Groth16 Implement Specification in BitVM2

Fiamma [1]

July 2024

Abstract

In this paper, we present our methodology for implementing Groth16 verification within the Bitcoin framework, utilizing the BitVM2 repository. We extend our gratitude to contributors such as Robin, Lukas, Weikeng, and the Zerosync team for their invaluable contributions to this project. The objectives of this paper are threefold:

- Peer Review: We invite the BitVM2 community to review our design and implementation, providing feedback to enhance its robustness and efficiency.
- Knowledge Dissemination: We aim to provide a detailed exposition on the integration of BitVM2 with Groth16 to enlighten further developers and researchers about its operational mechanics.
- Community Collaboration: By sharing our findings, we seek to catalyze collaborative development efforts within the community to expedite the safe production readiness of BitVM2.

Through this publication, we hope to contribute significantly to the evolution and adoption of BitVM2, fostering a safer and more efficient deployment in production environments.

Contents

1 Basic Data

This section outlines the fundamental information of Fiamma until now, it will be updated in the future.

- ZK algorithm: Groth16 [6]
- Script size: 2.605447015 GB
- Subscript number: 994
- Average script size: 2.621174 MB
- Max input size:
- Max output size:
- Signature: Winternitz
- Hash: Blake3

2 Basic Theory

This section outlines essential basic theories used in our implementation.

- Groth16 [6]: The algorithm we implementing currently
- On Proving Pairing [5]: an efficient solution for implementing ZKP on Bitcoin combined with BitVM2 [1]
- Non-deterministic computation: an efficient way to reduce the script size in terms of verification point
- Affine module: an efficient way to reduce the script size in terms of computation
- Montgomery reduction and Karatsuba multiplication: an efficient way to duce the script size of big integer multiplication

2.1 Groth16-verification-program

The verify progress of Groth16[6] as follows:

$$0/1 \Leftarrow Vrf(R, \sigma, a_1, ..., a_l, \pi) : Parse\pi = ([A]_1, [C]_1, [B]_2) \in G_1^2, G_2 \tag{1}$$

Accept the proof if and only if

$$[A]_1 \cdot [B]_2 = [\alpha_1] \cdot [\beta]_2 + \sum_0^l a_i [\frac{\beta\mu_i(x) + \alpha\upsilon_i(x) + \omega_i(x)}{\gamma}]_1 \cdot [\gamma]_2 + [C]_1 \cdot [\delta]_2 \tag{2}$$

Let

$$[msm]_1 = \sum_0^l a_i [\frac{\beta\mu_i(x) + \alpha\upsilon_i(x) + \omega_i(x)}{\gamma}]_1 \tag{3}$$

It should be noted that $a_0 = 1$

2.2 On-proving-pairing

This is a efficient ways to prove correctness of [5], the algorithm shows in page 25: Algorithm 9: Multi Miller loop with embedded c exponentiation

If we integrate this algorithm into Groth16, the entire process would be structured as follows:

$P_1 = [msm]_1; Q_1 = -[\gamma]_2$

$P_2 = [C]_1; Q_2 = -[\delta]_2$
$P_3 = [\alpha]_1; Q_3 = -[\beta]_2$
$P_4 = [A]_1; Q_4 = [B]_2$
Q_4 is non-fixed, Q_1, Q_2, and Q_3 is fixed.

Input: $A = [(P_1, Q_1), (P_2, Q_2), (P_3, Q_3), (P_4, Q_4)], c, c^{-1} \in F_{q^k}, s \in F_{q^3}, P_{Q_4} \leftarrow \mathcal{L}(Q_4)$

Output: 1 $if \prod_{i=0}^{n} e(P_i, Q_i) = 1$

(1) assert $c \cdot c^{-1} = 1$
(2) $f \leftarrow c^(- 1), lc \leftarrow 0$
(3) Initialize array T such that $T[4] = Q_4$
(4) for i = $L - 2$ to 0 do
(5) f = f^2
(6) for j = 1 to n do
(7) $l \leftarrow P_{Q_4}[lc]$
(8) $f = f \cdot l.evaluate(P_4)$
(9) Q_4 is not fixed then
(10) $T \leftarrow T[4]$
(11) assert $l.is_tangent(T)$
(12) $T[4] = l.double(T)$
(13) end
(14) if $bit^2 == 1$ then
(15) $f = f \cdot c^{-1}$ if bit == 1 else $f \cdot c$ end
(16) $l \leftarrow P_{Q_4}[lc + 1]$
(17) $f = f \cdot l.evaluate(P_4)$
(18) Q_4 is not fixed then
(19) $Q^{'} = Q_4$ if bit == 1 else $-Q_4$
(20) $T \leftarrow T[4]$
(21) assert $l.is_line(T, Q^{'})$
(22) $T[4] = l.add(T, Q^{'})$
(23) end
(24) end
(25) end
(26) $lc = lc + 2$
(27) $f \leftarrow f \cdot s \cdot (c^{-1})^q \cdot (c^{-1})^{q^2} \cdot (c^{-1})^{q^3}$
(28) for j = 0 to n do
(29) $l_{1..3} \leftarrow (P_{Q_j}[lc + i])_{i=0}^{2}$

(30) $f \leftarrow f \cdot l_1.evaluate(P_j) \cdot l_2.evaluate(P_j) \cdot l_3.evaluate(P_j)$

(31) Q_4 is not fixed then

(32) $Q_1 \leftarrow \pi_p(Q), Q_12 \leftarrow \pi_p(Q_1), Q_3 \leftarrow \pi_p(Q_2)$

(33) $T \leftarrow T[4]$

(34) assert $l_1.is_line(T, Q_1); T \leftarrow T + Q_1$

(35) assert $l_2.is_line(T, -Q_2); T \leftarrow T - Q_1$

(36) assert $l_3.is_line(T, Q_3)$

(37) end

(38) end

(39) end

(40) return $f == 1$?

2.3 Non-deterministic-computation

We reference this concept from cairo-vm [2], in one word: the prover may do additional work that is not part of the proven computation.

What if we want to compute a square root of a certain number x as part of a larger function? The deterministic approach is to use some square-root algorithm to compute $y = \sqrt{x}$ which means we need to include its execution trace in our script. But the nondeterministic approach is much more efficient: the prover computes the square-root y using the same algorithm, but does not include this computation in the script. Instead, the only thing the script proved is that y 2 = x, which can be done using a single SQUARE script gadget.

We adapt this concept with double and add operations based on G_1 and G_2

2.3.1 Double in G_1 and G_2

- Show that the pair (λ, μ), indeed define a tangent through T showing that $y_1 - \lambda x_1 - mu = 0$ and $2\lambda y_1 = 3x_1^2$. This step is dominated by $2\tilde{m}$ and one $\tilde{s}$
- Compute λ^2 which is simple one $\tilde{s}$
- Compute $x_3 = \lambda^2 - 2x_1$ and $2\lambda y_3 = -\mu - \lambda x_3$ which is dominated by computing λx_3

2.3.2 Add in G_1 and G_2

- Show that the pair (λ, μ), indeed define a tangent through T showing that $y_1 - \lambda x_1 - mu = 0$ and $y_2 - \lambda x_2 - mu = 0$. This step is dominated by $2\tilde{m}$
- Compute λ^2 which is simple one $\tilde{s}$
- Compute $x_3 = \lambda^2 - x_1 - x_2$ and $2\lambda y_3 = -\mu - \lambda x_3$ which is dominated by computing λx_3

2.4 Affine coordinates

Pairings can be computed over elliptic curves represented in any coordinate system, but popular choices have been homogeneous projective and affine coordinates, depending on the ratio between inversion and multiplication.

2.4.1 Homogeneous Projective coordinates

The choice of projective coordinates has proven especially advantageous at the 128-bit security level for single pairing computation, due to the typically large inversion/multiplication ratio in this setting. The tangent line evaluated at $P = (x_P, y_P)$ can be computed with the following formula:

$$g_{2\phi(T)}(P) = -2YZy_p + 3X^2x_pw + (3b^{'}Z^2 - Y^2)w^3 \tag{4}$$

2.4.2 Affine coordinates

The choice of affine coordinates has proven more useful at higher security levels and embedding degrees, due to the action of the norm map on simplifying the computation of inverses at higher extensions. The main advantages of affine coordinates are the simplicity of implementation and format of the line functions, allowing faster accumulation inside the Miller loop if the additional sparsity is exploited.

If $T = (x_1, y_1)$ is a point in $E^t(F_{p2})$, one can compute the point $2T := T + T$ with the following formula:

$$(1/y_p)g_{2\phi(T)}(P) = 1 + (-x_p/y_p)\lambda w + (1/y_p)(\lambda x_1 - y_1)w^3 \tag{5}$$

you can read this paper [9] to know more deatils about this, we big thanks to all contributors of PR [7] in BitVM2 [1].

2.5 Big integer multiplication

2.5.1 Montgomery reduction

Montgomery reduction [3], also known as REDC, is an algorithm that simultaneously computes the product by R' and reduces modulo N more quickly than the naïve method. Unlike conventional modular reduction, which focuses on making the number smaller than N, Montgomery reduction focuses on making the number more divisible by R. It does this by adding a small multiple of N which is sophisticatedly chosen to cancel the residue modulo R. Dividing the result by R yields a much smaller number. This number is so much smaller that it is nearly the reduction modulo N, and computing the reduction modulo N requires only a final conditional subtraction. Because all computations are done using only reduction and divisions with respect to R, not N, the algorithm runs faster than a straightforward modular reduction by division.

2.5.2 Karatsuba multiplication

Karatsuba multiplication [4] is a divide-and-conquer algorithm for (non-modular) multiplication, which for n-bit integers reduces cost from the $O(n^2)$ for classical multiplication to $O(n^{\log_2^3})$, that is $O(n^{1.58\cdots})$.

Applied to modular exponentiation with n-bit numbers including exponent, the cost goes from $O(n^3)$ to $O(n^{1+log_2^3})$, that is $O(n^{2.58\cdots})$. One of several methods for getting the benefits of Karatsuba multiplication during modular reduction is pre-computing the (non-modular) inverse of the modulus to slightly more than n bits, which can be done at cost $O(n^2)$ (thus negligible as far as O is concerned) with classical algorithms.

Karatsuba multiplication can be used together with Montgomery reduction. We highly appreciate at the work [8] of Robin and Zerosync team. It reduce the script size of multiplication to around 55% of native version.

3 script

3.1 Split basics

This section outlines essential basic knowledge used in our implementation.

3.1.1 Signature-hash-type

Signature Hash Types:

- 0x01 = SIGHASH_ALL
- 0x02 = SIGHASH_NONE
- 0x03 = SIGHASH_SINGLE
- 0x81 = SIGHASH_ANYONECANPAY | SIGHASH_ALL
- 0x82 = SIGHASH_ANYONECANPAY | SIGHASH_NONE
- 0x83 = SIGHASH_ANYONECANPAY | SIGHASH_SINGLE

3.1.2 Transaction-constructure

The transaction constructure of Bitcoin show in the following excel.

Field	Size	Description
Version	4 bytes	The version number for the transaction. Used to enable new features
Maker	1 bytes	Used to indicate a segwit transaction. Must be 00
Flag	1 bytes	Used to indicate a segwit transaction. Must be 01 or greater
Input Count	Variable	Indicates the number of inputs
Input-TXID	32 bytes	The TXID of the transaction containing the output you want to spend
Input-VOUT	4 bytes	The index number of the output you want to spend
Input-ScriptSig Size	Variable	The size in bytes of the upcoming ScriptSig
Input-ScriptSig	Variable	The unlocking code for the output you want to spend
Input-Sequnencer	4 bytes	Set whether the transaction can be replaced or when it can be mined
Output Count	Variable	Indicates the number of outputs
Output-Amount	8 bytes	The value of the output in satoshis
Output-ScriptPubKey Size	Variable	The size in bytes of the upcoming ScriptPubKey
Output-ScriptPubKey	Variable bytes	The locking code for this output
Witness-Stack Items	Variable	The number of items to be pushed on to the stack as part of the unlocking code.
Witness-Stack Items-Size	Variable	The size of the upcoming stack item
Witness-Stack Items-Item	Variable	The data to be pushed on to the stack
Locktime	4 bytes	Set a time or height after which the transaction can be mined

The blue part means it will be stored in segwit part. Any one could check more details in [10]

3.1.3 Block-size-calculation

Understanding how Bitcoin block size is calculated post-Taproot upgrade is crucial.

block size The block size calculation is depicted in the following illustration:

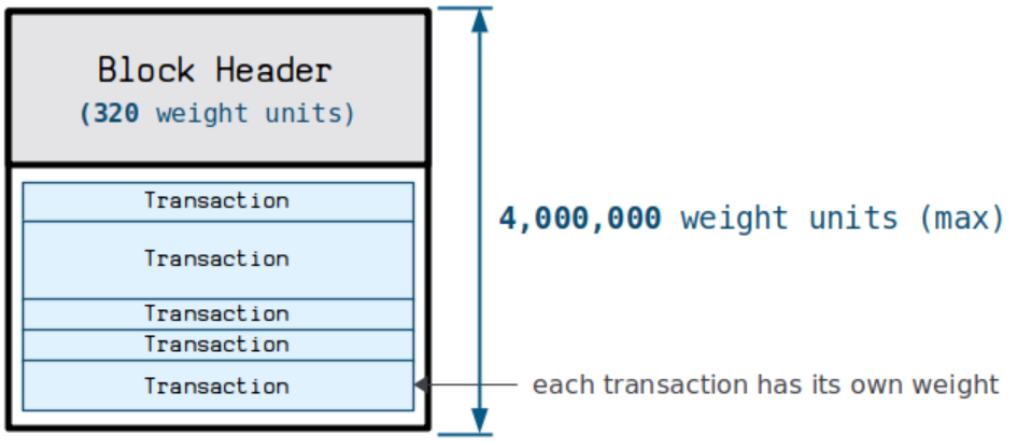

Figure 1: Block size

transaction size The transaction size calculation is illustrated below:

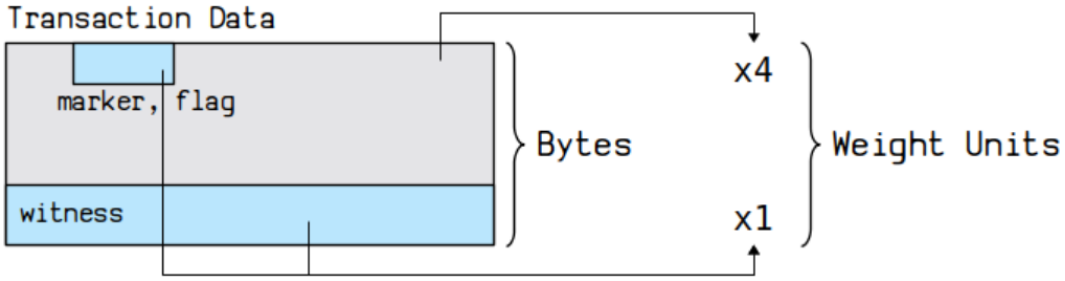

Figure 2: Transaction size

For more detailed information, refer to source in [11]

script chunk limitation Based on the deign of BitVM2 [1], we hope each script chunk could be packed into one block as a transaction. So, the transaction size could not exceed 4,000,000 - 320 = 3,999,680 weight units [11].

The disputed transaction is a 1 input and 2 outputs, the average size of non-witness data in this kind of transaction is around 464 weight units. So for the witness size, the limitation is 3,999,680 - 464 = 3,999,216 weight units.

As showed in BitVM2 [1], the disputed transaction needs to the signature of Committee, and the sign type is SIGNHASH_SINGLE. Let's assume that the number of Committee is 7 and the size of each schnorr signature is 65 bytes (64 bytes for SIGNHASH_ALL)

So the limitation will be 3,999,216 - 7 * 75 - 8(stack item size) = 3,998,683 weight units.

3.1.4 Limitations

Several constraints must be considered:

- Max script size: 4MB
- Max stack depth: 1000 (combined main stack and alternate stack);
- Max signatures number in one block: 1000 Winternitz signatures;

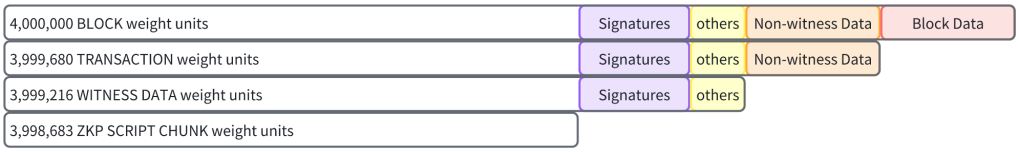

Figure 3: ZKP script chunk limitation

3.2 Split pricinple

We will mainly introduce why we select a manually way to split the ZKP verification script.

3.2.1 automated

Splitting the whole computation automately is a good story. Ideally, the whole progress shoule be like the follow picture:

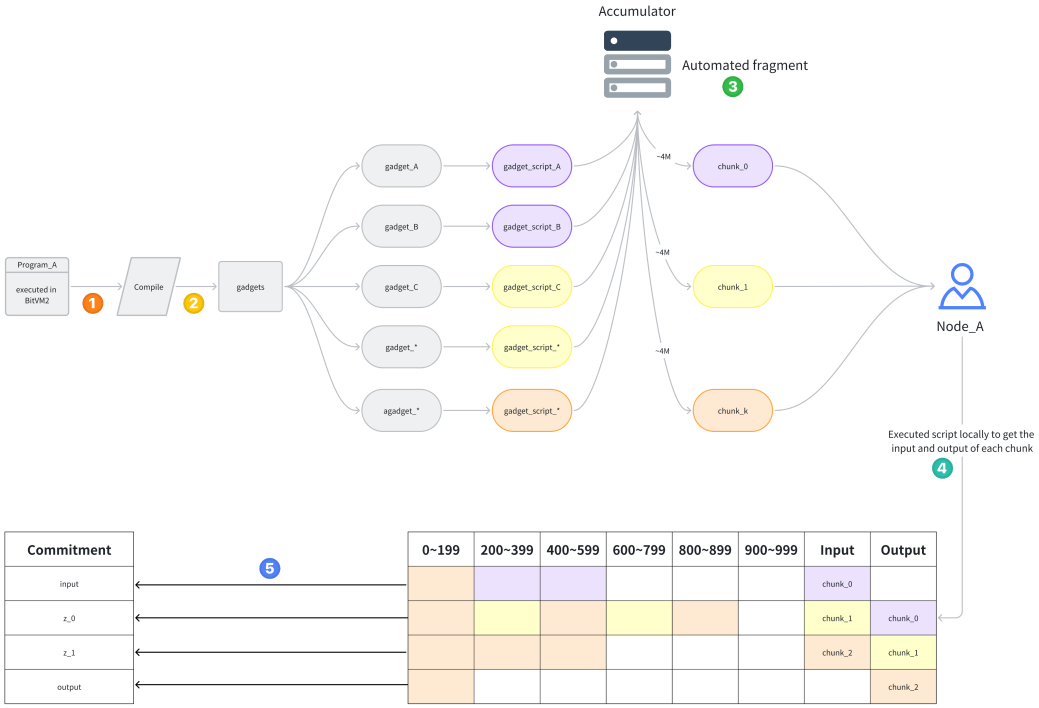

Figure 4: automated fragment

The overall flow shoule be as follows:

- The program will be complied into a set of customized gadgets first
- Each gadget will correspond with a script gadget;
- The accmulator begin to split the whole gadgets;
- Becasue each script gadget has a fixed size, so when the accumulated size is almost equal to 4M, these gadgets will spilt as one chunk;
- The Node A execute the script program locally to generates input and output for each chunk;
- All the input and output locate in stack, so The Node A have to commit all the value in the stack;

It is worth to note that stack depth is another factor shoule be taken into account. For the automated way, we think there have

a few constrains:

- It is easy to exceed the stack depth limitation;
- It committes lot of value which won'be used in current chunk;
- It has to implememnt enough gadgets to support any computation which means turing completed;
- Executing a big scirpt program is much slow;
- It adds the costs when verify the expected input and output on chain;
- The logic of each chunk is unreadable;

However, automated fragment has some advantages as well, like it will generate the minimal number of script chunk. But as we dont put all chunks into Bitcoin network, So we do can not mind the number of chunks unless the size is much big.

3.2.2 manually

Why do we select a manually way to split the whole program?

- It is not easy to exceed the stack depth limitation as we just put data into stack which is needed by current chunk ;
- It only committes data which only used in current chunk;
- We just need to implement gadgets to support ZKP verification as any computaion could generate a ZK proof;
- Executing the rust program to generate the input and output of each chunk;
- It keeps the lowest costs when verify the expected input and output on chain;
- The logic of each chunk is readable;

This approach maybe generate more chunks, but just like we said before, there will only one script chunk executed on Bitcoin. So it's acceptable. The overall flow of manually fragment as follow pictures:

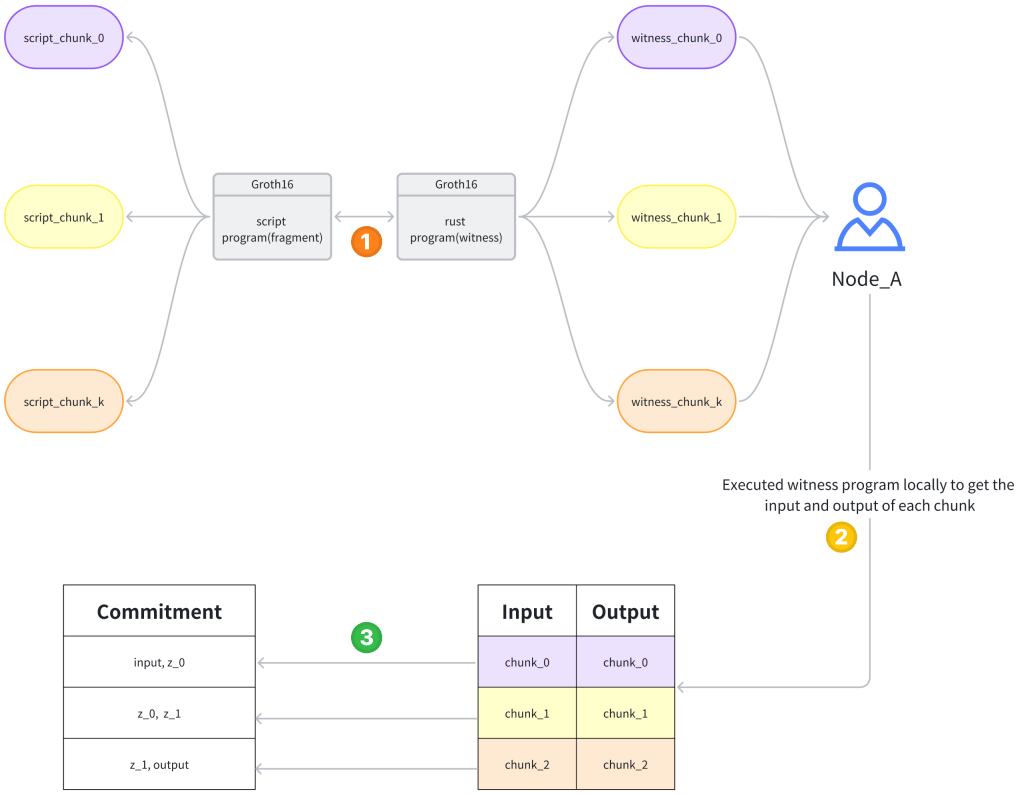

Figure 5: manually fragment

The overall flow shoule be as follows:

- We implememnt the rust version and script version of Groth16 ZKP verification cocurrently first;
- The rust verion includes the witness generation of each chunk;
- The script verion includes all chunks we split;
- We keep each chunk satisfies the size constraint and depth constraint;
- The Node A execute the rust program locally to generates input and output for each chunk;
- The Node A committes all the inputs and outputs;

3.3 Split data

We give the result directly how we split the script which the size exceeds the 4M limitation. As we showed in the 3.4, There only have some operations of F_{q12} need to be split after we optimize the operations for G_1 and G_2

operator typr	script size	max depth	exceed 4M?
$F_{q12} : a * b$	11,641,775 bytes	545	yes
$F_{q12} : frobenius\_map(1)$	4,541,887 bytes	-	yes
$F_{q12} : mul\_by\_034$	9,810,459 bytes	-	yes
$F_{q12} : ell\_by\_constant$	9,525,050 bytes	383	yes

We will show how we split these 4 big script one by one, as we split it manually, we try our best to satifying the following property cocurrently:

- Doesn't exceed the limitation of size and stack depth;
- Keeping the size of input and output is minimal;
- Try our best to make logic of each chunk is readable;

3.3.1 split-code

$F_{q12} : a \cdot b$

```
% Split Fq12 mul into small scripts. For each script
% size < 4M && max_stack_used < 1000
% Input: a0, a1, b0, b1
%
% Algorithm:
%     Final_a0 = a0 * b0 + a1 * b1 * \gamma
%     Final_a1 = (a0 + a1) * (b0 + b1) - (a0 * b0 + a1 * b1)
pub fn split_mul() -> Vec<Script> {
    % The degree-12 extension on BN254 Fq6 is under the polynomial z^2 - y

    let mut res = vec![];

    res.push(script! {
        % a0, b0
        { Fq6::mul(6, 0) }
        % a0 * b0
    });

    res.push(script! {
        % a1, b1
```

```
            { Fq6::mul(6, 0) }
            % a1 * b1
        });

        res.push(script! {
            % a0 * b0, a1 * b1, a0, a1, b0, b1,
            { Fq6::add(6, 0) }
            % a0 * b0, a1 * b1, a0, a1, b0 + b1,
            { Fq6::add(12, 6) }
            % a0 * b0, a1 * b1, b0 + b1, a0 + a1,
            { Fq6::mul(6, 0) }
            % a0 * b0, a1 * b1, (a0 + a1) * (b0 + b1)
            { Fq6::copy(12) }
            % a0 * b0, a1 * b1, (a0 + a1) * (b0 + b1), a0 * b0
            { Fq6::copy(12) }
            % a0 * b0, a1 * b1, (a0 + a1) * (b0 + b1), a0 * b0, a1 * b1
            { Fq12::mul_fq6_by_nonresidue() }
            % a0 * b0, a1 * b1, (a0 + a1) * (b0 + b1), a0 * b0, a1 * b1 * \gamma
            % z^2 - \gamma = 0
            { Fq6::add(6, 0) }
            % a0 * b0, a1 * b1, (a0 + a1) * (b0 + b1), a0 * b0 + a1 * b1 * \gamma
            { Fq6::add(18, 12)}
            % (a0 + a1) * (b0 + b1), a0 * b0 + a1 * b1 * \gamma, a0 * b0 + a1 * b1
            { Fq6::sub(12, 0) }
            % a0 * b0 + a1 * b1 * \gamma, (a0 + a1) * (b0 + b1) - (a0 * b0 + a1 * b
                1)
        });

        res
    }
```

$F_{q12} : frobenius\_map(1)$

```
    pub fn split_frobenius_map(i: usize) -> Vec<Script> {
        let mut res = vec![];
        if i == 1 {
            % [p.c0, p.c1]
            res.push(script! {
                { Fq6::frobenius_map(i) }
                { Fq6::roll(6) }
                { Fq6::frobenius_map(i) }
                % [p.c1 ^ p^i, p.c0 ^ p^i]
            });
            % [p.c1 ^ p^i]
            res.push(Fq6::mul_by_fp2_constant(
                &ark_bn254::Fq12Config::FROBENIUS_COEFF_FP12_C1
                    [i % ark_bn254::Fq12Config::FROBENIUS_COEFF_FP12_C1.len()],
            ));
        } else {
            res.push(Self::frobenius_map(i));
```

```
        }

        res
    }
```

F_{q12} : *mul_by_*034

```
    pub fn split_mul_by_034() -> Vec<Script> {
        let mut res = vec![];

        % compute b = p.c1 * (c3, c4)
        % [p.c1, c3, c4]
        res.push(Fq6::mul_by_01());
        % [b]

        % [c0, c3, b, p.c0, p.c1]
        % [Fq2, Fq2, Fq6, Fq6, Fq6]
        res.push(script! {
            % compute a = c0 * p.c0
            { Fq6::copy(6) }
            % [c0, c3, b, p.c0, p.c1, p.c0]
            { Fq2::copy(26) }
            % [c0, c3, b, p.c0, p.c1, p.c0, c0]
            { Fq6::mul_by_fp2() }
            % [c0, c3, b, p.c0, p.c1, c0 * p.c0]
            % [c0, c3, b, p.c0, p.c1, a]
            % compute gamma * b
            { Fq6::roll(18) }
            % [c0, c3, p.c0, p.c1, a, b]
            { Fq12::mul_fq6_by_nonresidue() }
            % [c0, c3, p.c0, p.c1, a, b * gamma]

            % compute final c0 = a + gamma * b
            % [c0, c3, p.c0, p.c1, a, b * gamma]
            { Fq6::copy(6) }
            % [c0, c3, p.c0, p.c1, a, b * gamma, a]
            { Fq6::add(6, 0) }
            % [c0, c3, p.c0, p.c1, a, a + b * gamma]
            % [c0, c3, p.c0, p.c1, a, final_c0]

            % compute e = p.c0 + p.c1
            { Fq6::add(18, 12) }
            % [c0, c3, a, final_c0, p.c0 + p.c1]
            % [c0, c3, a, final_c0, e]

            % compute c0 + c3
            { Fq2::add(20, 18) }
            % [a, final_c0, e, c0 + c3]
        });
```

```
        % [b, a, final_c0, e, c0 + c3, c4]
        res.push(script! {
            % update e = e * (c0 + c3, c4)
            { Fq6::mul_by_01() }
            % [b, a, final_c0, e]

            % sum a and b
            { Fq6::add(18, 12) }
            % [final_c0, e, b + a]

            % compute final c1 = e - (a + b)
            { Fq6::sub(6, 0) }
            % [final_c0, e - (b + a)]
            % [final_c0, final_c1]
        });

        res
    }
```

F_{q12} : *mul_by_constant*

```
    pub fn split_mul_by_034_with_4_constant(constant: &ark_bn254::Fq2) -> Vec<Script
        > {
        let mut res = vec![];

        % [p.c1, c3], constant = c4
        res.push(Fq6::mul_by_01_with_1_constant(constant));

        % compute a = p.c0 * c0
        % Input: [p.c0, c0]
        % Output: [p.c0 * c0]
        res.push(Fq6::mul_by_fp2());

        % [c0, c3, p.c0, p.c1, a, b]
        res.push(script! {
            { Fq6::copy(0) }
            % [c0, c3, p.c0, p.c1, a, b, b]
            % compute beta * b
            { Fq12::mul_fq6_by_nonresidue() }
            % [c0, c3, p.c0, p.c1, a, b, b * beta]

            % compute final c0 = a + beta * b
            { Fq6::copy(12) }
            % [c0, c3, p.c0, p.c1, a, b, b * beta, a]
            { Fq6::add(6, 0) }
            % [c0, c3, p.c0, p.c1, a, b, a + beta * b]
            % [c0, c3, p.c0, p.c1, a, b, final_c0]

            % compute e = p.c0 + p.c1
            { Fq6::add(24, 18) }
```

```
            % [c0, c3, a, b, final_c0, e]

            % compute c0 + c3
            { Fq2::add(26, 24) }
            % [a, b, final_c0, e, c0 + c3]

            % update e = e * (c0 + c3, c4)
            { Fq6::mul_by_01_with_1_constant(constant) }
            % [a, b, final_c0, e]

            % sum a and b
            { Fq6::add(18, 12) }
            % [final_c0, e, a + b]

            % compute final c1 = e - (a + b)
            { Fq6::sub(6, 0) }
            % [final_c0, final_c1]
        });

        res
    }
```

3.3.2 split-result

We give the split result directly as follow excel:

operator typr	chunks	script size	max depth	exceed 4M?
$F_{q12} : a * b$		11,641,775 bytes	545	yes
	chunk0	11,641,775 bytes	545	no
	chunk1	11,641,775 bytes	545	no
	chunk2	11,641,775 bytes	545	no
$F_{q12} : frobenius\_map(1)$		4,541,887 bytes	-	yes
	chunk0	11,641,775 bytes	545	no
	chunk1	11,641,775 bytes	545	no
$F_{q12} : mul\_by\_034$		9,810,459 bytes	-	yes
	chunk0	11,641,775 bytes	545	no
	chunk1	11,641,775 bytes	545	no
	chunk2	11,641,775 bytes	545	no
$F_{q12} : ell\_by\_constant$		9,525,050 bytes	383	yes
	chunk0	11,641,775 bytes	545	no
	chunk1	11,641,775 bytes	545	no
	chunk2	11,641,775 bytes	545	no

3.4 Bench data

This section mainly give some bench datas for some operators used in Groth16 verification process.

3.4.1 operators-script-size-origin

We give some initial bench data we test in current implement first. Including:

- Double and Add operators in G_1 group;
- Double and Add operators in G_2 group;
- Field operators in extension field;

G1 group

operator typr	script size	max depth	exceed 4M?
$2 \cdot g_1$	1,752,916 bytes	131	no
$g_1 \cdot g_1^{'}$	3,997,319 bytes	< 1000	no

G2 group

operator typr	script size	max depth	exceed 4M?
$2 \cdot g_2$	7,019891 bytes	815	yes
$g_2 \cdot g_2^{'}$	9,270,854 bytes	293	yes

field

operator typr	script size	max depth	exceed 4M?
$F_{q12} : a + b$	6,644 bytes	220	no
$F_{q12} : 2 * a$	6,793 bytes	217	no
$F_{q12} : a * b$	11,641,775 bytes	545	yes
$F_{q12} : mul\_fq6\_by\_nonresidue$	4,923 bytes	146	no
$F_{q12} : frobenius\_map(1)$	4,541,887 bytes	-	yes
$F_{q12} : frobenius\_map(2)$	2,224,363 bytes	-	yes
$F_{q12} : mul\_by\_034$	9,810,459 bytes	-	yes
$F_{q12} : ell\_by\_constant$	9,525,050 bytes	383	yes
$F_{q6} : a * b$	3,873,847 bytes	275	no
$F_{q6} : frobenius\_map(1)$	1,518,206 bytes	-	no
$F_{q6} : frobenius\_map(2)$	598,274 bytes	-	no
$F_{q6} : mul\_by\_01\_with\_1\_constant$	3,280,529 bytes	221	no
$F_{q6} : mul\_by\_fp2\_constant$	1,520,337 bytes	101	no
$F_{q6} : mul\_by\_01$	3,769,633 bytes	-	no
$F_{q6} : mul\_by\_fp2$	2,252,362 bytes	167	no
$F_{q2} : a * b$	750,883 bytes	113	no

3.4.2 operators-script-size-optimization

The less script chunks, the better. So before we split the big operators, we want to optimize them first. We will give our new data first and then clarification the pricinple.

- Double and Add operators in G_1 group;
- Double and Add operators in G_2 group;
- Field operators in extension field;

G1 group

operator typr	script size	optimized script size	exceed 4M?
$2 \cdot g_1$	1,752,916 bytes	1,251,319 bytes	no
$g_1 \cdot g_1^{'}$	3,997,319 bytes	1,001,977 bytes	no

G2 group

operator typr	script size	optimized script size	exceed 4M?
$2 \cdot g_2$	7,019891 bytes	3,262,334 bytes	yes
$g_2 \cdot g_2^{'}$	9,270,854 bytes	2,761,898 bytes	yes

field

operator typr	script size	max depth	exceed 4M?
$F_{q12} : a + b$	6,644 bytes	220	no
$F_{q12} : 2 * a$	6,793 bytes	217	no
$F_{q12} : a * b$	11,641,775 bytes	545	yes
$F_{q12} : mul\_fq6\_by\_nonresidue$	4,923 bytes	146	no
$F_{q12} : frobenius\_map(1)$	4,541,887 bytes	-	yes
$F_{q12} : frobenius\_map(2)$	2,224,363 bytes	-	yes
$F_{q12} : mul\_by\_034$	9,810,459 bytes	-	yes
$F_{q12} : ell\_by\_constant$	9,525,050 bytes	383	yes
$F_{q6} : a * b$	3,873,847 bytes	275	no
$F_{q6} : frobenius\_map(1)$	1,518,206 bytes	-	no
$F_{q6} : frobenius\_map(2)$	598,274 bytes	-	no
$F_{q6} : mul\_by\_01\_with\_1\_constant$	3,280,529 bytes	221	no
$F_{q6} : mul\_by\_fp2\_constant$	1,520,337 bytes	101	no
$F_{q6} : mul\_by\_01$	3,769,633 bytes	-	no
$F_{q6} : mul\_by\_fp2$	2,252,362 bytes	167	no
$F_{q2} : a * b$	750,883 bytes	113	no

4 Automated slashing

Automated slashing is another important part in BitVM2. Everyone should be slashed if he announced an invalid assertion.

5 Summary

Bibliography

[1] **BitVM2**. URL: https://bitvm.org/bitvm2.

[2] **CairoVM**. URL: https://eprint.iacr.org/2021/1063.pdf.

[3] **Modular Multiplication Without Trial Division**. URL: https://github.com/BitVM/BitVM/pull/75/commits/1467368e68649224325fb089319e818a26

[4] **Multidigit Multiplication for Mathematicians**. URL: https://cr.yp.to/papers/m3.pdf.

[5] **On Proving Pairings**. URL: https://eprint.iacr.org/2024/640.pdf.

[6] **On the Size of Pairing-based Non-interactive Arguments**. URL: https://eprint.iacr.org/2016/260.pdf.

[7] **PR**. URL: https://github.com/BitVM/BitVM/commit/d40e501cbcffb25a631b18ae6e7381e29f88a0fa.

[8] **PR**. URL: https://github.com/BitVM/BitVM/pull/75/commits/1467368e68649224325fb089319e818a26d7bdb2.

[9] **The Realm of the Pairings**. URL: https://eprint.iacr.org/2013/722.pdf.

[10] **Transaction constructure**. URL: https://learnmeabitcoin.com/technical/transaction/.

[11] **Transaction size**. URL: https://learnmeabitcoin.com/technical/transaction/size/.