Security of BLS batch verification

By JP Aumasson (Taurus), Quan Thoi Minh Nguyen, and Antonio Sanso (Ethereum Foundation)

Thanks to Vitalik Buterin for his feedback.

[

batch

886×564 101 KB

](https://ethresear.ch/uploads/default/original/2X/0/0a7eecbe5ee30052db925d0bbddf1aa0b800fab7.jpeg)

In a 2019 post Vitalik Buterin introduced Fast verification of multiple BLS signatures. Quoting his own words this is

a purely client-side optimization that can voluntarily be done by clients verifying blocks that contain multiple signatures

The original post includes some preliminary security analysis, but in this post we'd like to formalize it a bit and address some specific risks in the case of:

- Bad randomness
- Missing subgroup checking

We described several attacks that work in those cases, and provide proof-of-concept implementations using the Python reference code.

The batch verification construction

This technique works as follows, given n

aggregate signatures S_i

, respective public keys P_i

, each over a number m_i

of messages (note that each aggregate signature may cover a distinct number of messages, that is, we can have $m_i\neq m_j$

for $i\neq j$

):

$e(S_i, P) = \prod_{j=1}^{m_i} e(P_{i,j}, M_{i,j}), i=1,\dots,n$

The naive method thus consists in checking these n

equalities, which involves $n+\sum_{i=1}^n m_i$

pairings, the most calculation-heavy operation.

To reduce the number of pairings in the verification, one can further aggregate signatures, as follows: the verifier generates n

random scalars $r_i \geq 1$

, and aggregates the signatures into a single one:

$S^\star = r_1S_1 + \cdots r_n S_n$

the verifier also "updates" the signed messages (as their hashes to the curve) to integrate the coefficient of their batch, defining

$M_{i,j}'=r_i M_{i,j}, i=1,\dots,n, j=1,\dots,m_i$

Verification can then be done by checking

$e(S^\star,G)=\prod_{i=1}^n \prod_{j=1}^{m_i} e(P_{i,j},M_{i,j}')$

Verification thus saves $n-1$

pairing operations, but adds $n+\sum_{i=1}^n m_i$

scalar multiplications. Note that if the verification fails, then the verifier can't tell which (aggregate) signatures are invalid.

In particular, if $m_i=1, \forall i$

, then verification requires $n+1$

pairings and $2n$

multiplications, instead of $2n$

pairings. Depending on the relative cost of pairings vs. scalar multiplications, the speed-up may vary (see this post for pairings implementations benchmarks)

Security goals

Informally, the batch verification should succeed if and only if

all signatures would be individually successfully verified. One or more (possibly all) of the signers may be maliciously colluding. Note that this general definition implicitly covers cases where

- Attackers manipulate public keys (without necessarily knowing the private key),
- Malicious signers pick their signature/messages tuples depending on honest signers' input.

For a formal study of batch verification security, we refer to the 2011 paper of Camenisch, Hohenberger, Østergaard Pedersen.

Note that in the Ethereum context, attackers have much less freedom than this attack model allows, but we nonetheless want security against strong attackers to prevent any unsafe overlooked scenario.

The importance of randomness

The randomness has been already discussed in the original post. Buterin points out that if coefficients are constant, then an attacker could validate invalid signature:

the randomizing factors are necessary: otherwise, an attacker could make a bad block where if the correct signatures are C_1

and C_2

, the attacker sets the signatures to C_1+D

and C_2-D

for some deviation D

. A full signature check would interpret this as an invalid block, but a partial check would not.

This observation generalizes to predictable coefficients, and more specifically to the case where α

attackers collude and any subset of $\beta\leq \alpha$

coefficients are predictable among those assigned to attackers' input.

Note that the coefficient can't be deterministically derived from the batch to verify, for this would make coefficients predictable to an attacker.

Buterin further discusses the possible ranges of r_i

to keep the scheme safe:

We can also set other r_i

values in a smaller range, eg. $1…2^{64}$

, keeping the cost of attacking the scheme extremely high (as the ri values are secret, there's no computational way for the attacker to try all possibilities and see which one works); this would reduce the cost of multiplications by ~4x.

A simple attack

If coefficients are somehow predictable, then the above trivial attack can be generalized to picking as signatures $S_1=(C_1+D_1)$

and $S_2=(C_2+D_2)$

such that $r_1 D_1=-r_2 D_2$

.

If coefficients are uniformly random b

-bit signed

values, then there is a chance $1/2^{b-1}$

that two random coefficients satisfy this equality (1 bit being dedicated to the sign encoding), and thus that verification passes. Otherwise, the chance that $r_1 D_1=-r_2 D_2$

for random coefficients is approximately 2^{-n}

, with n

the size of the subgroup

in which lie D_1

and D_2

.

However, the latter attack, independent of the randomness size, will fail if signatures are checked to fall in the highest-order subgroup (see the section The importance of subgroup checks

).

Signature manipulation

In practice, BLS signatures are not purely used as signatures, but implementers take advantage of other informal and implicit security properties such as uniqueness and “commitment”. That is, if the private key is fixed, given a message not controlled by the signer, the signer can’t manipulate the signature. In the case of bad randomness, such use cases may be insecure.

For instance, let’s say there is a lottery based on the current time interval, i.e., at time interval t

outside of signers’ control if $S_i = \mathsf{Sign}(sk_i, t), (S_i)_x \mod N = 0$

(where N

is just some small number, e.g., the number of signers) then the signer i

wins the lottery. The first two signers can collude to win the lottery as follows. The first signer chooses a random point $P \in G_1$

and offline bruteforces a k

such that $(S_1')_x \mod N = (S_1 - kP)_x \mod N = 0$

where $S_1 = \mathsf{Sign}(sk_1, t)$

. The second signer computes $S_2' = S_2 + (k r_1/r_2)P$

where $S_2 = \mathsf{Sign}(sk_2, t)$

. We have

$r_1S_1' + r_2S_2' + r_3S_3 = r_1(S_1 - kP) + r_2(S_2 + kr_1/r_2P) + r_3S_3$

that is, calculating further:

$r_1S_1 - r_1kP + r_2S_2 + kr_1P + r_3S_3 = r_1S_1 + r_2S_2 + r_3S_3$

What it means is that the first and second signer can manipulate the signatures so that the first signer wins the lottery while making batch verification valid.

The importance of subgroup checks

The current state-of-the-art pairing curves such BLS12-381 are not prime-order, for this reason theBLS signatures IRTF document warns about it and mandates a subgroup_check(P)

while performing some operations. For batch verification, subgroup check is essential, aspreviously commented.

The lack of subgroup validation appears to have a greater impact with batch verification than with standard BLS validation, where points in small subgroup are not known to trivially be exploitable, as noted in the BLS signatures IRTF document:

For most pairing-friendly elliptic curves used in practice, the pairing operation e (Section 1.3) is undefined when its input points are not in the prime-order subgroups of E_1

and E_2

. The resulting behavior is unpredictable, and may enable forgeries.

and in the recent Michael Scott's paper:

The provided element is not of the correct order and the impact of, for example, inputting a supposed $\mathbb{G}_2$

point of the wrong order into a pairing may not yet be fully understood.

The difficulty of performing an actual attack is also due the fact that $\mathsf{gcd}(h_1,h_2)=1$

where h_1

and h_2

are the respective cofactors of E_1

and E_2

. Hence pushing the two pairing's input to lie in the same subgroup is not possible. Let's see an example based onBLS12-381, where

$h_1=3 \times 11^2 \times 10177^2 \times 859267^2 \times 52437899^2$

and

$h_2=13^2 \times 23^2 \times 2713 \times 11953 \times 262069 \times p_{136}$.

Choosing an invalid signature S_1

of order 13

and a public key $P_{1,1}$

of order 3

, a potential attack would succeed (to pass batch verification) with probability $1/39=1/3 \times 1/13$

. The attack assumes an implementation that multiplies the public keys by r_i

's rather than the messages (in order to gain speed when there a distinct messages signed by a same key)as described here and implemented for example by Milagro.

An attack then works as follows in order to validate a batch of signatures that includes a signature that would not have passed verification:

- Pick S_1

of order 13, and $P_{1,1}$

of order 3. The chance that $r_1 S_1 = r_1 P_{1,1} = \mathcal{O}$

is $1/39$

, which is the attack success rate. In such case, we have:

- S^\star = r_2S_2 + \cdots r_n S_n

.

- Suppose, without loss of generality, that m_1=1

(namely the first batch to verify is a single signature). That is, P'*{1,1} = r_1P*{1,1} = \mathcal{O}

.

- The right part of the verification equation becomes \prod_{i=1}^n \prod_{j=1}^{m_i} e(P_{i,j},M_{i,j}')

that is, e(P'*{1,1},M*{1,1})\prod_{j=2}^{m_i} e(P'*{i,j},M*{i,j})=1 \times \prod_{j=2}^{m_i} e(P'*{i,j},M*{i,j})

.

It follows that verification will pass when all other signatures are valid, even if P_1

's signature S_1

is not valid.

Note that the Ethereum clients (Lighthouse, Lodestar, Nimbus, Prysm, Teku) will already have performed subgroup validation upon deserialization preventing such an attack.

Implementations cheat sheet

Secure implementations of batch BLS verification must ensure that:

- Group elements (signatures, public keys, message hashes) do not represent the point to infinity and belong to their respective assigned group (BLS12-381's \mathbb{G}_1

or \mathbb{G}_2

).

- The r_i

coefficients are chosen of the right size, using a cryptographically random generator, without side channel leakage.

- The r_i

coefficients are non-zero, using constant-time comparison, and if zero reject it and generate a new random value (if zero is hit multiple times, verification should abort, for something must be wrong with the PRNG).

- The number of signatures matches the number of public keys and of message tuples.

Additionally, implementations may prevent DoS by bounding the number of messages signed.