Generic Optimistic Verifiable Computation on Bitcoin: BitVM2 Approach

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

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

Abstract. Assume some user wants to publicly execute a large program on the Bitcoin chain, but its implementation in the native for Bitcoin scripting language — Bitcoin Script — is larger than 4 megabytes (for example, a zero-knowledge verification logic), making it impossible to publish the complex logic on-chain input. This document is a fast recap of the BitVM2 doc, which proposes optimistic execution verification with fraud proofs in case of incorrectness.

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

Contents

1 Introduction

In the ever-evolving landscape of blockchain technology, the desire to execute increasingly complex and large-scale programs on the Bitcoin[2] network is growing. However, Bitcoin Script, the native programming language of Bitcoin, imposes strict size limits, such as the 4-megabyte cap on transaction inputs, which makes it challenging to implement certain advanced cryptographic proofs, like 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. This method leverages fraud proofs to ensure the validity of executions, offering a robust mechanism for verifying computations, while providing a fail-safe against incorrect or malicious operations. This document provides a concise overview of BitVM2, highlighting its potential to unlock new possibilities for secure and scalable program execution on Bitcoin.

2 Program splitting

Let there be a large program f, which can be described in Bitcoin Script. We want to compute it **on-chain**, i.e., find such an $f(x) = y$. To achieve this, we divide the program into n sub-programs $f_1, \ldots, f_n$ and their corresponding intermediate states $z_1, \ldots, z_n$, such that:

$$\begin{aligned} f_1(x) &= z_1 \\ f_2(z_1) &= z_2 \\ &\cdots \\ f_n(z_{n-1}) &= y \end{aligned} \tag{1}$$

However, the user (referred in BitVM2 as the operator) only needs to prove that the given program f indeed returns y for x, or **give others the opportunity to disprove this fact**. In our case, this means giving challengers the ability to prove that for at least one of the sub-programs statement $f_i(z_{i-1}) \neq z_i$ is true.

3 Assert transaction

To achieve this, the user publishes an **Assert** transaction, which has one output with multiple possible spending scenarios:

1. **PayoutScript** (**LockTime** + signature) — the transaction has passed verification, and the operator can spend the output, thereby confirming the statement $f(x) = y$.
2. **DisproveScript** — 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. p

3.1 `DispoveScript`

`DispoveScript` is part of the MAST tree in a Taproot address, which allows challengers to claim the transaction amount for states z_i, z_{i-1}, and sub-program f_i, is called $\mathsf{DisproveScript}_i$ in BitVM2:

```
// push z_i, z_{i-1} onto the stack
{ z_i     }
{ z_(i-1) }
// compute f(z_{i-1})
{ f_i     }
// ensure that f_i(z_{i-1}) != z_i
OP_EQUAL
OP_NOT
```

In fact, this script does not need a `script_sig`, as with the correct z_i and z_{i-1}, it will always execute successfully. Therefore, to restrict spending capability, a Winternitz signature and Covenants verification are added to the script.

3.1.1 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 1. 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 $\texttt{DisproveScript}_j$ 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:

Stack:	$\langle z_{j-1} \rangle$ `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)[1].

The correct implementation of `OP_WINTERNITZVERIFY` in Bitcoin Script would require the $\mathsf{Enc}(m)$ to split the state into d digit numbers or recover the state from d digit numbers on stack, which without the bitwise operations or `OP_CAT` for larger than 32-bit values is nearly impossible.

3.1.2 Winternitz Signatures in Bitcoin Script

Instead of bitwise operations, the Bitcoin script can use arithmetic ones. Still, this arithmetic 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

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

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.

By defining the $\mathsf{Enc}(m)$ as a domination free function $P(m)$, like described in [1], it is convenient to make $d+1$ the power of two. Therefore, from now on, we set $d+1 = 2^w$ for some $w \in \mathbb{N}$. Which splits m by some number of equal chunks of w bits, and k becomes the sum of n_0 — the number of d-digit numbers from m, and n_1 a checksum (see table 1).

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

w	n_0	n_1	k
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

If chunks are of equal lengths, the recovery from n_0 digits is simply:

$$\begin{aligned} w &:= \log_2(d+1) \\ m &= \sum_{i=0}^{n_0} e_i \cdot 2^{iw} \end{aligned} \tag{2}$$

Where multiplication by powers of two can be implemented in Bitcoin Script with sequence of `OP_DUP` and `OP_ADD` opcodes. So, for example, multiplication of e_j by 2^n in Bitcoin Script is:

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

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 before hand 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:

Stack: $\langle \sigma_j \rangle \underbrace{\texttt{OP\_DUP OP\_HASH}}_{d \text{ times}}$ $\langle e_j \rangle$ `OP_PICK`

Still the upper bound for script size in Bitcoin persists, but 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 $(y_1, \ldots, y_k)$ of the public key pk_m adds 40 bytes to the total script size. Additionaly, for calculating a lookup table for signature verification, $2d \cdot k$ opcodes are used. Further more, for message recovery $2\sum_{i=0}^{n_0} iw$ number of opcodes are added too. Also, note that:

$$2\sum_{i=0}^{n_0} iw = wn_0(n_0+1) \approx wn_0^2 \tag{3}$$

So the total script size, excluding utility opcodes, will be at least $40k + 2dk + wn_0(n_0+1)$ (the sizes for different d can be seen in table 2).

Table 2: Different script sizes depending on the d value per each 32-bit message

d	Public Key Size	Verification Script Size	Recovery Script Size	Total
3	800	120	240	1160
7	560	196	165	921
15	400	300	112	812
31	360	558	105	1023
63	320	1008	90	1418
127	280	1778	70	2128
255	240	3060	48	3348

3.2 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 $\mathsf{DisproveScript}_i$, where $i = \overline{1, n}$, 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.

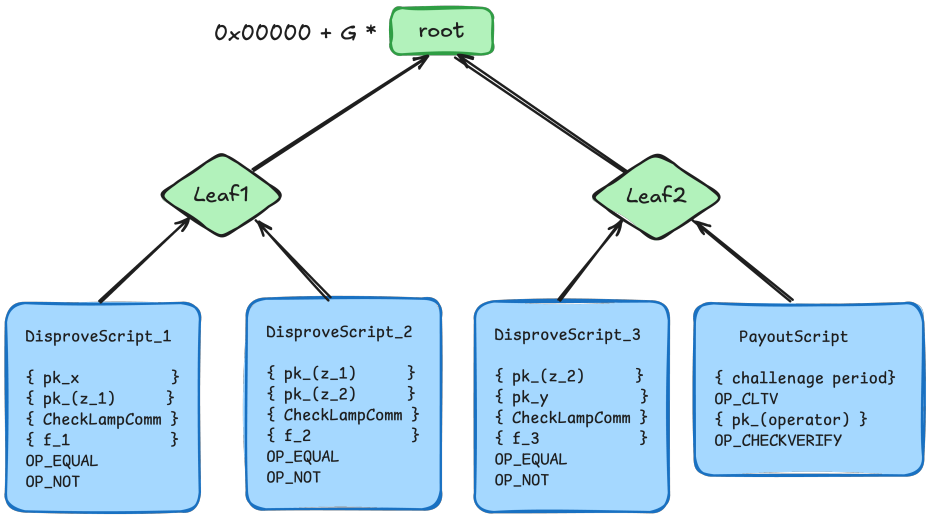

Figure 1: Script tree in a Taproot address with three sub-programs and two intermediate states.

4 `Disprove` and `Payout` transactions

These are transactions that spend the output from `Assert` via `DisproveScript` and `PayoutScript` respectively (see fig. 2). Their structure becomes more important when we consider the emulation of "covenants" through a committee.

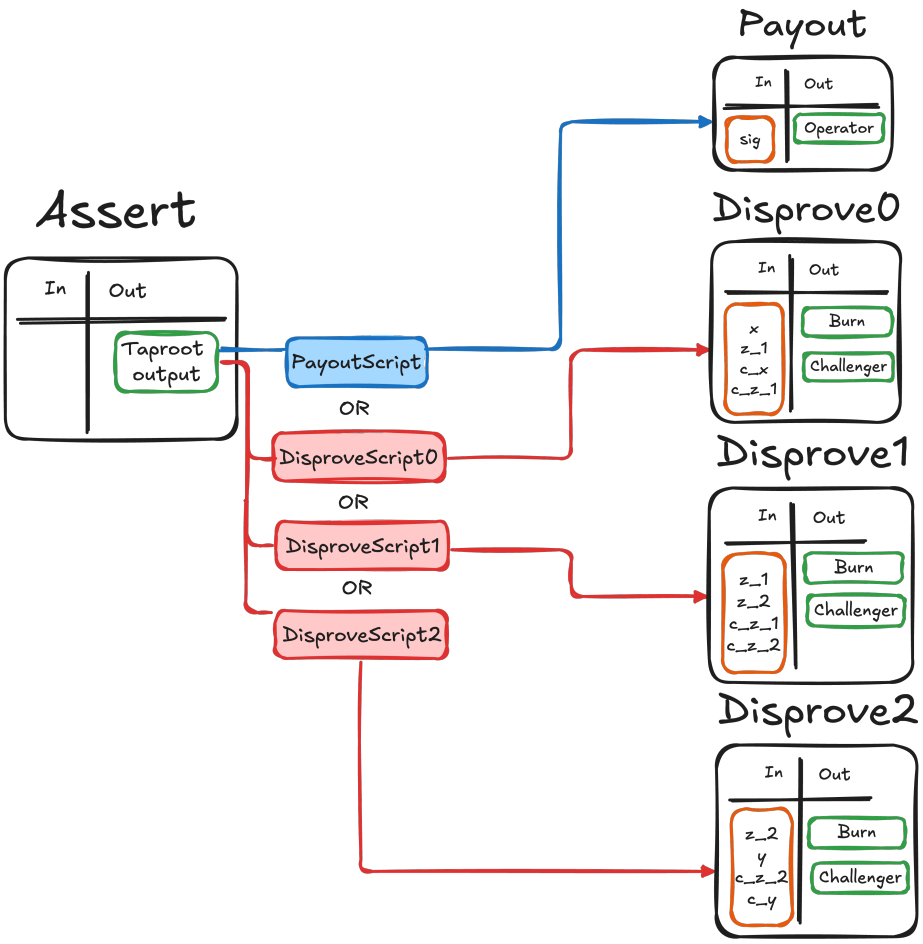

Figure 2: Sequance of transactions in BitVM2 with 3 subprograms and 2 intermediate states.

5 TODO Covenants emulation through comittee

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.