Optimizing Big Integer Multiplication on Bitcoin: Introducing w-windowed Approach

August 16, 2025

Dmytro Zakharov[1], Oleksandr Kurbatov[1], Manish Bista[2], Belove Bist[2]
[1]Distributed Lab, Ukraine. [2]Alpen Labs, USA

eprint.iacr.org/2024/1236

github.com/distributed-lab/bitcoin-window-mul

Introduction to Bitcoin Script

What Bitcoin Script is for?

Recall

Bitcoin Script is a scripting language used in Bitcoin to specify conditions on how the UTXO can be spent.

What Bitcoin Script is for?

Recall

Bitcoin Script is a scripting language used in Bitcoin to specify conditions on how the UTXO can be spent.

Example

The standard `pay-to-pubkey-hash` looks as follows.
`scriptPubKey`:

Script: `OP_DUP OP_HASH160` $\langle H(\mathsf{pk})\rangle$
`OP_EQUALVERIFY OP_CHECKSIG`

What Bitcoin Script is for?

Recall

Bitcoin Script is a scripting language used in Bitcoin to specify conditions on how the UTXO can be spent.

Example

The standard `pay-to-pubkey-hash` looks as follows.
`scriptPubKey`:

Script:	`OP_DUP OP_HASH160` $\langle H(\mathsf{pk})\rangle$ `OP_EQUALVERIFY OP_CHECKSIG`

As a `scriptSig`, the user provides $\{\ \langle\sigma\rangle\ \langle \mathsf{pk}\rangle\ \}$.

How the Bitcoin Script is executed

Consider the `pay-to-pubkey-hash`'s `scriptSig` || `scriptPubKey`:

Script: $\langle \sigma \rangle$ $\langle \mathrm{pk}' \rangle$ `OP_DUP` `OP_HASH160` $\langle H(\mathrm{pk}) \rangle$ `OP_EQUALVERIFY` `OP_CHECKSIG`

How the Bitcoin Script is executed

Consider the `pay-to-pubkey-hash`'s `scriptSig` $\|$ `scriptPubKey`:

Script: $\langle \sigma \rangle$ $\langle \mathrm{pk}' \rangle$ `OP_DUP` `OP_HASH160` $\langle H(\mathrm{pk}) \rangle$ `OP_EQUALVERIFY` `OP_CHECKSIG`

Script: $\langle \sigma \rangle$ $\langle \mathrm{pk}' \rangle$ $\langle \mathrm{pk}' \rangle$ `OP_HASH160` $\langle H(\mathrm{pk}) \rangle$ `OP_EQUALVERIFY` `OP_CHECKSIG`

How the Bitcoin Script is executed

Consider the `pay-to-pubkey-hash`'s `scriptSig` || `scriptPubKey`:

Script: $\langle\sigma\rangle$ $\langle\mathrm{pk}'\rangle$ `OP_DUP` `OP_HASH160` $\langle H(\mathrm{pk})\rangle$ `OP_EQUALVERIFY` `OP_CHECKSIG`

Script: $\langle\sigma\rangle$ $\langle\mathrm{pk}'\rangle$ $\langle\mathrm{pk}'\rangle$ `OP_HASH160` $\langle H(\mathrm{pk})\rangle$ `OP_EQUALVERIFY` `OP_CHECKSIG`

Script: $\langle\sigma\rangle$ $\langle\mathrm{pk}'\rangle$ $\langle H(\mathrm{pk}')\rangle$ $\langle H(\mathrm{pk})\rangle$ `OP_EQUALVERIFY` `OP_CHECKSIG`

How the Bitcoin Script is executed

Consider the `pay-to-pubkey-hash`'s `scriptSig` || `scriptPubKey`:

Script: $\langle\sigma\rangle$ $\langle \mathrm{pk}' \rangle$ `OP_DUP` `OP_HASH160` $\langle H(\mathrm{pk})\rangle$ `OP_EQUALVERIFY` `OP_CHECKSIG`

Script: $\langle\sigma\rangle$ $\langle \mathrm{pk}' \rangle$ $\langle \mathrm{pk}' \rangle$ `OP_HASH160` $\langle H(\mathrm{pk})\rangle$ `OP_EQUALVERIFY` `OP_CHECKSIG`

Script: $\langle\sigma\rangle$ $\langle \mathrm{pk}' \rangle$ $\langle H(\mathrm{pk}')\rangle$ $\langle H(\mathrm{pk})\rangle$ `OP_EQUALVERIFY` `OP_CHECKSIG`

Script: $\langle\sigma\rangle$ $\langle \mathrm{pk}' \rangle$ `OP_CHECKSIG`

How the Bitcoin Script is executed

Consider the `pay-to-pubkey-hash`'s `scriptSig` || `scriptPubKey`:

Script: $\langle\sigma\rangle$ $\langle \mathrm{pk}'\rangle$ `OP_DUP` `OP_HASH160` $\langle H(\mathrm{pk})\rangle$ `OP_EQUALVERIFY` `OP_CHECKSIG`

Script: $\langle\sigma\rangle$ $\langle \mathrm{pk}'\rangle$ $\langle \mathrm{pk}'\rangle$ `OP_HASH160` $\langle H(\mathrm{pk})\rangle$ `OP_EQUALVERIFY` `OP_CHECKSIG`

Script: $\langle\sigma\rangle$ $\langle \mathrm{pk}'\rangle$ $\langle H(\mathrm{pk}')\rangle$ $\langle H(\mathrm{pk})\rangle$ `OP_EQUALVERIFY` `OP_CHECKSIG`

Script: $\langle\sigma\rangle$ $\langle \mathrm{pk}'\rangle$ `OP_CHECKSIG`

Script: $\langle 1\rangle$

How the Bitcoin Script is executed

Consider the `pay-to-pubkey-hash`'s `scriptSig` || `scriptPubKey`:

Script: $\langle\sigma\rangle$ $\langle \mathrm{pk}' \rangle$ `OP_DUP` `OP_HASH160` $\langle H(\mathrm{pk})\rangle$ `OP_EQUALVERIFY` `OP_CHECKSIG`

Script: $\langle\sigma\rangle$ $\langle \mathrm{pk}' \rangle$ $\langle \mathrm{pk}' \rangle$ `OP_HASH160` $\langle H(\mathrm{pk})\rangle$ `OP_EQUALVERIFY` `OP_CHECKSIG`

Script: $\langle\sigma\rangle$ $\langle \mathrm{pk}' \rangle$ $\langle H(\mathrm{pk}')\rangle$ $\langle H(\mathrm{pk})\rangle$ `OP_EQUALVERIFY` `OP_CHECKSIG`

Script: $\langle\sigma\rangle$ $\langle \mathrm{pk}' \rangle$ `OP_CHECKSIG`

Script: $\langle 1 \rangle$

Note

One can spend the UTXO iff the output is `OP_1`.

Can we do more?

So typically, Bitcoin Script allows writing only basic smart contracts using **native** `OP_CODEs`:

✓ Hash Preimage Verification and Timelocks.

Can we do more?

So typically, Bitcoin Script allows writing only basic smart contracts using **native** `OP_CODEs`:

✓ Hash Preimage Verification and Timelocks.

✓ Basic Signatures (ECDSA for tx data, Schnorr for Taproot).

Can we do more?

So typically, Bitcoin Script allows writing only basic smart contracts using **native** `OP_CODEs`:

- ✓ Hash Preimage Verification and Timelocks.
- ✓ Basic Signatures (ECDSA for tx data, Schnorr for Taproot).
- ✓ Threshold/Multisignatures.

Can we do more?

So typically, Bitcoin Script allows writing only basic smart contracts using **native** `OP_CODE`s:

- ✓ Hash Preimage Verification and Timelocks.
- ✓ Basic Signatures (ECDSA for tx data, Schnorr for Taproot).
- ✓ Threshold/Multisignatures.
- ✓ Combination of those.

Can we do more?

So typically, Bitcoin Script allows writing only basic smart contracts using **native** `OP_CODEs`:

- ✓ Hash Preimage Verification and Timelocks.
- ✓ Basic Signatures (ECDSA for tx data, Schnorr for Taproot).
- ✓ Threshold/Multisignatures.
- ✓ Combination of those.

Question

Can we implement some non-native verifications? For example, zk-SNARKs (Groth16, fflonk), zk-STARKs, BLS Signatures?

Can we do more?

So typically, Bitcoin Script allows writing only basic smart contracts using **native** `OP_CODEs`:

- ✓ Hash Preimage Verification and Timelocks.
- ✓ Basic Signatures (ECDSA for tx data, Schnorr for Taproot).
- ✓ Threshold/Multisignatures.
- ✓ Combination of those.

Question

Can we implement some non-native verifications? For example, zk-SNARKs (Groth16, fflonk), zk-STARKs, BLS Signatures?

Significance: Having effective Bitcoin on-chain Groth16 verification enables the L2 over Bitcoin!

Can we do more?

- ✓ Groth16 is already implemented.

Can we do more?

- ✓ Groth16 is already implemented.
- ✓ fflonk is already implemented.

Can we do more?

✓ Groth16 is already implemented.

✓ fflonk is already implemented.

✗ zk-STARK cannot be currently implemented (requires `OP_CAT` for Fiat-Shamir transformation and Merkle Trees). Yet, assuming `OP_CAT`, the **Circle STARK** is implemented!

Can we do more?

✓ Groth16 is already implemented.

✓ fflonk is already implemented.

✗ zk-STARK cannot be currently implemented (requires `OP_CAT` for Fiat-Shamir transformation and Merkle Trees). Yet, assuming `OP_CAT`, the **Circle STARK** is implemented!

✓ Any discrete-log-based protocol that does not involve hashing (typically requiring concatenation) can be implemented: Σ-protocols, Bulletproofs, BLS Signatures.

Can we do more?

✓ Groth16 is already implemented.

✓ fflonk is already implemented.

✗ zk-STARK cannot be currently implemented (requires `OP_CAT` for Fiat-Shamir transformation and Merkle Trees). Yet, assuming `OP_CAT`, the **Circle STARK** is implemented!

✓ Any discrete-log-based protocol that does not involve hashing (typically requiring concatenation) can be implemented: Σ-protocols, Bulletproofs, BLS Signatures.

Note

In other words, currently, it is *theoretically possible* to build a Groth16 zk-SNARK verification of proof π in a form

Script: $\langle\pi\rangle$ $\langle$public statement$\rangle$ `OP_CHECKGROTH16`

Demystifying Math behind BitVM Groth16

✓ Arithmetic is allowed only over `u32` integers.

[1] With dropping variations such as `OP_1ADD` and such.

Demystifying Math behind BitVM Groth16

- ✓ Arithmetic is allowed only over `u32` integers.
- ✓ From arithmetic, one only[1] has `OP_ADD`, `OP_SUB`, `OP_NEGATE`, `OP_ABS`, `OP_LESSTHAN`, `OP_GREATERTHAN`, `OP_BOOLAND`, `OP_BOOLOR`.

[1] With dropping variations such as `OP_1ADD` and such.

Demystifying Math behind BitVM Groth16

- ✓ Arithmetic is allowed only over `u32` integers.
- ✓ From arithmetic, one only[1] has `OP_ADD`, `OP_SUB`, `OP_NEGATE`, `OP_ABS`, `OP_LESSTHAN`, `OP_GREATERTHAN`, `OP_BOOLAND`, `OP_BOOLOR`.
- ✓ Flow control is very limited: only `OP_IF`s/`OP_ELSE`s are allowed. No for/while loops, but almost full control of compile-time stack movement.

[1] With dropping variations such as `OP_1ADD` and such.

Demystifying Math behind BitVM Groth16

- ✓ Arithmetic is allowed only over `u32` integers.
- ✓ From arithmetic, one only[1] has `OP_ADD`, `OP_SUB`, `OP_NEGATE`, `OP_ABS`, `OP_LESSTHAN`, `OP_GREATERTHAN`, `OP_BOOLAND`, `OP_BOOLOR`.
- ✓ Flow control is very limited: only `OP_IF`s/`OP_ELSE`s are allowed. No for/while loops, but almost full control of compile-time stack movement.

Question

Currently, big integer multiplication takes 74.9k OPCODEs to implement. Can we implement a better approach?

[1] With dropping variations such as `OP_1ADD` and such.

Big Integer Arithmetic

Representing large integers

Recall

We can represent any integer x in arbitrary base b:

$$x = \sum_{j=0}^{n-1} x_j b^j, \quad 0 \leq x_j < b$$

Representing large integers

Recall

We can represent any integer x in arbitrary base b:

$$x = \sum_{j=0}^{n-1} x_j b^j, \quad 0 \leq x_j < b$$

Numbers $x_0, x_1, \ldots, x_{n-1}$ are called **limbs**, where n is the **limb-size** of x in base b.

Representing large integers

Recall

We can represent any integer x in arbitrary base b:

$$x = \sum_{j=0}^{n-1} x_j b^j, \quad 0 \leq x_j < b$$

Numbers $x_0, x_1, \ldots, x_{n-1}$ are called **limbs**, where n is the **limb-size** of x in base b.

Idea #1

If b is small enough, we can publish individual limbs $x_0, \ldots, x_{n-1}$ that constitute the whole number x.

Representing large integers

Recall

We can represent any integer x in arbitrary base b:

$$x = \sum_{j=0}^{n-1} x_j b^j, \quad 0 \leq x_j < b$$

Numbers $x_0, x_1, \dots, x_{n-1}$ are called **limbs**, where n is the **limb-size** of x in base b.

Idea #1

If b is small enough, we can publish individual limbs $x_0, \dots, x_{n-1}$ that constitute the whole number x.

Idea #2

Since we want to minimize the number of limbs, we take the largest b possible (with $b = 2^t$ for convenience). Thus, we set $b := 2^{30}$.

Representing large integers

Example

Consider the following 254-bit integer:

$$x = (\texttt{0xbe48fffd2a6f534dc}\\ \texttt{5b6a6901840fc0fb65827e6}\\ \texttt{efd22a8063cded681f5f7b2})$$

Representing large integers

Example

Consider the following 254-bit integer:

$$x = (\texttt{0xbe48fffd2a6f534dc}\\ \texttt{5b6a6901840fc0fb65827e6}\\ \texttt{efd22a8063cded681f5f7b2})$$

To add this integer to the stack, one uses the following script:

Script:

```
OP_PUSHBYTES_2 ⟨e40b⟩ OP_PUSHBYTES_4 ⟨a9f4ff23⟩
OP_PUSHBYTES_4 ⟨c54d532f⟩ OP_PUSHBYTES_4 ⟨06a4a92d⟩
OP_PUSHBYTES_4 ⟨fbc00f04⟩ OP_PUSHBYTES_4 ⟨9b9f6019⟩
OP_PUSHBYTES_4 ⟨802ad22f⟩ OP_PUSHBYTES_4 ⟨5a7bf318⟩
OP_PUSHBYTES_4 ⟨b2f7f501⟩
```

Representing large integers

Example

Consider the following 254-bit integer:

$$x = (\mathtt{0xbe48fffd2a6f534dc}$$
$$\mathtt{5b6a6901840fc0fb65827e6}$$
$$\mathtt{efd22a8063cded681f5f7b2})$$

To add this integer to the stack, one uses the following script:

Script:

OP_PUSHBYTES_2 ⟨e40b⟩ OP_PUSHBYTES_4 ⟨a9f4ff23⟩
OP_PUSHBYTES_4 ⟨c54d532f⟩ OP_PUSHBYTES_4 ⟨06a4a92d⟩
OP_PUSHBYTES_4 ⟨fbc00f04⟩ OP_PUSHBYTES_4 ⟨9b9f6019⟩
OP_PUSHBYTES_4 ⟨802ad22f⟩ OP_PUSHBYTES_4 ⟨5a7bf318⟩
OP_PUSHBYTES_4 ⟨b2f7f501⟩

Note: One needs **9 limbs** to represent a 254-bit integer.

BigInt Addition

Problem

Given two 254-bit integers x and y, find $z := x + y$, assuming overflowing does not occur.

BigInt Addition

Problem

Given two 254-bit integers x and y, find $z := x + y$, assuming overflowing does not occur.

Solution. We have two representations:

$$x = \sum_{j=0}^{8} x_j \times 2^{30j}, \quad y = \sum_{j=0}^{8} y_j \times 2^{30j}$$

BigInt Addition

Problem

Given two 254-bit integers x and y, find $z := x + y$, assuming overflowing does not occur.

Solution. We have two representations:

$$x = \sum_{j=0}^{8} x_j \times 2^{30j}, \quad y = \sum_{j=0}^{8} y_j \times 2^{30j}$$

Idea: add limb by limb, starting from the least significant one.

BigInt Addition

Problem

Given two 254-bit integers x and y, find $z := x + y$, assuming overflowing does not occur.

Solution. We have two representations:

$$x = \sum_{j=0}^{8} x_j \times 2^{30j}, \quad y = \sum_{j=0}^{8} y_j \times 2^{30j}$$

Idea: add limb by limb, starting from the least significant one.

1. On step i, calculate $t \leftarrow x_i + y_i + \text{carry}$ (start with zero carry).

BigInt Addition

Problem

Given two 254-bit integers x and y, find $z := x + y$, assuming overflowing does not occur.

Solution. We have two representations:

$$x = \sum_{j=0}^{8} x_j \times 2^{30j}, \quad y = \sum_{j=0}^{8} y_j \times 2^{30j}$$

Idea: add limb by limb, starting from the least significant one.

1. On step i, calculate $t \leftarrow x_i + y_i + \text{carry}$ (start with zero carry).
2. If $t < 2^{30}$, set $z_i \leftarrow t$, $\text{carry} \leftarrow 0$.

BigInt Addition

Problem

Given two 254-bit integers x and y, find $z := x + y$, assuming overflowing does not occur.

Solution. We have two representations:

$$x = \sum_{j=0}^{8} x_j \times 2^{30j}, \quad y = \sum_{j=0}^{8} y_j \times 2^{30j}$$

Idea: add limb by limb, starting from the least significant one.

1. On step i, calculate $t \leftarrow x_i + y_i + \text{carry}$ (start with zero carry).
2. If $t < 2^{30}$, set $z_i \leftarrow t$, carry $\leftarrow 0$.
3. If $t \geq 2^{30}$, set $z_i \leftarrow t - 2^{30}$, carry $\leftarrow 1$.

BitInt Addition: Bitcoin Script

Algorithm 7: Adding two integers assuming with no overflow

Input : Two integers on the stack: $\{ \langle x_{\ell-1} \rangle \dots \langle x_0 \rangle \; \langle y_{\ell-1} \rangle \dots \langle y_0 \rangle \}$

Output : Result of addition $z = x + y$ in a form $\{ \langle z_{\ell-1} \rangle \dots \langle z_0 \rangle \}$

{ OP_ZIP } ; /* Convert current stack $\{ \langle x_{\ell-1} \rangle \dots \langle x_0 \rangle \; \langle y_{\ell-1} \rangle \dots \langle y_0 \rangle \}$ to the form $\{ \langle x_{\ell-1} \rangle \; \langle y_{\ell-1} \rangle \dots \langle x_0 \rangle \; \langle y_0 \rangle \}$ */

$\{ \langle \beta \rangle \}$; /* Push base to the stack */

{ OP_LIMB_ADD_CARRY OP_TOALTSTACK }

for $\_ \in \{0, \dots, \ell - 3\}$ **do**

- /* At this point, stack looks as $\{ \langle x_n \rangle \; \langle y_n \rangle \; \langle \beta \rangle \; \langle c \rangle \}$. We need to add carry c and call OP_LIMB_ADD_CARRY */
- { OP_ROT }
- { OP_ADD }
- { OP_SWAP }
- { OP_LIMB_ADD_CARRY OP_TOALTSTACK }

end

/* At this point, again, stack looks as $\{ \langle x_n \rangle \; \langle y_n \rangle \; \langle \beta \rangle \; \langle c \rangle \}$. We need to drop the base, add carry, and conduct addition, assuming overflowing does not occur */

{ OP_NIP OP_ADD , OP_ADD }

/* Return all limbs to the main stack */

for $\_ \in \{0, \dots, \ell - 2\}$ **do**

- { OP_FROMALTSTACK }

end

BigInt Multiplication

Problem

Given two 254-bit integers x and y, find 508-bit $z := x \times y$.

BigInt Multiplication

Problem

Given two 254-bit integers x and y, find 508-bit $z := x \times y$.

Algorithm 2: Double-and-add method for integer multiplication

---

Input : x, y — two integers being multiplied
Output : Result of the multiplication $x \times y$

```
Decompose y to the binary form: (y_0, y_1, ..., y_{N-1})_2
r ← 0
t ← x
for i ∈ {0, ..., N − 1} do
    if y_i = 1 then
        r ← r + t
    end
    t ← 2 × t
end
```

Return : Integer r

Problem

Problem: If the cost of addition in A and of doubling is D, the total algorithm's cost is $NA + ND$.

Problem

Problem: If the cost of addition in A and of doubling is D, the total algorithm's cost is $NA + ND$.

Definition

The w-**width** form of a scalar $k \in \mathbb{Z}_{\geq 0}$ is the representation:

$$k = \sum_{i=0}^{L-1} k_i \times 2^{wi}, \quad 0 \leq k_i < 2^w,$$

where the decomposition length is $L = \lceil N/w \rceil$.

Problem

Problem: If the cost of addition in A and of doubling is D, the total algorithm's cost is $N\mathsf{A} + N\mathsf{D}$.

Definition

The w-**width** form of a scalar $k \in \mathbb{Z}_{\geq 0}$ is the representation:

$$k = \sum_{i=0}^{L-1} k_i \times 2^{wi}, \quad 0 \leq k_i < 2^w,$$

where the decomposition length is $L = \lceil N/w \rceil$.

Using multiplication in such form, we will reduce the complexity down to:

$$[2^{w-1}\mathsf{A} + 2^{w-1}\mathsf{D}] + \left[\frac{N}{w}\mathsf{A} + N\mathsf{D}\right]$$

w-width BigInt Multiplication

Algorithm 3: Double-and-add method for integer multiplication

Input : x, y — two integers being multiplied
Output : Result of the multiplication $x \times y$

Decompose y to the w-width form: $(y_0, y_1, \ldots, y_{L-1})_w$
Precompute $\{0x, 1x, 2x, \ldots, (2^w - 1)x\}$. Denote $\mathcal{T}[j] = jx$.
$q \leftarrow 0$
for $i \in \{L-1, \ldots, 0\}$ **do**
    for $\_ \in \{1, \ldots, w\}$ **do**
        $q \leftarrow 2q$
    end
    $q \leftarrow q + \mathcal{T}[y_i]$
end
Return : Integer q

Further Optimization

However, which value of w to choose? In our research, we optimized the cost function:

$$C(w) = 2^{w-1}(C_A + C_D) + \frac{NC_A}{w} + NC_D$$

Further Optimization

However, which value of w to choose? In our research, we optimized the cost function:

$$C(w) = 2^{w-1}(C_A + C_D) + \frac{NC_A}{w} + NC_D$$

- $2^{w-1}(C_A + C_D)$ — lookup table initialization cost.

Further Optimization

However, which value of w to choose? In our research, we optimized the cost function:

$$C(w) = 2^{w-1}(C_A + C_D) + \frac{NC_A}{w} + NC_D$$

- $2^{w-1}(C_A + C_D)$ — lookup table initialization cost.
- $NC_A/w + NC_D$ — cost of operations in the main loop.

Further Optimization

However, which value of w to choose? In our research, we optimized the cost function:

$$C(w) = 2^{w-1}(C_A + C_D) + \frac{NC_A}{w} + NC_D$$

- $2^{w-1}(C_A + C_D)$ — lookup table initialization cost.
- $NC_A/w + NC_D$ — cost of operations in the main loop.

Theorem

Optimal value is given by $\widehat{w}^2 2^{\widehat{w}} = \frac{2N}{\log 2} \cdot \frac{C_A}{C_A + C_D}$

For BN254, we have $N = 254$, $\widehat{w} = 4$, so we need $72\text{A} + 262\text{D}$ operations instead of $254\text{A} + 254\text{D}$.

What is left to optimize

How to prevent the overflowing when multiplying two N-bit integers x, y and getting an $2N$-bit integer z?

What is left to optimize

How to prevent the overflowing when multiplying two N-bit integers x, y and getting an $2N$-bit integer z?

Naive approach: Extend x, y to $2N$ bit and conduct the algorithm as usual.

What is left to optimize

How to prevent the overflowing when multiplying two N-bit integers x, y and getting an $2N$-bit integer z?

Naive approach: Extend x, y to $2N$ bit and conduct the algorithm as usual.

Better approach: Extend number of bits of x and y gradually throughout the loop.

Results

Approach	Overflowing Multiplication	Widening Multiplication
Cmpeq	N/A	201,879
BitVM `bigint`	106,026	200,334
BitVM Karatsuba	N/A	74,907
Our w-width method	55,710	71,757

- Besides better results in terms of OPCODEs #, this method allows to optimize MSM (batch multiplication).
- As a result, our implementation is included in the BitVM codebase and the total Groth16 script size is reduced 2×.

Thank you for your attention

♥

distributedlab.com

github.com/distributed-lab/bitcoin-window-mul