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[1] 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[2] 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.

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 Lamport signature and Covenants verification are added to the script.

3.2 Lamport signature

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

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

Thus, by taking the intermediate states z_i as the message for the Lamport signature, we form one-time key pairs: $(\mathsf{sk}_{z_0}, \mathsf{pk}_{z_0})$, $(\mathsf{sk}_{z_1}, \mathsf{pk}_{z_1})$, ..., $(\mathsf{sk}_{z_{n-1}}, \mathsf{pk}_{z_{n-1}})$ and signatures c_{z_1}, ..., $c_{z_{n-1}}$, respectivly. Then, to spend the output from the `Assert` transaction using the $DispoveScript_i$ script, challenger is required adding the corresponding states z_i, z_{i-1}, and signatures c_{z_i}, $c_{z_{i-1}}$ to the stack in the `script_sig`, making the `script_sig` of the transaction input like this:

```
// push z_i, z_{i-1} onto the stack
{ z_i         }
{ z_(i-1)     }
// push c_{z_i}, c_{z_{i-1}} onto the stack
{ c_(z_i)     }
{ c_(z_(i-1)) }
```

By adding the signatures verification code and public keys to the `DisproveScript`s, they become:

```
// Push the keys corresponding to states z_i, z_{i-1}
{ pk_(z_i)      }
{ pk_(z_(i-1))  }
// Verify the Lamport signature
{ CheckLampComm }
// compute f(z_{i-1})
{ f_i           }
// ensure that f_i(z_{i-1}) != z_i
OP_EQUAL
OP_NOT
```

where, as noted in BitVM2, `CheckLampComm` (or `Lamport.Vrfy`) is the verification of the Lamport signature (commitment), described in Bitcoin Script (as Bitcoin Script does not have a built-in "opcode" for Lamport signatures).

However, in newer implementations, it has been proposed to use the Winternitz signature, which has the same properties but requires fewer "opcodes".

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 $\mathsf{DisproveScript}_i$, 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.

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.

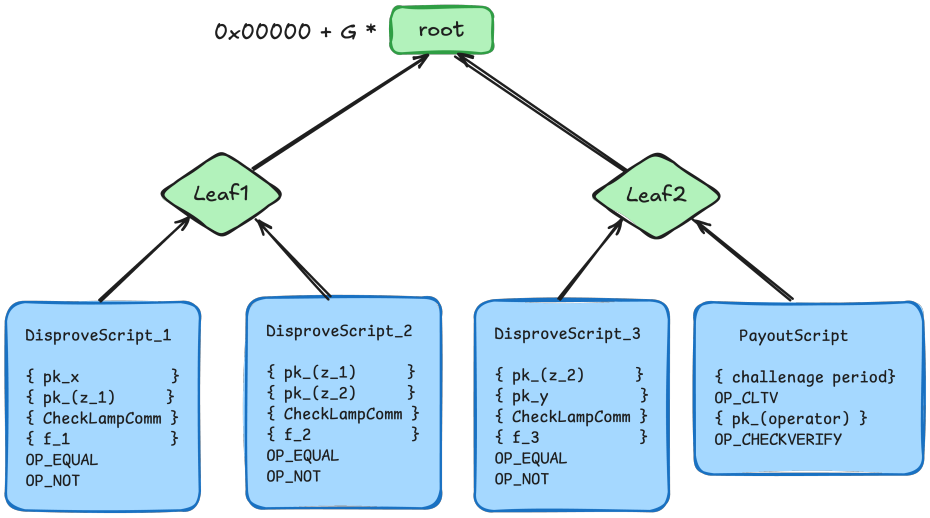

Figure 1: Script tree in a Taproot address with three sub-programs and two intermediate states.

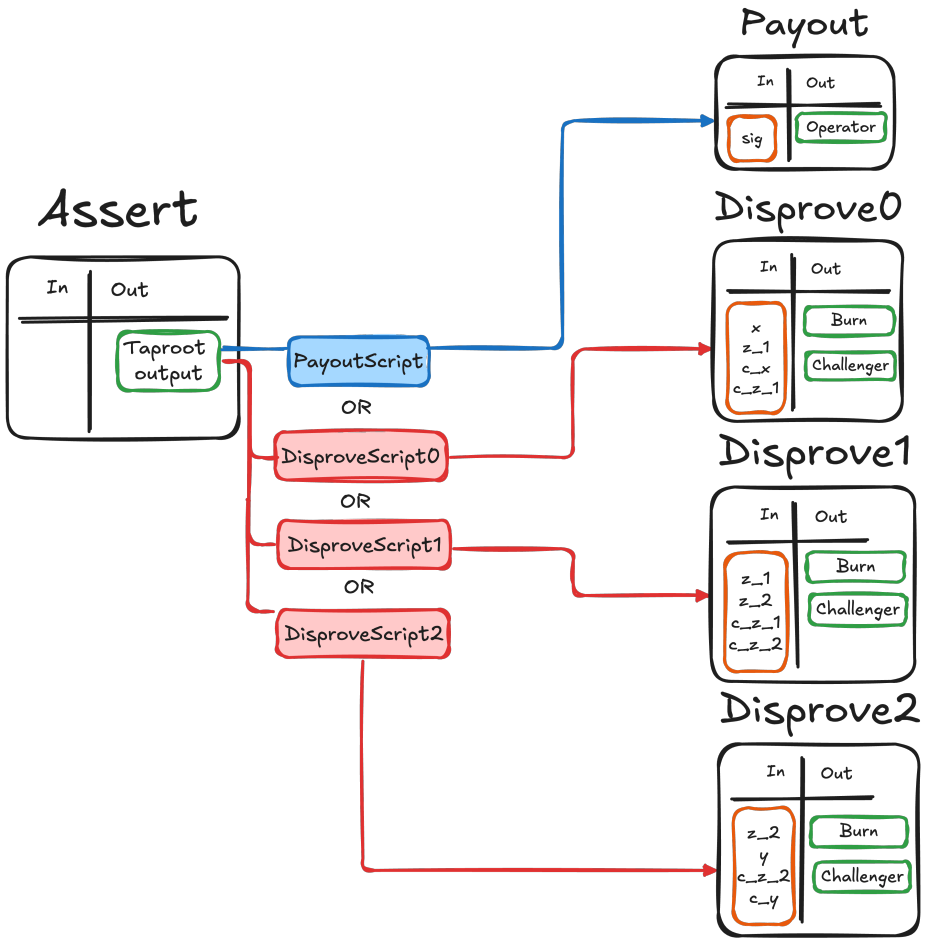

Figure 2: Sequance of transactions in BitVM2 with 3 subprograms and 2 intermediate states.

5 TODO Covenants emulation through comittee

References

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

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