$\mathcal{NeRO}$: On Developing Generic Optimistic Verifiable Computation on Bitcoin. BitVM2 Practical Research

Kyrylo Baibula[1], Oleksandr Kurbatov[1] and Dmytro Zakharov[1]

Distributed Lab dmytro.zakharov@distributedlab.com, ok@distributedlab.com, kyrylo.baybula@distributedlab.com

Abstract. One of Bitcoin's biggest unresolved challenges is the ability to execute a large arbitrary program on-chain. Namely, publishing a program written in Bitcoin Script that exceeds 4 MB is practically impossible. This is a strict restriction as, for instance, it is impossible to multiply two large integers, not even mentioning a zero-knowledge proof verifier. To address this issue, we narrow down the problem to the verifiable computation which is more feasible given the current state of Bitcoin. One of the ways to do it is the BitVM2 protocol. Based on it, we are aiming to create a generic library for the on-chain verifiable computations. This document is designated to state our progress, pitfalls, and pain... While most of the current efforts are put into transferring the *Groth16* verifier on-chain with the main focus on implementing bridge, we try to solve a broader problem, enabling a more significant number of potential use cases (including zero-knowledge proofs verification).

Keywords: Bitcoin, Bitcoin Script, Verifiable Computation, Optimistic Verification, BitVM2

Contents

Note

This is a very early version of the paper, development is still in active progress!

1 Introduction

The Bitcoin Network [2] is rapidly growing. However, the Bitcoin Script, the native programming language of Bitcoin, imposes strict size limits on transactions — only 4 MB are allowed, making it challenging to implement any advanced cryptographic (and not only) primitives, among which highly desirable zero-knowledge proofs verification on-chain. To address this limitation, the BitVM2 [3] proposal introduces an innovative approach that enables the optimistic execution of large programs on the Bitcoin chain.

The proposed method suggests that the executor (which is called an **operator**) splits the large program into smaller chunks (which we further refer to as **shards**) and commits to the intermediate values. This way, if the computation is incorrect, it must be incorrect in some shard, and it can be proven *concisely* due to the splitting mechanism.

This document provides a concise overview of our progress in implementing the library for generic, optimistic, verifiable computation on Bitcoin. Currently, we are focusing on reproducing the BitVM2 paper approach while not limiting the function and input/output format as much as possible.

2 Program Split

2.1 Public Verifiable Computation

Since the main goal of our research is to build the *public verifiable computation*, it is reasonable to start with a brief overview of this concept. A *public verifiable computation scheme* allows the (potentially) computationally limited verifier $\mathcal{V}$ outsource the evaluation of some function f on input x to the prover (worker) $\mathcal{P}$. Then, $\mathcal{V}$ can verify the correctness of the provided output y by performing significantly less work than f requires.

In the context of Bitcoin on-chain verification, $\mathcal{V}$ can be viewed as the Bitcoin smart contract which is heavily limited in computational resources (due to the inherit Bitcoin Script inexpressiveness). The prover $\mathcal{P}$ is the operator who executes the program on-chain. The program f is the Bitcoin Script, and the input x is the data provided by the operator.

Now, we define the *public verifiable computation scheme* as follows:

Definition 1. A public verifiable computation (VC) scheme Π_{VC} consists of three probabilistic polynomial-time algorithms:

- $\mathsf{Gen}(f, 1^\lambda)$: a randomized algorithm, taking the security parameter $\lambda \in \mathbb{N}$ and the function f as input, and outputting the prover and verifier parameters pp and vp.
- $\mathsf{Compute}(\mathsf{pp}, x)$: a deterministic algorithm, taking the prover parameters pp and the input x, and outputting the output y together with a "proof of computation" π.
- $\mathsf{Verify}(\mathsf{vp}, x, y, \pi)$: given the verifier parameters vp, the input x, the output y, and the proof π, the algorithm outputs accept or reject based on the correctness of the computation.

Such scheme should satisfy the following properties (informally):

- **Correctness**. Given any function f and input x,
$$\Pr\left[\mathsf{Verify}(\mathsf{vp}, x, y, \pi) = \mathsf{accept} \;\middle|\; \begin{array}{c}(\mathsf{pp}, \mathsf{vp}) \leftarrow \mathsf{Gen}(f, 1^\lambda) \\ (y, \pi) \leftarrow \mathsf{Compute}(\mathsf{pp}, x)\end{array}\right] = 1$$
- **Security**. For any function f and any probabilistic polynomial-time adversary $\mathcal{A}$,
$$\Pr\left[\mathsf{Verify}(\mathsf{vp}, \widetilde{x}, \widetilde{y}, \widetilde{\pi}) = \mathsf{accept} \;\middle|\; \begin{array}{c}(\mathsf{pp}, \mathsf{vp}) \leftarrow \mathsf{Gen}(f, 1^\lambda) \\ (\widetilde{x}, \widetilde{y}, \widetilde{\pi}) \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{vp}),\ f(\widetilde{x}) \neq \widetilde{y}\end{array}\right] \leq \mathsf{negl}(\lambda)$$
- **Efficiency**. Verify should be much cheaper than the evaluation of f.

2.2 Motivation for Verifiable Computation on Bitcoin

Suppose we have a large program f implemented inside the Bitcoin Script and want to verify its execution on-chain. Suppose the prover $\mathcal{P}$ claims that $y = f(x)$ for published x and y. Some of the examples include:

- **Field multiplication**: $f(a, b) = a \times b$ for $a, b \in \mathbb{F}_p$. Here, the input $x = (a, b) \in \mathbb{F}_p^2$ is a tuple of two field elements, while the output $y \in \mathbb{F}_p$ is a single field element.
- **EC points addition**: $f(x_1, y_1, x_2, y_2) = (x_1, y_1) \oplus (x_2, y_2) = (x_3, y_3)$. Input is a tuple (x_1, y_1, x_2, y_2) of four field elements, representing the coordinates of two elliptic curve points. The output is a point (x_3, y_3), represented by two field elements $\mathbb{F}_p$.
- **Groth16 verifier**: $f(\pi_1, \pi_2, \pi_3) = b$ for $b \in \{\mathsf{accept}, \mathsf{reject}\}$. Based on three provided points π_1,π_2,π_3, representing the proof, decide whether the proof is valid.

As mentioned before, publishing f entirely on-chain is not an option. Instead, the BitVM2 paper suggests splitting the program into shards $f_1, \ldots, f_n$ such that $f = f_n \circ f_{n-1} \circ \cdots \circ f_1$, where $\circ$ denotes the function composition. This way, both the prover $\mathcal{P}$ and verifier $\mathcal{V}$ can calculate all intermediate results as follows:

$$z_j = f_j(z_{j-1}), \text{ for each } j \in \{1, \ldots, n\}$$

Of course, we additionally set $z_0 := x$. If everything was computed correctly and the function was split into shards correctly, eventually, we will have $z_n = y$.

So recall that $\mathcal{P}$ (referred to in BitVM2 as the *operator*) only needs to prove that the given program f indeed returns y for x, otherwise **anyone can disprove this fact**. In our case, this means giving challengers (essentially, being verifiers $\mathcal{V}$) the ability to prove that at least one of the sub-program statements $f_j(z_{j-1}) = z_j$ is false.

So overall, the idea of BitVM2 can be described as follows:

1. The program f is decomposed into shards $f_1, \ldots, f_n$ of reasonable size[1].
2. $\mathcal{P}$ executes f on input x shard by shard, obtaining intermediate steps $z_1, \ldots, z_n$.
3. $\mathcal{P}$ commits to the given intermediate steps and publishes commitments on-chain.
4. $\mathcal{V}$, knowing x published by $\mathcal{P}$, executes the same program, obtaining his own states $\widetilde{z}_1, \ldots, \widetilde{z}_n$.
5. $\mathcal{V}$ checks whether $\widetilde{z}_j = z_j$. If this does not hold, the verifier publishes transactions corresponding to the disprove statement $z_j \neq f_j(z_{j-1})$ and claims funds.

2.3 Implementation on Bitcoin

This does not sound very hard; however, implementing this in Bitcoin is not obvious. The good news is that Bitcoin is a stack-based language, so the function f is just a string, where each word is the `OP_CODE`. Notice that, in the stack-based languages, the concatenation $f_1 \parallel f_2$ of two *valid* functions f_1 and f_2 is the same thing as their composition. In other words, executing the script $\{ \langle x \rangle \langle f_1 \rangle \langle f_2 \rangle \}$ is the same as calculating composition $f_2 \circ f_1(x)$. So all what remains is finding *valid* $f_1, \ldots, f_n$ such that $f = f_1 \parallel f_2 \parallel \cdots \parallel f_n$. All the intermediate steps $\{z_j\}_{0 \leq j \leq n}$ can be calculated as specified in Algorithm 1.

[1] By "size" we mean the number of `OP_CODES` needed to represent the logic.

Algorithm 1: Calculating intermediate steps from script shard decomposition

Input : Script f
Output : Intermediate steps $z_1, \ldots, z_n$
Decompose f into shards: $(f_1, \ldots, f_n) \leftarrow \mathsf{Decompose}(f)$;
for $i \in \{1, \ldots, n\}$ **do**
  $z_i \leftarrow \mathsf{Exec}(\{\, \langle z_{i-1} \rangle \, \langle f_i \rangle \,\})$;
end
Return : $z_1, \ldots, z_n$

Bad news is that Decompose function is quite tricky to implement. Namely, we believe that there are several issues:

- Decomposition must be valid, meaning each f_j must be valid itself. For example, f_j cannot contain unclosed `OP_IF`'s. This issue is easily fixed through a careful implementation of the splitting mechanism: for instance, whenever the number of `OP_IF`'s and `OP_NOTIF`'s is not equal to the number of `OP_ENDIF`'s, we continue the current shard until the balance is restored.

- Despite that each f_j might be small, not necessarily z_j is. In other words, optimizing the size of each f_j does not result in optimizing the size of z_j. Moreover, in the further sections, we show that optimizing the size of intermediate states is, in fact, a much more tricky and fundamental issue than optimizing the shards' sizes. In other words, we should find a balance between the size of f_j and the size of z_j.

- Some of z_j's might contain the same repetitive pieces: for example, the lookup table for certain algorithms or the number binary/w-width decomposition for arithmetic. We believe that there must be an optimal method to store commitments.

However, the default version proceeds as follows: suppose our script is of form $f = \{\, \langle s_1 \rangle \, \langle s_2 \rangle \, \ldots \, \langle s_k \rangle \,\}$ where $\langle s_j \rangle$ is either an `OP_CODE` or an element in the stack (added via, for example, `OP_PUSHBYTES`). Then, we start splitting the program from left to right and if the size of the current shard exceeds the limit (say, L), we stop and start a new shard. The only exception when we cannot stop is unclosed `OP_IF` and `OP_NOTIF`. This way, approximately, we will have $\lceil k/L \rceil$ shards each of size L.

2.3.1 Fuzzy Search

The basic version, though, does not guarantee the optimal intermediate stack sizes. One of the proposals to improve the splitting mechanism is to make program automatically choose the optimal size. In other words, we make the parameter L variable and try to find the optimal L that minimizes the certain "metric". What is this metric?

Since we want to potentially disprove the equality $z_{j+1} = f_j(z_j)$, the cost of such disproof is the total size of z_j, z_{j+1} and the shard f_j. Denote the size of the script/state by $|\star|$. Then, we want to minimize some sort of "average" of $\alpha(|z_j| + |z_{j+1}|) + |f_j|$. The factor α is introduced since, besides the cost of storing z_j, we also need to *commit* to these values which, as we will see, significantly increases the cost of a disprove script. In other words, α is a considerable factor in practice: currently, our estimate suggests $\alpha \approx 1000$.

Then, depending on the goal, we might choose different criteria of "averaging":

- **Maximal size**. Suppose we want to minimize the cost of the worst-case scenario. Suppose after the launching the splitting mechanism on the shard size L we get k_L shards $f_{L,1}, \ldots, f_{L,k_L}$ with intermediate states $z_{L,0}, \ldots, z_{L,k_L}$. Then, we choose L to be:

$$\hat{L} := \operatorname*{arg\,min}_{0 \le L \le L_{\max}} \left\{ \max_{0 \le j < k_L} \{\alpha(|z_{L,j}| + |z_{L,j+1}|) + |f_{L,j}|\} \right\} .$$

- **Average size**. Suppose we want to minimize the average cost of disproof. Then, we choose L to be:

$$\hat{L} := \underset{0 \le L \le L_{\max}}{\arg\min} \left\{ \frac{1}{k_L} \sum_{0 \le j < k_L} (\alpha(|z_{L,j}| + |z_{L,j+1}|) + |f_{L,j}|) \right\} .$$

Note, however, that this algorithm is still far from being the most optimal one. We assume that, in reality, in the majority of cases, the optimal shards sizes can significantly differ, which the automatic splitting can easily miss.

The ultimate solution would be to check every possible splitting and choose the one that minimizes the cost of disproof. However, this is not feasible in practice, as the number of possible splittings is enormous (even, say, for the fixed number of shards).

2.3.2 Current State

We implemented the basic splitting mechanism that finds $f_1, \ldots, f_k$ of almost equal size (which can be specified). It already produces valid shards and intermediate states on all of the following scripts:

- **Big Integer Addition** (of any bitsize).
- **Big Integer Multiplication** (of any bitsize).
- **SHA-256** hash function.
- **Square Fibonacci Sequence Demo**.
- **`u32` Multiplication**.

We will explore the last two functions in more detail a bit later. All the current implementation of test scripts can be found through the link below:

https://github.com/distributed-lab/bitvm2-splitter/tree/main/bitcoin-testscripts

3 Assert Transaction

When the splitting is ready, the prover $\mathcal{P}$ publishes an `Assert` transaction, which has one output with multiple possible spending scenarios:

1. `PayoutScript` (`CheckSig` + `CheckLocktimeVerify` + `Covenant`) — the transaction has passed verification, and the operator can spend the output, thereby confirming the statement $y = f(x)$.
2. `DisproveScript[`i`]` — one of the challengers has found a discrepancy in the intermediate states z_i, z_{i-1} and the sub-program f_i. In other words, they have proven that $f_i(z_{i-1}) \neq z_i$, and thus, they can spend the output.

While the `PayoutScript` is rather trivial, we need to specify how the `DisproveScript[`i`]` is constructed. `DispoveScript` is part of the MAST tree in a Taproot address, allowing the verifier to claim the transaction amount for states z_i, z_{i-1}, and sub-program f_i. We call it `DisproveScript[`i`]` and compose it as follows:

Script:	$\langle z_i \rangle$ $\langle z_{i-1} \rangle$ $\langle f_i \rangle$ `OP_EQUAL` `OP_NOT`

This script does not need a `CheckSig`, as with the correct z_i and z_{i-1}, it will consistently execute successfully. Therefore, we added a Winternitz signature and covenant verification to restrict the script's spending capability. Currently, we will simulate covenant through a committee of a single person (essentially, being a single signature verification), but this is easily extendable to the multi-threshold signature version (and, potentially, to `OP_CAT`-based version, but that is the next phase of our research).

3.1 Winternitz Signature

Unlike other digital signature algorithms, the Winternitz signature uses a pair of random secret and public keys $(\mathsf{sk}, \mathsf{pk})$ that can sign and verify only any message from the message space $\mathcal{M} = \{0,1\}^\ell$ of ℓ-bit messages.

However, once the signature σ_m is formed, where $m \in \mathcal{M}$ is the message being signed, $(\mathsf{sk}_m, \mathsf{pk}_m)$ become tied to m, because any other signature with these keys will compromise the keys themselves. Thus, for the message m, the keys $(\mathsf{sk}_m, \mathsf{pk}_m)$ are one-time use.

Now, let us define the Winternitz Signature. Further by $f^{(k)}(x)$ denote the composition of function f with itself k times: $f^{(k)}(x) = \underbrace{f \circ \cdots \circ f}_{k \text{ times}}(x)$.

Definition 2. The **Winternitz Signature Scheme** over parameters (k, d) with a hash function $H : \mathcal{X} \to \mathcal{X}$ is defined as follows:

- $\mathsf{Gen}(1^\lambda)$: secret key is generated as a tuple $(x_1, \ldots, x_k) \xleftarrow{R} \mathcal{X}$, while the public key is $(y_1, \ldots, y_k)$, where $y_j = H^{(d)}(x_j)$ for each $j \in \{1, \ldots, k\}$.
- $\mathsf{Sign}(m, \mathsf{sk})$: denote by $\mathcal{I}_{d,k} := (\{0, \ldots, d\})^k$ and suppose we have an encoding function $\mathsf{Enc} : \mathcal{M} \to \mathcal{I}_{d,k}$ that translates a message $m \in \mathcal{M} = \{0,1\}^\ell$ to the element in space $\mathcal{I}_{d,k}$. Now, set $e = (e_1, \ldots, e_k) \leftarrow \mathsf{Enc}(m)$. Then, the signature is formed as:
$$\sigma \leftarrow (H^{(e_1)}(x_1), H^{(e_2)}(x_2), \ldots, H^{(e_k)}(x_k))$$
- $\mathsf{Verify}(\sigma, m, \mathsf{pk})$: to verify $\sigma = (\sigma_1, \ldots, \sigma_k)$ on $m \in \mathcal{M}$ and $\mathsf{pk} = (y_1, \ldots, y_k)$, first compute encoding $(e_1, \ldots, e_k) \leftarrow \mathsf{Enc}(m)$ and then check whether:
$$H^{(d-e_j)}(\sigma_j) = y_j, \quad j \in \{1, \ldots, k\}.$$

That being said, by taking the intermediate states $\{z_j\}_{1 \le j \le n}$ as the message for the Winternitz signature, we form one-time key pairs $\{(\mathsf{sk}_j, \mathsf{pk}_j)\}_{1 \le j \le n}$ and signatures $\{\sigma_j\}_{1 \le j \le n}$, respectively (where each of pk_j, sk_j, and σ_j corresponds to the intermediate variable z_j). Then, to spend the output from the `Assert` transaction using the `DisproveScript[`j`]` script, the challenger is required to add the corresponding states z_j, z_{j-1}, and corresponding signatures σ_j, σ_{j-1} to the stack in the `scriptSig`, making the `scriptSig` of the transaction input like this:

Script:

$\langle z_{j-1} \rangle$ `OP_DUP` $\langle \sigma_{j-1} \rangle$ $\langle \mathsf{pk}_{j-1} \rangle$ `OP_WINTERNITZVERIFY`
$\langle z_j \rangle$ `OP_DUP` $\langle \sigma_j \rangle$ $\langle \mathsf{pk}_j \rangle$ `OP_WINTERNITZVERIFY`
$\langle f_j \rangle$ `OP_EQUAL OP_NOT`

where `OP_WINTERNITZVERIFY` is the verification of the Winternitz signature (commitment), described in Bitcoin Script (as Bitcoin Script does not have a built-in `OP_CODE` for Winternitz signatures)[2].

[2]Its implementation can be found here: https://github.com/distributed-lab/bitvm2-splitter/blob/feature/winternitz/bitcoin-winternitz/src/lib.rs.

3.1.1 Winternitz Signatures in Bitcoin Script

The first biggest issue with the provided approach is that the Winternitz Script requires encoding the message $\mathsf{Enc}(m)$, which splits the state into d digit number. For BitVM2, it means encoding each state z_j. However, the arithmetic in Bitcoin Script is limited and contains only basic opcodes such as `OP_ADD`. To make matters worse, all the corresponding operations can be applied to 32-bit elements only, and as the last one is reserved for a sign, only 31 bits can be used to store the state. This limitation can be considered strong, but most of the math can be implemented through 32-bit stack elements. So lets fix $\ell = 32$ — maximum size of the stack element in bits.

The first observation is that essentially z_j is a collection of 32-bit numbers (suppose this collection consists of n_j numbers). Denote this fact by $z_j = (u_{j,1}, u_{j,2}, \ldots, u_{j,n_j})$ where each $u_{j,k} \in \mathbb{Z}_{2^\ell}$. Therefore, one way to implement the message encoding is following:

1. Aggregate elements of z_j into a single hash digest $h_j \leftarrow H(u_{j,1} \parallel u_{j,2} \parallel \cdots \parallel u_{j,n_j})$.

2. Use dominant free function $P(h_j)$ as described in [1] to get the decomposition.

However, as of now, the Bitcoin does not have the `OP_CAT`, so there is no way we can effectively aggregate the intermediate state z_j into a single stack element. Meaning, we need to create a Winternitz keypair $(\mathsf{pk}_{j,k}, \mathsf{sk}_{j,k})$ for each $u_{j,k}$ where $k \in \{1, \ldots, n_j\}$.

However, there are couple of tricks to make the life easier. First, obviously, it is convenient to choose d such that $d+1 = 2^w$ for some $w \in \mathbb{N}$. This splits the signed message m by the fixed number of equal chunks of w bits. Let N be the sum of n_0 — the number of d-digit numbers in the decomposition of m, and n_1 a checksum (see Table 1).

Table 1: Different values of N depending on d for 32-bit message

w	n_0	n_1	N
2	16	4	20
3	11	3	14
4	8	2	10
5	7	2	9
6	6	2	8
7	5	2	7
8	4	2	6

Secondly, notice the following fact: *encoding the message m (essentially, being the number decomposition) is more expensive than decoding the message (being the number recovery from limbs).* In fact, if chunks are of equal lengths, the recovery of a message m from n_0 digits can be computed very easily: simply set $m \leftarrow \sum_{i=0}^{n_0} e_i \times 2^{wi}$. Note that multiplication by powers of two can be implemented in Bitcoin Script with sequence of `OP_DUP` and `OP_ADD` opcodes quite efficiently. So, for example, multiplication of e_j by 2^n in Bitcoin Script is:

Script: $\langle e_j \rangle \underbrace{\texttt{OP\_DUP OP\_ADD}}_{n \text{ times}}$

As Bitcoin Script has no loops or jumps, implementing dynamic number of operations, like hashing something $d - e_j$ times without knowing the e_j beforehand is challenging. That's why implementation uses the "lookup" table of all d hashes of signature's part σ_j and by using `OP_PICK` pop the $d - e_j$ one on the top of stack, like this:

Script:

$$\langle \sigma_j \rangle \underbrace{\texttt{OP\_DUP OP\_HASH}}_{d \text{ times}}$$
$$\langle e_j \rangle \texttt{ OP\_PICK}$$

So overall, the algorithm to sign the states looks as follows:

1. The prover $\mathcal{P}$ runs the program on all shards $\{f_j\}_{1 \le j \le n}$ to obtain the intermediate states $\{z_j\}_{0 \le j \le n}$: essentially, being the stack after executing $\{ \langle z_{j-1} \rangle \langle f_j \rangle \}$.
2. $\mathcal{P}$ interprets each z_j as a collection of n_j 32-bit numbers: $z_j = (u_{j,1}, u_{j,2}, \ldots, u_{j,n_j})$.
3. $\mathcal{P}$ encodes each $u_{j,k}$ and forms the Winternitz keypairs $\{(\mathsf{pk}_{j,k}, \mathsf{sk}_{j,k})\}_{1 \le k \le n_j}$.
4. Verifier $\mathcal{V}$, when publishing the `DisproveScript[`j`]`, will add the corresponding **encoded** states $\mathsf{Enc}(z_j)$, $\mathsf{Enc}(z_{j-1})$, and corresponding signatures σ_j and σ_{j-1}.
5. The script, in turn, besides the verification of the intermediate states signatures, will **recover** the original $u_{j,k}$ elements from the encoded states and verify the equality $f_j(z_{j-1}) = z_j$ after recovery of both z_j and z_{j-1}.

Script Size Analysis. Still, even with optimizations provided, the current implementation requires around 1000 bytes per 32-bit stack element, which is unfortunatly a lot. Parts of the public key make the largest contribution to the script size. Assuming that as H implementation uses `OP_HASH160`, each part $(y_1, \ldots, y_N)$ of the public key pk_m adds 20 bytes to the total script size. Additionaly, for calculating a lookup table for signature verification, $2d \times N$ opcodes are used. Furthermore, for message recovery, $2\sum_{i=0}^{n_0} iw$ opcodes are added too. Also, note that $2\sum_{i=0}^{n_0} iw = wn_0(n_0+1) \approx wn_0^2$, so the total script size, excluding utility opcodes, will be at least roughly $20N + 2dN + wn_0^2$.

For script's input stack elements, the size of encoding $\mathsf{Enc}(z_j)$ and corresponding signature σ_j, as each limb can be stored only as a whole byte, is around $N + 20N = 21N$ bytes. So the total size of the largest part of a disprove transaction — `witness`, is:

$$(41 + 2d) \cdot N + w(n_0 + 1)n_0$$

The sizes for different d can be seen in Table 3.

Table 2: Different script sizes depending on the d value per each 32-bit message. Note that, "Signature" column includes the encoding of 32-bit message in it.

d	Signature	Public Key	Verification Script	Recovery Script	Total
3	420	400	120	240	1180
7	294	280	196	165	935
15	210	200	300	112	822
31	189	180	558	105	1032
63	168	160	1008	90	1426
127	147	140	1778	70	2135
255	126	120	3060	48	3354

3.1.2 Compact Winternitz commitment scheme in Bitcoin

As we mentioned in prevoius section, the most contribution to script size comes from public key, signature, verification and recovery, but do we really need to do that for all of the limbs?

Let's check the checksum of winternitz singature described in [1]:

$$c \leftarrow dn_0 - (e_1 + \ldots + e_{n_0}) \tag{1}$$

Assuming that: $d = 15$ and $e_7 = 0$ — is the last limb of the $\mathsf{Enc}(m)$, from (3.1.2) the value of e_7 is obviously negliable in the sum of the checksum, so, without any security concerns, can be skipped. But to keep the right value after a recover, we can't remove all "zeroed" limbs. That's why instead, we propose *skipping* only the most significant limbs of a stack element.

This makes the recovery script size equal to zero in best case scenarios, public key's and signature's hashes number from $1 + n_1$ in best cases to N in worst, and the same applies to verification script size too, as less checks required for smaller public key.

Table 3: Different number of possible script sizes for $d = 15$ depending on the the number of non-zero limbs.

Non-zero limbs	Signature	Public Key	Verification	Recovery	Total
1	63	60	90	0	213
2	84	80	120	4	288
3	105	100	150	12	367
4	126	120	180	24	450
5	147	140	210	40	537
6	168	160	240	60	628
7	189	180	270	84	723
8	210	200	300	112	822

3.2 Disprove Script Specification

All in all, the `DisproveScript[`j`]` is formed as follows:

- **Witness:** $\Big\{\mathsf{Enc}(z_{j+1}),\ \sigma_{j+1},\ \mathsf{Enc}(z_j),\ \sigma_j\Big\}$.
- **Spending Condition**:

Script:

$\langle \mathsf{pk}_j \rangle$ `OP_WINTERNITZVERIFY OP_RESTORE OP_TOALTSTACK`
$\langle \mathsf{pk}_j \rangle$ `OP_WINTERNITZVERIFY OP_RESTORE FROM_TOALTSTACK`
$\langle f_j \rangle$ `OP_EQUAL OP_NOT`

Note on implementation. One more tricky part is that z_j, in fact, is not really a collection of stack elements, but two collections: one sitting in the `mainstack`, while the other is in the `altstack`. For that reason, when signing z_j, what we *really* mean is signing both elements in the `mainstack` and the `altstack`. Finally, one should carefully manage when to pop the elements in and out of the `altstack`. All of this is implemented in the current version of the code.

3.3 Structure of the MAST Tree in a Taproot Address

The inputs of the `Assert` transaction spend the output to a Taproot address, which consists of a MAST tree of Bitcoin scripts mentioned in Section 3. From the BitVM2 document, it is known that the first n scripts in the tree are all `DisproveScript[`i`]`, where $i \in \{1, \ldots, n\}$, and the last is a script that allows the operator who published the `Assert` transaction to spend the output after some time. A visualization of this tree can be seen in the Figure 1.

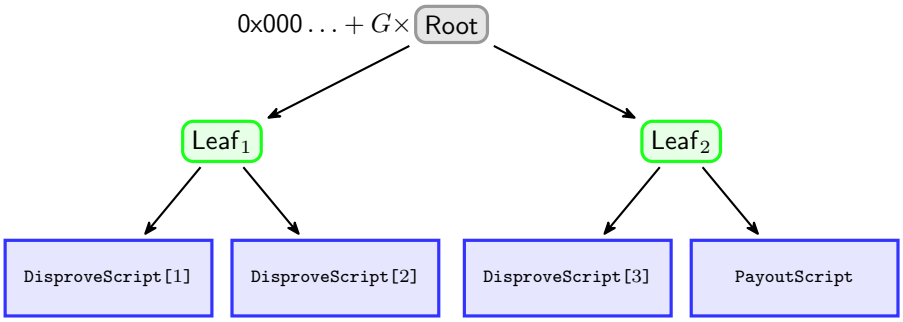

Figure 1: Script tree in a Taproot address with three sub-programs and two intermediate states. Here, G is the generator point of the elliptic curve.

4 Exploring BitVM2 Potential using Toy Examples

Finally, in this section, we explore the potential of BitVM2 using some toy examples. We will consider the following functions:

- `u32` **Multiplication** — a function that multiplies two 32-bit unsigned integers.
- **Square Fibonacci Sequence** — a simple function that calculates the n-th element of the square Fibonacci sequence.

We will demonstrate that the current implementation of BitVM2 and current approach to writing Mathematics (finite field arithmetic, elliptic curve operations etc.) cannot handle even the first example. Based on the second example, we will show that with the appropriate ideology, the BitVM2 can still be used to verify the execution of complex programs, but written in a different way. We call such functions as **BitVM-friendly functions**.

4.1 u32 Multiplication

The most basic example is the multiplication of two 32-bit unsigned integers. In our terminology, $f(x, y) = x \times y$, where the output is a 64-bit unsigned integer. Since using `u32` elements in the stack to represent limbs of the big integer typically results in overflowing, we use two 30-bit limbs to represent a `u32`. This way, the result, being `u64`, is represented by three `u30` limbs. We acknowledge that this is might not be the most efficient representation, but it should suffice for the demonstration purposes.

4.1.1 Implementation Notes

In this section, we give a brief recap of the BitVM implementation of the big integer multiplication in Bitcoin Script. Essentially, the Bitcoin script utilizes the double-and-add method, commonly used for elliptic curve arithmetic. The idea is following: we can first decompose one of the integers to the binary form (say, $y = (y_0, \ldots, y_{N-1})_2$ where N is the bitsize of y). Next, we iterate through each bit and on each step, we double the temporary variable and add it to the result if the corresponding bit in y is 1. The concrete algorithm is described in Algorithm 3.

Note that implementing long addition and doubling in Bitcoin Script is quite cheap, so algorithm turns out to be relatively efficient — you can read more in our recently published paper [4], where we analyze various strategies of big integer multiplication. In our particular case, we assume that `u32` is just a special case of the big integer with the total bitsize of $N = 32$.

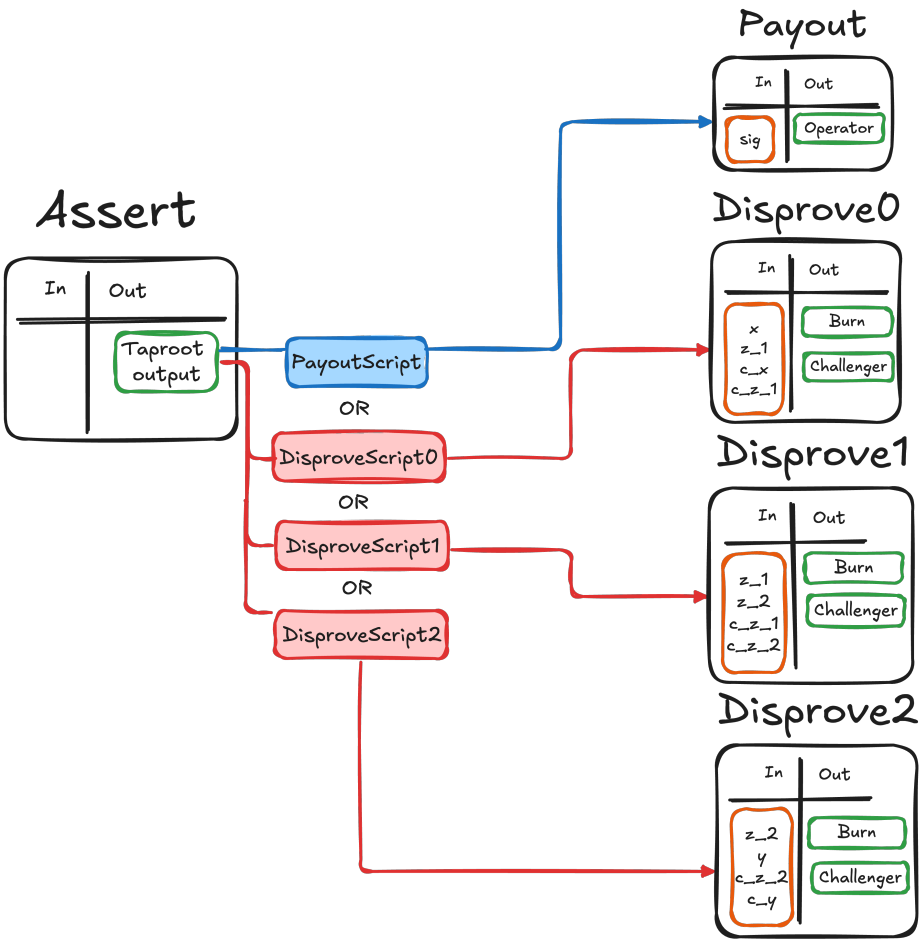

Figure 2: Sequance of transactions in BitVM2 with 3 subprograms and 2 intermediate states.

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

Input : x, y — two `u32` integers being multiplied, N — bitsize of y.
Output : Result of the multiplication $x \times y$

Decompose y to the binary form: $(y_0, y_1, \ldots, k_{N-1})_2$
$r \leftarrow 0$
$t \leftarrow x$
for $i \in \{0, \ldots, N-1\}$ **do**
  if $y_i = 1$ **then**
    $r \leftarrow r + t$
  end
  $t \leftarrow 2 \times t$
end
Return : Integer r

4.1.2 Split Cost Analysis

The total size of the script turns out to be roughly **4450 bytes** (4450B). Now suppose we want to split it into chunks of size 600. The result is depicted in Table 4. Note that due to the presence of `OP_IF`'s, we cannot split the program into *exactly* equal parts of size 600B, so the size insignificantly varies.

Table 4: The result of splitting the program into chunks of approximated size $600B$. The second column represents the shard size $|f_j|$, the third column number of elements in z_j and finally, the last column is the estimated cost of singing z_j.

Shard number	Shard Size	# Elements in state	Estimated Commitment Cost
1	623B	37	37kB
2	640B	32	32kB
3	640B	27	27kB
4	640B	22	22kB
5	640B	17	17kB
6	640B	12	12kB
7	627B	3	3kB

Notice an interesting fact: the cost of a single commitment exceeds the cost of the shard itself many times! For example, if we were to build the `DisproveScript` for transition from z_1 to z_2, we would need the script of size $37\text{kB} + 32\text{kB} + 623\text{B} \approx 69.6\text{kB}$! This leads us to the essential conclusion.

Takeaway. *Optimizing the intermediate states representation is crucial for the BitVM2. Even if the program is split into small chunks, the cost of the commitment can still be overwhelming.*

This leads us to the question: can we throw BitVM2 out of the window due to such inefficiency and wait for the `OP_CAT`? The answer is obviously no (for what other reason are we writing this paper?). We can still use BitVM2, but we need to change the way we write the programs. We call such programs **BitVM-friendly functions**. We provide the first example below.

4.2 Square Fibonacci Sequence

Let us consider a toy example of the Square Fibonacci Sequence. Suppose our input is a pair of elements (x_0, x_1) from the field $\mathbb{F}_q$. For the sake of convenience, we choose $\mathbb{F}_q$ to be the prime field of BN254 curve, which is frequently used for zk-SNARKs. Then, our program f_n consists in finding the $(n-1)^{\text{th}}$ element in the sequence:

$$x_{j+2} = x_{j+1}^2 + x_j^2, \text{ over } \mathbb{F}_q.$$

Such function has a very natural decomposition. Suppose our state is described by the tuple (x_j, x_{j+1}). Consider the transition function $\phi : (x_j, x_{j+1}) \mapsto (x_{j+1}, x_j^2 + x_{j+1}^2)$. In this case, our function f_n can be defined as:

$$f_n = \phi^{(n)}(x_0, x_1)[1],$$

where the index $(a, b)[1] = b$ means the second element in the tuple.

Suppose that we have `Fq` implemented in the Bitcoin script. Then, the state transition function can be implemented as:

Script: `FQ::DUP Fq::SQUARE ⟨2⟩ Fq::OP_ROLL Fq::SQUARE Fq::ADD`

The size of this transition is roughly *270 kB* and it requires the storage of 18 elements in the stack, costing additional *18 kB*. So the rough size of `DisproveScript` is **290 kB**, which is a lot, but still manageable. In turn, consider the function f_n, written in Bitcoin script:

Script:

for $i \in \{1, \dots, n\}$ **do**
  `FQ::DUP Fq::SQUARE` $\langle 2 \rangle$ `Fq::ROLL Fq::SQUARE Fq::ADD`
end
`Fq::SWAP Fq::DROP`

For $n = 128$, the size is roughly **35 MB**, which, in contrast, is not at all manageable. However, the decomposition of the function would make roughly n scripts, each of size **290 kB**.

Additionally, notice that regardless of n, the size of the disprove scripts is always the same. Even if we take, say, $n = 10000$, making the direct computation cost roughly 2GB, we would have 10000 disprove transactions, each of size **290 kB**. Moreover, since the cost of storing the disprove scripts in the Taptree is negligible, *it does not matter how many chunks we split the program into.*

5 Takeaways and Future Directions

All in all, we believe that, currently, in order to make BitVM2 practical, the whole Groth16 verifier should be written in the **BitVM-friendly** format. We give an informal definition below.

Definition 3. A function f is called **BitVM-friendly** if:

- It can be split into the shards $f_1, \dots, f_n$ of relatively small size.
- The intermediate states $\{z_j\}_{0 \le j \le n}$ contain a small number of elements, making the commitment cheap enough.

This way, the worst-case disprove script would cost $\max_{1 \le j \le n} (|f_j| + \alpha(|z_j| + |z_{j-1}|))$ for $\alpha \approx 1000$[3]. Note that the number of shards almost does not influence the cost since building the larger tree is typically not a problem.

It is a question, though, whether such BitVM-friendly function exists for all the Groth16 ingredients. However, we believe that many functions can be rewritten in such a way. Take, for example, the big integer multiplication. A great cost of such method is storing the bit decomposition of the number. So, if we have an N-bit integer, the cost of storing the decomposition is roughly αN (currently, this corresponds to N kB). Well, that is a lot, especially for 254-bit long integers, which are currently used in the Groth16 verifier. Moreover, there is no efficient way to split the program to avoid storing the decomposition: you initialize the table at the very beginning and drop at the very end.

So how to fix this? The answer is simple: manually construct the script so that at the end of each shard, the decomposition is dropped and y is, therefore, recovered. Then, at the beginning of the next shard, compute the decomposition again and proceed. Of course, this would result in the significantly larger f, but the thing is that *we do not care about the total size of f, as long as the commitment size with the shard size is small enough.* Informally, we present the Algorithm 3 that implements the double-and-add method in a BitVM-friendly way.

That being said, our future directions are the following:

[3]This constant, after further optimizations, is subject to change.

Algorithm 3: BitVM-friendly double-and-add method

Input : x, y — two u32 integers being multiplied, N — bitsize of y, s — parameter to regulate the number of shards.
Output : Result of the multiplication $x \times y$

for $i \in \{0, \ldots, N/s\}$ **do**
  Start the shard i
  Decompose y into the binary form: $y = (y_0, \ldots, y_{N-1})_2$
  for $j \in \{0, \ldots, s\}$ **do**
    if $y_{is+j} = 1$ **then**
      $r \leftarrow r + t$
    end
    $t \leftarrow 2 \times t$
  end
  Recover y back to the original form: $y \leftarrow \sum_{i=0}^{N-1} y_i 2^i$.
  End shard i
end
Return : Integer r

- Try writing the aforementioned algorithm in the BitVM-friendly way.
- Experiment whether w-width decomposition might make multiplication more friendly.
- Implement the cost-effective version of the architecture (Section 5.3 in [3]).
- Run the architecture with the simple demo function verification on the Bitcoin mainnet.

References

[1] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography*. 2023. URL: https://toc.cryptobook.us/book.pdf.

[2] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". In: (May 2009). URL: http://www.bitcoin.org/bitcoin.pdf.

[3] Linus Robin et al. "BitVM2: Bridging Bitcoin to Second Layers". In: 2024. URL: https://bitvm.org/bitvm_bridge.pdf.

[4] Dmytro Zakharov et al. *Optimizing Big Integer Multiplication on Bitcoin: Introducing w-windowed Approach*. Cryptology ePrint Archive, Paper 2024/1236. 2024. URL: https://eprint.iacr.org/2024/1236.