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

1 Introduction **1**

2 Program Split **2**
2.1 Public Verifiable Computation 2
2.2 Motivation for Verifiable Computation on Bitcoin 3
2.3 Implementation on Bitcoin 3

3 Assert Transaction **5**
3.1 Dispove Script 5
3.2 Winternitz Signature 5
3.3 Putting Disprove Transaction Together 8
3.4 Structure of the MAST Tree in a Taproot Address 8

4 Toy Example: Square Fibonacci Sequnce **10**

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 $f_1, \dots, f_n$ 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:

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

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 $f_1, \dots, f_n$ of reasonable size[1].

2. $\mathcal{P}$ executes f on input x shard by shard, obtaining intermediate steps $z_1, \dots, z_n$.

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 $\widetilde{z}_1, \dots, \widetilde{z}_n$.

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 $\left\{ \langle x \rangle \ \langle f_1 \rangle \ \langle f_2 \rangle \right\}$ is the same as calculating composition $f_2 \circ f_1(x)$. So all what remains is finding *valid* $f_1, \dots, f_n$ such that $f = f_1 \parallel f_2 \parallel \cdots \parallel f_n$. All the intermediate steps 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 $z_1, \dots, z_n$
1 Decompose $f$ into shards: $(f_1, \dots, f_n) \leftarrow \mathsf{Decompose}(f)$;
2 for $i \in \{1, \dots, n\}$ do
3     $z_i \leftarrow \mathsf{Exec}(\{\ \langle z_{i-1} \rangle\ \langle f_i \rangle\ \})$;
4 end
Return : $z_1, \dots, z_n$
```

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 is valid. For example, f_j cannot contain unclosed `OP_IF`'s.
- 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. Thus, there should be a balance between two and some clever algorithm to manage this.
- Some of z_j's might contain the same repetitive pieces: for example, the lookup table for certain algorithms. 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 $f = \{\ \langle s_1 \rangle\ \langle s_2 \rangle\ \dots\ \langle s_k \rangle\ \}$ 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 disprove script.

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 $f_{L,1}, \dots, f_{L,k(L)}$ with intermediate states $z_{L,0}, \dots, z_{L,k(L)}$. 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:
$$\hat{L} := \operatorname*{arg\,min}_{0 \le L \le L_{\max}} \left\{ \frac{1}{k(L)} \sum_{0 \le j < k(L)} (\alpha(|z_{L,j}| + |z_{L,j+1}|) + |f_{L,j}|) \right\} .$$

2.3.2 Current State

We implemented the basic splitting mechanism that finds $f_1, \dots, f_k$ 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.

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` — 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 Dispove Script

`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 $\texttt{DisproveScript}_i$ 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).

3.2 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: $f^{(k)}(x) = \underbrace{f \circ \dots \circ f}_{k \text{ times}}(x)$.

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 $(x_1, \dots, x_k) \xleftarrow{R} \mathcal{X}$, while the public key is $(y_1, \dots, y_k)$, where $y_j = H^{(d)}(x_j)$ for each $j \in \{1, \dots, k\}$.
- $\mathsf{Sign}(m, \mathsf{sk})$: denote by $\mathcal{I}_{d,k} := (\{0, \dots, d\})^k$ 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 $e = (e_1, \dots, e_k) \leftarrow \mathsf{Enc}(m)$. Then, the signature is formed as:

$$\sigma \leftarrow (H^{(e_1)}(x_1), H^{(e_2)}(x_2), \dots, H^{(e_k)}(x_k))$$

- $\mathsf{Verify}(\sigma, m, \mathsf{pk})$: to verify $\sigma = (\sigma_1, \dots, \sigma_k)$ on $m \in \mathcal{M}$ and $\mathsf{pk} = (y_1, \dots, y_k)$, first compute encoding $(e_1, \dots, e_k) \leftarrow \mathsf{Enc}(m)$ and then check whether:

$$H^{(d-e_j)}(\sigma_j) = y_j, \quad j \in \{1, \dots, k\}.$$

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].

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.

3.2.1 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 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. So k becomes the sum of n_0 — the number of d-digit numbers from m, and n_1 a checksum (see Table 1).

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

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

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

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

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:

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 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:

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

$\langle e_j \rangle$ `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 20 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:

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

So the total script size, excluding utility opcodes, will be at least $20k + 2dk + wn_0(n_0+1)$ (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	400	120	240	760
7	280	196	165	641
15	200	300	112	612
31	180	558	105	843
63	160	1008	90	1258
127	140	1778	70	1988
255	120	3060	48	3228

3.3 Putting Disprove Transaction Together

All in all, the `DisproveScript`$_j$ is formed as follows:

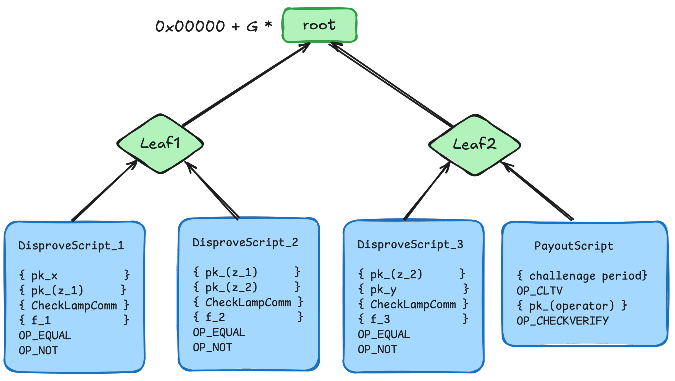
- **Witness:** $\left\{\mathsf{Enc}(z_{j+1}),\ \sigma_{j+1},\ \mathsf{Enc}(z_j),\ \sigma_j\right\}$.
- **Spending Condition**:

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`.

3.4 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 $i \in \{1, \ldots, n\}$, 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.



```
{ pk_x           }
{ pk_(z_1)       }
{ CheckLampComm  }
{ f_1            }
OP_EQUAL
OP_NOT
```

```
{ pk_(z_1)       }
{ pk_(z_2)       }
{ CheckLampComm  }
{ f_2            }
OP_EQUAL
OP_NOT
```

```
{ pk_(z_2)       }
{ pk_y           }
{ CheckLampComm  }
{ f_3            }
OP_EQUAL
OP_NOT
```

```
{ challenage period}
OP_CLTV
{ pk_(operator) }
OP_CHECKVERIFY
```

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

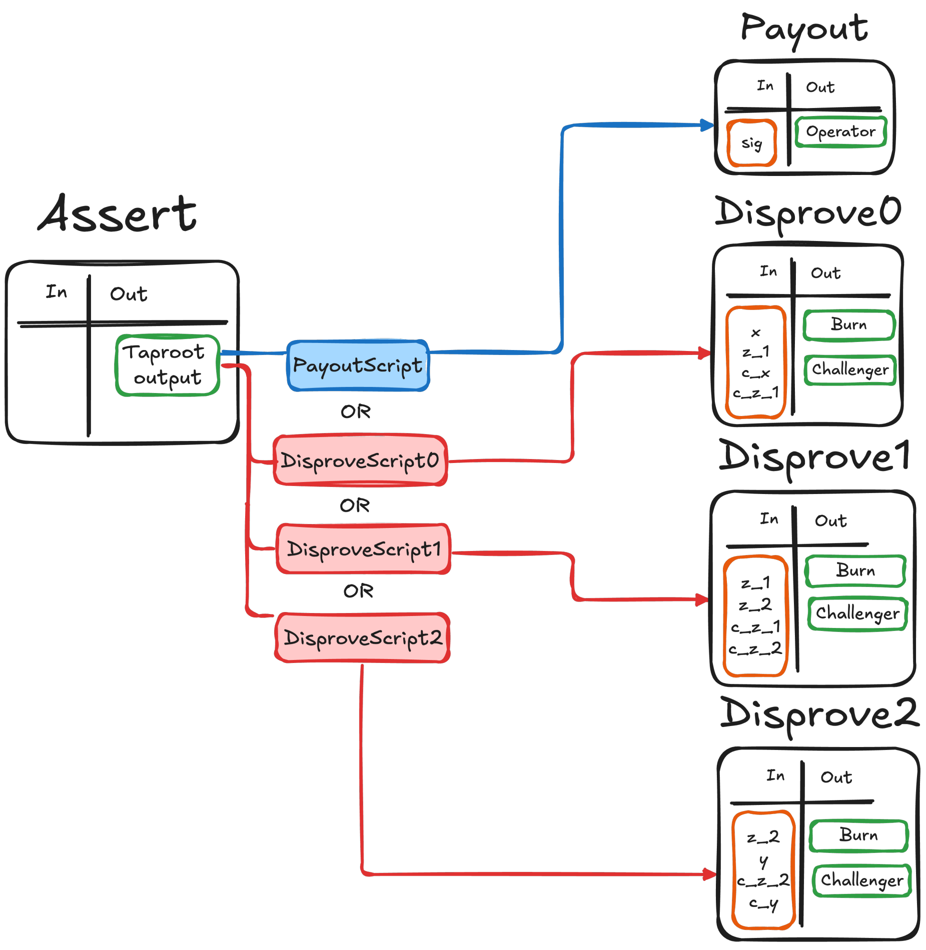


Figure 2: Sequance of transactions in BitVM2 with 3 subprograms and 2 intermediate states.

4 Toy Example: Square Fibonacci Sequnce

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 ⟨2⟩ Fq::OP_ROLL Fq::SQUARE Fq::ADD
end
Fq::SWAP Fq::OP_DROP
```

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

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.