BitVM: Compute Anything on Bitcoin

Robin Linus

robin@zerosync.org

October 9, 2023

Abstract

BitVM is a computing paradigm to express Turing-complete Bitcoin contracts. This requires no changes to the network's consensus rules. Rather than executing computations on Bitcoin, they are merely verified, similarly to optimistic rollups. A prover makes a claim that a given function evaluates for some particular inputs to some specific output. If that claim is false, then the verifier can perform a succinct fraud proof and punish the prover. Using this mechanism, any computable function can be verified on Bitcoin.

Committing to a large program in a Taproot address requires significant amounts of off-chain computation and communication, however the resulting on-chain footprint is minimal. As long as both parties collaborate, they can perform arbitrarily complex, stateful off-chain computation, without leaving any trace in the chain. On-chain execution is required only in case of a dispute.

1 Introduction

By design, the smart contract capabilities of Bitcoin are reduced to basic operations, such as signatures, timelocks, and hashlocks. The BitVM creates a novel design space for more expressive Bitcoin contracts and also off-chain computation. Potential applications include games like Chess, Go, or Poker, and particularly, verification of validity proofs in Bitcoin contracts. Additionally, it might be possible to bridge BTC to foreign chains, build a prediction market, or emulate novel opcodes.

The main drawback of the model proposed here is that it is limited to the two-party setting with a prover and a verifier. Another limitation is that, for both the prover and the verifier, significant amounts of off-chain computation and communication is required to execute programs. However, these issues seem likely to be solved by further research. In this work, we focus solely on the key components of the two-party BitVM.

https://t.me/learningnets

2 Architecture

Similar to Optimistic Rollups[1] and the MATT proposal (Merkelize All The Things)[2], our system is based on fraud proofs and a challenge-response protocol. However, BitVM requires no changes to Bitcoin's consensus rules. The underlying primitives are relatively simple. It's mostly based on hashlocks, timelocks, and large Taproot trees.

The prover commits to the program literally bit-by-bit, however verifying all of that on-chain would be too computationally expensive, so the verifier performs a sequence of carefully crafted challenges to succinctly disprove a false claim of the prover. Prover and verifier jointly pre-sign a sequence of challenge-and-response transactions, which they can later use to resolve any dispute.

The model is designed to simply illustrate that this approach allows for universal computations on Bitcoin. For practical applications we should consider more efficient models.

The protocol is simple: Firstly, prover and verifier compile the program into a huge binary circuit. The prover commits to that circuit in a Taproot address which has a leaf script for every logic gate in the circuit. Additionally, they pre-sign a sequence of transactions, enabling a challenge-response game between the prover and the verifier. Now they have exchanged all of the required data, so they can make their on-chain deposits to the resulting Taproot address.

This activates the contract and they can start exchanging off-chain data to trigger state changes in the circuit. If the prover makes any incorrect claim, the verifier can take their deposit. This guarantees attackers always lose their deposits.

3 Bit Value Commitment

The *bit value commitment* is the most elementary component of the system. It allows the prover to set the value of a particular bit to either "0" or "1". Especially, it allows the prover to set the value of a variable across different Scripts and UTXOs. This is key, as it extends the execution runtime of Bitcoin's VM by splitting it across multiple transactions.

The commitment contains two hashes, *hash0* and *hash1*. At some later point, the prover sets the bit's value either to "0" by revealing *preimage0*, the preimage of *hash0* – or the prover sets the bit's value to "1" by revealing *preimage1*, the preimage of *hash1*. If, at some point, they reveal both preimages *preimage0* and *preimage1*, then the verifier can use them as a fraud proof, and take the prover's deposit. That is called *equivocation*. Being able to punish equivocation is what makes the commitment binding – it is an "incentive-based commitment".

Combining bit value commitments with timelocks allows the verifier to force the prover

https://t.me/learningnets

to decide the value of a particular bit within some given time frame.

Stack Elements

```
1 // Opening this bit commitment to the value "1"
2 <0x47c31e611a3bd2f3a7a42207613046703fa27496>
3 <1>
4
```

Witness Script

```
1  OP_IF
2      OP_HASH160
3      <0xf592e757267b7f307324f1e78b34472f8b6f46f3>
4      OP_EQUALVERIFY
5      <1>
6  OP_ELSE
7      OP_HASH160
8      <0xb157bee96d62f6855392b9920385a834c3113d9a>
9      OP_EQUALVERIFY
10     <0>
11 OP_ENDIF
12
13 // Now the bit value is on the stack
```

Figure 1: A concrete implementation for a 1-bit commitment. To unlock this script, the prover has to reveal either the preimage of hash0 or of hash1. In this example execution, the prover reveals hash1, and sets the bit's value to "1". We can have copies of this commitment to enforce a specific value across different scripts.

For simplicity, from here on, we assume there's an opcode OP_BITCOMMITMENT, which is shorthand for the script above. The opcode consumes two hashes and a preimage of one of the hashes. It puts a bit value on the stack, according to which hash is matched by the preimage.

4 Logic Gate Commitment

Any computable function can be represented as a Boolean circuit. The NAND gate is a universal logic gate, so any Boolean function can be composed from them. To keep our model simple, we show that our method works for simple NAND gates. Additionally, we show how to compose gates arbitrarily. Together this demonstrates BitVM can express any circuit.

The implementation of a NAND gate commitment is simple. It contains two bit commitments representing the two inputs and a third bit commitment representing the output. The Script computes the NAND value of the two inputs to ensure that it matches the committed output bit.

https://t.me/learningnets

```
 1 // Reveal preimage of hash C1 or hash C0
 2 <0xC468A29472CACF3EF179BA2352F88587B91E3E15>
 3 <0x829923B22B9E831822E0A783F92687D27128157B>
 4 OP_BITCOMMITMENT
 5 // Now the bit value of "C" is on the stack
 6 // ... we put it to the altstack for now
 7 OP_TOALTSTACK
 8
 9 // Reveal preimage of hash B1 or hash B0
10 <0x34F0132278E874836DA82F8A6C1E10A21A153D17>
11 <0xF9FCE46CEFE9D9392108480AD42B4CE69557D27D>
12 OP_BITCOMMITMENT
13 // Now there's the bit value of "B" on the stack
14 // ... we put it to the altstack for now
15 OP_TOALTSTACK
16
17 // Reveal preimage of hash A0 or hash A1
18 <0x5ACFDE72A8E37111CBA96D3DD705BA983F47AF4D>
19 <0xA0172816A2D1B20EF0D5A1093958E9564E590BAF>
20 OP_BITCOMMITMENT
21 // Now the bit value of "A" is on the stack
22
23 //
24 // Verify that "A NAND B == C"
25 //
26
27 // Read "B" from altstack
28 OP_FROMALTSTACK
29
30 // Compute "A NAND B"
31 OP_NAND
32
33 // Read "C" from altstack
34 OP_FROMALTSTACK
35 // ... and check that it is correct
36 OP_EQUALVERIFY
```

Figure 2: Logic gate commitment for a NAND operation. Executing this script requires to reveal values for the bit commitments A, B, and C, such that A NAND $B = C$ holds.

(Here, we assume for simplicity, that an opcode for OP_NAND exists. Actually it does not exist, however, it can be easily implemented using OP_BOOLAND and OP_NOT.)

5 Binary Circuit Commitment

In the previous section we defined NAND gate commitments. We can express any circuit by composing gate commitments. Every step of the execution is committed to in a Tapleaf. They're are all combined into the same Taproot address, such that the prover could execute any gate in the circuit. Executing a gate requires the prover to open the corresponding gate commitment and set values for its inputs and output bits.

https://t.me/learningnets

The Taptree might become huge and have a billion Tapleaf Scripts, but its on-chain footprint is minimal.

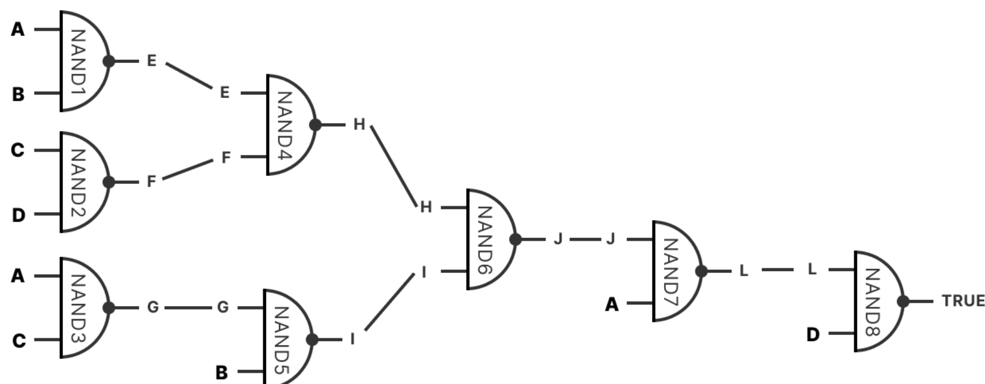

Figure 3: A random example circuit which has 8 different NAND gates, and 4 inputs A,B,C, and D. Using billions of gates would allow us to define basically any function.

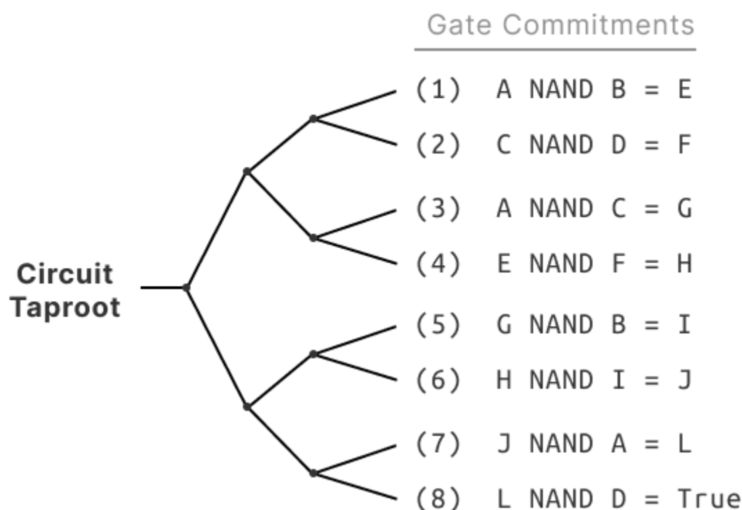

Figure 4: For each gate, the prover's Taproot address contains a leaf script with a corresponding gate commitment. This allows the prover to set the values of the circuit's inputs, (here, A,B,C, and D), at any point later in time.

6 Challenges and Responses

Committing to a circuit is not enough. To disprove an incorrect claim, the verifier has to be able to challenge the prover's statement. This is possible by them pre-signing a sequence of transactions during setup. The transactions are linked like challenge $\rightarrow$ response $\rightarrow$ challenge $\rightarrow$ response $\rightarrow \ldots$. If one of the parties stops engaging then, after

https://t.me/learningnets

some timeout, the other party wins the challenge and can take both deposits. As long as both parties are cooperative, they can jointly settle any contract with a 2-of-2 signature. The following mechanism is required only in case of fraud.

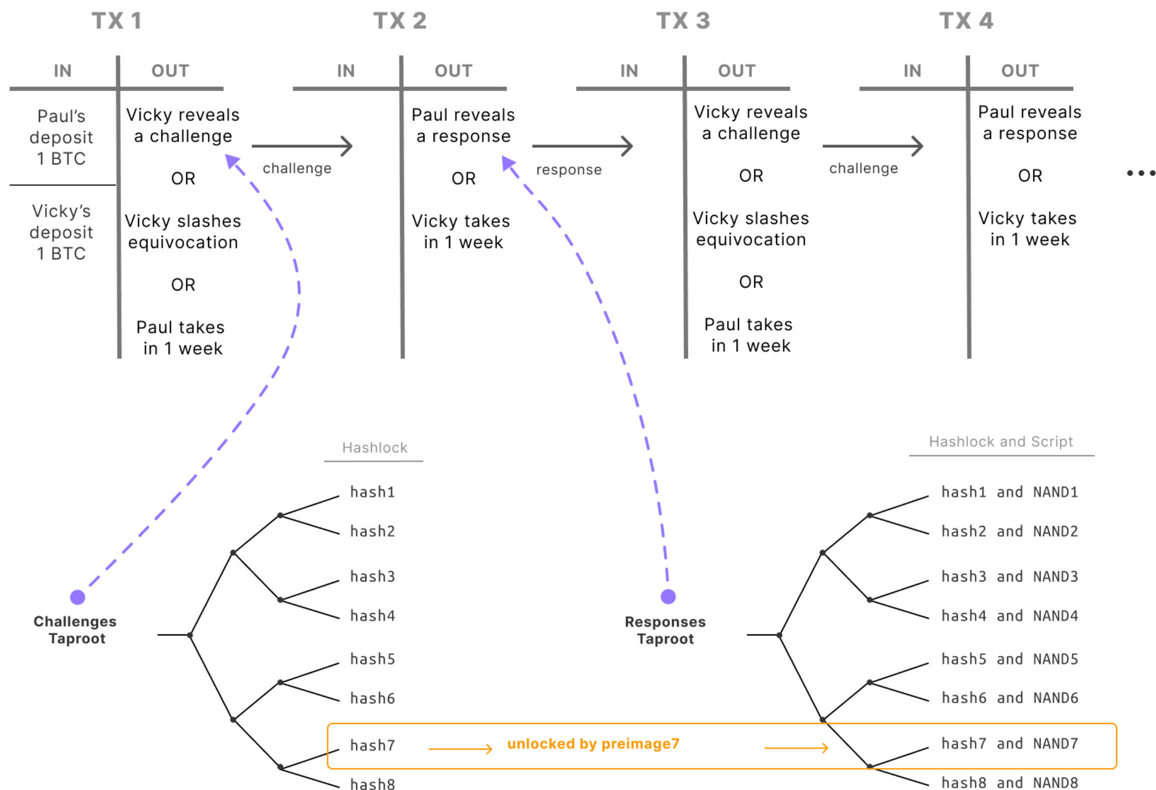

Figure 5: A pre-signed sequence of transactions to perform multiple rounds of challenge-and-response. This sequence is generated during setup.
Vicky chooses a challenge by opening one of the hashlocks in her Tapscript leaves. This unlocks for Paul a specific Tapscript and forces him to execute it. The script forces Paul to reveal the gate commitment challenged by Vicky. Any inconsistent claim can be disproven quickly by repeating this procedure for a few rounds of queries.

If the prover stops collaborating with the verifier off-chain, the verifier needs a way to force his hand on-chain. The verifier does this by unlocking a hashlock: each of the NAND Tapleaves in the prover's UTXO can only be spent if the prover knows a preimage held by the verifier. Therefore, the prover can prove that a given Tapleaf executes correctly by revealing its inputs and outputs, but only if the verifier "unlocks" it for him by revealing the preimage to the hash that guards that Tapleaf. Applying binary search, the verifier can quickly identify the prover's error after just a few rounds of challenge-and-response.

https://t.me/learningnets

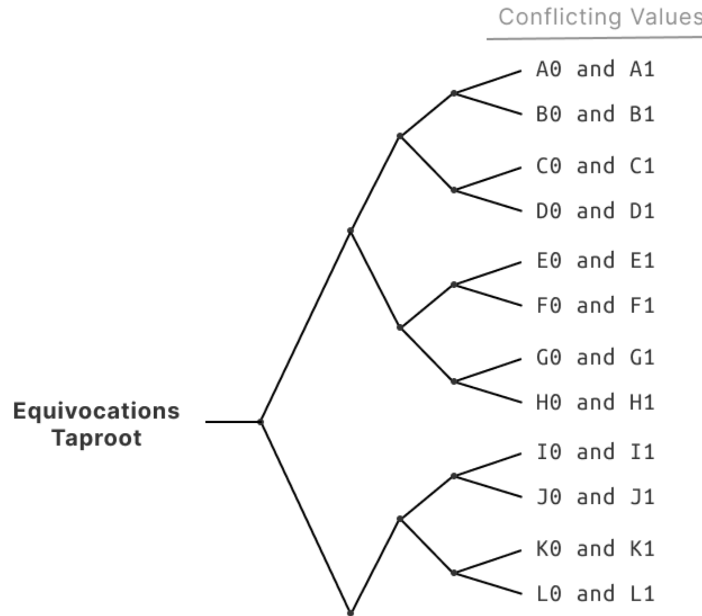

Figure 6: After each response, Vicky can punish equivocation. If Paul ever reveals two conflicting values for a variable, then Vicky immediately wins the challenge and is allowed to take his deposit. Vicky proves Paul's equivocation by revealing for any of his bit commitments both of the preimages.

7 Inputs and Outputs

The prover can set inputs by revealing the corresponding bit commitments. Ideally, they reveal the commitments off-chain to minimize their on-chain footprint. In the non-cooperative case the verifier can force the prover to reveal their inputs on-chain.
It is possible to process large amounts of data by exchanging it upfront, but encrypted. This way the prover can reveal the decryption key at a later point in time.
Multi-party inputs are also possible. Gates can have bit commitments from both parties.

8 Limitations and Outlook

It is inefficient to express functions in simple NAND circuits. Programs can be expressed more efficiently by using more high-level opcodes. E.g., Bitcoin script supports adding 32-bit numbers, so we need no binary circuit for that. We could also have larger bit commitments, e.g. it is possible to commit to 32 bits in a single hash. Additionally, scripts can be up to about 4 MB in size. Thus, we can implement substantially more than a single NAND instruction per leaf script.

The model proposed here is limited to two parties. However, it might be possible to have

https://t.me/learningnets

two-way channels, and chain them to form a network similar to Lightning. Exploring the two-party setting might yield interesting possibilities for generalization. For example, we can explore a 1-to-n star topology for the network. Another research question is if we can apply our model to the n-of-n setting and create more sophisticated channel factories. Furthermore, we could combine our system with different off-chain protocols, e.g., the Lightning Network or rollups.

Other directions of research include cross-application memory, how to make statements about arbitrary data inscribed into the chain, or off-chain programmable circuits, i.e. an off-chain VM. It also might be possible to apply more sophisticated sampling techniques, similar to STARKs, to check a circuit in a single round.

The next major milestone is to complete a design and an implementation of a concrete BitVM and also of *Tree++*, a high-level language to write and debug Bitcoin contracts.

9 Conclusion

Bitcoin is Turing-complete in the sense that encoding fraud proofs in large Taptrees allows to verify the execution of any program. A major constraint of the model outlined here is that it is limited to the two-party setting. Hopefully, this can be generalized in further works.

Acknowledgments

Special thanks to Super Testnet and Sam Parker, who always kept refusing to believe that Bitcoin would not be Turing-complete.

References

[1] Ethereum Research. Optimistic rollups. `https://ethereum.org/en/developers/docs/scaling/optimistic-rollups/`, 2022.

[2] Salvatore Ingala. Merkleize all the things. `https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2022-November/021182.html`, 2022.

Sponsor BitVM developers: `bc1qf5g6z0py2t3t49gupeqrlewga0qz2etalu4xf9`

https://t.me/learningnets