Fast \mathbb{G_2}

subgroup check in BN254

Authors: Antonio Sanso and Chih Cheng Liang (Ethereum Foundation -https://ethereum.foundation/)

BLS12-381 is now universally recognized as the pairing curve

to be used given our present knowledge

. As propably know though BN254 is currently the only curve with precompiled contracts on Ethereum for elliptic curve addition, scalar multiplication, and pairings (EIP 196, EIP 197), this until the inclusion of EIP-2537.

Did you say \mathbb{G_2}

?

One problem with using BN254 in smart contract though is the expensive G2

operations. In particular the required validation of checking that a point lies in the correct subgroup is prohibitively expensive.

Subgroup membership testing for \mathbb{G_2}

The naive way to check that a group lies in the correct subgroup is to multiply by the order. This might be particular expensive for G2

on BN254. The order of the curve is a 254 bits number. According to the benchmark data provided wih the BN256G2 contract. Concrete number shows that the gas cost is about 4,000,000 for a 256-bit scalar multiplication

. So the cost of the naive subgroup check is dominated by the scalar multiplication:

function isOnSubgroupG2Naive(uint256[4] memory point) internal view returns (bool) { uint256 t0; uint256 t1; uint256 t2; uint256 t3; // solium-disable-next-line security/no-inline-assembly assembly { t0 := mload(add(point, 0)) t1 := mload(add(point, 32)) // y0, y1 t2 := mload(add(point, 64)) t3 := mload(add(point, 96)) } uint256 xx; uint256 xy; uint256 yx; uint256 yy;

```
    (xx,xy,yx,yy) = BN256G2.ECTwistMul(r,t0,t1,t2,t3);
    return xx==0 && xy==0 && yx==0  && yy==0;
}
```

Can we do any better?

Probably yes. BN254 has j

-invariant equal to zero so is equipped with an extra endomorphism (in addition to scalar multiplication and Frobenius). So at the very least we could rely on GLV.

Better than GLV

The last section of BN254 For The Rest Of Us already suggests a nice improvements that leverages the untwist-Frobenius-twist endomorphism introduced by Galbraith-Scott:

\psi = \Psi^{-1}\circ\phi_p\circ\Psi

where \Psi

is the twist map and \phi_p

is the Frobenius morphism.

Now thanks to a Theorem by El Housni et al in order to verify that a point is in the correct subgroup is enough to check that \psi(P) = [6x^2]P

where x

is the seed of the BN254 curve. Given the fact that computing the endomorphism on the left side is really fast this is really a good improvement. We passed from a 256-bit scalar multiplication (the order) to a 125-bit scalar multiplication

(the seed x

is relatively small 63 bits

).

…and even better

This is not the end of a story though. Arecent paper improved even further the status quo (employing the same endomorphism).

The work required is a single point multiplication by the BN seed parameter x

(with some corner case that we are going to check below)!!

Indeed assuming that the seed x

satisfies that x \not\equiv 4~(\textrm{mod}~13)

and x \not\equiv92~(\textrm{mod}~97)

in order to verify that a point is in the correct subgroup is enough to check that [x+1]P + \psi([x]P) + \psi^2(xP)= \psi^3([2x]P)

. Basically a 64-bit scalar multiplication

.

For BN254 we have

sage: x 4965661367192848881 sage: assert x % 13 != 4 sage: assert x % 97 != 92

At this point we went ahead and implemented these function in solidity going from a 4,000,000 gas cost to about a 1,000,000

function _FQ2Mul(uint256 xx, uint256 xy, uint256 yx, uint256 yy) internal pure returns (uint256, uint256) { return (submod(mulmod(xx, yx, N), mulmod(xy, yy, N), N), addmod(mulmod(xx, yy, N), mulmod(xy, yx, N), N)); }

```
function _FQ2Conjugate(
    uint256 x, uint256 y
) internal pure returns (uint256, uint256) {
    return (x, N-y);
}

function endomorphism(uint256 xx, uint256 xy,
    uint256 yx, uint256 yy) internal pure returns (uint256[4] memory end) {

    // Frobenius x coordinate
    (uint256 xxp,uint256 xyp) = _FQ2Conjugate(xx,xy);
    // Frobenius y coordinate
    (uint256 yxp,uint256 yyp) = _FQ2Conjugate(yx,yy);
    // x coordinate endomorphism
    (uint256 xxe,uint256 xye) = _FQ2Mul(epsExp0x0,epsExp0x1,xxp,xyp);
    // y coordinate endomorphism
    (uint256 yxe,uint256 yye) = _FQ2Mul(epsExp1x0, epsExp1x1,yxp,yyp);

    end[0] = xxe;
    end[1] = xye;
    end[2] = yxe;
    end[3] = yye;
}

function isOnSubgroupG2(uint256[4] memory point)
    internal
    view
    returns (bool)
{
    uint256 t0;
    uint256 t1;
    uint256 t2;
    uint256 t3;
    // solium-disable-next-line security/no-inline-assembly
    assembly {
        t0 := mload(add(point, 0))
        t1 := mload(add(point, 32))
        // y0, y1
        t2 := mload(add(point, 64))
        t3 := mload(add(point, 96))
    }
```

```
    uint256 xx;
    uint256 xy;
    uint256 yx;
    uint256 yy;
    //s*P
    (xx,xy,yx,yy) = BN256G2.ECTwistMul(s,t0,t1,t2,t3);

    uint256 xx0;
    uint256 xy0;
    uint256 yx0;
    uint256 yy0;
    //(s+1)P
    (xx0,xy0,yx0,yy0) = BN256G2.ECTwistAdd(t0,t1,t2,t3,xx,xy,yx,yy);
    //phi(sP)
    uint256[4] memory end0 = endomorphism(xx,xy,yx,yy);
    //phi^2(sP)
     uint256[4] memory end1 = endomorphism(end0[0],end0[1],end0[2],end0[3]);
    //(s+1)P + phi(sP)
    (xx0,xy0,yx0,yy0) = BN256G2.ECTwistAdd(xx0,xy0,yx0,yy0,end0[0],end0[1],end0[2],end0[3]);
    //(s+1)P + phi(sP) + phi^2(sP)
    (xx0,xy0,yx0,yy0) = BN256G2.ECTwistAdd(xx0,xy0,yx0,yy0,end1[0],end1[1],end1[2],end1[3]);
    //2sP
    (xx,xy,yx,yy) = BN256G2.ECTwistAdd(xx,xy,yx,yy,xx,xy,yx,yy);
    //phi^2(2sP)
    end0 = endomorphism(xx,xy,yx,yy);
    end0 = endomorphism(end0[0],end0[1],end0[2],end0[3]);
    end0 = endomorphism(end0[0],end0[1],end0[2],end0[3]);

    return xx0 == end0[0] && xy0==end0[1] && yx0==end0[2]  && yy0==end0[3];
}
```

You can find the full code in[GitHub - asanso/bls_solidity_python]

Acknowledgments:

we would like to thank Mustafa Al-Bassam, Kobi Gurkan, Simon Masson and Michael Scott for fruitful discussions.