Thank @kilic, @jannikluhn, and @Mikerah for the review and the advice

This article targets developers who want to perform BLS signature verification in Eth1 contracts.

For readers interested in the BLS signature in Eth2, I highly recommend "BLS12-381 For The Rest Of Us"^ben-bls by Ben Edgington, which provides long but not too long answers to common questions.

The article also assumes the readers heard of the terms like G_1

, G_2

, and pairings. "BLS12-381 For The Rest Of Us" is also a good source for understanding those.

Introduction

When we talk about the BLS signature, we are talking about the signature aggregation technique. The BLS here is the name of three authors Boneh, Lynn, and Shacham. ^bls

Which curve should I use?

BLS signature aggregation works on pairing friendly curves. We have to choose which of the following curves to use.

- alt_bn128: Barreto-Naehrig curve. ^bn
- BLS12-381: This BLS is Barreto, Lynn, and Scott.^bls-curve

At the time of writing this document (2020-Aug), we can only use alt_bn128 curve in contracts. The alt_bn128 curve is specified in EIP-196^eip196 (curve point addition and multiplication) and EIP-197^eip197 (curve pairing) and introduced in Byzantium hardfork October 16, 2017^byzantium-fork.

The cost is further reduced in EIP-1108^eip1108, and was live on mainnet in Istanbul-fork^istanbul-fork.

The BLS12-381 curve is specified in EIP-2537^eip2537 and expected to go live in the Berlin hardfork^eip2070.

The alt_bn128(BN256) is shown in 2016 to have less than 100-bits of security^ietf. To target 128-bits security use BLS12-381 whenever possible.

We use alt_bn128 in the examples hereafter.

alt_bn128, bn256, bn254, why so many names and so confusing?

- Originally named alt_bn128 for targeting theoretic security at 128 bits.
- It then be shown to have security level less than 128 bits^ietf, so people call it bn254.
- Some also call it bn256 for unknown reason.

Notes on size of the elements

alt_bn128

BLS12-381

Note

F_q

254 bits (32 bytes)

381 bits (48 bytes)

has leading zero bits

F_{q^2}

64 bytes

96 bytes

$\mathbb{G}_1$

Uncompressed

64 bytes

96 bytes

has x and y coordinates as F_q

$\Bbb G_2$

Uncompressed

128 bytes

192 bytes

has x and y coordinates as F_{q^2}

A curve point’s y coordinate can be compressed to 1 bit and recovered by x.

alt_bn128

BLS12-381

$\Bbb G_1$

compressed

32 bytes

48 bytes

$\Bbb G_2$

compressed

64 bytes

96 bytes

Big Signatures or Big public keys?

This is a decision you’ll face in the project.

Choose either:

- Use $\Bbb G_2$

for signatures and messages, and $\Bbb G_1$

for public keys, or

- Use $\Bbb G_2$

for public keys, and $\Bbb G_1$

for signatures and messages.

$\Bbb G_2$

is 128 bytes (uint256[4]

in Solidity) and $\Bbb G_1$

is 64 bytes (uint256[2]

in Solidity). Also field/curve operations on $\Bbb G_2$

are more expensive.

The option saves more gas in your project is better.

We’ll use the big public keys in the examples hereafter.

BLS cheat sheet

We have a curve. The points on the curve can form a subgroup. We define $\Bbb G_2$ to be the subgroup of points where the curve's x and y coordinates defined on F_{q^2}. And subgroup $\Bbb G_1$ with coordinates on F_q.

Tips

- Field operations

: We are talking about arithmetics of field elements F_q or F_{q^2}. F_q can be added, subtracted, multiplied, and divided by other F_q. Same applies for F_{q^2}.

- Curve operations

: We are talking about operations of curve elements $\Bbb G_1$ or $\Bbb G_2$. An element in $\Bbb G_1$ can be added to another element in a geometrical way. An element can be added to itself multiple times, so we can define multiplications of an element in $\Bbb G_1$ with an integer in Z_q [^ecmul](#). Same applies for $\Bbb G_2$.

Pairing function

$e:\Bbb G_1 \times \Bbb G_2 \to \Bbb G_T$

G_1, G_2 are the generators of $\Bbb G_1$ and $\Bbb G_2$ respectively

Pairing function has bilinear property, which means for $a, b \in Z_q$, $P \in \Bbb G_1$, and $Q \in \Bbb G_2$, we have:

$e(a \times P, b \times Q) = e(ab \times P, Q) = e(P, ab \times Q)$

Hash function

It maps the message to an element of $\Bbb G_1$

$H_0: \mathcal{M} \to \Bbb G_1$

Key Generation

Secret key:

$\alpha \gets \Bbb Z_q$

Public key:

$h \gets \alpha \times G_2 \in \Bbb G_2$

Signing

$\sigma \gets \alpha \times H_0(m) \in \Bbb G_1$

Verification

$e(\sigma, G_2)\stackrel{?}{=} e(H_0(m), h)$

Proof of why verification works

$e(\sigma, G_2) \ = e(\alpha \times H_0(m), G_2) \ = e(H_0(m), \alpha \times G_2) \ = e(H_0(m), h)$

Contracts / Precompiles

Developers usually work with a wrapped contract that has clear function names and function signatures. However, the core of the implementation could be confusing. Here we provide a simple walkthrough.

Verify Single

Below shows an example Solidity function that verifies a single signature. It is the small signature and big public key setup. A signature is a 64 bytes $\Bbb G_1$

group element, and its calldata is a length 2 array of uint256. On the other hand, a public key is a 128 bytes $\Bbb G_2$

group element, and its calldata is a length 4 array of uint256.

EIP 197 defined a pairing precompile contract at address 0x8

and requires input to a multiple of 192. This assembly code calls the precompile contract at address 0x8

with inputs.

```
function verifySingle( uint256[2] memory signature, \ small signature uint256[4] memory pubkey, \ big public key: 96 bytes uint256[2] memory message ) internal view returns (bool) { uint256[12] memory input = [ signature[0], signature[1], nG2x1, nG2x0, nG2y1, nG2y0, message[0], message[1], pubkey[1], pubkey[0], pubkey[3], pubkey[2] ]; uint256[1] memory out; bool success; // solium-disable-next-line security/no-inline-assembly assembly { success := staticcall(sub(gas(), 2000), 8, input, 384, out, 0x20) switch success case 0 { invalid() } } require(success, ""); return out[0] != 0; }
```

We translate the above code into math formula. Where the nG2

is the negative “curve” operation of the G_2

group generator.

$e(\text{signature}, neg(G_2)) e(\text{message}, \text{pubkey}) \stackrel{?}{=} 1$

If the above formula is not straight forward, let’s derive from the pairing check we usually see. The message here is the raw message hashed to G_1

.

$e(\text{message}, \text{pubkey}) \ = e(\text{message}, \text{privkey} \times G_2) \ = e(\text{privkey} \times \text{message}, G_2) \ = e(\text{signature}, G_2)$

$e(\text{message}, \text{pubkey}) \stackrel{?}{=} e(\text{signature}, G_2)$

Hash to curve

We can choose from Hash and pray approach or Fouque Tibouchi approach:

- Hash and pray approach is easy to implement, but since it is non-constant time to run it has a security issue^hash-and-pray-issue. Each iteration is expensive in solidity and attacker can grind a message that's too expensive to check on chain.
- Fouque Tibouchi is constant time, but more difficult to implement.

The following discussion are for hash to G1. For the case of bn254 in G2, see musalbas's implementation

Attempts to fix hash and pray

Avg 30k gas for hash and pray https://github.com/thehubbleproject/RedditHubble/runs/1011657548#step:7:35

A sqrt iteration takes 14k gas due to the call to modexp precompile.

Attempt to replace modexp with a series of modmul. optimized cost is 6.7k

GitHub

ChihChengLiang/modexp

Contribute to ChihChengLiang/modexp development by creating an account on GitHub.

Kobi has a proposal that user provides outputs of modexp and onchain we use 1 modmul to verify. https://gist.github.com/kobigurk/b9142a4755691bb12df59fbe999c2a1f#file-bls_with_help-sol-L129-L154

Gas Consumption

Post EIP-1108, k pairings cost 34 000 * k + 45 000

gas.

So as the above example, to validate a single signature takes 2 pairings and thus 113000 gas.

Validating n different messages takes n + 1 pairings, which costs 80 000 + 34000*n

gas.

In comparison, ECDSA takes 3000 gas, see ecrecover

^ethgastable.

Here are cases to consider the aggregate signature:

- When there's only one message, but many signatures to verify. The aggregate signature takes 2 pairings (113000 gas), and it wins ECDSA when you have 38 more signatures to verify.
- When you need to store or log signatures on chain. Storing a word (32 bytes) costs 20000 or 5000 gas, and logging costs 8 gas per byte. The aggregate signature (48 bytes * 1 sig) wins ECDSA (65 bytes * n sigs) easily in this case.

In the Hubble project, we use BLS signature to achieve 3000 TPS on ropsten

Packages to do BLS

JavaScript/TypeScript

GitHub

kilic/evmbls

Contribute to kilic/evmbls development by creating an account on GitHub.

Python

GitHub

ChihChengLiang/bls_solidity_python

Contribute to ChihChengLiang/bls_solidity_python development by creating an account on GitHub.

Cookbook

Aggregations

TODO

- how to create private keys, public keys, and signatures (just listing the formulas should be enough, everyone can then use their favorite library to implement them).
- test data for one cycle (a private key, a public key, a message, and a signature)
- a note on encodings: The solidity code just uses the plain uints, but at least for BLS12-381 there are standardized encodings, aren't there? Not sure if it makes sense to go into details, but just mentioning that they exist with maybe a link could be helpful
- public key aggregation (my problem basically): My understanding from our discussion yesterday is that it's not easily possible on-chain with bn128 and public keys from G2. I looked into the EIP fro BLS12-381 and they have a precompile for G2 additions, so with that it should be easy.