Fast Big Integer Multiplication using w-windowed method on Bitcoin

Dmytro Zakharov[1], Oleksandr Kurbatov[1], Manish Bista[2] and Belove Bist[2]

[1] Research Department at Distributed Lab dmytro.zakharov@distributedlab.com, ok@distributedlab.com

[2] Alpen Labs manish@alpenlabs.io, belove@alpenlabs.io

Abstract. A crucial component of any SNARK system is performing finite field arithmetic, which inherently involves the fundamental task of multiplying two large integers. Performing such arithmetic on Bitcoin is particularly challenging due to the limitations of Bitcoin's scripting language. Bitcoin Script is intentionally non-Turing-complete and stack-based, designed with simplicity and security in mind. Hence, it lacks built-in support for complex arithmetic operations and has constraints on the size and number of stack elements. Implementing efficient big integer multiplication requires innovative techniques to work within these constraints while minimizing the number of opcodes used.

This paper introduces the w-windowed method for multiplying two 254-bit prime q (BN254 curve) integers, along with additional optimization techniques. Building on this fast multiplication, we implement modular multiplication with further enhancements. Our approach significantly reduces the number of opcodes compared to state-of-the-art methods, making it a substantial improvement in performing finite field arithmetic on Bitcoin. This primitive, central to protocols like SNARKnado, brings us closer to practical SNARK verification on Bitcoin.

Keywords: Bitcoin, Bitcoin Script, Fast Multiplication, Elliptic Curve Scalar Multiplication, BitVM

Contents

1 Introduction

2 Preliminaries

2.1 Bitcoin Script

2.1.1 Basic Structure

Bitcoin Script is a stack-based, not Turing-complete language used for specifying conditions on how UTXO can be spent [1]. Informally, this condition is called `scriptPubKey`, while the data that must be provided to meet this condition is called `scriptSig`[1]. To verify that the condition is met based on `scriptSig` provided, one should first concatenate `scriptSig` ‖ `scriptPubKey`, execute the script and verify that the resultant stack contains a non-false value (meaning, anything except for `0`).

The stack consists of the values placed in the script and the so-called **opcodes** — keywords that operate with the elements in the stack. Let us consider some examples to introduce notation and describe how the script gets executed.

Example 1. The script $\{\ \langle a\rangle\ \langle b\rangle\ \texttt{OP\_ADD}\ \langle c\rangle\ \texttt{OP\_EQUAL}\ \}$ verifies whether given a, b, c satisfy $a + b = c$. We first push two integers a and b to the stack, then `OP_ADD` will consume a and b (meaning, they get removed) and output $s \leftarrow a + b$, so the stack becomes $\{\ \langle s\rangle\ \langle c\rangle\ \texttt{OP\_EQUAL}\ \}$. Finally, `OP_EQUAL` takes s and c and outputs `OP_TRUE` if $a + b = c$, and `OP_FALSE`, otherwise. Note that such notation is commonly called the *Reverse Polish Notation* in the literature [4].

Example 2. Suppose our condition on spending the coins is providing the pre-image of the given hash value h (that is, providing a message m such that $h = \mathsf{H}(m)$), which is called the *Hashlock Script*. In this case, our `scriptPubKey` looks as follows[2]:

Stack:	`OP_HASH160` $\langle h\rangle$ `OP_EQUAL`

Suppose we brought a message m, our `scriptSig`. Concatenating `scriptSig` and `scriptPubKey` would result in the following script:

Stack:	$\langle m\rangle$ `OP_HASH160` $\langle h\rangle$ `OP_EQUAL`

Execution in this case would proceed as follows:

1. First, m is added to the stack.
2. Next, `OP_HASH160` will hash the provided value $h' \leftarrow \mathsf{H}(m)$, so the stack would become $\{\ \langle h'\rangle\ \langle h\rangle\ \texttt{OP\_EQUAL}\ \}$.
3. Finally, after executing `OP_EQUAL`, we will either get `OP_TRUE` on the top of the stack if $h = h'$, or `OP_FALSE` otherwise.

Note that we get `OP_TRUE` (meaning, we can spend the coins) only if $h' = h$ or, equivalently, $\mathsf{H}(m) = h$, what was needed from the start.

2.1.2 Arithmetic in Bitcoin

To implement the SNARK verifier on Bitcoin, one must implement the finite field arithmetic over the elliptic curve scalar field $\mathbb{F}_q$. The bitsize of such scalar field is typically from 254 bits (as for *BN254* [2]) to 381 bits and more (as for *BLS12-381* [5]). Currently, the

[1] Formally, `scriptSig` might contain the logic as well, but we omit the details here.

[2] It should be noted, though, that in the placeholder $\langle h\rangle$ we should push `0x20` followed by 20 bytes of h.

common choice is the *BN254* based on 254-bit prime order q, which, for example, is currently used for elliptic curve precompiles in *Ethereum* [8]. Although further discussion is valid for any fairly large q, our implementation was focused on 254-bit q.

Finite field arithmetic over N-bit q (where $N = 254$ for *BN254*, for example) includes implementing the widening multiplication of two N-bit numbers, resulting in a $2N$-bit integer. Why is this a problem in Bitcoin at all? The main issue is that Bitcoin does not have a multiplication opcode[3]. To make matters worse, integers on the stack are 32-bit, meaning that representing large integers requires some additional workload. Therefore, we will use the **base** β representation of an integer.

Definition 1. Given positive integer $x \in \mathbb{Z}_{\geq 0}$, **base** β representation is an expression

$$x = \sum_{k=0}^{\ell-1} x_k \times \beta^k, \tag{1}$$

where each **limb** x_k is between 0 and $\beta - 1$, and ℓ is the length of such representation. We further denote such representation by $(x_0, x_1, \ldots, x_{\ell-1})_\beta$.

Empirically, it seems that using larger bases results in smaller scripts. The main reason is that larger bases result in the shorter representation of integers. However, this does not mean better methods with shorter integers will not produce shorter scripts in the future. Therefore, we pick $\beta = 2^{30}$: it is the power of two, which would come in handy later, and we will not run out of 32 bits when performing arithmetic (doublings, additions, etc.). Also, assume the limb size in bits is $n = 30$.

Moreover, Bitcoin does not have loops (recall that Bitcoin Script is not Turing complete!), meaning that the length of our representation must be fixed. It means that $\ell = \lceil N/n \rceil$, or, $\ell = 9$ in our particular case.

All things combined, Algorithm 1 shows how to preprocess the given integer x and push the representation to the stack.

Algorithm 1: Pushing given integer to the stack

Input : Integer x of bit size up to N
Output : Representation $(X_0, X_1, \ldots, X_{\ell-1})_\beta$ for $\beta = 2^n$ which can be inserted to the stack (meaning $n \leq 32$).

1 Decompose x to the binary form: $(x_0, x_1, \ldots, x_{N-1})_2$
2 Split the form into chunks of size n (the last chunk would be of size $N - (\ell - 1)n$)
3 For k^{th} chunk $(c_0, \ldots, c_{m-1})$ set $X_k \leftarrow \sum_{j=0}^{m-1} c_j 2^j$

Return : $(X_0, X_1, \ldots, X_{\ell-1})$

2.2 Multiplication Methods

2.2.1 Karatsuba Algorithm

The **Karatsuba Algorithm** is a fast multiplication algorithm to multiply two integers using *divide and conquer* approach [7]. In contrast to naive $O(N^2)$ complexity, the Karatsuba method allows to reduce the asymptotic complexity to $O(N^{\log_2 3})$.

Assume that we have integers x and y, represented in base β with ℓ limbs. We divide each number into two halves: high bits x_H, y_H and low bits x_L, y_L as follows:

$$x = x_H \beta^{\lceil \ell/2 \rceil} + y_L, \; y = y_H \beta^{\lceil \ell/2 \rceil} + y_L \tag{2}$$

[3] At some point, Bitcoin did have `OP_MUL`, but it was later disabled.

Then, a simple multiplication formula gives us:

$$xy = x_H y_H \beta^{\ell} + (x_H y_L + x_L y_H)\beta^{\lceil \ell/2 \rceil} + x_L y_L \tag{3}$$

Which requires multiplying four times: $x_H y_H, x_H y_L, x_L y_H, x_L y_L$. Now, the Karatsuba algorithm consists in calculating these four expressions using only three multiplications. Indeed, calculate: $c_0 = x_H y_H, c_1 = x_L y_L$, then $c_2 = (x_H + x_L)(y_H + y_L) - c_1 - c_0$, and then

$$xy = c_0 \beta^{\ell} + c_2 \beta^{\lceil \ell/2 \rceil} + c_1 \tag{4}$$

The Karatsuba Algorithm is used in the current *BitVM* approach, where to represent the 254-bit number, one uses 29×9 representation (that is, $n = 29, \ell = 9$), resulting in roughly 74.9k opcodes [6].

2.2.2 Elliptic Curve Scalar Multiplication

Ideas from methods used for Elliptic curve scalar multiplication will be helpful in further optimizations. Subsequent methods will be primarily based on explanations from [3].

Assume that $(E(\mathbb{F}_q), \oplus)$ is the group of points on an elliptic curve under operation $\oplus$ over some prime field $\mathbb{F}_q$ of a prime order r. Suppose $P \in E(\mathbb{F}_q)$ and $k \in \mathbb{Z}_r$ and denote by $[k]P$ adding P to itself k times (for $k = 0$ assume $[0]P = \mathcal{O}$ where $\mathcal{O}$ is the point at infinity). Also, assume that k is, again, N-bit sized for notation simplicity.

The basic classical approach of multiplying point P by k is specified in Algorithm 2.

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

Input : $P \in E(\mathbb{F}_q)$ and $k \in \mathbb{Z}_r$
Output : Result of scalar multiplication $[k]P \in E(\mathbb{F}_q)$

Decompose k to the binary form: $(k_0, k_1, \ldots, k_{N-1})$
$R \leftarrow \mathcal{O}$
$T \leftarrow P$
for $i \in \{0, \ldots, N-1\}$ **do**
  if $k_i = 1$ **then**
    $R \leftarrow R \oplus T$
  end
  $T \leftarrow [2]T$
end
Return : Point R

As can be seen, the complexity of such an approach is $O(\log_2 k)$. Specifically, suppose A is the cost of addition while D is the cost of doubling[4]. In this case, the maximal total cost is roughly $N\mathsf{A} + N\mathsf{D}$. However, we can do better by using the w-width approach. The main idea is to decompose the scalar k into the w-width format.

Definition 2. The **w-width** form of a scalar $k \in \mathbb{Z}_{\geq 0}$ is a base 2^w representation, that is

$$k = \sum_{i=0}^{L-1} k_i \times 2^{wi}, \quad 0 \leq k_i < 2^w \tag{5}$$

Let the **length** of such decomposition be $L := \lceil N/w \rceil$. We denote such decomposition by $(k_0, k_1, \ldots, k_{L-1})_w$.

[4]Of course, D is slightly easier to perform than A since doubling is a special case of addition.

Now, what does this form give us? Let us consider Algorithm 3. At first glance, the overall complexity is still $O(\log_2 k)$, but a closer inspection reveals that the number of additions is significantly lower for a suitable choice of w. Indeed, the number of doublings is still roughly N, but the number of additions is now approximately N/w. Of course, this comes at a cost of initializing the lookup table: to initialize 2^w values we need roughly 2^{w-1} additions and 2^{w-1} doublings (to calculate $[2m]P$ we can always double $[m]P$, while for calculating $[2m+1]P$, add P to already precomputed $[2m]P$). So the overall cost is:

$$\left[2^{w-1}\mathsf{A} + 2^{w-1}\mathsf{D}\right] + \left[\frac{N}{w}\mathsf{A} + N\mathsf{D}\right] \tag{6}$$

Note that the cost of initializing the lookup table grows exponentially with respect to w, so typically the best choice is $w = 4$. This way, instead of having roughly 254 additions maximum, we get 64 instead.

Algorithm 3: w-width windowed method for scalar multiplication

Input : $P \in E(\mathbb{F}_q)$ and $k \in \mathbb{Z}_r$
Output : Result of scalar multiplication $[k]P \in E(\mathbb{F}_q)$

Decompose k to the w-width form: $(k_0, k_1, \ldots, k_{L-1})_w$
Precompute values $\{[0]P, [1]P, [2]P, \ldots, [2^w - 1]P\}$ (in other words, implement the lookup table). Denote by $\mathcal{T}[j] = [j]P$ – referencing the lookup table at index j.
$Q \leftarrow \mathcal{O}$
for $i \in \{L-1, \ldots, 0\}$ **do**
  for $\_ \in \{1, \ldots, w\}$ **do**
    $Q \leftarrow [2]Q$
  end
  $Q \leftarrow Q \oplus \mathcal{T}[k_i]$
end
Return : Q

Yet another effective approach is w-width non-adjacent form (NAF). Let us introduce it first.

Definition 3. Again, assume $w \geq 2$. A **width-w** NAF of $k \in \mathbb{Z}_{\geq 0}$ is an expression $k = \sum_{i=0}^{L-1} k_i 2^i$ where each non-zero coefficient k_i is odd, $|k_i| < 2^{w-1}$, and at most one of any w consecutive digits is non-zero.

The main properties of width-w NAF are listed in the next theorem.

Theorem 1. *Let $k \in \mathbb{Z}_{\geq 0}$. Then,*

1. *k has a unique width-w NAF, denoted by $(k_0, \ldots, k_{L-1})_{w,\mathsf{NAF}}$.*
2. *The length of width-w NAF is at most one more than the binary representation of k.*
3. *The average density of non-zero digits in width-w NAF is approximately $1/(w+1)$.*

Among three listed properties, probably the most important is the third one. Indeed, if we take a random L-sized width-w NAF of some integer, most likely it would have only $L/(w+1)$ non-zero digits, so the average number of additions would be $L/(w+1)$ – this is slightly lower than L/w which we had before. The resultant algorithm is identical to Algorithm 3 except for the fact that it suffices to precompute only odd products $\{[1]P, [3]P, \ldots, [2^{w-1}-1]P\}$ and their negatives (where negative is easily computed in case of $E(\mathbb{F}_q)$ using relation $\ominus P = \ominus(x_P, y_P) = (x_P, -y_P)$).

However, this method has not provided us with fewer opcodes for the reasons provided in subsequent sections.

3 Implementation

3.1 Binary and Window Decomposition

First things first, we need to decompose our integer to the binary form using *Bitcoin Script*. Since we have chosen our base to be the power of two, it suffices to decompose the limbs to the binary form and then concatenate the result (this is the primary reason for using $\beta = 2^n$ and not any other limb base). The implementation is specified in Algorithm 4.

Algorithm 4: Decomposing a limb to the binary form

Input : A single n-bit integer x ($n \leq 32$)
Output : Bits $(x_0, x_1, \ldots, x_{n-1})$ in `altstack`

```
1  { OP_TOALTSTACK } ;                          /* Moving limb to altstack */
2  for i ∈ {0, ..., n − 1} do
3  |  { ⟨2 ≪ i⟩ } ;                            /* Pushing powers of two */
4  end
5  { OP_FROMALTSTACK } ;                        /* Getting element back */
6  for _ ∈ {0, ..., n − 1} do
7  |  { OP_2DUP OP_LESSTHANOREQUAL }
8  |  { OP_IF }
9  |     { OP_SWAP OP_SUB ⟨1⟩ }
10 |  { OP_ELSE }
11 |     { OP_NIP ⟨0⟩ }
12 |  { OP_ENDIF }
13 |  { OP_TOALTSTACK }
14 end
```

The idea here is quite straightforward: we first make the stack in a form

Stack:	$\langle 2^1 \rangle \ \langle 2^2 \rangle \ \langle 2^3 \rangle \ \ldots \ \langle 2^n \rangle \ \langle x \rangle$

Then, we duplicate top-stack elements to get $\{ \ldots \langle 2^n \rangle \ \langle x \rangle \ \langle 2^n \rangle \ \langle x \rangle \}$, then checking whether $2^n \leq x$. If not, we remove 2^n and push $\langle 0 \rangle$ to the `altstack`, otherwise we modify x to be $x - 2^n$, push $\langle 1 \rangle$ to the `altstack` and proceed.

We then repeat this process for each limb $(x_0, x_1, \ldots, x_{\ell-1})_\beta$. This way, we have a script `OP_TOBEBITS_TOALTSTACK` which takes an N-bit integer in the main stack and pushes all bits to the `altstack` in the big endian format.

Having this expansion, we can easily convert it to the w-width form using Algorithm 5. The idea is similar to one used in Algorithm 1 from Section 2.1.2: we split the binary expansion to the chunks of size w (except for, maybe, the last chunk, which might have a size less than w), suppose that the chunk is $\{c_j\}_{j=0}^{m-1}$, then the corresponding limb in w-width representation is $\sum_{j=0}^{m-1} c_j 2^j$. Then, having all limbs in the main stack, we can easily, if needed (which is the case), push it to the `altstack`.

All things considered, to get the w-width format, we simply call `OP_TOBEBITS_TOALTSTACK` and Algorithm 5 sequentially, and push resultant limbs to the `altstack`.

3.2 Addition and Doubling

To implement multiplication, we need to implement two additional "opcodes": `OP_ADD`, which takes two N-bit integers and adds them up, and `OP_2MUL`, which takes N-bit integer

Algorithm 5: Decomposing a limb to the w-width form

Input : Binary decomposition of a given limb x in the `altstack`
Output : w-width decomposition $(x_0, x_1, \ldots, x_{L-1})_w$ in the main stack

Prepare chunk sizes $\{c_j\}_{j=0}^{L-1}$ where the last chunk is of size $c_{L-1} := n - (L-1)w$, while others are of size w.
for $i \in \{0, \ldots, L-1\}$ **do**
  for $j \in \{0, \ldots, c_i - 1\}$ **do**
    $\{$ `OP_FROMALTSTACK` $\}$
    $\{$ `OP_IF` $\langle 1 \ll j \rangle$ `OP_ELSE` $\langle 0 \rangle$ `OP_ENDIF` $\}$
  end
  for $\_ \in \{0, \ldots, c_i - 2\}$ **do**
    $\{$ `OP_ADD` $\}$
  end
end

and doubles it. In both cases, we assume no overflow occurs (which will be the case for our multiplication algorithm), meaning that the result is still an N-bit integer.

Addition. Let us start with addition. We will do addition limb-wise with handling the carry bit. For that reason, we need an intermediate opcode `OP_LIMB_ADD_CARRY`, which takes $\{ \langle a \rangle \langle b \rangle \langle \beta \rangle \}$ – two limbs a, b and base β, and outputs $\{ \langle \beta \rangle \langle c \rangle \langle s \rangle \}$, where c is the carry bit, while s is the sum ($a + b$ if $c = 0$ and $(a + b) - \beta$ if $c = 1$). We specify the algorithm in Algorithm 6.

Algorithm 6: Adding two limbs with carry bit

Input : $\{ \langle a \rangle \langle b \rangle \langle \beta \rangle \}$ – two limbs a, b and base β
Output : $\{ \langle \beta \rangle \langle c \rangle \langle s \rangle \}$, where c is the carry bit, while s is the sum ($a + b$ if $c = 0$ and $(a + b) - \beta$ if $c = 1$)

$\{$ `OP_ROT OP_ROT` $\}$
$\{$ `OP_ADD OP_2DUP` $\}$
$\{$ `OP_LESSTHANOREQUAL` $\}$
$\{$ `OP_TUCK` $\}$
$\{$ `OP_IF` $\}$
  $\{ \langle 2 \rangle$ `OP_PICK OP_SUB` $\}$
$\{$ `OP_ENDIF` $\}$

Now we are ready to add two integers: see Algorithm 7. Note that we use the helper opcode `OP_ZIP`, which converts the stack

Stack: $\langle x_{\ell-1} \rangle \langle x_{\ell-2} \rangle \ldots \langle x_1 \rangle \langle x_0 \rangle \langle y_{\ell-1} \rangle \langle y_{\ell-2} \rangle \ldots \langle y_1 \rangle \langle y_0 \rangle$

to the following stack:

Stack: $\langle x_{\ell-1} \rangle \langle y_{\ell-1} \rangle \langle x_{\ell-2} \rangle \langle y_{\ell-2} \rangle \ldots \langle x_1 \rangle \langle y_1 \rangle \langle x_0 \rangle \langle y_0 \rangle$

which makes it easy to perform subsequent element-wise operations. We do not concretize its implementation, but it is quite straightforward. Also, since we rely on the fact that $x + y$ is still an N-bit integer (which, of course, is not always the case), when processing the last two limbs $\{ \langle x_{\ell-1} \rangle \langle y_{\ell-1} \rangle \langle c \rangle \}$ with a carry bit c, we do not need to handle the case when $x_{\ell-1} + y_{\ell-1} + c \geq \beta$.

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 { ⟨x_{ℓ-1}⟩ ... ⟨x_0⟩ ⟨y_{ℓ-1}⟩ ... ⟨y_0⟩ } to the form
   { ⟨x_{ℓ-1}⟩ ⟨y_{ℓ-1}⟩ ... ⟨x_0⟩ ⟨y_0⟩ } */
{ ⟨β⟩ } ; /* Push base to the stack */
{ OP_LIMB_ADD_CARRY OP_TOALTSTACK }
for _ ∈ {0, ..., ℓ − 3} do
    /* At this point, stack looks as { ⟨x_n⟩ ⟨y_n⟩ ⟨β⟩ ⟨c⟩ } . 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 { ⟨x_n⟩ ⟨y_n⟩ ⟨β⟩ ⟨c⟩ } . 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 _ ∈ {0, ..., ℓ − 2} do
    { OP_FROMALTSTACK }
end
```

Doubling. The doubling is performed similarly to addition, but we can avoid making the `OP_ZIP` operation and simply duplicate the last limb in the stack at each step. In this particular case, we need an additional opcode `OP_LIMB_DOUBLING_STEP`, which takes $\{\langle x\rangle \langle\beta\rangle \langle c\rangle\}$ – limb, base, and carry bit, and outputs $\{\langle\beta\rangle \langle c'\rangle \langle d\rangle\}$ – base, new carry bit c', and $d = 2x + c$. The implementation is specified in Algorithm 8. Additionally, we need the same version, but without c, which is executed at the beginning of the doubling, which we call `OP_LIMB_DOUBLING_INITIAL`. The corresponding implementation is specified in Algorithm 9.

Algorithm 8: Doubling the limb with carry bit

Input : $\{\langle x\rangle \langle\beta\rangle \langle c\rangle\}$ – limb, base, and carry bit
Output: $\{\langle\beta\rangle \langle c'\rangle \langle d\rangle\}$ – base, new carry bit c', and $d = 2x + c$

```
{ OP_ROT }
{ OP_DUP OP_ADD } ; /* Multiplying a 32-bit integer by 2 */
{ OP_ADD }
{ OP_2DUP }
{ OP_LESSTHANOREQUAL }
{ OP_TUCK }
{ OP_IF }
    { ⟨2⟩ OP_PICK OP_SUB }
{ OP_ENDIF }
```

Now, all we are left to do is performing the algorithm similar to Algorithm 7, but with small optimizations, accounting for the fact that we do not need `OP_ZIP`. The implementation is specified in Algorithm 10.

Algorithm 9: Doubling the limb without the carry bit

Input : $\{\ \langle x\rangle\ \langle\beta\rangle\ \}$ – limb and base
Output : $\{\ \langle\beta\rangle\ \langle c\rangle\ \langle d\rangle\ \}$ – base, new carry bit c, and limb doubled

```
1 { OP_SWAP }
2 { OP_DUP OP_ADD } ;                          /* Multiplying a 32-bit integer by 2 */
3 { OP_2DUP }
4 { OP_LESSTHANOREQUAL }
5 { OP_TUCK }
6 { OP_IF }
7     { ⟨2⟩ OP_PICK OP_SUB }
8 { OP_ENDIF }
```

Algorithm 10: Doubling the integer without overflowing

Input : $\{\ \langle x_{\ell-1}\rangle\ \langle x_{\ell-2}\rangle\ \ldots\ \langle x_1\rangle\ \langle x_0\rangle\ \}$ – N-bit integer to be doubled
Output : $\{\ \langle z_{\ell-1}\rangle\ \langle z_{\ell-2}\rangle\ \ldots\ \langle z_1\rangle\ \langle z_0\rangle\ \}$ – integer doubled $z = 2x$

```
1 { ⟨β⟩ } ;                                                  /* Base β = 2^n */
  /* Double the limb, take the result to the altstack, and add initial carry */
2 { OP_LIMB_DOUBLING_INITIAL OP_TOALTSTACK }
3 for _ ∈ {0, ..., ℓ − 3} do
  |  /* Since we have { ⟨x⟩ ⟨β⟩ ⟨c⟩ } in the stack, we need to double the limb x and
  |     add an old carry c to it.                                                  */
4 |  { OP_LIMB_DOUBLING_STEP OP_TOALTSTACK }
5 end
  /* At the end, we again get { ⟨x⟩ ⟨β⟩ ⟨c⟩ } where x is a limb in the stack. We drop
     the base and add the carry to the limb and double it without caring about
     overflowing.                                                                 */
6 { OP_NIP OP_SWAP }
7 { OP_DUP OP_ADD } ;                          /* Multiplying a 32-bit integer by 2 */
8 { OP_ADD }
  /* Take all limbs from the altstack to the main stack                           */
9 for _ ∈ {0, ..., ℓ − 2} do
10 |  { OP_FROMALTSTACK }
11 end
```

3.3 Binary Multiplication

Now comes the most interesting part: we will use methods from elliptic curve scalar multiplication to implement the product of two integers. Indeed: in Algorithm 2 and Algorithm 3 we might easily change $E(\mathbb{F}_q)$ to any other set, equipped with the addition operation (for example, any abelian group). In our particular case, when implementing $x \times y$, we will interpret the y as a scalar, while x as an element to be added/doubled. So let us implement the Algorithm 2 in *Bitcoin Script* first. Note the following: although our initial number is N-bit, we expect the product $x \times y$ to be $2N$-bit, so in the intermediate steps, when performing additions and doublings, we should account for the fact that they can easily overflow N bits. Straightforward workaround is to simply perform operations over the extended big integer of size $2N$. This is of course not the best approach and we will revisit it in Section 3.5 later on.

Since currently we have multiple various integers to work with, we will use notation `BigInt<`N`>::{OPCODE}` to denote calling the `OPCODE` of an N-bit big integer. So, calling `BigInt<`$2N$`>::{OPCODE}` would call the `OPCODE` of a $2N$-bit integer. Additionally, assume `OP_PICK`, `OP_ROLL` and `OP_DROP` are implemented for integers of arbitrary bitlength. These methods are relatively trivial compared to `OP_ADD` and `OP_2MUL`, considered before: all one needs to do is to operate with integers "limbwise".

So the implementation of Algorithm 2 in *Bitcoin Script* is specified in Algorithm 11. As can be seen, the cost (in opcodes) of conducting the double-and-add algorithm is $N\mathsf{A}+(N-1)\mathsf{D}$. Note that when analysing the cost in Section 2.2, we specified the *maximal* number of additions which get performed, but here the situation is different: the number of additions is exactly N, despite the fact that the `OP_IF` branch might be executed only a few times.

This is the primary reason why NAF methods did not significantly boost our performance: although additions might be called fewer times, we still need to include the logic in the script for each loop iteration. Therefore, we are interested in reducing the number of places where we need to place addition operations, not the number of times they get executed.

3.4 Windowed Multiplication

Now, let us implement the windowed method from Algorithm 3. Again, similarly to how it was done in Section 3.3, we conduct the following steps:

1. Decompose y to the width-w form using opcode from Algorithm 5.
2. Push the resultant decomposition to the `altstack`. Call first and second steps as `T::OP_TOBEWINDOWEDFORM_TOALTSTACK`.
3. Extend x to be $2N$-bit by appending zero limbs.
4. Precompute lookup table $\{0, x, 2x, 3x, \ldots, (2^w-1)x\}$.
5. Conduct the rest as described in Algorithm 3, assuming that additions and doublings never overflow (all intermediate are less than xy, which is a $2N$-bit number at worst).

Steps 1-3 were already covered in our discussion, so let us discuss our strategy to implementing the lookup table. It looks as follows:

1. Push 0 and x to the stack.
2. On each step if we need to calculate $2n \times x$, simply `BigInt<`$2N$`>::OP_PICK` the element $n \times x$ and double it using $\{$ `BigInt<`$2N$`>::OP_DUP BigInt<`$2N$`>::OP_ADD` $\}$.

Algorithm 11: Double-and-add integer multiplication

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

Output : $2N$-bit integer $z = x \times y$ on the stack: $\{\langle z_{\ell'-1}\rangle \ldots \langle z_1\rangle\ \langle z_0\rangle\}$

```
1  { BigInt<N>::OP_TOBEBITS_TOALTSTACK }
2  { BigInt<N>::OP_EXTEND::<BigInt<2N>> } ;        /* Extend N-bit integer to
   2N-bit integer by appending ℓ' − ℓ zero limbs */
3  { BigInt<2N>::OP_0 } ;                  /* Pushing 2N-bit zero to the stack */
4  { OP_FROMALTSTACK }
5  { OP_IF }
6     { ⟨1⟩ BigInt<2N>::OP_PICK }
7     { BigInt<2N>::OP_ADD }
8  { OP_ENDIF }
9  for _ ∈ {1, ..., N − 2} do
10    { ⟨1⟩ BigInt<2N>::OP_ROLL }
11    { BigInt<2N>::OP_2MUL }
12    { ⟨1⟩ BigInt<2N>::OP_ROLL }
13    { OP_FROMALTSTACK }
14    { OP_IF }
15       { ⟨1⟩ BigInt<2N>::OP_PICK }
16       { BigInt<2N>::OP_ADD }
17    { OP_ENDIF }
18 end
19 { ⟨1⟩ BigInt<2N>::OP_ROLL }
20 { BigInt<2N>::OP_2MUL }
21 { OP_FROMALTSTACK }
22 { OP_IF }
23    { BigInt<2N>::OP_ADD }
24 { OP_ELSE }
25    { BigInt<2N>::OP_DROP }
26 { OP_ENDIF }
```

3. If, instead, we need to calculate $(2n+1) \times x$, copy the last element in the stack via `BigInt<`$2N$`>::OP_DUP` (which is $2n \times x$), then copy x and add them together via `OP_ADD`.

The aforementioned strategy, as discussed before, costs $(2^{w-1}-1)$A and $(2^{w-1}-1)$D, which reduces to 7A and 7D for $w = 4$. Let us further encapsulate the logic of pushing $\{0x, 1x, \ldots, (2^w-1)x\}$ to the stack as `BigInt<`$2N$`>::OP_INITWINDOWEDTABLE`(w).

Now we are ready to define the algorithm itself: see Algorithm 12.

Algorithm 12: Windowed integer multiplication

Input : Parameter w; two N-bit integers on the stack: $\{ \langle x_{\ell-1} \rangle \ldots \langle x_0 \rangle \ \langle y_{\ell-1} \rangle \ldots \langle y_0 \rangle \}$

Output : $2N$-bit integer $z = x \times y$ on the stack: $\{ \langle z_{\ell'-1} \rangle \ldots \langle z_1 \rangle \ \langle z_0 \rangle \}$

1 { `BigInt<`N`>::OP_TOBEWINDOWEDFORM_TOALTSTACK` }
2 { `BigInt<`N`>::OP_EXTEND::<BigInt<`$2N$`>>` } ; /* Extend N-bit integer to $2N$-bit integer by appending $\ell' - \ell$ zero limbs */
3 { `BigInt<`$2N$`>::OP_INITWINDOWEDTABLE`(w) } ; /* Precomputing $\{0, x, \ldots, ((1 \ll w) - 1)x\}$ */
4 { `OP_FROMALTSTACK` $\langle 1 \rangle$ `OP_ADD` } ; /* Picking first limb from the altstack +1 */
5 { $\langle 1 \ll w \rangle$ `OP_SWAP OP_SUB BigInt<`$2N$`>::OP_PICKSTACK` } ; /* Picking the corresponding value from the precomputed table */
6 **for** $\_ \in \{1, \ldots, L-1\}$ **do**
  /* Double the result w times */
7 **for** $\_ \in \{0, \ldots, w-1\}$ **do**
8 { `BigInt<`$2N$`>::OP_2MUL` }
9 **end**
  /* Picking limb from the altstack and picking the corresponding element from the lookup table. After picking an element, the stack would look like $\{ \langle 0 \rangle \ \langle x \rangle \ \ldots \ \langle ((1 \ll w) - 1)x \rangle \ \langle r \rangle \ \langle y_i \rangle \}$, where r is the temporary variable, being the final result, and y_i is the limb at step i */
10 { $\langle 1 \ll w \rangle$ `OP_SWAP OP_SUB` }
11 { `BigInt<`$2N$`>::OP_PICKSTACK BigInt<`$2N$`>::OP_ADD` }
12 **end**
  /* Clearing the precomputed values from the stack. */
13 { `BigInt<`$2N$`>::OP_TOALTSTACK` }
14 **for** $\_ \in \{0, \ldots, ((1 \ll w) - 1)\}$ **do**
15 { `BigInt<`$2N$`>::OP_DROP` }
16 **end**
17 { `BigInt<`$2N$`>::OP_FROMALTSTACK` }

3.5 Gradual Bitsize Increase

Finally, notice that extending an integer from N bits to $2N$ bits from the very beginning is not optimal. For example, consider the first iteration of a loop in the windowed integer multiplication, where we multiply by 2^w and then add the precomputed value. Notice that if we begin from the 256-bit number, for instance, multiplying by 16 and adding the 256-bit number would result in the 261-bit number maximum (in fact, 260-bit number as we will see later). Similarly, when conducting the next iteration, we would not exceed 264 bits and so on. This motivates us to handle the size dynamically: when ℓ limbs are not sufficient to conduct the operations without overflowing, we would push the zero limb (to

extend an integer to $\ell + 1$ limbs) and conduct the rest as usual. This would save tons of opcodes as the number of useless additions of zero limbs is considerable.

Now, let us consider the following theorem.

Theorem 2. *Suppose that Algorithm 12 is conducted using two N-bit integers, the window size of w with $L = \lceil N/w \rceil$ limbs. For each k^{th} step, it suffices to extend the temporary variable q to $\lambda + kw$ bits, resulting in $\lceil (\lambda + kw)/n \rceil$ limbs for $\lambda = 2N - w(L-1)$.*

Proof. Let us examine the first step. We decompose y to the width-w form, resulting in $y = \sum_{i=0}^{L-1} y_i 2^{wi}$, where each $0 \leq y_i < 2^w$. Next, we initialize the lookup table which involves calculating $\{0, x, 2x, \ldots, (2^w - 1)x\}$. Finally, we initialize the temporary variable $q \leftarrow 0$ and set it to the value $y_{L-1}x$ (since multiplication by 2^w would leave $q = 0$ unchanged).

Now, x is N bits in size. An interesting question is the size of y_{L-1} in bits. Recall that $y = y_{L-1}2^{w(L-1)} + y_{L-2}2^{w(L-2)} + \cdots + y_0$ is an N-bit number which means that $y_{L-1}2^{w(L-1)}$ should also be N bits. If the size of y_{L-1} in bits is λ, then the size of $y_{L-1}2^{w(L-1)}$ is $\lambda + w(L-1)$ which is N maximum. Meaning, $\lambda \leq N - w(L-1) = (N + w) - wL$.

All in all, we conclude that the size of q in the beginning (call it λ) is $2N - w(L-1)$. Then, suppose that we are at step k with a value q_k. In this case,

$$q_{k+1} = 2^w q_k + y_{L-k}x, \; q_0 = y_{L-1}x \tag{7}$$

This is a recurrence relation which is quite tough to solve generically as y_{L-k} term is different for each step. For that reason, assume the worst case: suppose $y_{L-k} = 2^w - 1$ for each $k > 1$ and consider the recurrence relation

$$Q_{k+1} = 2^w Q_k + (2^w - 1)x, \; Q_0 = q_0 = y_{L-1}x \tag{8}$$

In this case, $q_k < Q_k$ for each $k > 1$, so Q_k is our upper bound. Now, Equation (8) is an equation of form $z_{k+1} = \alpha z_k + \beta$, which has a closed solution $z_n = \alpha^n z_0 + \frac{\alpha^n - 1}{\alpha - 1}\beta$, so we get

$$Q_k = 2^{wk} Q_0 + (2^{wk} - 1)x \tag{9}$$

Notice that $2^{wk}Q_0$ has a bitsize of $wk + \lambda$, while $(2^w - 1)x$ is $N + w$ bits in size. Notice this addition always result in the integer of bitsize $wk + \lambda$. Indeed:

$$Q_k < 2^{wk}(2^\lambda - 1) + (2^{wk} - 1)(2^N - 1) < 2^{wk+\lambda} + 2^{wk+N} < 2^{wk+\lambda+1}, \tag{10}$$

so Q_k fits in $\lambda + wk$ bits. Thus, as $q_k < Q_k$, q_k also fits in $\lambda + wk$ bits, concluding the proof. □

With Theorem 2 in hand, we are ready to optimize the Algorithm 12 by introducing Algorithm 13.

3.6 Mulmod Optimization

In previous sections, we discussed optimizing multiplication. Now, we shift our focus to modular multiplication, which is crucial for performing arithmetic in finite fields on Bitcoin. Specifically, we aim to carry out operations in the finite field defined by the BN254 curve, a pairing-friendly curve used in cryptographic protocols like SNARKs.

Optimizing modular multiplication aims to efficiently compute products within a finite field. For prime field $\mathbb{F}_q$, instead of simply calculating $z = x \times y$, what we need to compute is $z = x \times y \pmod{q}$, where q is the prime field of the BN254 curve.

We can perform modular multiplication through the division-remainder decomposition:

$$\begin{aligned} x \times y &= q \times t + z \\ z &= (x \times y) - (q \times t) \end{aligned} \tag{11}$$

Algorithm 13: Windowed integer multiplication with gradual bitsize increase

Input : Parameter w; two N-bit integers on the stack: $\{ \langle x_{\ell-1} \rangle \ \ldots \ \langle x_0 \rangle \ \langle y_{\ell-1} \rangle \ \ldots \ \langle y_0 \rangle \}$

Output : $2N$-bit integer $z = x \times y$ on the stack: $\{ \langle z_{\ell'-1} \rangle \ \ldots \ \langle z_1 \rangle \ \langle z_0 \rangle \}$

1 { BigInt<N>::OP_TOBEWINDOWEDFORM_TOALTSTACK }

/* Important note: here we assume that all precomputed values still fit in ℓ limbs, so there is no need to extend an integer from N to λ bits. Yet, this can be easily accounted for if needed. */

2 { BigInt<N>::OP_INITWINDOWEDTABLE (w) } ; /* Precomputing $\{0, x, \ldots, ((1 \ll w) - 1)x\}$ */

3 { OP_FROMALTSTACK $\langle 1 \rangle$ OP_ADD } ; /* Picking first limb from the altstack $+1$ */

4 { $\langle 1 \ll w \rangle$ OP_SWAP OP_SUB BigInt<N>::OP_PICKSTACK } ; /* Picking the corresponding value from the precomputed table */

5 **for** $i \in \{1, \ldots, L-1\}$ **do**

  /* Extend the result from $\lambda + (i-1)w$ bits to $\lambda + iw$ */

6 { BigInt<$\lambda + (i-1)w$>::OP_EXTEND::<BigInt<$\lambda + iw$>> }

  /* Double the result w times */

7 **for** $\_ \in \{0, \ldots, w-1\}$ **do**

8 { BigInt<$\lambda + iw$>::OP_2MUL }

9 **end**

  /* Picking limb from the altstack and picking the corresponding element from the lookup table. After picking an element, the stack would look like $\{ \langle 0 \rangle \ \langle x \rangle \ \ldots \ \langle ((1 \ll w) - 1)x \rangle \ \langle r \rangle \ \langle y_i \rangle \}$, where r is the temporary variable, being the final result, and y_i is the limb at step i */

10 { $\langle 1 \ll w \rangle$ OP_SWAP OP_SUB }

11 { BigInt<$\lambda + iw$>::OP_PICKSTACK }

12 { BigInt<λ>::OP_ADD } ; /* Since we need to only care about last limbs, we do not extend the result */

13 **end**

/* Clearing the precomputed values from the stack. */

14 { BigInt<$2N$>::OP_TOALTSTACK }

15 **for** $\_ \in \{0, \ldots, ((1 \ll w) - 1)\}$ **do**

16 { BigInt<λ>::OP_DROP }

17 **end**

18 { BigInt<$2N$>::OP_FROMALTSTACK }

where t is the quotient. Here q is the characteristic of the pairing curve used for SNARK verification and t is passed as an auxiliary parameter to the script. This decomposition shows that we can achieve modular multiplication with two executions of the multiplication script. Thus, the number of opcodes required for modular multiplication is roughly twice that of simple multiplication. However, we can further optimize this process.

See the implementation in Algorithm 14. Again, similarly to Section 2.2, the algorithm is specified for calculating the equivalent expression $[y]X \ominus [q]T$ on an elliptic curve, but we can easily convert the algorithm to work with integers.

The modular multiplication algorithm specified below shares similarities with Algorithm 3 for multiplication, particularly in the number of addition operations required. However, it offers savings on doubling costs. Instead of performing $2N$ doubling for two separate multiplications, this approach allows us to compute modular multiplication with just N doublings. The trade-off is an increased precomputation cost, as we need to maintain two lookup tables. That being said, the overall cost is:

$$2 \times \left[2^{w-1}\mathsf{A} + 2^{w-1}\mathsf{D}\right] + \left[\frac{2N}{w}\mathsf{A} + N\mathsf{D}\right] \tag{12}$$

Throughout this paper, we have primarily discussed addition operations within the double-and-add algorithm for scalar multiplication. It is important to note that, in our implementation, finite field elements are represented in the form of two's complement. This allows us to handle signed integers efficiently. Consequently, addition operations implicitly cover the subtraction of signed integers as well.

Algorithm 14: w-width windowed method for modular multiplication

Input : $X, T \in E(\mathbb{F}_q)$ and $y, q \in \mathbb{Z}_r$
Output : Result of $[y]X \ominus [q]T \in E(\mathbb{F}_q)$

Decompose y to the w-width form: $(y_0, y_1, \ldots, y_{L-1})_w$
Decompose q to the w-width form: $(q_0, q_1, \ldots, q_{L-1})_w$
Precompute values for two lookup tables:

$$\{[0]X, [1]X, [2]X, \ldots, [2^w - 1]X\},$$
$$\{[0]T, [1]T, [2]T, \ldots, [2^w - 1]T\}.$$

Denote by $\mathcal{T}_X[j] = [j]X$ – referencing the lookup table for X at index j.
Denote by $\mathcal{T}_T[j] = [j]T$ – referencing the lookup table for T at index j.
$Z \leftarrow \mathcal{O}$
for $i \in \{L-1, \ldots, 0\}$ **do**
  for $\_ \in \{1, \ldots, w\}$ **do**
    $Z \leftarrow [2]Z$
  end
  $Z \leftarrow Z \oplus (\mathcal{T}_X[y_i] \ominus \mathcal{T}_T[q_i])$
end
Return : Point $Z \in E(\mathbb{F}_q)$

4 Discussion

4.1 Window Width Choice

One of our key claims is that the width parameter $w = 4$ gives the best performance. In this section, we justify this claim. For that reason, we provide the following theorem.

Theorem 3. *Suppose that Algorithm 12 is performed over two N-bit integers, and the cost of the addition of $2N$-bit integers is $C_{\mathsf{A}} \in \mathbb{N}$ and the cost of doubling is $C_{\mathsf{D}} \in \mathbb{N}$. Then, the optimal width parameter w is approximately $\hat{w} \in \mathbb{R}$, where $\hat{w}$ satisfies:*

$$\hat{w}^2 2^{\hat{w}} = \frac{2N}{\log 2} \cdot \frac{C_{\mathsf{A}}}{C_{\mathsf{A}} + C_{\mathsf{D}}} \tag{13}$$

In particular, if $C_{\mathsf{A}} \approx C_{\mathsf{D}}$, then this reduces to $\hat{w}^2 2^{\hat{w}} = N/\log 2$.

Remark 1. To simplify the analysis, we consider the Algorithm 12, which operates over extended integers. The analysis for optimized version Algorithm 13 would be ideologically similar but quite cumbersome, so let us stick to the simpler version.

Proof. The total cost C of width-w multiplication is, as mentioned in Section 2.2 is approximately (without accounting for operations not depending on the chosen w) given by the following formula:

$$C(w) = 2^{w-1}(C_{\mathsf{A}} + C_{\mathsf{D}}) + \frac{NC_{\mathsf{A}}}{w} + NC_{\mathsf{D}} \tag{14}$$

Therefore, it suffices to apply a simple calculus to find the optimal value of w. If $\hat{w} \in \mathbb{R}$ is the optimal width, it should satisfy $C'(\hat{w}) = 0$ which gives us:

$$C'(w) = (C_{\mathsf{A}} + C_{\mathsf{D}})2^{w-1}\log 2 - \frac{NC_{\mathsf{A}}}{w^2} \implies \hat{w}^2 2^{\hat{w}} = \frac{2N}{\log 2} \cdot \frac{C_{\mathsf{A}}}{C_{\mathsf{A}} + C_{\mathsf{D}}} \tag{15}$$

To see why this gives a minimum, compute the second derivative:

$$C''(w) = (C_{\mathsf{A}} + C_{\mathsf{D}})2^{w-1}\log^2 2 + \frac{2NC_{\mathsf{A}}}{w^3}, \tag{16}$$

which is positive for any $w > 0$ (which is the case). The relation $\hat{w}^2 2^{\hat{w}} = N/\log 2$ follows immediately after substituting $C_{\mathsf{A}} = C_{\mathsf{D}}$. □

So, now, let us substitute values corresponding to our implementation. We use $N = 254$, and the cost of the addition is 363 bytes (so we set $C_{\mathsf{A}} := 363$), while doubling takes 245 bytes (thus we set $C_{\mathsf{D}} := 245$)[5]. Thus, approximately, $\hat{w}^2 2^{\hat{w}} \approx 437.5$, yielding $\hat{w} \approx 4.45$. After checking both $w = 4$ and $w = 5$, we conclude that $w = 4$ is the optimal choice.

Out of curiosity, we plot the dependence $C(w)$ for different N's and w's. The result is depicted in Figure 1. Interestingly, for larger integers (in particular, for $N = 512$ or $N = 1024$), $w = 4$ most likely would no longer be the optimal choice.

4.2 Performance Comparison

Now, we compare our implementation with the state-of-the-art approaches currently used. To our knowledge, implementations include:

1. *BitVM "Overflow" Multiplication*[6]: BitVM provides the default library to operate with big integers (thereafter, called `bigint`) that implements the `mul` operation. The catch is that, based on two N-bit integers, this function also returns a N-bit integer, reduced modulo 2^N (essentially, the lower limb in $2N$-bit integer representation $c_0 + c_1 \times 2^N$) — we call this "overflow multiplication". Therefore, for comparison, we adapted algorithm Algorithm 12 to have the same functionality, and also tweaked the *BitVM*'s implementation to give the $2N$-bit integer as the result.

[5] In fact, it does not matter which units to use to represent C_{A} and C_{D} since at the end of the day, all that matters is the fraction $\frac{C_{\mathsf{A}}}{C_{\mathsf{A}}+C_{\mathsf{D}}}$, which depends solely on ratio $C_{\mathsf{D}}/C_{\mathsf{A}}$.

[6] https://github.com/BitVM/BitVM, Accessed: 25 July 2024

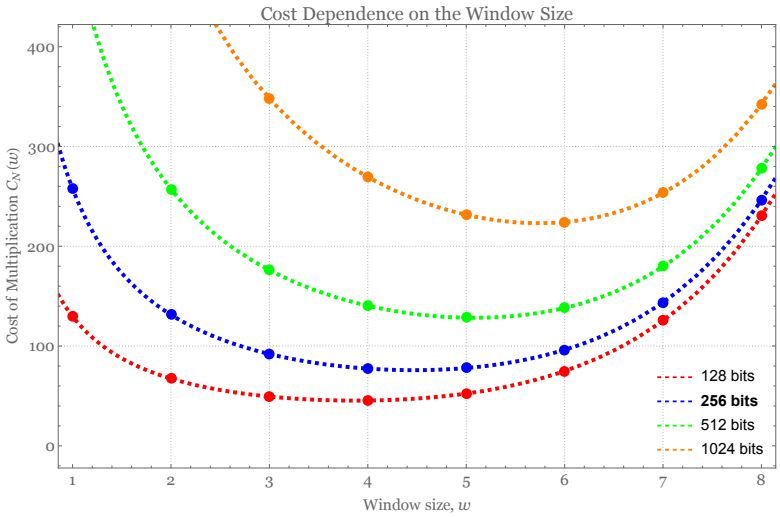

Figure 1: Dependence of multiplication cost $C_N(w)$ on the window size w for various integer bit-sizes (N). We plotted the dependence for four integers: **128 bits**, **256 bits**, **512 bits**, **1024 bits**. The dashed line in **blue** is most closely related to our case ($N = 254$). Here, we assumed that $C_\mathsf{D}/C_\mathsf{A} \approx 0.675$, corresponding to our multiplication.

Table 1: Comparison of our method with the current state-of-the-art. N/A means "non-applicable": that is, the algorithm is not adapted to the corresponding type of task.

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

2. *Cmpeq's Implementation*[7]: quite recently, on Bitcoin Forum, *cmpeq* claimed to have roughly 100k opcodes in his multiplication of two 255-bit integers. The result is a 510-bit integer, compared to `bigint` multiplication from *BitVM*.

3. *BitVM* 29×9 *Karatsuba Multiplication*: This is the most recent version that BitVM mostly relies on that uses the Karatsuba multiplication (see Section 2.2.1) with $(n = 29, \ell = 9)$ to represent a 254-bit integer.

The comparison results are depicted in Table 1.
Some comments about the achieved results:

1. Although Cmpeq claimed to have roughly 100k opcodes, after uploading the script, it appears that the real number of opcodes is, in fact, 200k. This probably happens because pushing a single integer to the stack does not always cost one opcode. For example, pushing 10^3 costs 3 opcodes while 10^5 costs 4.

2. Our algorithm outperforms current multiplication methods for both overflowing and widening versions.

3. As mentioned before, original BitVM's `bigint` does not support the widening version, so we extended the used approach (namely, double-and-add method from Algorithm 2) to handle $2N$-bit result.

[7] https://bitcointalk.org/index.php?topic=5477449.0, Accessed: 25 July 2024

4.3 Future Directions

Most likely, our current version is not best-optimized. In particular, we list what can help to possibly reduce the number of opcodes even further:

1. Small polishes in gadgets used underneath (extending big integers to handle larger limbs, more effective addition or doubling, etc.).

2. We have not achieved any boost using NAF methods, but that does not mean these methods are not applicable: it is curious whether something can be achieved with them. In particular, w-NAF form might possibly decrease the number of additions from $\frac{N}{w}$ to $\frac{N}{w+1}$ and the cost of precomputing values. On the other hand, this would require implementing subtraction and sign handling, which might be troublesome.

3. Using different bases: we achieved the best results using 30-bit limbs to represent an integer, but maybe smaller limbs might result in something more effective.

References

[1] Andreas M. Antonopoulos. *Mastering Bitcoin: Unlocking Digital Crypto-Currencies*. O'Reilly Media, Inc., 1st edition, 2014.

[2] Augusto Devegili, Michael Scott, and Ricardo Dahab. Implementing cryptographic pairings over barreto-naehrig curves. volume 4575, pages 197–207, 07 2007.

[3] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[4] Predrag V. Krtolica and Predrag S. Stanimirović. Reverse polish notation method. *International Journal of Computer Mathematics*, 81(3):273–284, 2004.

[5] Mahender Kumar and Satish Chand. Pairing-friendly elliptic curves: Revisited taxonomy, attacks and security concern, 2022.

[6] Robin Linus. Bitvm: Compute anything on bitcoin. 2023.

[7] Andre Weimerskirch and Christof Paar. Generalizations of the karatsuba algorithm for efficient implementations. *IACR Cryptology ePrint Archive*, 2006:224, 01 2006.

[8] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.