

Este documento ha sido redactado
editado e imprimido con la herramienta tex2html
y el servidor web http-server con entorno integrado de programación JHS