Groth16 Implementation 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 **3**

2 Basic Theory **3**

2.1 Groth16 verification program . . . 3

2.2 On proving pairing . . . 3

2.3 Non-deterministic computation . . . 5

2.3.1 Double in G_1 and G_2 . . . 5

2.3.2 Add in G_1 and G_2 . . . 5

2.4 Affine coordinates . . . 5

2.4.1 Homogeneous projective coordinates . . . 6

2.4.2 Affine coordinates . . . 6

2.5 Big integer multiplication . . . 6

2.5.1 Montgomery reduction . . . 6

2.5.2 Karatsuba multiplication . . . 6

3 script **7**

3.1 Split basics . . . 7

3.1.1 Signature hash type . . . 7

3.1.2 Transaction structure . . . 7

3.1.3 Block size calculation . . . 8

3.1.4 Limitations . . . 9

3.2 Split pricinple . . . 10

3.2.1 automated . . . 10

3.2.2 manually . . . 10

3.3 Split data . . . 11

3.3.1 split code . . . 12

3.3.2 split result . . . 15

3.4 Benchmark data . . . 16

3.4.1 Operators script size origin . . . 16

3.4.2 Operator script size optimization . . . 17

4 Automated slashing **18**

5 Summary **18**

Bibliography **18**

1 Basic Data

This section outlines key details about Fiamma to date and will be updated as needed.

- ZK algorithm: Groth16 [8]
- Script size: 2.605447015 G
- Subscript number: 994
- Max subscript szie: 3.710443 M
- Min subscript szie: 0.562906 M
- Average subscript size: 2.621174 MB
- Max input size: 1152 Bytes
- Max output size: 576 Bytes
- Signature: Winternitz
- Hash: Blake3

2 Basic Theory

This section outlines essential basic theories used in our implementation.

- Groth16 [8]: The algorithm that we are implementing currently
- On Proving Pairing [7]: an efficient solution for implementing ZKP on Bitcoin combined with BitVM2 [2]
- Non-deterministic computation: an efficient way to reduce the script size in terms of verification point
- Affine module: an efficient way to decrease the script size in terms of computation
- Montgomery reduction and Karatsuba multiplication: an efficient way to reduce the script size of big integer multiplication

2.1 Groth16 verification program

The verification progress of Groth16[8] proceeds 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

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

Let

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

It should be noted that $a_0 = 1$

2.2 On proving pairing

This is an efficient way to prove correctness of [7], 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 [4]. Simply put, the prover may do additional work that is not part of the proven computation.

Consider the task of computing a square root of a 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 that we need to include its execution trace in our script. In contrast, 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 script only need to prove that is that y2 = x, which can be accomplished 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 defines 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 defines 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 particularly advantageous at the 128-bit security level for single pairing computation, due to the typically high 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, primarily due to the norm map's role in simplifying the computation of inverses at higher extensions. The main advantages of affine coordinates lie in their ease of implementation and the straightforward format of the line functions. These features enable faster accumulation within the Miller loop, particularly when 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 [11] for more details on this topic. We extend our heartfelt thanks to all contributors of the PR [9] in BitVM2 [2].

2.5 Big integer multiplication

2.5.1 Montgomery reduction

Montgomery reduction [5], 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 aims at making the number more divisible by R. It achieves this by adding a sophisticatedly chosen small multiple of N 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 close enough to 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 straightforward modular reduction by division.

2.5.2 Karatsuba multiplication

Karatsuba multiplication [6] is a divide-and-conquer algorithm for (non-modular) multiplication that reduces the computational cost for n-bit integers from $O(n^2)$ in classical multiplication to $O(n^{\log_2^3})$ which is $O(n^{1.58\cdots})$.

When applied to modular exponentiation with n-bit numbers, including the exponent, the cost is reduced from $O(n^3)$ to $O(n^{1+log_2^3})$ which is $O(n^{2.58\cdots})$. One of several methods to leverage the benefits of Karatsuba multiplication during modular reduction is to pre-compute the (non-modular) inverse of the modulus to slightly more than n bits, achievable at a cost of $O(n^2)$, this is considered negligible in the context of O, using classical algorithms.

Karatsuba multiplication can be used in conjunction with Montgomery reduction. We highly appreciate the work [10] of Robin and the Zerosync team, which has reduced for multiplication to approximately 55% of the native version.

3 script

3.1 Split basics

This section outlines essential basic knowledge used in our implementation.

3.1.1 Signature hash type

A SIGNHASH [1] flag is used to indicate which part of the transaction is signed. The mechanism provides a flexibility in constructing transactions. There are in total 6 different flag combinations that can be added to a digital signature in a transaction.

- 0x01 = SIGHASH_ALL : Sign all inputs and outputs
- 0x02 = SIGHASH_NONE : Sign all inputs and no output
- 0x03 = SIGHASH_SINGLE : Sign all inputs and the output with the same index
- 0x81 = SIGHASH_ANYONECANPAY | SIGHASH_ALL : Sign its own input and all outputs
- 0x82 = SIGHASH_ANYONECANPAY | SIGHASH_NONE : Sign its own input and no output
- 0x83 = SIGHASH_ANYONECANPAY | SIGHASH_SINGLE : Sign its own input and the output with the same index

3.1.2 Transaction structure

The transaction structure of Bitcoin is illustrated in the following table.

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-Sequencer	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 the segwit part. Anyone could check more details in [12]

3.1.3 Block size calculation

Understanding the calculation of Bitcoin block size following the Taproot upgrade is essential.

block size The following illustration depicts the block size calculation:

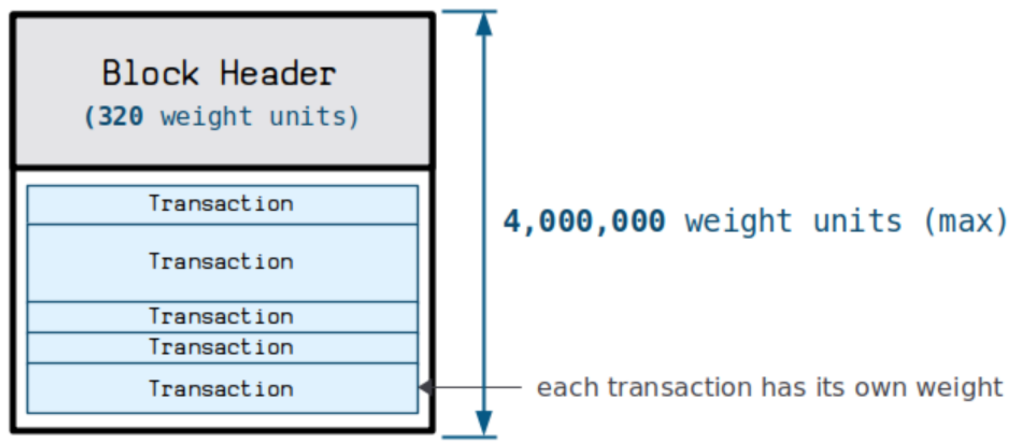

Figure 1: Block size

transaction size The transaction size calculation is illustrated below:

For more detailed information, please refer to [13]

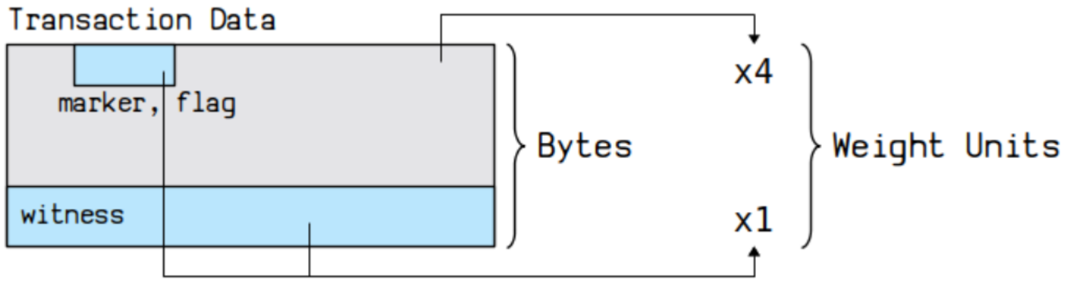

Figure 2: Transaction size

script chunk limitation Based on the design of BitVM2 [2], We aim for each script chunk to be packed into one block as a transaction. So, the transaction size could not exceed 4,000,000 - 320 = 3,999,680 weight units [13].

A disputed transaction, characterized by 1 input and 2 outputs, typically has an average non-witness data size of approximately 464 weight units. Consequently, the witness size limitation for such transactions is calculated as 3,999,680 - 464 = 3,999,216 weight units.

As outlined in BitVM2 [2], the disputed transaction needs the signature of Committee, and the signature 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 * 65 - 8(stack item size) = 3,998,753 weight units.

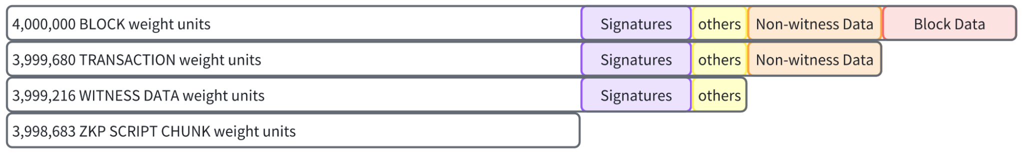
4,000,000 BLOCK weight units
Signatures
others
Non-witness Data
Block Data
3,999,680 TRANSACTION weight units
Signatures
others
Non-witness Data
3,999,216 WITNESS DATA weight units
Signatures
others
3,998,683 ZKP SCRIPT CHUNK weight units

Figure 3: ZKP script chunk limitation

As for the structure of subscript, the normal flow should be as follows:

- check the hash [3] is consistent with the input and output
- check the Winternitz [14] signature
- execute the subscript

So, if we take these factors into account, the subscript structure should be like the following picture

It should be noted that: we need to reduce the maxmial input and putput size of subscript less than 1024 to invoke at most 2 rounds blake3 [3] hash progress

3.1.4 Limitations

Several constraints must be considered:

- Max script size: 4MB
- Max stack depth: 1000 (main stack and alternate stack combined);
- Max stack item size: 520 bytes
- Max input size for arithmetic opcodes: 32-bit words
- Max input bytes for Blake3: 512 Bytes

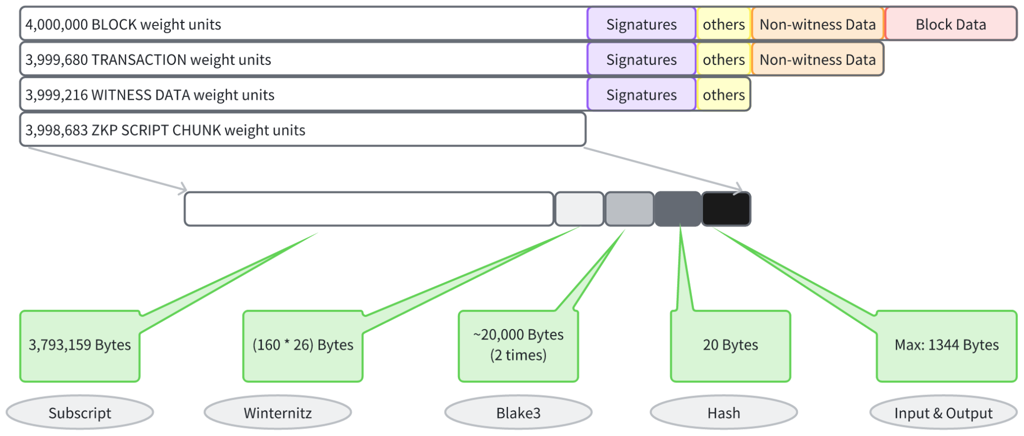

Figure 4: structure of subscript

3.2 Split pricinple

In this section, we will primarily discuss why we chose a manual method for splitting the ZKP verification script.

3.2.1 automated

Automating the splitting of the entire computation is appealing. Ideally, the whole process shoule be similar to the following figure:

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 will begin to split all gadgets;
- Becasue each script gadget has a fixed size, these gadgets will be spilt as one chunk when their accumulated size is almost equal to 4M;
- Node A executes the script program locally to generate input and output for each chunk;
- All the input and output locate in stack, so Node A has to commit all the value in the stack;

It is important to note that stack depth is another factor that should be considered. In the automated approach, there are several constraints:

- It is easy to exceed the stack depth limitation;
- It commits many values which will not be used in the current chunk;
- It must implememnt enough gadgets to support any computation, which means achieving turing completeness;
- Executing a large scirpt program is much slower;
- It increases the costs when verifying the expected input and output on-chain;
- The logic of each chunk is unreadable;

However, automated fragmentation has its advantages as well, such as generating the minimal number of script chunks. But since we do not upload all chunks to the Bitcoin network, the number of chunks is less of a concern unless their size becomes excessively large.

3.2.2 manually

Why do we select a manual way to split the entire program?

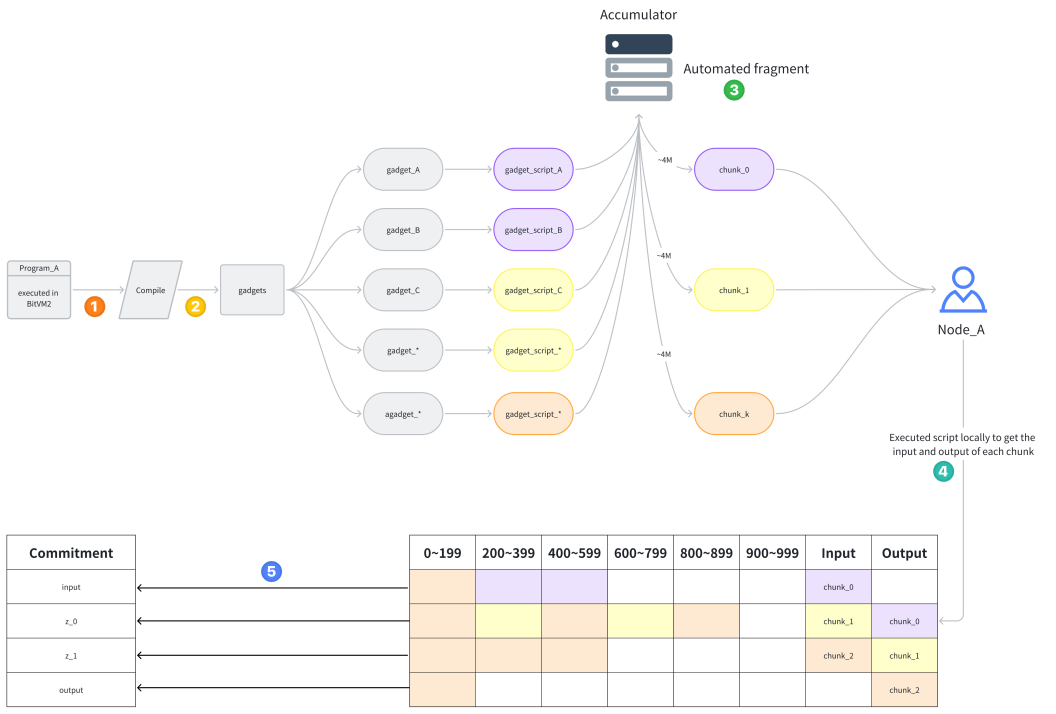

Figure 5: automated fragment

- We avoid exceeding the stack depth limitation by only placing necessary data for the current chunk on the stack;
- We commit only the data used in the current chunk;
- We just need to implement gadgets to support ZKP verification as any computaion could generate a ZK proof;
- We use a Rust program to generate the input and output for each chunk;
- This approach minimizes costs when verifying the expected input and output on-chain;
- The logic of each chunk is readable;

While this approach may generate more chunks, as mentioned earlier, only one script chunk is executed on Bitcoin, so this is acceptable. The overall manual fragmentation process is illustrated in the following figure:

The overall flow proceeds as follows:

- We first concurrently implement the Rust and script versions of the Groth16 ZKP verification;
- The rust version includes the witness generation of each chunk;
- The script version includes all split chunks;
- We ensure each chunk meets the size and depth constraints;
- Node A executes the Rust program locally to generate inputs and outputs for each chunk;
- Node A commits all the inputs and outputs;

3.3 Split data

We present the results directly on how we split the script whose size exceeds the 4M limitation, as demonstrated in **??**, there are only some operations of F_{q12} that need to be split after we optimize the operations for G_1 and G_2

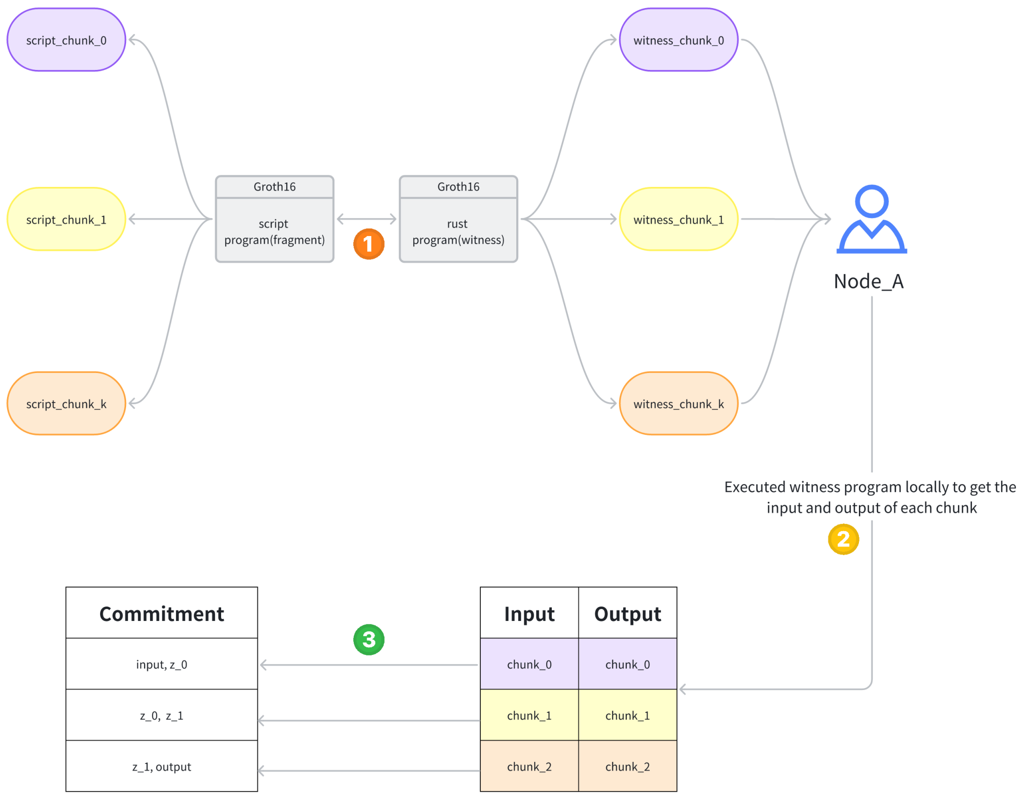

Figure 6: manually fragment

operator type	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 demonstrate how we manually split these four large scripts one by one, striving to satisfy the following properties concurrently:

- Ensuring that size and stack depth limitations are not exceeded;
- Minimizing the size of inputs and outputs;
- Making the logic of each chunk as clear and readable as possible;

3.3.1 split code

We will give an example to explain how the split process works based on $Fn\ \ double\_ell\_affine()$, there have 2 types of code:

- Split_code: it will generate all subscripts;
- Witness_code: it will generate all inputs and outputs for all subscripts

Split_pseudocode

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

        // [p.c0, p.c1, c3, c4, lamda, mu, Q.x, Q.y, 1]
        // [p.c0, p.c1, c3, c4, x3, y3]
        // [final_c0] | [y3, x3]
        // [final_c0, x3, y3]
        res.push(script! {

            { Pairing::double_line_g2_optimization() }

            { Fq2::toaltstack() }
            { Fq2::toaltstack() }

            // [p.c0, p.c1, c3, c4,]
            // compute b = p.c1 * (c3, c4)
            { Fq6::mul_by_01() }
            // [p, c3, c4, b]

            // [p.c0, b]
            { Fq12::mul_fq6_by_nonresidue() }
            // [p.c0, b * beta]

            // compute final c0 = a + beta * b
            { Fq6::add(6, 0) }

            { Fq2::fromaltstack() }
            { Fq2::fromaltstack() }

        });

        // [c0, c3, c4, b, a, p.c0, p.c1, lamda, p.x, -mu, p.y]
        // [c0, c3, c4, b, a, p.c0, p.c1, p.y, -mu, lamda * p.x]
        // [c0, c3, c4, b, a, p.c0, p.c1, lamda * p.x, p.y * -mu]
        // [c0, c3, c4, b, a, p.c0 + p.c1]
        // [c3, c4, b, a, e, c0]
        // [c4, b, a, e, c0, c3]
        // [c4, b, a, e, c0 + c3]
        // [b, a, e, c0 + c3, c4]
        // [b, a, e]
        // [e, a + b]
        // [final_c1]
        res.push(script! {

            { Fq2::mul_by_fq(1, 0) }
            { Fq2::copy(29) }
            { Fq2::equalverify() }
            { Fq2::mul_by_fq(1, 0) }
            { Fq2::copy(28) }
            { Fq2::equalverify() }
            { Fq6::add(6, 0) }
            { Fq2::roll(22) }
            { Fq2::roll(22) }
            { Fq2::add(2, 0) }
            { Fq2::roll(20) }
            { Fq6::mul_by_01() }
            { Fq6::add(12, 6) }
            { Fq6::sub(6, 0) }
        });
        unsafe {
            FQ12_SPLIT_DOUBLE_ELL_COUNTER += 1;
        }
        res
    }
```

Witness_code

```
    pub fn witness_add_ell_affine(
        f: &Fq12,
        p1: &ark_G1Affine,
        affi_t4: &G2Affine,
        affi_q4: &G2Affine,
    ) -> (Fq12, Vec<ScriptContext>, G2Affine) {
        let mut contexts = vec![];
        /////////////////// script-1 //////////////
        // non-fixed (non-constant part) P4
        let (lambda, mu, x3, y3) = Self::line_add_g2(affi_t4, affi_q4);

        let affi_t4_new = G2Affine::new(x3, y3);

        let mut p = ark_G1Affine::new(p1.x, p1.y);

        p.x = -p1.x / p1.y;
        p.y = Fq::from(1) / p1.y;
        /////////////////// script-1 //////////////
        let mut context = ScriptContext::default();

        let c0 = ark_bn254::Fq2::from(1);

        let mut c3 = lambda.clone();
        c3.mul_assign_by_fp(&p.x);

        let mut c4 = -mu.clone();
        c4.mul_assign_by_fp(&p.y);

        let mut b = f.c1 as Fq6;
        b.mul_by_01(&c3, &c4);

        let mut a = f.c0 as Fq6;
        a.mul_by_fp2(&c0);
        let mut beta_b = b;
        <Fq12Config as Fp12Config>::mul_fp6_by_nonresidue_in_place(&mut beta_b);

        let final_c0 = a + beta_b;

        // fill inputs for subscript - 1
        context
            .inputs
            .extend(f.c0.to_base_prime_field_elements().collect::<Vec<Fq>>());
        ...
        // fill outputs for subscript - 1
        context
            .outputs
            .extend(final_c0.to_base_prime_field_elements().collect::<Vec<Fq>>());
        ...

        //////////////////// script-2 ///////////////
        let mut context = ScriptContext::default();
        // fill inputs for subscript - 2
        context
            .inputs
            .extend(c0.to_base_prime_field_elements().collect::<Vec<Fq>>());
        ...

        let e = f.c0 + f.c1;
        let mut e = e;
        let c0_c1 = c0 + c3;
        e.mul_by_01(&c0_c1, &c4);

        let a_b = a + b;
        let final_c1 = e - a_b;

        // fill outputs for subscript - 2
        context
            .outputs
            .extend(final_c1.to_base_prime_field_elements().collect::<Vec<Fq>>());
        contexts.push(context);

        (Fq12::new(final_c0, final_c1), contexts, affi_t4_new)
    }
```

3.3.2 split result

We present the split result directly in the following table:

operator typr	subscript	script size	max depth	exceed 4M?
$F_{q12} : a * b$		6,727,971 bytes	545	yes
	chunk0	3,085,202 bytes	545	no
	chunk1	3,642,594 bytes	545	no
$F_{q12} :\ a * a$		4,495,790 bytes	-	yes
	chunk0	2,246,763 bytes	545	no
	chunk1	2,249,027 bytes	545	no
$F_{q12} :\ doube_ell$		6,416,951 bytes	-	yes
	chunk0	3,708,344 bytes	545	no
	chunk1	2,708,607 bytes	545	no
$F_{q12} :\ add_ell$		6,415,463 bytes	-	yes
	chunk0	3,706,856 bytes	545	no
	chunk1	2,708,607 bytes	545	no
$F_{q12} :\ ell\_by\_constant$		4,714,973 bytes	383	yes
	chunk0	2,568,521 bytes	545	no
	chunk1	2,145,297 bytes	545	no

Note: we combine Fq12_ell and double_g2, add_g2 operation to get a smaller subscript number

3.4 Benchmark data

This section primarily provides benchmark data for various operators used in the Groth16 verification process.

3.4.1 Operators script size origin

We will first present some initial benchmark data from our current implementation, 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 type	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 type	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 type	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
$F_{q12} : a\ *\ a$	7,772,080 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 Operator script size optimization

The fewer script chunks, the better. Therefore, before we split the large operators, we aim to optimize them first. We will present our new data initially, followed by an explanation of the principle.

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

G1 group

operator type	script size	optimized script size	exceed 4M?
$2 \cdot g_1$	1,752,916 bytes	699,519 bytes	no
$g_1 \cdot g_1^{'}$	3,997,319 bytes	562,445 bytes	no

G2 group

operator type	script size	optimized script size	exceed 4M?
$2 \cdot g_2$	7,019,891 bytes	1,847,059 bytes	yes
$g_2 \cdot g_2^{'}$	9,270,854 bytes	1,563,501 bytes	yes

field

operator type	script size	optimized script size	exceed 4M?
$F_{q12} : a\ +\ b$	6,644 bytes	6,644 bytes	no
$F_{q12} : 2 * a$	6,793 bytes	6,793 bytes	no
$F_{q12} : a\ *\ b$	11,641,775 bytes	**6,727,971** bytes	yes
$F_{q12} : a\ *\ a$	7,772,080 bytes	**4,495,790** bytes	yes
$F_{q12} :\ mul\_fq6\_by\_nonresidue$	4,923 bytes	4,923 bytes	no
$F_{q12} :\ frobenius\_map(1)$	4,541,887 bytes	2,879,118 bytes	no
$F_{q12} :\ frobenius\_map(2)$	2,224,363 bytes	2,791,858 bytes	no
$F_{q12} :\ mul\_by\_034$	9,810,459 bytes	**4,289,505** bytes	yes
$F_{q12} :\ ell\_by\_constant$	9,525,050 bytes	**4,714,973** bytes	yes
$F_{q6} :\ a\ *\ b$	3,873,847 bytes	2,236,303 bytes	no
$F_{q6} :\ frobenius\_map(1)$	1,518,206 bytes	933,606 bytes	no
$F_{q6} :\ frobenius\_map(2)$	598,274 bytes	958,404 bytes	no
$F_{q6} :\ mul\_by\_01\_with\_1\_constant$	3,280,529 bytes	1,925,215 bytes	no
$F_{q6} :\ mul\_by\_fp2\_constant$	1,520,337 bytes	960,147 bytes	no
$F_{q6} : mul\_by\_01$	3,769,633 bytes	2,136,209 bytes	no
$F_{q6} :\ mul\_by\_fp2$	2,252,362 bytes	1,273,623 bytes	no
$F_{q2} :\ a\ *\ b$	750,883 bytes	424,433 bytes	no

4 Automated slashing

Automated slashing is a crucial feature of BitVM2, and anyone who declares an invalid assertion should be slashed.

5 Summary

Bibliography

[1] **all signhash types**. URL: `https://wiki.bitcoinsv.io/index.php/SIGHASH_flags`.

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

[3] **blake3**. URL: `https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf`.

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

[5] **Modular Multiplication Without Trial Division**. URL: `https://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/S0025-5718-1985-0777282-X.pdf`.

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

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

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

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

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

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

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

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

[14] **witernitz signature**. URL: `https://toc.cryptobook.us/book.pdf`.