OPTIMIZING BIG INTEGER MULTIPLICATION ON BITCOIN: INTRODUCING w-WINDOWED APPROACH

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

[1] Distributed Lab, Kyiv, Ukraine
[2] Alpen Labs, New York, USA

ABSTRACT

A crucial component of any zero-knowledge system is operations with finite fields. This, in turn, leads to the implementation of the fundamental operation: multiplying two big integers. In the realm of Bitcoin, this problem gets revisited, as Bitcoin utilizes its own stack-based and Turing-complete scripting system called Bitcoin Script. Inspired by Elliptic Curve scalar multiplication, this paper introduces the w-windowed method for multiplying two numbers. We outperform state-of-the-art approaches, including BitVM's implementation. Finally, we also show how the windowed method can lead to optimizations not only in big integer arithmetic but in more general arithmetic problems.

KEYWORDS

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

1. INTRODUCTION

Introduced in 2009, Bitcoin has drastically changed the world of digital finance and led to the broad adoption of blockchain technology [1]. Being the first cryptocurrency, it put into action numerous novel concepts such as decentralization, digital security, and programmable conditions for operating with digital currency [2],[3]. However, by design, Bitcoin Smart Contract functionality is very limited. Essentially, one can only perform verifications on basic primitives such as ECDSA/Schnorr signatures, hashlocks, or timelocks. Despite such a limited set of tools, the Bitcoin community managed to come up with a multitude of exciting and complex protocols such as Atomic Swaps [4],[5], Anonymized Taprootized Swaps [6], Lightning Network [7],[8], RGB protocol [9], LRC-20 [10] etc.

Despite all the community's efforts, one of the most anticipated technologies yet to be fully developed is the *L2 zero-knowledge (zk) rollup* on top of Bitcoin. Currently, the adoption of *L2 zk-rollups* allows achieving much better scalability on Ethereum [11], resulting in lower fees and a higher number of transactions per second while maintaining the same security as in the *L1 layer* (that is, Ethereum blockchain itself). This is achieved through zero-knowledge technology, which allows for the formation of succinct validity proof for verification. One of the most widely used *L2 zk-rollups* are Aztec, Scroll, Polygon zkEVM, zkSync, Starknet, etc[12]. The majority of them (besides Starknet), one way or another, rely on the Succinct Non-Interactive Argument of Knowledge (SNARK), allowing building proofs of certain statements with the size $O(\log|C|)$

David C. Wyld et al. (Eds): SESBC, AIFL, NLPTT – 2025
pp. 45-63, 2025. CS & IT - CSCP 2025 DOI: 10.5121/csit.2025.151503

where $|C|$ is the number of gates in the arithmetic circuit C, describing arbitrary logic that we want to prove and verify [13].

While there are many endeavors to achieve a similar SNARK-based zk-rollup on Bitcoin, currently, to the best of our knowledge, there is yet to be a production-ready system on top of Bitcoin Mainnet. For the most part, as we mentioned, the primary reason is the limitation of Bitcoin Script. In spite of all the limitations, there is significant progress in writing the full zero-knowledge SNARK verifier in Bitcoin Script. One of notable examples include BitVM[14] and Alpen Labs with SNARKnado[15], but current implementations still require more optimizations of the underlying primitives.

1.1. Our Contribution

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. 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.

Inspired by Elliptic Curve optimization tricks, this paper introduces the w-windowed method for multiplying two 254-bit prime (BN254 curve [16]) integers, along with additional optimization techniques. Our approach improves upon the work done by the *BitVM* team, reducing script size for multiplication by roughly 3.2k opcodes and is integrated into the BitVM code base due to superior performance. In addition, our optimization can be used in any application involving big integer arithmetic, suchas Groth16 [17] or fflonk [18], which are both currently implemented in BitVM. Even more notably, our approach can lead to even more optimizations for more general tasks such as multiple integer multiplication or fixed integer multiplication, so we expect that the methods considered are not limited to the multiplication of two integers solely.

All the code with implementation is available through the following link:https://github.com/distributed-lab/bitcoin-window-mul

The paper is structured as follows: in 2 we will give a basic overview of Bitcoin Script and fast multiplication methods. In 3 we will list scripts to conduct the windowed multiplication (our primary proposed method). Finally, in 4 and 5, we will compare our performance with state-of-the-art and draw a conclusion.

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 [19]. Informally, this condition is called scriptPubKey, while the data that must be provided to meet this condition is called scriptSig. 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 $\{ \langle a \rangle\ \langle b \rangle$ OP_ADD $\langle c \rangle$ 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 $\{ \langle s \rangle\ \langle c \rangle$ 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 [20].

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:

OP_HASH160 $\langle h \rangle$ OP_EQUAL

Note, though, that in the placeholder $\langle h \rangle$we should push 0x20 followed by 20 bytes of h.Suppose we brought a message m, our scriptSig. Concatenating scriptSig and scriptPubKey would result in the following script:

$\langle m \rangle$ 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 $\{ \langle h' \rangle\ \langle h \rangle$ 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*[16]) to 381 bits and more (as for *BLS12-381*[21]). Currently, the common choice is the *BN254* based on 254-bit prime order q, which, for example, is currently used for elliptic curve precompiles in *Ethereum*[11]. 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. 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

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

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, Figure 1 shows how to preprocess the given integer x and push the representation 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 $n \leq 32$).
Decompose x to the binary form: $(x_0, x_1, \ldots, x_{N-1})_2$
Split the form into chunks of size n (the last chunk would be of size $N \bmod n$)
For k^{th} chunk with bits $(c_0, \ldots, c_{m-1})$ (where m is either n or $N \bmod n$) set $X_k \leftarrow \sum_{j=0}^{m-1} c_j 2^j$
Return : $(X_0, X_1, \ldots, X_{\ell-1})$

Figure 1. Pushing given integer to the stack.

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 [22]. 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:

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

Then, a simple multiplication formula gives us:

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

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

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

Karatsuba Multiplication is currently widely used in various applications [23, 24, 25, 26]such as implementation of Arithmetical Logic Unit (ALU), modulators or cryptosystemsdue to much better assymptotics. Due to such efficiency, 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 [14]However, as it turns out, there is a more efficient wayto implement the multiplication. Interestingly, we will consider the optimizations usedin the implementation of elliptic curve arithmetic and how they relate to regular integer arithmetic.

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 bemainly based on the explanations of [27]which are heavily employed in moderncryptography frameworks such as **gnark** [28].

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 Figure 2.

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

Figure 2. Double-and-add method for scalar multiplication

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. Remark, that$\mathsf{D}$ is slightly easier to perform than A since doubling is a special case of addition.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

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

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

Now, what does this form give us? Let us consider algorithm shownin Figure 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:

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

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.

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: $(k_0, k_1, \dots, k_{L-1})_w$
Precompute values $\{[0]P, [1]P, [2]P, \dots, [2^w - 1]P\}$ (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 $i \in \{L-1, \dots, 0\}$ **do**
  for $\_ \in \{1, \dots, w\}$ **do**
    $Q \leftarrow [2]Q$
  end
  $Q \leftarrow Q \oplus \mathcal{T}[k_i]$
end
Return : Q

Figure 3. w-width windowed method for scalar multiplication

Yet another effective approach is w-width non-adjacent form (NAF). Let us introduce it first.
Definition 3. Again, assume $w \ge 2$. A **width-w** NAF of $k \in \mathbb{Z}_{\ge 0}$ 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 $k \in \mathbb{Z}_{\ge 0}$. Then,

1. k has a unique width-w NAF, denoted by $(k_0, ..., k_{L-1})_{w,\mathsf{NAF}}$.
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 the 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 the algorithm shownin Figure 3, except for the fact that it suffices to precompute only odd products $\{[1]P, [3]P, ..., [2^{w-1}-1]P\}$ 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 Figure 4.

```
Input  : A single n-bit integer x (n ≤ 32)
Output : Bits (x_0, x_1, ..., x_{n-1}) in altstack
{ OP_TOALTSTACK } ;                          /* Moving limb to altstack */
for i ∈ {0, ..., n − 1} do
  | { ⟨2 ≪ i⟩ } ;                            /* Pushing powers of two */
end
{ OP_FROMALTSTACK } ;                        /* Getting element back */
for _ ∈ {0, ..., n − 1} do
  | { OP_2DUP OP_LESSTHANOREQUAL }
  | { OP_IF }
  |     { OP_SWAP OP_SUB ⟨1⟩ }
  | { OP_ELSE }
  |     { OP_NIP ⟨0⟩ }
  | { OP_ENDIF }
  | { OP_TOALTSTACK }
end
```

Figure 4. Decomposing a limb to the binary form

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

$$\langle 2^1 \rangle \ \langle 2^2 \rangle \ \langle 2^3 \rangle \dots \langle 2^n \rangle \ \langle x \rangle$$

Then, we duplicate top-stack elements to get $\{\dots \langle 2^n \rangle \ \langle x \rangle \ \langle 2^n \rangle \ \langle x \rangle\}$, 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 the algorithm shown in Figure 5. The idea is similar to one used in algorithm shown in Figure 1 from 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 algorithmin Figure 5sequentially, and push resultant limbs to the altstack.

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

Figure 5. Decomposing a limb to the w-width form

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 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 Figure 6.

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 }
    { ⟨2⟩ OP_PICK OP_SUB }
{ OP_ENDIF }
```

Figure 6. Adding two limbs with carry bit

Now we are ready to add two integers: seein Figure 7. Note that we use the helper opcode OP_ZIP, which converts the 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:

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

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, handling the case when $x_{\ell-1} + y_{\ell-1} + c \geq \beta$ is unnecessary.

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

```
{ 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
```

Figure 7. Adding two integers assuming with no overflow

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 }
```

Figure 8. Doubling the limb with carry bit

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 Figure 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 Figure 9.
Now, all we are left to do is perform the algorithm similar to algorithm in Figure 7, but with small optimizations, accounting for the fact that we do not need OP_ZIP. The implementation is specified in Figure 10.

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

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

Figure 9. Doubling the limb without the carry bit

```
Input  : { ⟨x_{ℓ−1}⟩ ⟨x_{ℓ−2}⟩ ... ⟨x_1⟩ ⟨x_0⟩ } – N-bit integer to be doubled
Output : { ⟨z_{ℓ−1}⟩ ⟨z_{ℓ−2}⟩ ... ⟨z_1⟩ ⟨z_0⟩ } – integer doubled z = 2x
{ ⟨β⟩ } ;                                                      /* Base β = 2^n */
/* Double the limb, take the result to the altstack, and add initial carry     */
{ OP_LIMB_DOUBLING_INITIAL OP_TOALTSTACK }
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.                                               */
    { OP_LIMB_DOUBLING_STEP OP_TOALTSTACK }
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.                                                                */
{ OP_NIP OP_SWAP }
{ OP_DUP OP_ADD } ;                           /* Multiplying a 32-bit integer by 2 */
{ OP_ADD }
/* Take all limbs from the altstack to the main stack                          */
for _ ∈ {0, ..., ℓ − 2} do
    { OP_FROMALTSTACK }
end
```

Figure 10. Doubling the integer without overflowing

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 in Figure 2 and algorithm in Figure 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. The straightforward workaround is simply performing operations over the extended big integer of size $2N$. This is, of course, not the best approach, and we will revisit it in 3.5later on.

```
Input  : Two N-bit integers on the stack: { ⟨x_{ℓ−1}⟩ ... ⟨x_0⟩ ⟨y_{ℓ−1}⟩ ... ⟨y_0⟩ }
Output : 2N-bit integer z = x × y on the stack: { ⟨z_{ℓ'−1}⟩ ... ⟨z_1⟩ ⟨z_0⟩ }
{ BigInt<N>::OP_TOBEBITS_TOALTSTACK }
{ BigInt<N>::OP_EXTEND::<BigInt<2N>> } ;        /* Extend N-bit integer to
  2N-bit integer by appending ℓ' − ℓ zero limbs */
{ BigInt<2N>::OP_0 } ;                          /* Pushing 2N-bit zero to the stack */
{ OP_FROMALTSTACK }
{ OP_IF }
   { ⟨1⟩ BigInt<2N>::OP_PICK }
   { BigInt<2N>::OP_ADD }
{ OP_ENDIF }
for _ ∈ {1, ..., N − 2} do
   { ⟨1⟩ BigInt<2N>::OP_ROLL }
   { BigInt<2N>::OP_2MUL }
   { ⟨1⟩ BigInt<2N>::OP_ROLL }
   { OP_FROMALTSTACK }
   { OP_IF }
      { ⟨1⟩ BigInt<2N>::OP_PICK }
      { BigInt<2N>::OP_ADD }
   { OP_ENDIF }
end
{ ⟨1⟩ BigInt<2N>::OP_ROLL }
{ BigInt<2N>::OP_2MUL }
{ OP_FROMALTSTACK }
{ OP_IF }
   { BigInt<2N>::OP_ADD }
{ OP_ELSE }
   { BigInt<2N>::OP_DROP }
{ OP_ENDIF }
```

Figure 11. Double-and-add integer multiplication

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 Figure 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 analyzing the cost in 2.2, we specified the *maximal* number of additions that 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 in Figure 3. Again, similarly to how it was done in 3.3, we conduct the following steps:

1. Decompose y to the width-w form using opcode from algorithm in Figure 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 for 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 } .
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 $(2^{w-1} - 1)\mathtt{A}$ and $(2^{w-1} - 1)\mathtt{D}$, which reduces to $7\mathtt{A}$ and $7\mathtt{D}$ 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 Figure 12.

```
Input  : Parameter w; two N-bit integers on the stack:
         { ⟨x_{ℓ-1}⟩ ... ⟨x_0⟩ ⟨y_{ℓ-1}⟩ ... ⟨y_0⟩ }
Output : 2N-bit integer z = x × y on the stack:  { ⟨z_{ℓ'-1}⟩ ... ⟨z_1⟩ ⟨z_0⟩ }
{ BigInt<N>::OP_TOBEWINDOWEDFORM_TOALTSTACK }
{ BigInt<N>::OP_EXTEND::<BigInt<2N>> } ;   /* Extend N-bit integer to
    2N-bit integer by appending ℓ' − ℓ zero limbs */
{BigInt<2N>::OP_INITWINDOWEDTABLE(w)} ;   /* Precomputing
    {0, x, ..., ((1 ≪ w) − 1)x} */
{ OP_FROMALTSTACK ⟨1⟩ OP_ADD } ;   /* Picking first limb from the altstack +1 */
{ ⟨1 ≪ w⟩ OP_SWAP OP_SUB BigInt<2N>::OP_PICKSTACK } ;   /* Picking the
    corresponding value from the precomputed table */
for _ ∈ {1, ..., L − 1} do
    /* Double the result w times */
    for _ ∈ {0, ..., w − 1} do
        { BigInt<2N>::OP_2MUL }
    end
    /* Picking limb from the altstack and picking the corresponding element from the
       lookup table.  After picking an element, the stack would look like
       { ⟨0⟩ ⟨x⟩ ... ⟨((1 ≪ w) − 1)x⟩ ⟨r⟩ ⟨y_i⟩ } , where r is the temporary variable, being
       the final result, and y_i is the limb at step i */
    { ⟨1 ≪ w⟩ OP_SWAP OP_SUB }
    { BigInt<2N>::OP_PICKSTACK BigInt<2N>::OP_ADD }
end
/* Clearing the precomputed values from the stack. */
{ BigInt<2N>::OP_TOALTSTACK }
for _ ∈ {0, ..., ((1 ≪ w) − 1)} do
    { BigInt<2N>::OP_DROP }
end
{ BigInt<2N>::OP_FROMALTSTACK }
```

Figure 12. Windowed integer multiplication

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 insufficient to conduct the operations without overflowing, we 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 $0 \le y_i < 2^w$. Next, we initialize the lookup table, which involves calculating $\{0, x, 2x, \dots, (2^w - 1)x\}$. 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 $y = y_{L-1}2^{w(L-1)} + y_{L-2}2^{w(L-2)} + \dots + y_0$ 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, $\lambda \le N - w(L-1) = (N+w) - wL$.

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,

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

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

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

In this case, $q_k < Q_k$ for each $k > 1$, so Q_k is our upper bound. Now, this 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

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

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 results in the integer of bitsize $wk + \lambda$. Indeed:

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

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 in Figure 12 by introducingthe algorithm in Figure 13.

```
Input  : Parameter w; two N-bit integers on the stack:
         { ⟨x_{ℓ−1}⟩ ... ⟨x_0⟩ ⟨y_{ℓ−1}⟩ ... ⟨y_0⟩ }
Output : 2N-bit integer z = x × y on the stack: { ⟨z_{ℓ'−1}⟩ ... ⟨z_1⟩ ⟨z_0⟩ }
{ 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.                                                */
{ BigInt<N>::OP_INITWINDOWEDTABLE(w) } ;                          /* Precomputing
  {0, x, ..., ((1 ≪ w) − 1)x} */
{ OP_FROMALTSTACK ⟨1⟩ OP_ADD } ;   /* Picking first limb from the altstack +1 */
{ ⟨1 ≪ w⟩ OP_SWAP OP_SUB BigInt<N>::OP_PICKSTACK } ;             /* Picking the
  corresponding value from the precomputed table */
for i ∈ {1, ..., L − 1} do
    /* Extend the result from λ + (i − 1)w bits to λ + iw                      */
    { BigInt<λ + (i − 1)w>::OP_EXTEND::<BigInt<λ + iw>> }
    /* Double the result w times                                                */
    for _ ∈ {0, ..., w − 1} do
        { BigInt<λ + iw>::OP_2MUL }
    end
    /* Picking limb from the altstack and picking the corresponding element from the
       lookup table. After picking an element, the stack would look like
       { ⟨0⟩ ⟨x⟩ ... ⟨((1 ≪ w) − 1)x⟩ ⟨r⟩ ⟨y_i⟩ } , where r is the temporary variable, being
       the final result, and y_i is the limb at step i                           */
    { ⟨1 ≪ w⟩ OP_SWAP OP_SUB }
    { BigInt<λ + iw>::OP_PICKSTACK }
    { BigInt<λ>::OP_ADD } ;   /* Since we need to only care about the last limbs,
      we do not extend the result */
end
/* Clearing the precomputed values from the stack.                              */
{ BigInt<2N>::OP_TOALTSTACK }
for _ ∈ {0, ..., ((1 ≪ w) − 1)} do
    { BigInt<λ>::OP_DROP }
end
{ BigInt<2N>::OP_FROMALTSTACK }
```

Figure 13. Windowed integer multiplication with gradual bitsize increase

4. DISCUSSION

4.1. Window Width Choice

One of our key claims is that the width parameter $w = 4$ provides the best performance. which is a common choice in the elliptic curve arithmetic literature [27]. Let usinformally see why we use this particular value. Note that for BN254 we have $N = 254$,so the algorithm in Figure12 requires approximately $2w + 254/w$ addition operations accordingto Section 2.2: the first $2w$ additions initialize the lookup table, while the latter number$254/w$ corresponds to the number of additions in the loop. Notice that although increasing w results in a smaller number of addition operations, it exponentially increases the cost ofinitializing the lookup table: so we do not expect

w to be significantly larger than~6–7. The simple substitutions $w \in \{1, \dots, 6\}$shows that $w = 4$ indeed provides the best cost.

Now, we rigorously justify this claim. For that reason, we provide the following theorem.

Theorem 3. Suppose that algorithm in Figure12 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 $\widehat{w} \in \mathbb{R}$, where $\widehat{w}$ satisfies:

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

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

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 2.2 is approximately (without accounting for operations not depending on the chosen w) given by the following formula:

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

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

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

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

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

which is positive for any $w > 0$ (which is the case). The relation $\widehat{w}^2 2^{\widehat{w}} = N/log2$ 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 $C_{\mathsf{D}} := 245$). Thus, approximately, $\widehat{w}^2 2^{\widehat{w}} \approx 437.5$, yielding $\widehat{w} \approx 4.45$. After checking both $w = 4$ and $w = 5$, we conclude that $w = 4$ isthe optimal choice.

Out of curiosity, we plot the dependence $C(w)$ for different N's and w's. The result is depicted in Figure 14. Interestingly, for larger integers (in particular, for $N = 512$ or $N = 1024$), $w = 4$ most likely would no longer be the optimal choice.

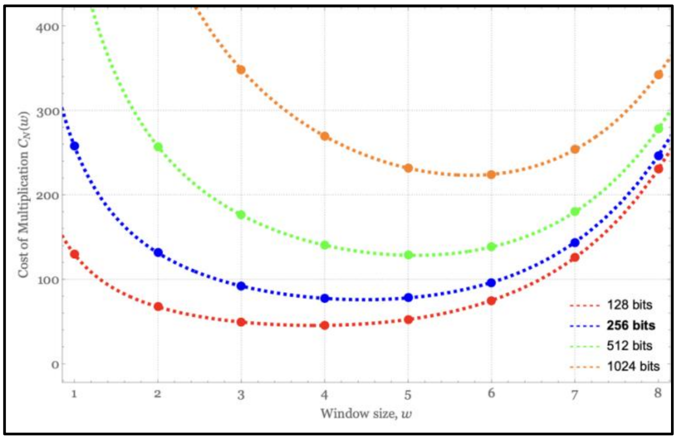

Figure 14. Dependence of multiplication cost $C_N(w)$ on the window size w for various integer bit-sizes (N). We plotted the dependence for four integers: **128 bits**, **256 bits**, **512 bits**, **1024 bits**. The dashed line in **blue** is most closely related to our case ($N = 254$). Here, we assumed that $C_{\mathrm{D}}/C_{\mathrm{A}} \approx 0.675$, corresponding to our multiplication.

4.2. Performance Comparison

Now, we compare our multiplication implementation with the state-of-the-art approaches currently used.

1. *BitVM "Overflow" Multiplication*: *BitVM* provides the default library to operate with big integers (therefore, 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 12 to have the same functionality, and also tweaked the *BitVM*'s implementation to give the $2N$-bit integer as the result.
2. *Cmpeq's Implementation*: 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*. Although it was 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.
3. *BitVM* 29×9 *Karatsuba Multiplication*: This is the most recent version that *BitVM* mostly relies on that uses the Karatsuba multiplication (see 2.2.1) with $(n = 29, \ell = 9)$ to represent a 254-bit integer.

The comparison results are depicted in Table 1.

Table 1. Comparison of our multiplication implementation 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	201879
BitVM bigint	106026	200334
BitVM Karatsuba	N/A	74907

Our w-width method	55710	71757

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.

4.3. Limitations

Although our implementation greatly optimizes the fundamental mathematical block inzk-SNARKs — big integer arithmetic, to achieve practicality (such as launching L2 ontop of Bitcoin), much more research on the Bitcoin-based zk-SNARK scheme is needed. Indeed, although with our approach one needs only∼74.9k opcodes for a single multiplication, this is typically done hundreds of times when launching more complex structures suchas bilinear pairing. This is exactly what BitVM2 [29] and recently publishedBitVM3 [30] try to achieve. We believe that our implementation will serve as the fundamental way to optimize native big integer arithmetic in subsequent protocol implementations (not limited to BitVM), but it cannot solve the aforementioned issue exclusively.

5. CONCLUSIONS

This paper introduced an innovative approach to performing big integer arithmetic within Bitcoin Script using the w-windowed method for multiplying 254-bit integers. Inspired by Elliptic Curve optimization techniques, our method reduces the *BitVM*'s script size needed for multiplication, reducing approximately 3.2k opcodes. Moreover, we believe the applied approach opens the door to other optimizations involving multiple integer multiplication or fixed integer multiplication, which are frequently used in the realm of Elliptic Curves arithmetic.

Our findings enable more efficient complex arithmetic operations. This advancement opens new possibilities for integrating advanced cryptographic protocols (and, in particular, *L2 zk-rollup*) within the Bitcoin ecosystem. For those interested in the technical details, our implementation code is available on GitHub through the provided link (see 1.1).

REFERENCES

[1] Nakamoto, S., (2009) "Bitcoin: A Peer-to-Peer Electronic Cash System."
[2] Mungoli, N., (2023) "Deciphering the Blockchain: A Comprehensive Analysis of Bitcoin's Evolution, Adoption, and Future Implications", arXiv preprint, abs/2304.02655.
[3] Böhme, R., Christin, N., Edelman, B. & Moore, T., (2015) "Bitcoin: Economics, Technology, and Governance", Journal of Economic Perspectives, Vol. 29, No. 2, pp. 213–238.
[4] Thyagarajan, S. A. K., Malavolta, G. & Moreno-Sánchez, P., (2021) "Universal Atomic Swaps: Secure Exchange of Coins Across All Blockchains", Cryptology ePrint Archive, Paper 2021/1612.
[5] Herlihy, M., (2018) "Atomic Cross-Chain Swaps", arXiv preprint, abs/1801.09515.

[6] Kurbatov, O., Zakharov, D., Levochko, A., Riabov, K. &Skriabin, B., (2024) "Multichain Taprootized Atomic Swaps: Introducing Untraceability Through Zero-Knowledge Proofs", arXiv preprint, 2024.

[7] Poon, J. & Dryja, T., (2016) "The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments."

[8] Seres, I. A., Gulyás, L., Nagy, D. A. &Burcsi, P., (2020) "Topological Analysis of Bitcoin's Lightning Network", in Mathematical Research for Blockchain Economy: 1st International Conference MARBLE 2019, Santorini, Greece, pp. 1–12. Springer.

[9] Orlovsky, M., Todd, P., Zucco, G., Tenga, F. &Ukolova, O., (2023) "RGB Blackpaper", https://blackpaper.rgb.tech/ (accessed 2024-08-02).

[10] Mia, A., (2024) "LRC-20: Scalable and Fast Tokenization on Lightning", GitHub Technical Whitepaper, https://github.com/akitamiabtc/LRC-20.

[11] Wood, G., et al., (2014) "Ethereum: A Secure DecentralisedGeneralised Transaction Ledger", Ethereum Project Yellow Paper, 151(2014):1–32.

[12] Chaliasos, S., Reif, I., Torralba-Agell, A., Ernstberger, J., Kattis, A. & Livshits, B., (2024) "Analyzing and Benchmarking ZK-Rollups", Cryptology ePrint Archive, Paper 2024/889, https://eprint.iacr.org/2024/889.

[13] Chen, T., Lu, H., Kunpittaya, T. & Luo, A., (2023) "A Review of ZK-SNARKs."

[14] Linus, R., (2023) "BitVM: Compute Anything on Bitcoin."

[15] Gautam, S., Kandel, P., Subedi, C. & Chhetri, A., (2023) "ZK Rollup on Bitcoin: Technical Whitepaper", GitHub Technical Whitepaper, https://github.com/alpenlabs/Technical-Whitepaper.

[16] Devegili, A., Scott, M. & Dahab, R., (2007) "Implementing Cryptographic Pairings Over Barreto-Naehrig Curves", Lecture Notes in Computer Science, Vol. 4575, pp. 197–207.

[17] Groth J., (2016)"On the size of pairing-based non-interactive arguments", InMarc Fischlin and Jean-Sébastien Coron, editors, Advances in Cryptology –EUROCRYPT 2016, pages 305–326.

[18] Gabizon A.,& Williamson Z., (2021)"fflonk: a fast-fourier inspired verifierefficient version of PlonK", Cryptology ePrint Archive, Paper 2021/1167.

[19] Antonopoulos, A. M., (2014) "Mastering Bitcoin: Unlocking Digital Crypto-Currencies", O'Reilly Media, Inc., 1st ed.

[20] Krtolica, P. V. &Stanimirović, P. S., (2004) "Reverse Polish Notation Method", International Journal of Computer Mathematics, Vol. 81, No. 3, pp. 273–284.

[21] Kumar, M. & Chand, S., (2022) "Pairing-Friendly Elliptic Curves: Revisited Taxonomy, Attacks and Security Concern", arXiv preprint, abs/2212.01855.

[22] Weimerskirch, A. & Paar, C., (2006) "Generalizations of the Karatsuba Algorithm for Efficient Implementations", IACR Cryptology ePrint Archive, 2006:224.\

[23] Eyupoglu C., (2015)"Performance analysis of karatsuba multiplication algorithm fordifferent bit lengths", Procedia - Social and Behavioral Sciences, 195:1860–1864, World Conference on Technology, Innovation and Entrepreneurship.

[24] Patil S., Manjunatha D. V., & Kiran D., (2014)"Design of speed and powerefficient multipliers using vedic mathematics with vlsi implementation", In2014 International Conference on Advances in Electronics Computers andCommunications, pages 1–6.

[25] Madke D. &Zafar S.,(2014) "Polynomial multiplication using karatsuba and nikhilamsutra", International Journal of Advanced Research in Computer Science andSoftware Engineering, 4(6):1423–1428.

[26] Mishra S. & Pradhan P., (2012)"Synthesis comparison of karatsubamultiplierusing polynomial multiplication, vedic multiplier and classicalmultiplier", International Journal of Computer Applications.

[27] Hankerson, D., Menezes, A. J. & Vanstone, S., (2010) "Guide to Elliptic Curve Cryptography", Springer Publishing Company, Inc., 1st ed.

[28] Botrel G., Piellard T., Housni Y., Kubjas I., &Tabaie A., (2025) Consensys/gnark: v0.13.0

[29] Linus R., Aumayr L., Avarikioti Z., Maffei M.,Pelosi A.,Litos O., Stefo C., Tse D., & Zamyatin A., (2025) "Bridging bitcoin to second layers via BitVM2", Cryptology ePrint Archive,Paper 2025/1158.

[30] Linus R., (2025) "Bitvm 3s – garbled circuits for efficient computation on bitcoin", bitvm.org.

AUTHORS

Dmytro Zakharov is an Applied Mathematics student at V. N. Karazin Kharkiv National University. His research interests include biometric systems, applied cryptography, AI safety and security, and zero-knowledge cryptography. He has published articles in Engineering Applications of Artificial Intelligence (2024), Multimedia Tools and Applications (2023), and Computers & Security (2024).

Oleksandr Kurbatov is a PhD Candidate in Karazin Kharkiv National University, Ukraine. His research focuses on public-key infrastructure, blockchain technologies, and anonymous decentralized voting systems. He has presented work at the 2019 IEEE International Conference on Advanced Trends in Information Theory (ATIT), CMiGIN 2019, and the 2023 IEEE 1st Ukrainian Distributed Ledger Technology Forum (UADLTF). His publications include a paper in ISCI 2019: Information Security in Critical Infrastructures. Currently working as the Lead Cryptography Researcher at Distributed Lab.

Manish Bista is a cryptography and Bitcoin developer at Alpen Labs, active in zero-knowledge research and implementation of zk-enabled infrastructure.

Belove Bist is a researcher at Alpen Labs, where he focuses on Applied Artificial Intelligence, Blockchain Scalability, and Zero-Knowledge Cryptography. He holds a Bachelor of Engineering in Computer Engineering from Tribhuvan University's Institute of Engineering. His publications include Automatic Vehicular License Trial Monitoring System (presented at NaScoIT 2016) and Clinical Risk Prediction Using Language Models: Benefits and Challenge.