Implementación de ECDSA en magma

Aritmética modular

Zelzin Marcela Márquez Navarrete
CINVESTAV – Zacatenco
zmarquez@computacion.cs.cinvestav.mx 14 de diciembre de 2017

Resumen

En este reporte se presenta una breve descripción de ECDSA y los algoritmos de multiplicación modular que se utilizarlo para implementarlo en Magma.

Palabras clave: EDSA, Multiplicación escalar, Escalera de Motgomery

1. El problema del logaritmo discreto de una curva elíptica

La dificultad del problema del logaritmo discreto de una curva elíptica (ECDLP por sus siglas en inglés) es esencial para la seguridad de todos los sistemas criptográficos basados en curvas elípticas. El problema se define como sigue:

Dada una curva elíptica E definida sobre un campo finito $\mathbb{F}_q$, un punto $P \in E(\mathbb{F}_q)$ de orden n, y un punto $Q \in \langle P \rangle$, halle el entero $l \in [0, n-1]$ tal que $Q = lP$. El entero l es llamado el logaritmo discreto de Q para la base P, denotado como $l = \log_P Q$

Los parámetros de la curva elíptica para un sistema criptográfico deben ser cuidadosamente elegidos de manera que resistan todos los ataques sobre el problema. El algoritmo más simple para resolver el ECDLP es una búsqueda exhaustiva donde se calcula la secuencia de puntos $P, 2P, 3P, 4P, \ldots$ hasta que se encuentra Q. El tiempo de ejecución de esta búsqueda requiere aproximadamente de n pasos en el peor de los casos, y $n/2$ en promedio. La búsqueda exhaustiva puede ser derrotada seleccionando una n lo suficientemente grande para representar una cantidad poco factible de cómputos a realizar, e.g., $n \geq 2^{80}$. El mejor ataque de propósito-general sobre ECDLP es una combinación del algoritmo de Pohlig-Hellman y el algoritmo rho de Polard, dicha combinación tienen un tiempo de ejecución completamente exponencial de $O(\sqrt{p})$, donde p es el divisor primo más grande de n. Para resistir este ataque, los parámetros de la curva elíptica deben elegirse de modo que n sea divisible por un primo p lo suficientemente grande para que realizar $\sqrt{p}$ pasos resulte en una cantidad poco factible de cómputos, e.g., $p \geq 2^{160}$. Si además, los parámetros de la curva elíptica son elegidos cuidadosamente de manera que se pueda derrotar cualquier otro tipo de ataque conocido entonces podemos decir que el problema ECDLP es intratable.

Debe notarse que no existe una demostración matemática de que ECDLP sea intratable. Dicha demostración implicaría que $P \neq NP$, lo que respondería una de las preguntas fundamentales de la teoría de la computación.

2. Signature scheme

Este tipo de firmas son el equivalente de una firma escrita a mano, pueden utilizarse para autentificar el origen y la integridad de los datos. Este tipo de firmas son utilizadas comúnmente por autoridades certificadoras para firmar certificados digitales que asocian una entidad y su llave pública [1].

Un signature scheme consiste de cuatro algoritmos:

1. Un *algoritmo de generación de parámetros de dominio* que se encarga de generar los parámetros del dominio.
2. Un *algoritmo de generación de llave* que toma como entrada un conjunto D de parámetros de dominio y genera pares de llaves (Q, d).
3. Un *algoritmo de generación de firma* que toma como entrada los parámetros de dominio D, una llave privada d, un mensaje m, y produce una firma Σ.
4. Un *algoritmo de verificación* que toma como entrada los parámetros de dominio D, una llave pública Q, un mensaje m y una "supuesta" firma Σ. El algoritmo se encarga de aceptar o rechazar dicha firma.

Asumimos que los parámetros de dominio D son válidos, y que la llave pública Q es válida y está asociada a D. El algoritmo de verificación de la firma siempre acepta la entrada (D, Q, m, Σ) si Σ fue generada por el algoritmo de generación de firma a partir de la entrada (D, d, m).

2.1. Seguridad de un Signature Scheme

Se dice que un Signature Scheme es seguro (o GMR-seguro) si no se puede falsificar por un adversario computacional que pueda montar un ataque *adaptive chosen-message*. En otras palabras, un adversario que pueda obtener firmas para cualquier mensaje de su elección de un firmante legítimo debe ser incapaz de producir una firma válida para un mensaje nuevo a menos que la solicite del firmante legitimo.

3. ECDSA

El Algoritmo de Firma Digital de Curva Elíptica (ECDSA por sus siglas en inglés) es el análogo con curvas elípticas del Algoritmo de Firmas Digitales (DSA por sus siglas en inglés). Sea H una función hash criptográfica cuya salida no tiene una longitud en bits mayor que n. Si esta condición no pudiera satisfacerse, entonces es posible truncar la salida de H.

3.1. Demostración de que que la firma ECDSA funciona

Si una firma (r, s) en un mensaje m fue generada por un firmante legítimo, entonces $s \equiv k^{-1}(e + dr)(\mod n)$. Reacomodando se tiene:

Algoritmo 1 Generación de firma mediante ECDSA

1: Seleccione $k \in [1, n-1]$
2: Calcule $kP = (x_1, y_1)$ y convierta x_1 aun entero $\bar{x}_1$.
3: Calcule $r = \bar{x}_1 \mod n$. Si $r = 0$ entonces repita el paso 1.
4: Calcule $e = H(m)$.
5: Calcule $s = k^{-1}(e + dr) \mod n$. Si $s = 0$ entonces repita el paso 1.
6: **return** (r, s).

Algoritmo 2 Generación de firma mediante ECDSA

1: Verifique que r y s son enteros en el intervalo $[1, n-1]$. Si alguna verificación falla entonces regrese "Rechace la firma".
2: Calcule $e = H(m)$.
3: Calcule $w = s^{-1} \mod n$
4: Calcule $u_1 = ew \mod n$ y $u_2 = rw \mod n$
5: Calcule $X = u_1 P + u_2 Q$
6: Si $X = \infty$ entonces **return** "Rechace la firma".
7: Convierta la coordenada x, x_1 de x a un entero $\bar{x}_1$; calcule $v = \bar{x}_1 \mod n$.
8: Si $v = r$ entonces regrese "Acepte la firma" de otra manera
9: **return** "Rechace la firma"

$$k \equiv s^{-1}(e + dr) \equiv s^{-1}e + s^{-1}rd \equiv we + wrd \equiv u_1 + u_2 d(\mod n)$$

4. Montgomery Ladder

Consideramos el problema general de calcular $y = g^k$ en un grupo abeliano $\mathbb{G}$ (escrito multiplicativamente), sobre una entrada g y k. Sea $\sum_{i=0}^{t-1} k_i 2^i$ la expansión binaria del exponente k. La escalera de Montgomery se basa en la siguiente observación. Definiendo $L_j = \sum_{i=j}^{t-1} k_i 2^{i-j}$ y $H_j = L_j + 1$, tenemos

$$L_j = 2L_{j+1} + k_j = L_{j+1} + H_{j+1} + k_j - 1 = 2H_{j+1} + k_j - 2$$

y por lo tanto obtenemos

$$(L_j, H_j) = \begin{cases} (2L_{j+1}, L_{j+1} + H_{j+1}) & \text{si } k_j = 0, \\ (L_{j+1} + H_{j+1}, 2H_{j+1}) & \text{si } k_j = 1. \end{cases} \quad (1)$$

Supongamos que, en cada iteración, un primer registro, digamos R_0, es usado para contener el valor de g^{L_j} y que un segundo registro, digamos R_1, es usado para contener el valor de g^{H_j}. La ecuación 1 implica que

$$(g^{L_j}, g^{H_j}) = \left((g^{L_{j+1}})^2, g^{L_{j+1}} \cdot g^{H_{j+1}}\right) \quad \text{si } k_j = 0$$

y

$$(g^{L_j}, g^{H_j}) = \left(g^{L_{j+1}} \cdot g^{H_{j+1}}, (g^{H_{j+1}})^2\right) \quad \text{si } k_j = 1$$

Algoritmo 3 Montgomery ladder

$R_0 \leftarrow 1$;
$R_1 \leftarrow g$;
for $j = t - 1$ **down to** 0 **do**
    if $k_j = 0$ **then**
        $R_1 \leftarrow R_0 R_1$
        $R_0 \leftarrow (R_0)^2$
    else if $k_j = 1$ **then**
        $R_0 \leftarrow R_0 R_1$
        $R_1 \leftarrow (R_1)^2$
    end if
end for
return R_0

Nótese que $L_0 = k$, esto genera un algoritmo elegante para evaluar $y = g^k$: la escalera de Montgomery[2].
Para aplicaciones criptográficas, el grupo $\mathbb{G}$ puede tomarse como $\mathbb{Z}_N^*$ (por ejemplo, para encriptación/firma RSA o Rabin), $\mathbb{F}_u^*$ (por ejemplo, para intercambio de llave DH), los elementos de una secuencia de Lucas (por ejemplo para firma LUC), los puntos de una curva elíptica (por ejemplo, para firma ECDSA),. . .. Otras aplicaciones prácticas incluyen tests de primalidad y algoritmos de factorización.

5. Código de la implementación

Código 1: Implementación de ECDSA en magma.

```
1  /*Curve Building  */
2  P256    := 2^256 - 2^224 + 2^192 +2^96 -1;
3
4  /* Field */
5  Fp := GF(2^255 - 19);
6  //Fp := GF(P256);
7
8  /* Curve */
9  E := EllipticCurve([Fp | 0, 486662, 0, 1, 0]);
10
11 /* Base-point */
12 P := E![9, 14781619447589544791020593568409986887264606134616475\
13     2889648818377555862374 01];
```

```
14
15 /* Order of P */
16 n := 2^252 + 27742317777372353535851937790883648493;
17 Fn := GF(n);
18
19 /* Functions */
20 BinaryL2R := function(k, p)
21     kbits   := IntegerToSequence(k, 2);
22     L       := #kbits;
23     Q       := 0;
24     G       := p;
25     for i   := L to 1 by -1 do
26         Q   := 2 * Q;
27         if (kbits[i] eq 1) then
28             Q := Q + G;
29         end if;
30     end for;
31     return Q;
32 end function;
33
34 BinaryL2RC := function(k, p)
35     kbits   := IntegerToSequence(k, 2);
36     L       := #kbits;
37     Q       := E!0;
38     G       := p;
39     for i   := L to 1 by -1 do
40         Q   := 2 * Q;
41         if (kbits[i] eq 1) then
42             Q := Q + G;
43         end if;
44     end for;
45     return Q;
46 end function;
47
48 BinaryR2L := function(k, g)
49     kbits   := IntegerToSequence(k, 2);
50     L       := #kbits;
51     Q       := 0;
52     G       := g;
53     for i   := 1 to L do
54         if (kbits[i] eq 1) then
55             Q := Q + G;
56         end if;
57         G := 2 * G;
58     end for;
59     return Q;
60 end function;
61
62 BinaryR2LC := function(k, g)
```

```
    kbits   := IntegerToSequence(k, 2);
    L       := #kbits;
    Q       := E!0;
    G       := g;
    for i   := 1 to L do
        if (kbits[i] eq 1) then
            Q := Q + G;
        end if;
        G := 2 * G;
    end for;
    return Q;
end function;

MontgomeryLadderL2R := function(k, g)
    kbits   := IntegerToSequence(k, 2);
    L       := #kbits;
    Q       := 0;
    G       := g;
    for i   := L to 1 by -1 do
        if (kbits[i] eq 1) then
            Q := Q + G;
            G := 2 * G;
        else
            G := Q + G;
            Q := 2 * Q;
        end if;
    end for;
    return Q;
end function;

MontgomeryLadderL2RC := function(k, g)
    kbits   := IntegerToSequence(k, 2);
    L       := #kbits;
    Q       := E!0;
    G       := g;
    for i   := L to 1 by -1 do
        if (kbits[i] eq 1) then
            Q := Q + G;
            G := 2 * G;
        else
            G := Q + G;
            Q := 2 * Q;
        end if;
    end for;
    return Q;
end function;

KeyPairGen := function(n)
    d   := Random(n - 1);
```

```
    Q   := d * P;
    return d, Q;
end function;

KeyPairGenL2R := function(n)
    d   := Random(n - 1);
    Q   := BinaryL2RC(d, P);
    return d, Q;
end function;

KeyPairGenR2L := function(n)
    d   := Random(n - 1);
    Q   := BinaryR2LC(d, P);
    return d, Q;
end function;

KeyPairGenLadderL2R := function(n)
    d   := Random(n - 1);
    Q   := MontgomeryLadderL2RC(d, P);
    return d, Q;
end function;

ECDSASignatureGen := function(M, d)
    k := Random(n - 1);
    R := k * P;
    r := IntegerRing()!R[1];
    s := Fn!(M + d * r)/k;
    return r, IntegerRing()!s;
end function;

ECDSASignatureGenL2R := function(M, d)
    k := Random(n - 1);
    R := BinaryL2RC(k, P);
    r := IntegerRing()!R[1];
    s := Fn!(M + BinaryL2R(d, r))/k;
    return r, IntegerRing()!s;
end function;

ECDSASignatureGenR2L := function(M, d)
    k := Random(n - 1);
    R := BinaryR2LC(k, P);
    r := IntegerRing()!R[1];
    s := Fn!(M + BinaryR2L(d, r))/k;
    return r, IntegerRing()!s;
end function;

ECDSASignatureGenLadderL2R := function(M, d)
    k := Random(n - 1);
    R := MontgomeryLadderL2RC(k, P);
```

```
    r := IntegerRing()!R[1];
    s := Fn!(M + MontgomeryLadderL2R(d, r))/k;
    return r, IntegerRing()!s;
end function;

ECDSA_Verify := function(M, Q, r, s)
    w   := IntegerRing()!Fn!(1/s);
    u1  := M * w;
    u2  := r * w;
    X   := u1 * P + u2 * Q;
    v   := X[1];
    if (v eq r) then
        printf "Verified\n";
    else
        printf "Verification failed\n";
    end if;
    return 0;
end function;

ECDSA_VerifyL2R := function(M, Q, r, s)
    w   := IntegerRing()!Fn!(1/s);
    u1  := BinaryL2R(M, w);
    u2  := BinaryL2R(r, w);
    X   := BinaryL2RC(u1, P) + BinaryL2RC(u2, Q);
    v   := X[1];
    if (v eq r) then
        printf "Verified\n";
    else
        printf "Verification failed\n";
    end if;
    return 0;
end function;

ECDSA_VerifyR2L := function(M, Q, r, s)
    w   := IntegerRing()!Fn!(1/s);
    u1  := BinaryR2L(M, w);
    u2  := BinaryR2L(r, w);
    X   := BinaryR2LC(u1, P) + BinaryR2LC(u2, Q);
    v   := X[1];
    if (v eq r) then
        printf "Verified\n";
    else
        printf "Verification failed\n";
    end if;
    return 0;
end function;

ECDSA_VerifyLadderL2R := function(M, Q, r, s)
    w   := IntegerRing()!Fn!(1/s);
```

```
    u1  := MontgomeryLadderL2R(M, w);
    u2  := MontgomeryLadderL2R(r, w);
    X   := MontgomeryLadderL2RC(u1, P) + MontgomeryLadderL2RC(u2, Q);
    v   := X[1];
    if (v eq r) then
        printf "Verified\n";
    else
        printf "Verification failed\n";
    end if;
    return 0;
end function;

/* Main */
//Keys generation
PrivateKey, PublicKey := KeyPairGen(n);
PrivateKey, PublicKey := KeyPairGenL2R(n);
PrivateKey, PublicKey := KeyPairGenR2L(n);
PrivateKey, PublicKey := KeyPairGenLadderL2R(n);

// Message to be signed
M       := Random(n - 1);
r, s    := ECDSASignatureGen(M, PrivateKey);
r, s    := ECDSASignatureGenL2R(M, PrivateKey);
r, s    := ECDSASignatureGenR2L(M, PrivateKey);
r, s    := ECDSASignatureGenLadderL2R(M, PrivateKey);

// Verification
ECDSA_Verify(M, PublicKey, r, s);
ECDSA_VerifyL2R(M, PublicKey, r, s);
ECDSA_VerifyR2L(M, PublicKey, r, s);
ECDSA_VerifyLadderL2R(M, PublicKey, r, s);
```

Referencias

1. D. Hankerson, A. J. Menezes, y S. Vanstone, *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.
2. M. Joye y S.-M. Yen, «The montgomery powering ladder.» Springer.