


Ataque

Criptografía de Curva Elíptica: Ataque de Rho de Pollard

Daniel Lerch 

Grado de dificultad



En este artículo se pretende dar a conocer la teoría en la que se basa la criptografía de curva elíptica de forma sencilla e intentando que sean necesarios los mínimos conocimientos de matemáticas posibles.

Sin embargo, la teoría asociada a las curvas elípticas es extensa y compleja, por lo que en la sección de referencias se presentan algunos recursos adecuados para que el lector pueda ampliar sus conocimientos.

Por otra parte, las bases aquí expuestas deberían ser suficientes para comprender el artículo.

En los apartados prácticos se usa GNU/Linux y C++. Por lo que es necesario disponer de conocimientos de ambos para seguir el artículo sin problemas.

El artículo empieza presentando las curvas elípticas y las operaciones básicas que se pueden realizar sobre ellas: la suma y la multiplicación, así como una técnica utilizada para contar los puntos que hay en una curva. Estas son las bases para comprender el problema del logaritmo discreto en curvas elípticas (ECDLP) que se presenta posteriormente y es la base de la criptografía de curva elíptica. A continuación, conociendo las bases necesarias, se implementan algunos algoritmos en C++ y se inicia el desarrollo del ataque Rho de Pollard, actualmente considerado el más rápido contra los criptosistemas de curva elíptica.

Criptografía de clave pública

La criptografía de clave pública permite a dos usuarios mantener una conversación privada sin tener que acordar inicialmente una clave común (por otros medios) como ocurre con la criptografía simétrica (o de clave secreta). Así, un usuario generará un par de claves (una pública y una privada) y distribuirá la clave pública entre todos los usuarios susceptibles de enviarle un mensaje. Cualquier mensaje que se cifre con la clave pública sólo podrá

En este artículo aprenderás...

- Las bases de la criptografía de Curva Elíptica,
- Qué es el problema del logaritmo discreto en Curvas Elípticas,
- Algoritmos aplicados a las curvas elípticas,
- El Ataque Rho de Pollard.

Lo que deberías saber...

- Matemáticas nivel medio,
- C++ nivel medio,
- Criptografía básica.

ser descifrado con la clave privada. Como sólo el usuario inicial dispone de la clave privada nadie más podrá leer sus mensajes. La misma operación se utiliza a la inversa para firmar mensajes. Es decir, un mensaje firmado con la clave privada se podrá verificar con la clave pública. De esta manera sólo el usuario original podrá firmar mensajes, mientras que cualquiera podrá verificarlos.

Normalmente los sistemas de cifrado de clave pública son más lentos que los de clave privada, por lo que los primeros suelen usarse como sistema de intercambio de claves. Posteriormente un cifrado simétrico será el encargado de mantener el resto de la comunicación segura.

La criptografía de clave pública se caracteriza por el uso de problemas matemáticos computacionalmente difíciles, de manera que romper el cifrado suele ser el equivalente a resolver estos problemas. A lo largo de la historia de la criptografía de clave pública, los dos problemas matemáticos más usados son el problema de la factorización y el problema del logaritmo discreto. De los algoritmos más conocidos de clave pública des-

tacan RSA [3] y el intercambio de claves de Diffie-Hellman. El primero está basado en el problema de factorización, mientras que el segundo está basado en el problema del logaritmo discreto.

Ambos problemas han sido extensamente estudiados debido a su importancia en criptografía. Actualmente para ambos casos existen algoritmos de tiempo subexponencial que permiten resolverlos. En el caso del problema de factorización, el algoritmo más rápido hasta la fecha es el *Number Field Sieve* [3]. En el caso del logaritmo discreto es el *Index Calculus* [20].

Esto no es suficiente para romper completamente el cifrado, pero obliga a utilizar claves muy grandes lo que en algunos casos puede ser problemático. Si se encontrase un algoritmo de tiempo polinómico para resolver estos problemas, el criptosistema asociado a ellos quedaría fuera de combate. Por otra parte, si se encuentra un problema matemático que sólo se pueda resolver en tiempo exponencial, podríamos decir que el criptosistema sería igual de seguro que los anteriores con tamaños de clave inferiores.

Tiempo exponencial >> Tiempo subexponencial >> Tiempo polinómico

Aquí es donde entran en juego las curvas elípticas. Pues nos permiten definir un problema matemático similar al del logaritmo discreto, al que llamaremos problema del logaritmo discreto en curvas elípticas (ECDLP). Actualmente el algoritmo más rápido que se conoce para resolver este problema es de tiempo

exponencial, lo que nos permite construir un criptosistema igual de seguro que RSA (por Listado) con tamaños de clave inferiores. De hecho se estima que una clave RSA de 4096 bits da el mismo nivel de seguridad que una clave de 313 bits de un sistema de curva elíptica. Esta diferencia resulta realmente notable cuando se trabaja con dispositivos móviles, dado que una operación como generar una clave, que tardaría unos pocos segundos mediante un sistema de curva elíptica, podría demorarse varios minutos utilizando un sistema como RSA.

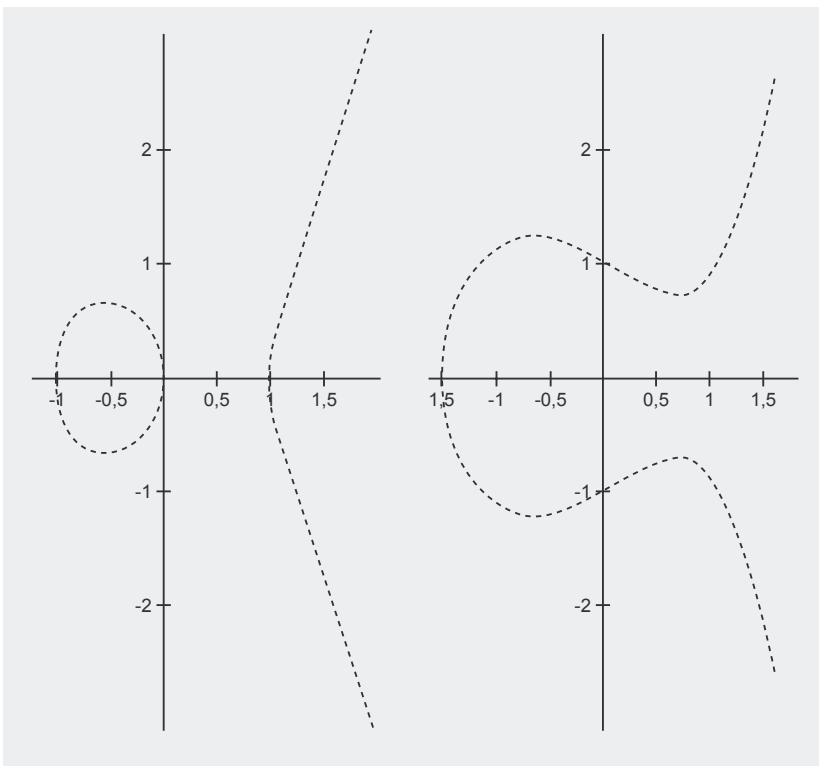


Figura 1. Forma básica

Curvas elípticas

Cuando hablamos de criptografía, una curva elíptica se define como una ecuación $y^2 = x^3 + Ax + B$, donde A y B son constantes y cumplen $4A^3 + 27B^2 \neq 0$. Esta ecuación es conocida como ecuación de *Weierstrass*. En la Figura 1 se muestran dos Listados de la forma que puede tomar la gráfica de una curva elíptica.

Se puede observar que la forma que toma la Figura 1.a es ligeramente diferente de la 1.b. Esta diferencia dependerá de los valores que tomen A y B en la ecuación de Weierstrass, pues para cada uno de ellos existe una curva diferente.

Por razones técnicas, a parte de los puntos que pertenecen a la curva se considera un punto en el infinito. Este punto, el (∞, ∞) suele denotarse simplemente como ∞O . Como veremos más adelante nos resultará realmente útil.



Suma de puntos

Para empezar nuestra introducción al fascinante mundo de las curvas elípticas veamos como se suman dos puntos situados en una curva. Lo haremos de forma gráfica, pues nos permite comprender rápidamente este concepto. Supongamos que tenemos dos puntos P y Q en una curva. Observe la Figura 2. Si trazamos una línea entre estos dos puntos veremos que corta a la curva en un tercer punto. Si reflejamos este punto a través del eje (cambiamos el signo de su coordenada y), obtendremos un punto R . Dicho punto R corresponde a la suma de P y Q .

$$P+Q=R$$

Esto se conoce como ley del grupo. Pero no es necesario disponer de dos puntos para encontrar un tercero. Pues a partir de un solo punto podemos obtener otro. Imaginemos que en el caso anterior $P=Q$. Entonces la línea en lugar de cortar la curva por los puntos P y Q solo lo haría por un punto. Es decir, sería una tangente. En la Figura 3 se ilustra este caso y algunos más, donde puede verse claramente como funciona este concepto.

En la Figura 3.a se ilustra el caso con el que hemos empezado. $P+Q=-R$ (anteriormente hemos cambiado el signo de R reflejando el punto a través del eje). En la Figura 3.b se ilustra el caso comentado anteriormente en el que se parte de un único punto P . En este caso la línea que dibujamos es una tangente a la curva que corta en el punto $-R$. De esta manera $P+P = 2P = -R$. En la Figura 3.c se ilustra el caso en el que no existe una línea que pase por P y por Q que corte en otro punto a la curva elíptica. En este caso se dice que dicha línea corta la curva en un punto O situado en el infinito, es decir $P+Q=0$ o $P+Q=\infty$. En cuarto y último lugar representamos un caso similar, en el que un único punto al ser sumado por el mismo corta la curva en el infinito.

Multiplicación

En el apartado anterior hemos aprendido a sumar puntos en una curva elíptica. Estas operaciones nos dan la base que nos permitirá realizar operaciones de multiplicación por un escalar. Es decir un número k por un punto P . Esto es más sencillo de lo que puede parecer. Supongamos que queremos calcular kP donde $k=27$. Podemos realizar un doblado de puntos de la forma siguiente:

$$P, \quad 2P = P + P, \quad 4P = 2P + 2P, \\ 8P = 4P + 4P, \quad 16P = 8P + 8P, \\ 27P = 16P + 8P + 2P + P$$

obteniendo el punto resultante de multiplicar P por un escalar k . Este procedimiento permite calcular kP para valores de k de varios cientos de dígitos muy rápidamente. El único problema consiste en el gran tamaño que adquieren las coordenadas de los puntos que se van calculando. Pero esto no ocurre al trabajar en campos finitos como suele hacerse en criptografía de curva elíptica. Consulte la Tabla *Campos Finitos* para saber algo más sobre ellos.

Contando puntos en curvas elípticas: el algoritmo SEA

Una curva definida sobre un campo finito tiene un número finito de puntos en ella. A este número de puntos lo llamamos orden de la curva. Por

otra parte, el número k más pequeño (pero mayor que 0) que multiplicado por un punto P da (∞) , se conoce como orden de ese punto. ¡No confundir el orden de la curva con el orden del punto!

Un resultado básico que nace del enunciado anterior es el siguiente: el orden de un punto P en una curva, divide al orden de la curva.

Estos conceptos son de vital importancia en la teoría asociada a las curvas elípticas y los usaremos en el Ataque Rho de Pollard que se explica más adelante.

En este apartado vamos a ver como se puede calcular el orden de una curva. Para tal propósito utilizaremos el algoritmo SEA (*Schoof-Elkies-Atkin*) que es de tiempo polinómico. Su implementación y teoría asociada queda fuera del alcance de este artículo, pero el lector puede consultar [5], [6] y [14] para obtener más detalles.

No es fácil encontrar una implementación libre de este algoritmo. Por suerte existe el paquete ELLSEA [15] implementado por Christophe Doche y Sylvain Duquesne. ELLSEA está desarrollado como script para PARI/GP [16], un sistema de álgebra computacional GPL muy utilizado. Veamos un Listado. Lo primero es instalar PARI/GP, para ello lo mejor es seguir las instrucciones de la página del proyecto o usar las herramientas de la distribución GNU/Linux utilizada (*apt-get, yum, emerge, etc*).

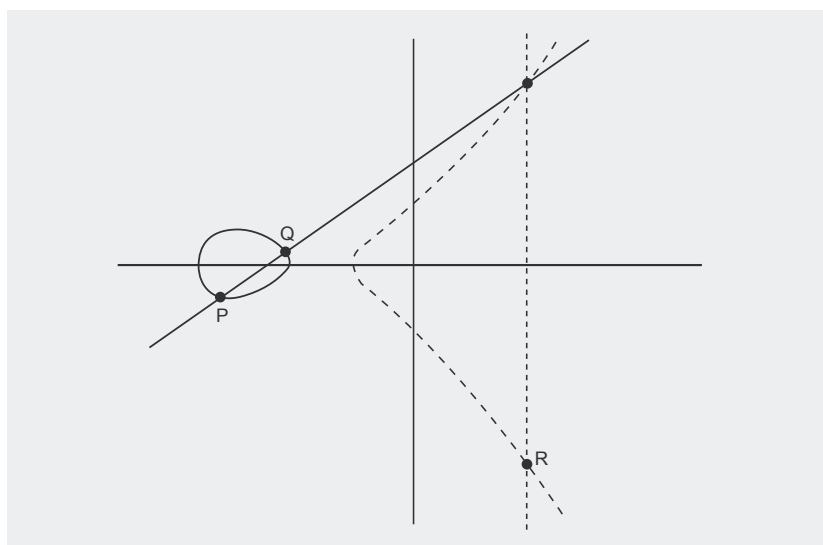


Figura 2. Suma de puntos

Una vez instalado PARI/GP se descomprime ELLSEA [15] y desde dentro del directorio SEA se llama a PARI/GP (Listado 1). Lo primero será leer el contenido de *sea.gp*. Lo haremos con la instrucción *read (sea)*. A continuación utilizaremos la instrucción *ellinit()* para crear una curva. Los dos últimos parámetros están formadas por las variables *A* y *B* de la ecuación de *Weirestrass* (Consultar Listado 1). En el Listado se utiliza *A=1* y *B=1*, es decir la curva $y^2 = x^3 + x + 1$. Finalmente para obtener el orden solo tenderemos que llamar a la función *ellsea()* a la que le pasaremos como parámetro la curva elíptica y el número primo que define el campo finito. Si usamos el número primo 43 vemos que el orden es 34 (Listado 1). Es decir, la curva mencionada dispone de 34 puntos diferentes.

Algoritmos para curvas elípticas: Implementación

En los apartados anteriores hemos estudiado cómo realizar las operaciones habituales en curvas elípticas de forma gráfica. Pero esta no es la forma más adecuada para un ordenador. Así que en este apartado veremos algunos algoritmos que realizan estas operaciones.

El lenguaje escogido para la implementación ha sido C++. El motivo principal es la posibilidad de utilizar la sobrecarga de operadores para realizar un programa matemáticamente más claro. Esta ventaja la encontramos principalmente al usar la librería GMP

[13]. Los mismo algoritmos, implementados en C correrían sustancialmente más rápido, pues la ventaja *visual* de la sobrecarga de operadores, no es tal cuando se trata de rendimiento. En cualquier caso, los siguientes algoritmos están desarrollados con objetivo docente, por lo que C++ parece ser la opción correcta.

La librería GMP nos permitirá usar números grandes sin problemas y nos ofrecerá todas las funciones necesarias para nuestra implementación de curvas elípticas. Así que empezaremos definiendo algunas estructuras de datos. Necesitaremos manejar curvas elípticas y puntos en una curva. En el Listado 2 vemos como podemos defi-

Listado 1. Pari/GP

```
$ gp
Reading GPRC: /etc/gprc ...Done.
GP/PARI CALCULATOR Version 2.3.1 (released)
i686 running linux (ix86 kernel) 32-bit version
compiled: Feb 28 2007, gcc-4.0.3 (Ubuntu 4.0.3-lubuntu5)
(readline v5.1 enabled, extended help available)
Copyright (C) 2000-2006 The PARI Group
PARI/GP is free software, covered by the GNU General Public License, and
comes WITHOUT ANY WARRANTY WHATSOEVER.
Type ? for help, \q to quit.
Type ?12 for how to get moral (and possibly technical) support.
parisize = 4000000, primelimit = 500000
?
? read("sea")
? E = ellinit([0,0,0,1,1]);
? ellsea(E,43)
? 34
```

Listado 2. Estructuras de datos

```
typedef struct point_t
{
    point_t() { infinity = false; }
    mpz_class x; // Coordenada X
    mpz_class y; // Coordenada Y
    bool infinity; // Es un punto en el infinito?
} point_t;
typedef struct
{
    mpz_class a; // Parametro A en la ecuación de Weierstrass
    mpz_class b; // Parametro B en la ecuación de Weierstrass
    mpz_class n; // Campo finito de la curva
    mpz_class o; // Orden de la curva
} elliptic_curve_t;
```

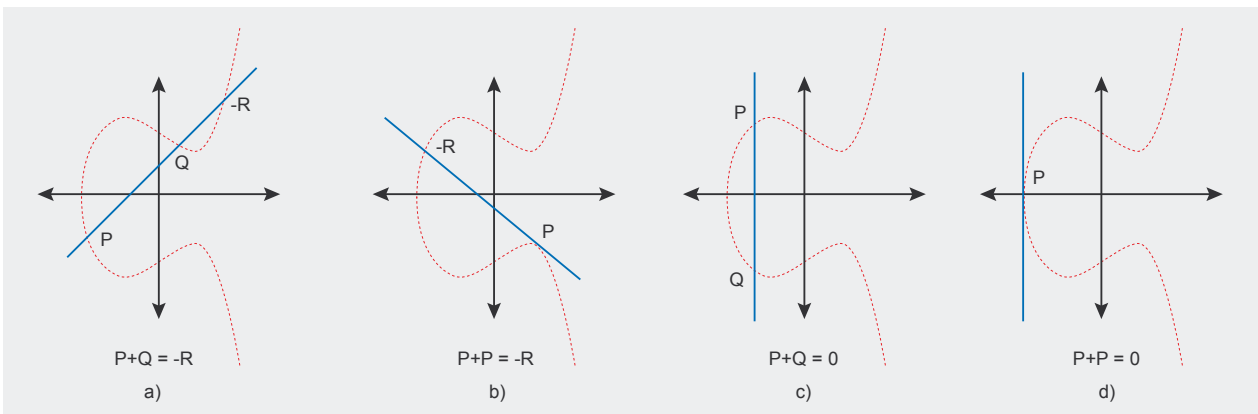


Figura 3. Ejemplos de suma



nir un punto y una curva. Usaremos el tipo `mpz_class` de la librería GMP que representa un número entero.

Veamos el tipo `point_t`. El punto cuenta con sus dos coordenadas X e Y, así como con un booleano que nos indica si se trata de un punto en el infinito.

Una curva elíptica queda definida por los parámetros A y B de la ecuación de Weirestrass que hemos explicado anteriormente. El tipo `elliptic_curve_t` define una curva a partir de los parámetros A y B, del campo finito de la curva y del orden de la misma. Asimilar adecuadamente el contenido del Listado 2 es necesario para el correcto seguimiento de los algoritmos posteriores.

Una vez disponemos de los tipos de datos básicos con los que trabajar: el punto y la curva, ya podemos iniciar el desarrollo de los algoritmos necesarios. Empezaremos por la suma de puntos, como en los apartados anteriores.

El algoritmo de suma del Listado 3 recibe como parámetros dos puntos $p1$ y $p2$, un punto r donde retornar el resultado y una curva en la que se realizan las operaciones. El algoritmo empieza verificando los datos de entrada en busca de puntos en el infinito, pues la suma de un punto P con un punto O (infinito) da el mismo punto P.

A continuación se calculan las nuevas coordenadas del punto resultante de sumar $p1$ y $p2$. En este caso el proceso se realiza despejando adecuadamente la ecuación de Weirestrass.

En los algoritmos propuestos en las Tablas se hace uso de funciones de la librería GMP como `mpz_mod()` para calcular el módulo o `mpz_invert()` para calcular la inversa modular. Si desconoce estas u otras funciones de la librería GMP puede consultar [13].

Partiendo del algoritmo de la suma es muy sencillo realizar el algoritmo de la resta. Pero para poder hacerlo necesitaremos una función que nos permita negar un punto. Para negar un punto solo tenemos que cambiar el signo de su coordenada Y (Listado 4).

Ahora con unas pocas líneas podemos realizar una función que nos permite restar dos puntos (Listado 5).

Otra función interesante para realizar operaciones con puntos y que nos resultará realmente útil en la implementación de la multiplicación es el doblado de puntos. Respecto a esta función, sobran los comentarios (Listado 6).

Solo nos queda el algoritmo de la multiplicación, que permite multiplicar un escalar por un punto en una curva. Aunque es relativamente corto, es un poco más difícil de entender que los anteriores. Se recomienda leerlo con detenimiento, pues aunque su comprensión no es necesaria para seguir el resto del artículo, es muy instructivo (Listado 7).

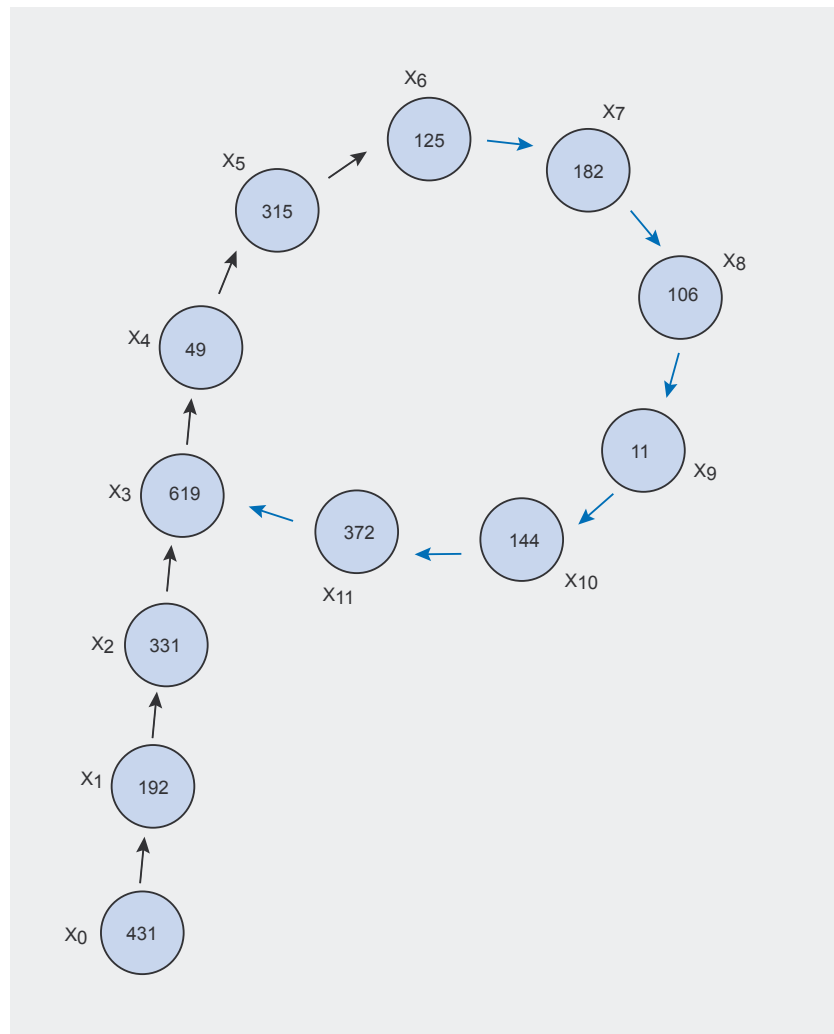


Figura 4. Rho

Campos Finitos

Un *campo finito* consiste en un conjunto finito de elementos con dos operaciones, la suma y la multiplicación y que satisface ciertas propiedades aritméticas. Si tenemos un número primo p , todos los números enteros $\text{mod } p$ forman el campo finito F_p . Veamos un Listado sencillo para entender el concepto. El campo finito F_5 está formado por los siguientes elementos: $F_5 = \{0, 1, 2, 3, 4\}$

Recordemos que una operación modular consistía en el resto de una división entera. De manera que $1 \text{ mod } 5 = 1$, $2 \text{ mod } 5 = 2$, $3 \text{ mod } 5 = 3$, $4 \text{ mod } 5 = 4$, $5 \text{ mod } 5 = 0$, $6 \text{ mod } 5 = 1$ y así sucesivamente. De manera que para cualquier entero n , al calcular $n \text{ mod } 5$ obtendremos un valor en F_5 .

Listado 3. Suma

```

void ec_add(point_t *r, const point_t *p1, const point_t *p2, const elliptic_
            curve_t *e)
{
    mpz_class m;
    mpz_class t;
    point_t res;
    if( (p1->infinity)&&(p2->infinity) )
    {
        r->x = 0;
        r->y = 0;
        r->infinity = true;
        return;
    }
    else if(p1->infinity)
    {
        r->x = p2->x;
        r->y = p2->y;
        r->infinity = false;
        return;
    }
    else if(p2->infinity)
    {
        r->x = p1->x;
        r->y = p1->y;
        r->infinity = false;
        return;
    }
    if(p1->x == p2->x)
    {
        if( (p1->y + p2->y) % e->n == 0)
        {
            r->x = 0;
            r->y = 0;
            r->infinity = true;
            return;
        }
        // m = (3x1^2+a) (2y1)^(-1)
        m = 3*(p1->x)*(p1->x)+e->a;
        t = 2*(p1->y);
        if(mpz_invert(t.get_mpz_t(), t.get_mpz_t(), e->n.get_mpz_t())==0)
        {
            r->x = 0;
            r->y = 0;
            r->infinity = true;
            return;
        }
        m *= t;
    }
    else
    {
        // m = (y2-y1) (x2-x1)^(-1)
        m = p2->y - p1->y;
        t = p2->x - p1->x;
        if(mpz_invert(t.get_mpz_t(), t.get_mpz_t(), e->n.get_mpz_t())==0)
        {
            r->x = 0;
            r->y = 0;
            r->infinity = true;
            return;
        }
        m *= t;
    }
    // x = m^2 - x1 -x2
    res.x = m*m - p1->x - p2->x;
    mpz_mod(res.x.get_mpz_t(), res.x.get_mpz_t(), e->n.get_mpz_t());
    // y = m(x1-x) -y1
    res.y = (m*(p1->x - res.x) - p1->y) % e->n;
    mpz_mod(res.y.get_mpz_t(), res.y.get_mpz_t(), e->n.get_mpz_t());
    r->x = res.x;
    r->y = res.y;
}

```

Hasta aquí hemos desarrollado los algoritmos necesarios para trabajar con curvas elípticas. En los siguientes apartados se utilizarán para construir un ataque contra el problema del logaritmo discreto, base de la criptografía de Curva Elíptica.

El problema del logaritmo discreto en curvas elípticas

El problema del logaritmo discreto para curvas elípticas (conocido como ECDLP) es la base de los criptosistemas de curva elíptica. En apartados anteriores hemos estudiado la manera de realizar operaciones como $Q=kP$ partiendo de k y de P . Esta operación es computacionalmente fácil. Sin embargo obtener k a partir de P y Q es un problema difícil incluso para un ordenador. De hecho si utilizamos valores de k lo suficientemente grandes, la tarea se vuelve computacionalmente imposible. Al menos con los algoritmos y máquinas actuales. Pues los algoritmos conocidos para resolver este problema son de tiempo exponencial. En el siguiente apartado veremos como aprovechar este problema para realizar un intercambio de claves seguro siguiendo el algoritmo de Diffie-Hellman.

Intercambio de claves de Diffie-Hellman en curvas elípticas

El intercambio de claves de Diffie-Hellman es un protocolo basado en el problema del logaritmo discreto que permite intercambiar claves de forma segura. Posteriormente se utilizará la clave intercambiada como clave de cifrado simétrico. Este algoritmo se puede adaptar a las curvas elípticas, formando el algoritmo ECDH o *Elliptic Curve Diffie-Hellman*. Veamos paso a paso como funciona el algoritmo.

- Un usuario A y un usuario B acuerdan usar una curva elíptica E sobre un campo finito F_q de manera que el problema del logaritmo discreto sea difícil en $E(F_q)$. También acordarán un punto P perteneciente a la curva de manera que su orden sea un número primo grande.

**Listado 4. Negación**

```
void ec_neg(point_t *p)
{
    p->y *= -1;
}
```

Listado 5. Resta

```
void ec_sub(point_t *r,
            const point_t *p1,
            const point_t *p2,
            const elliptic_curve_t *e)
{
    point_t res;
    res.x = p2->x;
    res.y = p2->y;
    ec_neg(&res);
    ec_add(&res, p1, &res, e);
    r->x = res.x;
    r->y = res.y;
}
```

Listado 6. Doblado

```
void ec_double(point_t *r,
               const point_t *p,
               const elliptic_curve_t *e)
{
    ec_add(r, p, p, e);
}
```

Listado 7. Multiplicación

```
void ec_mul(point_t *Q,
            const mpz_class k,
            const point_t *P,
            const elliptic_curve_t *e)
{
    mpz_class k3;
    unsigned int B;
    point_t R;
    if( (k==0) || (P->infinity))
    {
        Q->x = 0;
        Q->y = 0;
        Q->infinity = true;
        return;
    }
    R.x = P->x;
    R.y = P->y;
    k3 = k*3;
    B = mpz_sizeinbase(k3.get_mpz_t(), 2);
    int j;
    for(j=B-2; j>=1; j--)
    {
        ec_double(&R, &R, e);
        if( (mpz_tstbit(k3.get_mpz_t(),j)==1) && (mpz_tstbit(k.get_mpz_t(),j)==0) )
            ec_add(&R, &R, P, e);
        if( (mpz_tstbit(k3.get_mpz_t(),j)==0) && (mpz_tstbit(k.get_mpz_t(),j)==1) )
            ec_sub(&R, &R, P, e);
    }
    mpz_mod(Q->x.get_mpz_t(), R.x.get_mpz_t(), e->n.get_mpz_t());
    mpz_mod(Q->y.get_mpz_t(), R.y.get_mpz_t(), e->n.get_mpz_t());
}
```

- El usuario A escoge un entero secreto a , calcula $P_a = aP$ y envía P_a al usuario B.
- El usuario B escoge un entero secreto b , calcula $P_b = bP$ y envía P_b al usuario A.
- El usuario A calcula $aP_b = abP$.
- El usuario B calcula $bP_a = abP$.

Finalizado el algoritmo, tanto el usuario A como el usuario B disponen de abP . Un usuario que *escucha* el canal de comunicación ha podido obtener P_a y P_b pero estos no son suficientes para obtener abP a menos que se resuelva el problema del logaritmo discreto.

Ahora el usuario A y el usuario B pueden comunicarse utilizando una clave de cifrado que solo ellos conocen extrayéndola de abP , por Listado los X últimos dígitos o el resultado de una operación de hash.

Otros criptosistemas de curva elíptica

Existen varios criptosistemas de curva elíptica que se pueden encontrar en [5] o en Internet sin demasiado esfuerzo. Los más conocidos son: El intercambio de claves de *Diffie-Hellman* (explicado en el apartado anterior), el algoritmo de firma digital *ECDSA*, el cifrado *Massey-Omura*, *ElGamal*, etc.

Software Libre disponible

Existen varias implementaciones libres de criptosistemas de curva elíptica como el parche *eccGnuPG* para la conocida herramienta *GnuPG*[12] o la herramienta *SKS* [11] que destaca por su simplicidad.

Como no es mi intención escribir aquí un manual de herramientas de cifrado me limitaré a poner un Listado de uso de *SKS*. En la web del proyecto se proporciona una muy buena documentación que permitirá al lector probar todas sus funcionalidades sin dificultad alguna. En el Listado 8 se muestra el uso de *SKS* para firma digital. *SKS* tiene las funcionalidades de cifrado (y compresión), resumen (hash) y firma digital.

Listado 9. Factorización Rho de Pollard

```
// Compilar: g++ rho.cpp -lgmpxx -o rho
#include <gmpxx.h>
#include <iostream>
// F(x) = x^2k + a mod n
mpz_class F(mpz_class &x, mpz_class &k, mpz_class &a, mpz_class &n)
{
    mpz_class res;
    mpz_class pow;
    pow = 2*k;
    mpz_powm(res.get_mpz_t(), x.get_mpz_t(), pow.get_mpz_t(), n.get_mpz_t());
    res += a;
    return res;
}
int main(int argc, char *argv[])
{
    using namespace std;
    mpz_class n, U, V, s, g, k, a, p, count;
    if (argc!=3)
    {
        cout << argv[0] << " [n number] [B bound]" << endl;
        return 0;
    }
    n = argv[1];
    int B = atoi(argv[2]);
    count = 0;
    s = 2;
    U = s;
    V = s;
    g = 1;
    mpz_fac_ui(k.get_mpz_t(), B);
    a = 3;
    while(g==1)
    {
        U = F(U, k, a, n);
        V = F(V, k, a, n);
        V = F(V, k, a, n);
        g = U-V;
        mpz_gcd(g.get_mpz_t(), g.get_mpz_t(), n.get_mpz_t());
        if(g==n)
        {
            cout << "bad seed" << endl;
            U++;
            V++;
            g=1;
        }
        count ++;
        cout << "count: " << count << "\r";
    }
    cout << endl << g << endl;
    return 0;
}
```

Listado 8. SKS

```
$ ./sks -kg
SKS. Introduce etiqueta identificadora: miclave
SKS. Introduce tu contraseña:
-----SKS KEY-----
key: 7fZl9ptfprjM6wgU3iLqjbFbEzTwUa6jebi
miclave
[f7e2e68d]: miclave
$ ./sks -kl
[f7e2e68d]: miclave
$ ./sks -s msg.txt msg.sig
SKS. Introduce tu contraseña:
SKS. Clave de firma: [ed960057]
$ ./sks -v msg.txt msg.sig
[ed960057]: miclave
SKS. Firma correcta realizada en: 2007-03-07, 19:38
```

Ataques al logaritmo discreto

Resumiendo, podemos decir que resolver el problema del algoritmo discreto consiste en obtener de la fórmula $Q = kP$ a partir de Q y P . Como siempre en estos casos el primer ataque que se le viene a uno a la cabeza es la fuerza bruta, es decir, probar con todos los valores de k .

Un ejemplo de [8] que puede resultar ilustrativo es el siguiente:

Supongamos la curva elíptica sobre F_{23} , $Q = (4,5)$ y $P = (16,5)$. Para resolver el problema del logaritmo discreto $Q = kP$ calcularemos todos los múltiplos de P hasta resolver el problema:

y $9P = (4,5)$. Dado que $9P = (4,5) = Q$ sabemos que $k = 9$. Lógicamente en

$$\begin{aligned}
 P &= (16,5), \\
 2P &= (20,20), \quad 3P = (14,14), \\
 4P &= (19,20), \quad y^2 = x^3 + 9x + 17 \\
 5P &= (13,10), \quad 6P = (7,3), \\
 7P &= (8,7) \\
 8P &= (12,17)
 \end{aligned}$$

una aplicación criptográfica real el tamaño de k será lo suficientemente grande como para que esta forma de actuar sea impracticable incluso para miles de ordenadores trabajando en paralelo.

Pero esta no es la única técnica que permite atacar el logaritmo discreto. Existen muchas más como el algoritmo *Baby Step, Giant Step*, el ataque *MOV*, el método *Pohlig-Hellman*, etc. Algunos de ellos pueden encontrarse en [5] y [6]. Pero el ataque de carácter general más rápido es el algoritmo Rho de Pollard. Este algoritmo es de tiempo exponencial con una complejidad de \sqrt{N} , donde N es el orden de la curva. Este algoritmo es de tipo probabilístico y una de sus mayores ventajas es que no requiere almacenar grandes cantidades de datos. Además es fácilmente paralelizable.

Lo que queda de artículo está destinado a la comprensión e implementación del ataque Rho de Pollard a los Criptosistemas de Curva Elíptica basados en el ECDLP.



Introducción al algoritmo Rho de Pollard

Una de las principales cualidades del método Rho de Pollard es el tiempo medio que tarda en encontrar una solución (\sqrt{N}), por encima de los otros algoritmos que permiten atacar el problema del logaritmo discreto en curvas elípticas. Otra ventaja importante es que el sistema Rho es de carácter general, lo que permite usar este algoritmo en cualquier campo finito. Un buen Listado es la implementación de este sistema como algoritmo de factorización. En el Listado 9 se presenta dicho algoritmo implementado en C++/GMP. No entraremos en detalle, pues no es el objetivo de este artículo estudiar el problema de la factorización. Pero dado que vamos a centrarnos en el algoritmo Rho de Pollard para curvas elípticas, la implementación de este mismo algoritmo para factorización puede servirnos de introducción, pues es considerablemente más simple. Lo más relevante de este programa es el bucle principal y el uso que se hace en el de la función $F()$, dado que es idéntico al algoritmo que implementaremos más adelante para atacar el ECDLP.

Utilizamos campos finitos, por lo que el número de elementos (orden) con el que trabajaremos será también finito. Así pues, si definimos una función que realice un mapeo entre dos elementos del grupo y nos dedicamos a ir realizando mapeos uno tras otro, de todos los elementos, llegaremos a un punto donde se repetirán. Por Listado, supongamos que tenemos un grupo de elementos en F_{35} . Supongamos también que definimos una función $f(x) = x^2 + 1 \pmod{35}$, entonces si nos dedicamos a hacer mapeos con esta función: $f(0)=1$, $f(1)=2$, $f(2)=5$, $f(5)=26$, $f(26)=12$, $f(12)=5$ repitiendo de nuevo el tercer elemento y generando así un ciclo. Nuestro rango de soluciones se puede dividir en dos partes, la primera son los elementos {1,2} y la segunda los elementos {5, 26, 12} que forman un ciclo. Podríamos

representar la forma en la que se van generando los elementos como la letra griega rho, de ahí el nombre del algoritmo. En la Figura 4 se reproduce un Listado de Wikipedia. En este Listado se usa como valor inicial el 431 (cualquier valor sirve) y como función $f(x) = x^2 + 23 \pmod{703}$. Vemos como en el elemento 619 se inicia el ciclo, que queda cerrado en el elemento 372.

Volviendo al ejemplo del Listado 9 y al uso de la función $F()$ es interesante observar como la función es llamada dos veces con el parámetro V y una sola vez con el parámetro U . Esta técnica, llamada búsqueda de ciclos de Floyd, permite acelerar el algoritmo buscando una colisión entre V y U .

En curvas elípticas el esquema es el mismo, pero utilizando puntos en una curva en lugar de números enteros.

Atacando el logaritmo discreto

El apartado anterior puede resultar adecuado para comprender las bases del algoritmo Rho de Pollard. Pero se ha aplicado al problema de la factorización, y no al problema ECDLP que es el que nos ocupa. En este apartado aplicaremos el algoritmo a la resolución del ECDLP con un Listado.

Lo primero que necesitamos es encontrar una función $F(x)$ análoga a la utilizada en el apartado anterior. En este caso se tratará de $F(S)$ donde S será un punto aleatorio inicial. Dado que posteriormente se dedica un apartado completo a estudiar la optimización de la función F , para empezar usaremos una función simple como la siguiente: $f(S) = S + aP + bQ$ para valores de a y b aleatorios. El punto S inicial lo

Listado 10. Funciones útiles

```
bool solve(const mpz_class &U_mP, const mpz_class &U_mQ,
           const mpz_class &V_mP, const mpz_class &V_mQ,
           const point_t *P, const point_t *Q,
           const elliptic_curve_t *e)
{
    mpz_class k;
    mpz_class u = U_mP - V_mP;
    mpz_class v = V_mQ - U_mQ;
    point_t R;
    if(!mpz_invert(k.get_mpz_t(), v.get_mpz_t(), e->o.get_mpz_t()))
        return false;
    k *= u;
    mpz_mod(k.get_mpz_t(), k.get_mpz_t(), e->o.get_mpz_t());

    // Verificar k
    // ...
}

void seed_F(point_t *U, point_t *V, mpz_class &U_mP, mpz_class &U_mQ,
            mpz_class &V_mP, mpz_class &V_mQ, const point_t *P, const point_t *Q,
            const elliptic_curve_t *e)
{
    // ec_rand_number() Busca un número aleatorio < n usando la API de GMP
    ec_rand_number(U_mP, e->n);
    ec_rand_number(U_mQ, e->n);
    V_mP = U_mP;
    V_mQ = U_mQ;
    // P0 = rand(a)P + rand(b)Q
    point_t tmp1, tmp2;
    ec_mul(&tmp1, U_mP, P, e);
    ec_mul(&tmp2, U_mQ, Q, e);
    ec_add(U, &tmp1, &tmp2, e);
    V->x = U->x;
    V->y = U->y
    V->infinity = U->infinity;
    // ...
}
```

buscaremos de la misma manera, es decir, partiendo de dos valores a y b aleatorios, multiplicaremos por P y Q respectivamente y sumaremos los resultados. De momento utilizaremos como función $f(S) = S + 4P + 3Q$.

Como Listado tomaremos la curva definida por $y^2 = x^3 + 12x + 36 \pmod{43}$ de orden 53, y un punto en ella $P = (0, 6)$. Supongamos ahora que $Q = (3, 23)$. En los sistemas de criptografía de curva elíptica todos estos datos son públicos. Nuestro objetivo será encontrar un valor k que cumpla $kP = Q$. Lo que en un sistema criptográfico supondría encontrar la clave.

Basándonos en lo que hemos aprendido sobre el algoritmo Rho de Pollard sabemos que tenemos que manejar un punto U y un punto V , aplicando la función $F(S)$ dos veces a V por cada una que la apliquemos a U . Efectuaremos este procedimiento en repetidas ocasiones hasta encontrar una colisión.

Una vez encontramos la colisión disponemos de $F(S_i) = F(S_j)$, que también puede representarse como $a_i P + b_i Q = a_j P + b_j kP$. Como sabemos que $kP = Q$ podemos sustituir en la ecuación obteniendo $a_i P + b_i kP = a_j P + b_j kP$ y despejar $k = (a_i - a_j)(b_j + b_i)^{-1}$.

Existe otra manera de encontrar el resultado sin necesidad de encontrar una colisión. Esta consiste en la posibilidad de localizar durante la ejecución del algoritmo ecuaciones como $aP + Q = 0$, $aP + Q = P$, $aP + Q = Q$ o similares. Pues igual que en el caso anterior permitirían resolver k .

Finalmente veamos como resolver el problema planteado. Empezamos en un punto aleatorio $P_0 = 21P + 26Q = (14, 29)$. Aplicamos $F(U) = 25P + 29Q = (13, 29)$ y $F(F(V)) = 29P + 32Q$. No hay colisión. Aplicamos de nuevo $F(U)$ y $F(F(V))$ y durante el cálculo de la segunda vemos que $F(F(V)) = P = 37P + 38Q$. Lo que ya nos permite encontrar un resultado. Despejamos $k = 36(-38)^{-1} \pmod{53}$ (notar que es una operación modulo 53, es decir el orden de la curva). Rea-

lizando el cálculo obtenemos $k = 13$. En este punto podemos verificar que $13(0, 6) = (3, 23)$, resolviendo el problema en dos iteraciones, lejos de las 13 necesarias en un ataque de fuerza bruta.

Ataque Rho de Pollard: Implementación

Hemos estudiado como funciona el algoritmo Rho de Pollard y como usarlo para atacar el ECDLP. En

este apartado implementaremos en C++ este algoritmo, que nos servirá para atacar problemas en campos finitos mayores.

Partiendo de la implementación del algoritmo presentado en el Listado 9 podemos desarrollar mediante un esquema similar una implementación para curvas elípticas.

Lo primero que necesitamos es una función que nos permita inicializar los valores con los que traba-

Listado 11. Ataque Rho

```
// ...
int F(mpz_class &R_mP, mpz_class &R_mQ, point_t *R, const point_t *P,
      const point_t *Q, const point_t *S, const elliptic_curve_t *e)
{
    k = 4;
    ec_mul(&M, k, P, e);
    k = 3;
    ec_mul(&T, k, Q, e);
    ec_add(&M, &M, &T, e);
    R_mP += 4;
    R_mQ += 3;
    // ...
}
// ...
mpz_class U_mQ;
mpz_class V_mP;
mpz_class V_mQ;
point_t R;
point_t P;
point_t Q;
point_t U;
point_t V;
elliptic_curve_t E;
// Inicialización de valores
seed_F(&U, &V, U_mP, U_mQ, V_mP, V_mQ, &P, &Q, &E);
for(;;)
{
    if(!F(U_mP, U_mQ, &U, &P, &Q, &U, &E))
    { seed_F(&U, &V, U_mP, U_mQ, V_mP, V_mQ, &P, &Q, &E); continue; }
    if(!F(V_mP, V_mQ, &V, &P, &Q, &V, &E))
    { seed_F(&U, &V, U_mP, U_mQ, V_mP, V_mQ, &P, &Q, &E); continue; }
    if(!F(V_mP, V_mQ, &V, &P, &Q, &V, &E))
    { seed_F(&U, &V, U_mP, U_mQ, V_mP, V_mQ, &P, &Q, &E); continue; }
    // Colisión?
    if( (V.x==U.x) && (V.y==U.y) )
    {
        // Busca una solución
        if(solve(U_mP, U_mQ, V_mP, V_mQ, &P, &Q, &E))
        {
            break;
        }
        else
        {
            seed_F(&U, &V, U_mP, U_mQ, V_mP, V_mQ, &P, &Q, &E);
            continue;
        }
    }
}
// ...
```



jaremos. He llamado a esta función `seed_F()` y se encargará de buscar un punto aleatorio en la curva con el que empezar. Otra función intere-

Listado 12. Casos especiales

```
// ...
// aP+bQ=P
if( (R->x==P->x) && (R->y==P->y) )
{
    // ...
}
// aP+bQ=Q
if( (R->x==Q->x) && (R->y==Q->y) )
{
    // ...
}
// aP+bQ=0
if(R->infinity)
{
    // ...
}
// ...
```

Listado 13. Función F

```
// ...
switch (mpz_mod_ui(k, get_mpz_t(),
                S->x.get_
                mpz_t(),
                3))
{
    /* M0 = 4P + 3Q */
    case 0:
        k = 4;
        ec_mul(&M, k, P, e);
        k = 3;
        ec_mul(&T, k, Q, e);
        ec_add(&M, &M, &T, e);
        R_mP += 4;
        R_mQ += 3;
        break;
    /* M1 = 9P + 17Q */
    case 1:
        k = 9;
        ec_mul(&M, k, P, e);
        k = 17;
        ec_mul(&T, k, Q, e);
        ec_add(&M, &M, &T, e);
        R_mP += 9;
        R_mQ += 17;
        break;
    /* M2 = 19P + 6Q */
    case 2:
        k = 19;
        ec_mul(&M, k, P, e);
        k = 6;
        ec_mul(&T, k, Q, e);
        ec_add(&M, &M, &T, e);
        R_mP += 19;
        R_mQ += 6;
        break;
}
// ...
```

sante es la que calculará la solución cuando encontremos la colisión. A esta función le he llamado `solve()`. Encontrar una colisión no implica siempre encontrar la solución, por lo que `solve()` deberá verificar que $kP = Q$. En caso contrario se continuará con la búsqueda de colisiones (Listado 10).

En el caso de las curvas elípticas hay algunos factores adicionales a tener en cuenta. Por ejemplo, para

cada punto guardado, es necesario disponer de su relación con P y Q , pues es necesaria si queremos despejar k como se mostró en el apartado anterior. Con este propósito usaremos las variables U y V para almacenar los puntos, y las variables U_mP , U_mQ , V_mP y V_mQ para guardar los multiplicadores. De esta manera cada punto V podrá representarse también como $V_{mP}P + V_{mQ}Q$, y lo mismo con U (Listado 11).

Referencias

- [1] <http://daniellerch.com>,
- [2] <http://daniellerch.com/dfact.html>,
- [3] Daniel Lerch, Ataque de factorización a RSA. hakin9 – nº19,
- [4] Wikipedia, Curva Elíptica: http://es.wikipedia.org/wiki/Curva_el%C3%ADptica,
- [5] Lawrence C. Washington, Elliptic Curves, Number Theory and Cryptography. Chapman & Hall/CRC,
- [6] Richard Crandall, Carl Pomerance, Prime Numbers, A computational Perspective. 2Ed. Springer,
- [7] Alfred Menezes, Evaluation of Security Level of Cryptography: The Elliptic Curve Discrete Logarithm Problem (ECDLP),
- [8] Certicom, Online Elliptic Curve Cryptography Tutorial: http://www.certicom.com/index.php?action=ecc_tutorial,home,
- [9] The Certicom ECC Challenge: http://www.certicom.com/index.php?action=res,ecc_challenge,
- [10] Certicom, Challenge Lists and Prizes: http://www.certicom.com/index.php?action=res,ecc_solution,
- [11] SKS, Criptografía de Bolsillo: <http://pagina.de/sks>,
- [12] eccGnuPG. Implementación GnuPG con curvas elípticas: <http://www.calcurco.cat/eccGnuPG/document.es.html>,
- [13] GMP, GNU Multiple Precision Arithmetic Library: <http://gmplib.org/>,
- [14] Google: <http://google.com>,
- [15] Ellsea: <http://www.ufr-mi.u-bordeaux.fr/~belabas/pari/scripts/ellsea.tar.gz>,
- [16] PARI/GP: <http://pari.math.u-bordeaux.fr/>,
- [17] The Certicom ECC Challenge: http://www.certicom.com/index.php?action=ecc,ecc_challenge,
- [18] P.C. Oorschot, M.J. Wiener, Parallel Collision Search with Cryptanalytic Applications: <http://cr.yt.to/bib/1999/vanoorschot.pdf>,
- [19] Nicholas Lamb, An investigation into Pollard's Rho Method for attacking Elliptic Curve Cryptosystems: <http://www.cs.ualberta.ca/~nlamb/PollardRhoDiscussion.pdf>,
- [20] Wikipedia, Index Calculus: http://en.wikipedia.org/wiki/Index_calculus_algorithm.

Sobre el Autor

Daniel Lerch (<http://daniellerch.com>) es Ingeniero de Sistemas por la Universidad Oberta de Catalunya y Master en Cisco Networking, Wireless & Network Security. Actualmente trabaja en el sector de las telecomunicaciones como Ingeniero de Software C/C++ en plataformas GNU/Linux.

En su tiempo libre coordina el proyecto *OpenDomo* (<http://opendomo.org>) e investiga en seguridad informática y criptografía.

Con estas indicaciones el lector ya puede construir una primera versión del ataque Rho de Pollard.

Verificación de valores intermedios

Durante el uso normal del programa, principalmente en las llamadas a `seed_F()` y `solve()` nos podemos encontrar con valores que podrían darnos una solución inmediata, sin tener que esperar una colisión. Como se ha explicado anteriormente estos valores se producen al encontrar ecuaciones como $aP+Q=0$, $aP+Q=P$, $aP+Q=Q$ o similares. En el Listado 12 se muestra un Listado de como detectarlas. En estos casos, se llamaría a `solve()` con los parámetros adecuados intentando encontrar una solución.

Optimización

A continuación se plantean algunas posibles mejoras que se pueden aplicar a nuestra implementación:

El rendimiento del algoritmo depende en gran parte de la función $F()$. Para un análisis detallado consultar [19]. La función $F()$ que hemos usado en los Listados es la más básica. Un paso más hacia una mejora en el rendimiento consiste en realizar particiones. Cada partición consistirá en una función diferente a evaluar, de manera que al aplicar $F(S)$ se escoja una partición en función de las características de S y se aplique. Para hacerlo más claro veamos un Listado utilizando tres particiones. Sabremos cuál de las tres particiones utilizar en función de la coordenada X del punto de entrada. Así si realizamos $x \bmod 3$, podremos elegir entre las tres particiones (Listado 13).

Otra forma sencilla de aumentar las posibilidades de encontrar una colisión, consiste en mantener en memoria un listado de los últimos resultados obtenidos. De esta forma cada vez que comparemos el resultado ($U=V?$) actual para ver si hay colisión, realmente lo compararemos con un listado de resultados

anteriores ($U=V1$ o $V2$...?). Así aumentaremos considerablemente las posibilidades de encontrar una colisión. Por otra parte, mantener una lista de resultados demasiado larga podría llevar a producir un descenso del rendimiento por la pérdida de tiempo consultando dicha lista en cada iteración. Por lo que será necesario encontrar un valor adecuado para el tamaño de buffer.

Por otra parte, una mejora interesante es la de utilizar operaciones basadas en coordenadas Montgomery. Entre otros beneficios, eliminan la necesidad de utilizar operaciones de inversión, las cuales son bastante lentas. Consulte [6] y [14].

Para atacar criptosistemas de curva elíptica como los planteados en The Certicom ECC Challenge [17] es necesario recurrir a versiones distribuidas del ataque Rho de Pollard [18]. No entraremos en detalle, pues hay materia suficiente para otro artículo. En cualquier caso, el lector puede consultar [17] para más información sobre el reto.

Conclusiones

Aunque la criptografía de curva elíptica parece ser más segura que los algoritmos de clave pública más utilizados (con claves de menor tamaño), lo cierto es que no se ha investigado tanto. Por lo que los mas de 25 años que avalan la seguridad de criptosistemas como RSA continúan pesando mucho.

Actualmente la criptografía de curva elíptica está creciendo en uso, aunque se mantiene todavía a la sombra del famoso RSA. Quizás en no demasiado tiempo su uso se extienda tanto como merece.

Por otra parte existen diferentes líneas de investigación que hacen pensar en la posibilidad de la existencia de un algoritmo subexponencial que permita resolver el *ECDLP*, pero por ahora el ataque más rápido es el método Rho de Pollard, un algoritmo de tiempo exponencial. ●