Attacks and weaknesses of BLS aggregate signatures

Nguyen Thoi Minh Quan *†

Abstract

This article discusses existing attacks and known weaknesses of BLS aggregate signatures. The goal is clarify the threat model of BLS aggregate signatures, what security properties that they have and do *not* have. It's unfortunate that the weaknesses are *not* documented anywhere in BLS RFC draft v4 [1]. Confusion, ambiguity, misunderstanding all may cause security issues in practice. We hope that this article can help cryptographic practitioners make *informed decisions* when using BLS aggregate signatures and deploy mitigations at the application/protocol layer because BLS aggregate signatures might not have security guarantees that you need.

Contents

1 Introduction **2**

2 Weaknesses of BLS aggregate signature's definition **3**
2.1 BLS aggregate signature's definition 3
2.2 What does the definition say? 3
2.3 What does the definition *not* say? 3
2.4 Which use case the aggregate signature definition is appropriate? 4
2.5 Which use case the aggregate signature definition is *not* appropriate? 4

3 Known attacks against BLS aggregate signatures **4**
3.1 "Splitting zero" attack 4
3.2 Consensus attacks 5
3.2.1 *Equivalent* interfaces return *different* verification results 5
3.2.2 FastAggregateVerify's *aggregation order* leads to *different* verification results 6
3.2.3 Is batch verification a rescue? 8
3.3 Denial-of-service attack 8
3.4 Do we know all the attacks? 8

4 Acknowledgements **8**

A Proof of concept attacks **8**

*https://www.linkedin.com/in/quan-nguyen-a3209817, https://scholar.google.com/citations?user=9uUqJ9IAAAAJ, https://github.com/cryptosubtlety, msuntmquan@gmail.com, 2021-03-22

†Disclaimer: This is my personal research, and hence it does not represent the views of my employer.

1 Introduction

Besides "splitting zero" attack[2], there is also consensus bug in BLS RFC draft v4 standard [2], [3] where 2 equivalent interfaces return conflicting results: $FastAggregateVerify((pk_1, pk_2), m, 0) = false$c, $AggregateVerify((pk_1, pk_2), (m, m), 0) = true$, $pk_1 + pk_2 = 0$. Please read the article [2] for background as this article is not self-contained. Consensus bugs looked hilarious to me so I dug deeper in that direction. After a month, I thought I found new consensus attacks in the standard. Later, I've learned that the "new" consensus attacks that I found have been discovered 14 years ago by Jan Camenisch et al. [4]. Jan Camenisch et al. [4] deserve *full credit* for their consensus attack in this article. In summary, there are 2 known types of attacks against BLS aggregate signature:

1. "Splitting zero" attack[2]. Proof of concept attacks were provided in [2].
2. Consensus attacks by Jan Camenisch et al.[4]. I will provide concrete proof of concept attack in this article.

After seeing the above attacks, I was confused. How can those attacks exist? BLS aggregate signatures standardized in BLS RFC draft v4 have security proofs. I dug deeper into BLS aggregate signature definition and read Jan Camenisch et al. [4] article and I've realized that the threat model in BLS aggregate signature definition might not be appropriate for certain practical applications/protocols. I want to raise awareness to cryptographic practitioners so that you can make informed decisions. I don't say that we shouldn't use BLS aggregate signature, but using cryptographic primitives without knowing their weaknesses is dangerous.

2 Weaknesses of BLS aggregate signature's definition

2.1 BLS aggregate signature's definition

Let's take a look at the definition from Dan Boneh and Victor Shoup's book [5].

The scheme is correct if for all $\boldsymbol{pk} = (pk_1, \ldots, pk_n)$, $\boldsymbol{m} = (m_1, \ldots, m_n)$, *and* $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_n)$, *if* $V(pk_i, m_i, \sigma_i) = \mathsf{accept}$ *for* $i = 1, \ldots, n$ *then* $\Pr\big[VA\big(\boldsymbol{pk}, \boldsymbol{m}, A(\boldsymbol{pk}, \boldsymbol{\sigma})\big) = \mathsf{accept}\big] = 1$.

Definition of secure aggregation. Our security definition must properly model rogue public key attacks as above. In the security game the adversary is asked to create a valid aggregate signature over a number of public keys, where the adversary controls all but one of the public keys. Before outputting the aggregate forgery, the adversary can issue signing queries for the one public key that it does not control.

Attack Game 15.2 (Aggregate signature security). For a given aggregate signature scheme $\mathcal{SA} = (G, S, V, A, VA)$ with message space $\mathcal{M}$, and a given adversary $\mathcal{A}$, the attack game runs as follows:

- The challenger runs $(pk, sk) \xleftarrow{\text{R}} G()$ and sends pk to $\mathcal{A}$.
- $\mathcal{A}$ queries the challenger. For $i = 1, 2, \ldots,$ the ith *signing query* is a message $m^{(i)} \in \mathcal{M}$. The challenger computes $\sigma_i \xleftarrow{\text{R}} S(sk, m^{(i)})$, and then gives σ_i to $\mathcal{A}$.
- Eventually $\mathcal{A}$ outputs a candidate aggregate forgery $(\boldsymbol{pk}, \boldsymbol{m}, \sigma_{\text{ag}})$ where $\boldsymbol{pk} = (pk_1, \ldots, pk_n)$ and $\boldsymbol{m} = (m_1, \ldots, m_n) \in \mathcal{M}^n$.

We say that the adversary wins the game if the following conditions hold:

- $VA(\boldsymbol{pk}, \boldsymbol{m}, \sigma_{\text{ag}}) = \mathsf{accept}$,
- there is at least one $1 \leq j \leq n$ such that (1) $pk_j = pk$, and (2) $\mathcal{A}$ did not issue a signing query for m_j, meaning that $m_j \notin \{m^{(1)}, m^{(2)}, \ldots\}$.

We define $\mathcal{A}$'s advantage with respect to $\mathcal{SA}$, denoted $\mathrm{ASIG}\mathsf{adv}[\mathcal{A}, \mathcal{SA}]$, as the probability that $\mathcal{A}$ wins the game. □

2.2 What does the definition say?

+ If individual signatures are correct then the aggregate signature is correct.

+ The attack game models a *malicious aggregator* that tries to forge a valid signature for a signer's key it doesn't control.

2.3 What does the definition *not* say?

1. If the aggregate signature is correct then individual signatures are correct with high probability.

2. What happens when the signers collude with each other?

3. What happens when the signers are malicious?

In regard to "existential unforgeability" security property, the aggregate definition does cover 2. and 3. However, the definition doesn't cover what malicious (colluded) signers can achieve with regard to other security problems such as message binding or causing conflicting views among verifiers.

2.4 Which use case the aggregate signature definition is appropriate?

The classic use case in aggregate signature is aggregation of certificate authorities (CA)'s signatures. This use case has the correct threat model because the client (e.g. browser and pre-installed cert store in iOS/Android) completely trust the CAs. In other words:

+ The CAs are modeled as honest signers.

+ Each client's view is local, i.e., the global view of signatures among clients doesn't matter.

+ The only potential malicious actor is the aggregator.

Aggregate signature's definition perfectly captures the above scenario.

2.5 Which use case the aggregate signature definition is *not* appropriate?

Cryptocurrency is the use case where aggregate signature's threat model is *not* appropriate.

+ Cryptocurrency always has to deal with malicious signers.

+ Cryptocurrency has to deal with colluded signers. For instance, proof-of-work or proof-of-state have to deal with 1/2 or 1/3 colluded signers' attacks.

+ The global view of signatures is important as it needs consensus.

It's worth stressing that I don't say we shouldn't use BLS aggregate signature in cryptocurrency. What I'm saying is that the cryptographic layer (BLS aggregate signature) fails to model the threat that we need and we have to achieve certain security properties by other means at the application/protocol layer. I also don't claim that attacks at the cryptographic layer lead to attacks at the application/protocol layer. This is my and offensive security engineers' future research. What I'm saying is that attacks at the cryptographic layers are real and you have to design applications/protocols with those attacks in mind.

3 Known attacks against BLS aggregate signatures

Note that rogue public key attack [6] [1] is captured in the aggregate signature's definition, so it will be ignored in this article. The following 2 attacks in the next sections do not violate BLS aggregate signatures' definition, i.e., BLS aggregate signatures' definition fails to capture them.

3.1 "Splitting zero" attack

See the article [2]. This attack needs *colluded* signers whose sum of their public keys is 0.

3.2 Consensus attacks

At a high level, the attacker's goal is to create a set of invalid individual signatures, but their aggregate signature is valid. Therefore, some users will see valid signatures while others see invalid signatures, i.e., the views among users are split. Note that the attack only requires a few (2,3 or 4) malicious individual signatures to cause conflicting views [1] among users as long as the set of to-be-verified signatures include attacker-controlled signatures. In other words, the attack does not require majority signatures.

Jan Camenisch et al. [4] found the attack 14 years ago. I'll describe the attack from the BLS standard's *interfaces* perspective. The attacks work for all schemes: basic scheme, proof-of-possession, message augmentation and there are multiple variants. For clarity, I'll only describe 2 simplest variants for proof-of-possession.

3.2.1 *Equivalent* interfaces return *different* verification results

Let's say there are 4 messages and signatures where $m_1 = m_2, m_3 = m_4$:

$$
\begin{aligned}
m_1 &= "message0", \sigma_1 \\
m_2 &= "message0", \sigma_2 \\
m_3 &= "message1", \sigma_3 \\
m_4 &= "message1", \sigma_4
\end{aligned}
$$

The attacker's goal is create malicious signatures so that 2 different users who see the *same* malicious signatures, but use 2 *equivalent interfaces* will get 2 different verification results. The attacker creates the following malicious signatures

$$
\begin{aligned}
\sigma'_1 &= \sigma_1 - 2P \\
\sigma'_2 &= \sigma_2 + P \\
\sigma'_3 &= \sigma_3 - P \\
\sigma'_4 &= \sigma_4 + 2P
\end{aligned}
$$

where P is a valid point in the subgroup. You may ask how does the attacker create $\sigma'_1, \sigma'_2, \sigma'_3, \sigma'_4$? Here are a few scenarios:

+ The attacker has the private keys sk_1, sk_2, sk_3, sk_4, the attacker signs m_1, m_2, m_3, m_4 and immediately modify $\sigma_1, \sigma_2, \sigma_3\sigma_4$ to $\sigma'_1, \sigma'_2, \sigma'_3, \sigma'_4$ to mount consensus attack.

+ The attacker doesn't have the private keys sk_1, sk_2, sk_3, sk_4 but the attack can observe the traffic on the wire. For instance, the attacker is a proxy who sees $\sigma_1, \sigma_2, \sigma_3, \sigma_4$ and the attacker modifies the signatures to $\sigma'_1, \sigma'_2, \sigma'_3, \sigma'_4$ to mount consensus attack.

Note that $\sigma'_1 + \sigma'_2 + \sigma'_3 + \sigma'_4 = \sigma_1 + \sigma_2 + \sigma_3 + \sigma_4$. Furtheremore, as $\sigma_1, \sigma_2, \sigma_3, \sigma_4, P$ are all in the subgroup, $\sigma'_1, \sigma'_2, \sigma'_3, \sigma'_4$ are in the correct subgroup. In other words, *the subgroup check can't detect and prevent the attack.*

User1 uses the $AggregateVerify$ interface to verify 4 messages together $AggregateVerify$ $([pk_1, pk_2, pk_3, pk_4]$, $[m_1, m_2, m_3, m_4]$, $\sigma'_1 + \sigma'_2 + \sigma'_3 + \sigma'_4) = true$

[1] In term of terminology, consensus bugs in this article means *different* interpretation/views of the *same* data such as the *same* signatures.

User2 wants to achieve the same goal but wants to optimize the verification, so user2 decides to FastAggregateVerify the first 2 messages and the last 2 messages and AggregateVerify the final result.

In the first step, user2 computes the following

$$\begin{aligned}
&\sigma'_{12} = Aggregate([\sigma'_1, \sigma'_2]) \\
&\sigma'_{34} = Aggregate([\sigma'_3, \sigma'_4]) \\
&pk_{12} = AggregatePKs([pk_1, pk_2]) \\
&pk_{34} = AggregatePKs([pk_3, pk_4]) \\
&FastAggregateVerify([pk_1, pk_2], m1, \sigma'_{12}) \\
&FastAggregateVerify([pk_3, pk_4], m3, \sigma'_{34})
\end{aligned}$$

In the second step, user2 computes the following $AggregateVerify$ $([pk_{12}, pk_{34}], [m_1, m3], \sigma'_{12} + \sigma'_{34})$. However, we notice that

$$\begin{aligned}
\sigma'_{12} &= \sigma_1 - 2P + \sigma_2 + P = \sigma_1 + \sigma_2 - P \\
\sigma'_{34} &= \sigma_3 - P + \sigma_4 + 2P = \sigma_3 + \sigma_4 + P
\end{aligned}$$

Therefore $FastAggregateVerify([pk_1, pk_2], m_1, \sigma'_{12}) = false$, $FastAggregateVerify([pk_3, pk_4], m_3, \sigma'_{34}) = false$. It means that user2 sees invalid signatures in the first step and so it won't continue the second step. In other words, user1 sees valid signatures while user2 sees invalid signatures.

We can succinctly explain the bug by using math as follows. We expect that

$$\begin{aligned}
&AggregateVerify([pk_1, pk_2, pk_3, pk_4], [m_1, m_2, m_3, m_4], \sigma'_1 + \sigma'_2 + \sigma'_3 + \sigma'_4) \\
&= FastAggregateVerify([pk_1, pk_2], m_1, \sigma'_{12}) \\
&\text{and } FastAggregateVerify([pk_3, pk_4], m_3, \sigma'_{34}) \\
&\text{and } AggregateVerify([pk_{12}, pk_{34}], [m_1, m_3], \sigma'_{12} + \sigma'_{34})
\end{aligned}$$

Both sides of the equation should return the same result because the right hand side is just an optimization of the left hand side. However, with the above test vector, the left hand side returns true, while the right hand side returns false.

3.2.2 FastAggregateVerify's *aggregation order* leads to *different* verification results

Similar to the above section, let's say users want to FastAggregateVerify the message m with 3 signatures $\sigma_1, \sigma_1, \sigma_3$. The attacker creates the following malicious signatures

$$\begin{aligned}
\sigma'_1 &= \sigma_1 - 2P \\
\sigma'_2 &= \sigma_2 - P \\
\sigma'_3 &= \sigma_3 + 3P
\end{aligned}$$

Note that $\sigma'_1 + \sigma'_2 + \sigma'_3 = \sigma_1 + \sigma_2 + \sigma_3$. Furtheremore, as $\sigma_1, \sigma_2, \sigma_3, P$ are all in the subgroup, we have $\sigma'_1, \sigma'_2, \sigma'_3$ are in the correct subgroup. In other words, *the subgroup check can't detect and prevent the attack.*

User1 computes $FastAggregateVerify([pk_1, pk_2, pk_3,\ m, \sigma'_1 + \sigma'_2 + \sigma'_3)) = true$

User2 first FastAggregateVerify first 2 signatures $\sigma'_{12} = \sigma'_1 + \sigma'_2 = \sigma_1 + \sigma_2 - 3P$ and then FastAggregateVerify the whole thing in the final step:

1. $FastAggregateVerify([pk_1, pk_2], m, \sigma'_{12})$
2. $FastAggregateVerify([pk_{12}, pk_3], m, \sigma'_{12} + \sigma'_3)$

As the first step return false, user2 discards σ'_{12} and won't continue with the second step. In other words, user2 sees invalid signatures while user1 sees valid signatures.

3.2.3 Is batch verification a rescue?

It's worth mentioning that while aggregate signatures is vulnerable to the above consensus attacks, it's not the weakness of stronger definition "Definition 2.1 (Batch Verification of Signatures)" in the Jan Camenisch et al.'s paper [4]

Definition 2.1 (Batch Verification of Signatures) *Let ℓ be the security parameter. Suppose* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ *is a signature scheme, $k, n \in \mathrm{poly}(\ell)$, and $(pk_1, sk_1), \ldots, (pk_k, sk_k)$ are generated independently according to* $\mathsf{Gen}(1^\ell)$. *Let $PK = \{pk_1, \ldots, pk_k\}$. Then we call probabilistic* Batch *a batch verification algorithm when the following conditions hold:*

- *If $pk_{t_i} \in PK$ and* $\mathsf{Verify}(pk_{t_i}, m_i, \sigma_i) = 1$ *for all $i \in [1, n]$, then* $\mathsf{Batch}((pk_{t_1}, m_1, \sigma_1), \ldots, (pk_{t_n}, m_n, \sigma_n)) = 1$.
- *If $pk_{t_i} \in PK$ for all $i \in [1, n]$ and* $\mathsf{Verify}(pk_{t_i}, m_i, \sigma_i) = 0$ *for some $i \in [1, n]$, then* $\mathsf{Batch}((pk_{t_1}, m_1, \sigma_1), \ldots, (pk_{t_n}, m_n, \sigma_n)) = 0$ *except with probability negligible in ℓ, taken over the randomness of* Batch.

One might ask whether batch verification a rescue? Not really. It's true that batch verification definition prevents consensus attacks. However, you have to pay a significant cost. The advantage of aggregate signature is you only need 1 aggregate signature which saves bandwidth and storage. On the other hand, batch verification requires *all* individual signatures, so it doesn't save bandwidth or storage.

3.3 Denial-of-service attack

This attack was discussed by Justin Drake and DavidYakira [7]. Aggregate signature's verification is fast in the good case when all signatures are valid. However, if there are invalid individual signatures, the verifier has to hunt down the invalid individual signatures. It's worth mentioning that *anyone* can create invalid signatures, not just the malicious signers. Therefore, we never be sure who create invalid signatures to pusish them. It's not clear what is the worst complexity of finding k invalid signatures among n signatures.

3.4 Do we know all the attacks?

As discussed in section "What does the definition *not* say?", there are multiple scenarios that the aggregate signatures' definition and its proof doesn't capture. It's not clear whether we found all the attacks that are outside the scope of the definition.

4 Acknowledgements

Thanks twitter account @psiv_ [2] for valuable feedback.

A Proof of concept attacks

The proof of concept attacks should only be used for educational purposes. I make PoC attacks in py_ecc [8] because it's easy to use, but it's not the libraries' bugs. It's the inherent weakness of BLS aggregate signatures.

[2] Sorry, I don't know the real name to give a proper thank you.

```
git clone -n https://github.com/ethereum/py_ecc.git && cd py_ecc
git checkout -b poc 8ddea32b693f7c71fff3b68fca9fd8804ebf33cb
pip install .

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_primitives 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)

# 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('\n\nConsensus attack against proof-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_prime
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])))


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])))
```

References

[1] Dan Boneh, Sergey Gorbunov, Riad S. Wahby, Hoeteck Wee, and Zhenfei Zhang. https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-04.

[2] Nguyen Thoi Minh Quan. 0. https://eprint.iacr.org/2021/323.pdf.

[3] Nguyen Thoi Minh Quan. Proposals to fix bls rfc v4's message binding security property. https://github.com/cfrg/draft-irtf-cfrg-bls-signature/issues/38.

[4] Jan Camenisch, Susan Hohenberger, and Michael Østergaard Pedersen. Batch verification of short signatures.

[5] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography.*

[6] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps.

[7] Justin Drake and DavidYakira. Pragmatic signature aggregation with bls. https://ethresear.ch/t/pragmatic-signature-aggregation-with-bls/2105.

[8] https://github.com/ethereum/py_ecc/commit/8ddea32b693f7c71fff3b68fca9fd8804ebf33cb.