cartesi

Bringing ZK verifiers to Bitcoin using BitVM – ?

Warning: this is all totally beyond bleeding edge and should really not be attempted at home without protective gear and sufficient mental health insurance

cartesi.io

Carsten Munk

8 / 11 / 2023

cartesi

In order to do ZK verification..

(Taking basis from RiscZero terminology)

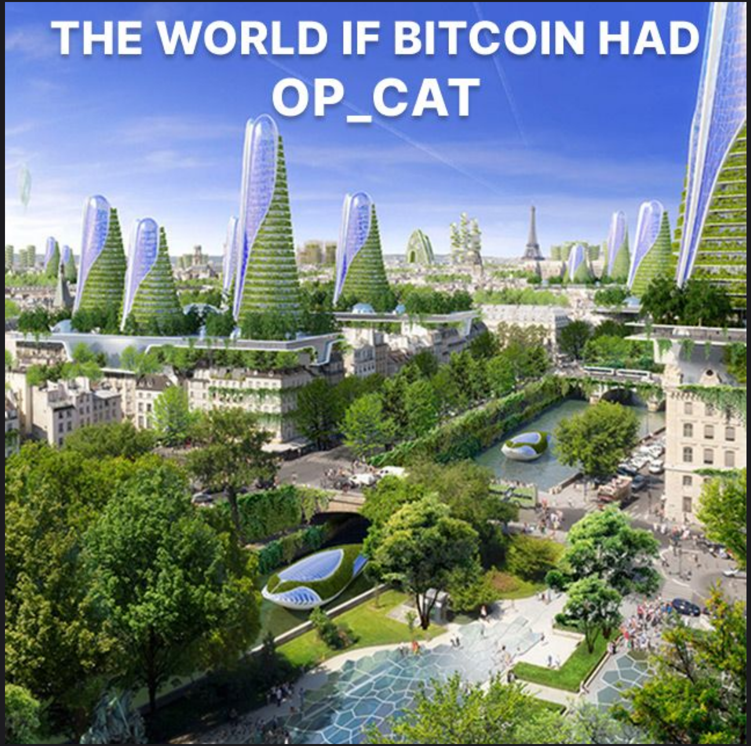
- Input the VM/circuit hash
- Input the proof
- Input the committed data (journal)
- Compute the verification that the journal is proven by the proof as having been done by the circuit/program with certain hash
- Return true/false
- It's probably a larger computation / large amount of instruction steps

Aim of this talk is to give you a baseline to potentially develop ZK verification on Bitcoin and present a few building blocks how it probably can done.

cartesi

Bitcoin Script 101

- Two stacks - main and alternative stack
- Stack contains byte vectors (0 to max 520b in length)
- Stack max 1000 elements
- if/notif/else/endif structures
- Stack operations, the usual + stack depth + pick X element on stack + some specialist ones
- equal/not equals
- No loops/jumps
- 32-bit signed integer arithmetic [4 byte vector] but no 32-bit XOR/AND/INVERT/OR, DIV or MOD or bit shifts
- SHA256, SHA1, RIPEMD160. Some signature checking opcodes.
- But cannot concatenate two byte vectors (disabled opcodes) - OP_CAT - so no merkle tree verification. A lot of "useful" opcodes were disabled by Satoshi Nakomoto in a panic.
- https://en.bitcoin.it/wiki/Script



cartesi

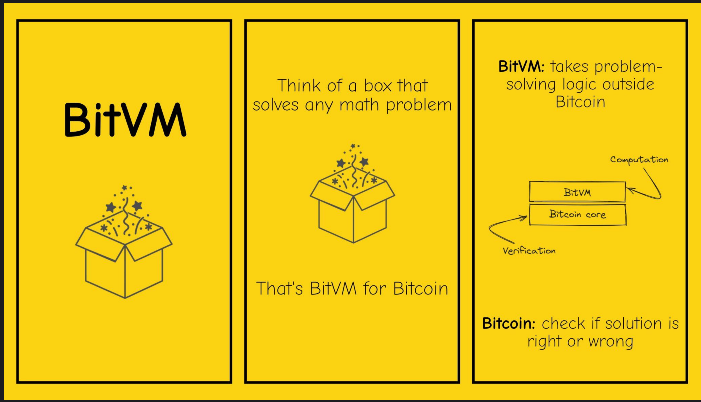
Script addresses

- You can submit a certain input of data to a Bitcoin address and ‘spend’ the value in that address if the script that hashes to that script address allows you (have to send script along on spend)
- Taproot (upgrade to Bitcoin which was deployed) script addresses can contain a merkle tree of scripts and you pick a ‘path’ (merkle proof) at tx submission to pick which script to execute
- It also enables 4MB script sizes
- You can send stack elements in the transaction to be put on the stack when the script executes

cartesi

BitVM

- https://bitvm.org/bitvm.pdf - Introduced October 9, 2023 by Robin Linus from Zerosync
- BitVM introduces three new concepts:
- Bit Commitments (a way to keep a value same across multiple Bitcoin transactions)
- Logic Gate Commitments
- (Binary) Circuit Commitments
- Almost 1000 people in the BitVM telegram group - it has really captured the imagination of many



Bit commitments – the first giant leap

- Enables Write-once memory across multiple transactions
- Why is this important? Cross-transaction state was not previously possible in Bitcoin
- Party which commits to a certain value is penalized if the other commitment value is opened within a certain timeframe
- Henceforth OP_BITCOMMITMENT - 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.

Stack Elements

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

Witness Script

```
OP_IF
    OP_HASH160
    <0xf592e757267b7f307324f1e78b34472f8b6f46f3>
    OP_EQUALVERIFY
    <1>
OP_ELSE
    OP_HASH160
    <0xb157bee96d62f6855392b9920385a834c3113d9a>
    OP_EQUALVERIFY
    <0>
OP_ENDIF

// Now the bit value is on the stack
```

Logic gate commitments

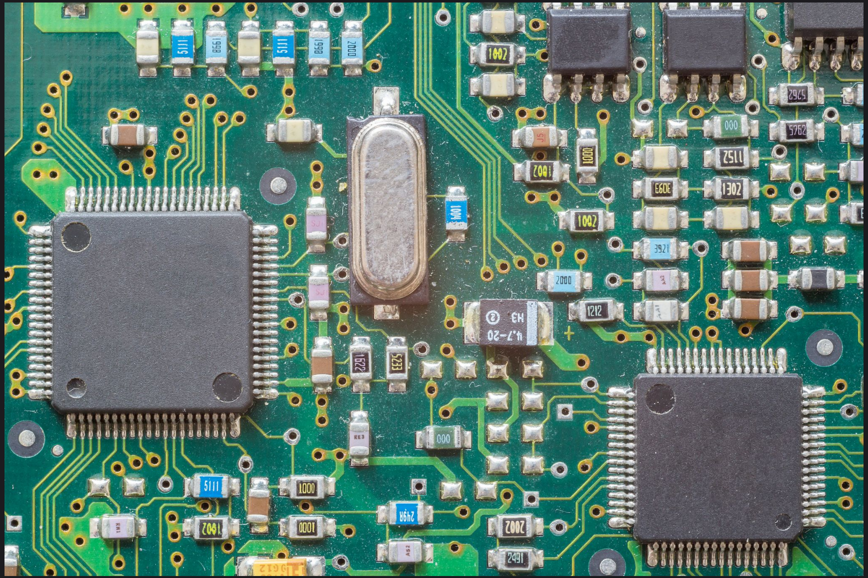
- Executing this script requires to reveal values for the bit commitments A, B, and C, such that A NAND B = C holds
- Beyond NAND, can be
  bitwise AND (OP_BOOLAND)
  OR (OP_BOOLOR)
  XOR (OP_NUMNOTEQUALS)
  NOT (OP_NOT)

```
// Reveal preimage of hash C1 or hash C0
<0xC468A29472CACF3EF179BA2352F88587B91E3E15>
<0x829923B22B9E831822E0A783F92687D27128157B>
OP_BITCOMMITMENT
// Now the bit value of "C" is on the stack
// ... we put it to the altstack for now
OP_TOALTSTACK

// Reveal preimage of hash B1 or hash B0
<0x34F0132278E874836DA82F8A6C1E10A21A153D17>
<0xF9FCE46CEFE9D9392108480AD42B4CE69557D27D>
OP_BITCOMMITMENT
// Now there's the bit value of "B" on the stack
// ... we put it to the altstack for now
OP_TOALTSTACK

// Reveal preimage of hash A0 or hash A1
<0x5ACFDE72A8E37111CBA96D3DD705BA983F47AF4D>
<0xA0172816A2D1B20EF0D5A1093958E9564E590BAF>
OP_BITCOMMITMENT
// Now the bit value of "A" is on the stack

//
// Verify that "A NAND B == C"
//

// Read "B" from altstack
OP_FROMALTSTACK

// Compute "A NAND B"
OP_NAND

// Read "C" from altstack
OP_FROMALTSTACK
// ... and check that it is correct
OP_EQUALVERIFY
```

cartesi

(Binary) circuit commitments

- Combining many gates together, with the circuit having inputs and outputs that are bit commitments (individual gates don't need)
- So you also can execute a circuit with input based on output from another circuit
- Not all need to be logic gates, for example, Bitcoin Script can do 32-bit addition just fine
- Handwriting circuits in Bitcoin Script.. or



Compiling C to circuits to Bitcoin script suitable for bit commitments

- HyCC (https://github.com/stskeeps/HyCC) is old academic software that uses bounded model checking (CBMC tool) to generate boolean circuits from C into various circuit description formats (incl. Bristol Circuits and SCD)
- https://github.com/stskeeps/bristol2btcscript is a script made by me that converts SCD format into Bitcoin Script
- Current Restrictions: amount of inputs, amount of outputs, amount of gate outputs active on stack
- Future: Compile such that single circuit execution/evaluation can extend over multiple bitcoin scripts using bit commitments; cut circuits to fit within scripts.

How do we compile programs to circuits?

1. Eliminate mutation
2. Eliminate branches
3. Unroll loops
4. Inline function calls
5. Functionalize arrays
6. ...

Similar to Bounded Model Checking!

cartesi

The hard way: do all computation on Bitcoin

- By executing circuits and evaluating a true/false bit in the end to allow or disallow a spend.
- How? When setting up initial script address, Prover and verifier signs a number of 2of2 transactions that forces prover to move the value locked in the computation through the computation step transactions. Verifier can be a m-of-m signature, so we rely on one honest party in the set of m verifiers
- At end of computation step, prover is required to wait a week for challenge of abuse of bit commitments, unless verifier agrees already
- Note: if you have everything in same script (big big expensive script) you don't need bit commitments (maybe able to fit ZK verifier in the future?) or to wait
- A verifier can then step in to challenge abuse of bit commitments (different value opened in different scripts) [and ensure prover can't spend the input, but exact mechanism TBA]
- After a week, the prover can spend the value guarded by the script how he pleases if he wasn't caught doing fraud
- **We can now** run some (larger) amount of computation on Bitcoin, implemented with hand written circuits or (bounded loops) C compiled to Bitcoin script! [might be a ZK verifier!]

cartesi

Virtual machine and optimistic proving

- If we can’t run a complete ZK verification within Bitcoin script currently with current constraints, we probably need a virtual machine of sorts made as one or more connected circuits/scripts and then:
- Use optimistic proving: find a particular machine state of disagreement and run a single step of the computation on Bitcoin
- Why do we possibly need ZK? Because of limits on inputs/commitment amounts/etc

cartesi

Machine state merkelization

- As of a few days ago, the BitVM team brought a Blake3 hash function circuit to Bitcoin script, that fits within reasonably computable transaction size
- What does this mean?
- We can now do merkle proof verification in Bitcoin Script!
- We can take the entire state of a virtual machine, put for example every word, or every page, into a merkle tree and prove the transition of:
  - Previous state + a memory write ⇒ new state

```
//
// Input: A 64-byte message in the unlocking script
//
`,
bytesFromText('OP_CAT can be used as a tool to liberate and protect people 🐱'),
`

//----------------------------------------------------------

//
// Program: A Blake3 hash lock
//

`,

// Sanitize the 64-byte message
sanitizeBytes(64),

// Compute Blake3
blake3(),

// Uncomment the following line to inspect the resulting hash
// 'debug;',

// Push the expected hash onto the stack
bytesFromHex('e72f095723bff66ad953e65b64bdf956aeeba11b628d7a44079a78e7dbff2654'),

// Verify the result of Blake3 is the expected hash
u256_equalverify,
```

cartesi

Dispute or not

- Between a prover and a verifier there's really three scenarios:
- Prover and verifier agrees on computation result and no need to prove it on chain (most cases)
- Verifier 'times out' while doing the protest and prover
- Prover does fraud/times out because verifier forces him to prove it and his fraud gets caught
- In case of fraud, or verifier wasting prover's time, a deposit can be slashed

cartesi

Bisection

- Assuming an agreed starting state (virtual machine state hash)
- We agree that the ‘max’ computation cycle as well
- We then conduct a bit of dance: a binary search on-chain where prover posts his belief of the state at a particular cycle of execution and the verifier states his/her agreement or disagreement to that state
- The binary search will then result the state hash where both parties agree and the subsequent state they disagree about, in log2(max steps), so for 3 million steps, that’d be 21 bisections, challenge-and-respond
- This bisection dance can be enforced with 2of2 signed transactions like before

cartesi

Step

- Once we have found an agreed state hash and one which there is disagreement about, we can then run that one single cycle in our “VM” on chain
- Prover sends ‘access logs’ + proofs to the step
- The state accesses (memory) made during the step are checked against the agreed state hash using merkle (multi?) proofs
- The state writes done to the state during the step should then result in a new state hash
- This new state can then be compared against what prover claimed it should be and conclude the winner of the dispute
- This step could be anything that will fit within reasonable Bitcoin computation - and the BitVM team is currently working on a ‘toy’ VM for this to document how this works

cartesi

RISC-V

- An open standard instruction set architecture - royalty-free open source license
- Support in GCC, LLVM/clang, tooling, Rust, etc
- 32-bit and 64-bit variants (RV32, RV64)
- Instruction set variants underneath:
- “I” - 32 registers or “E” - 16 registers
- “F” “D” floating point support
- “M” multiplication and division
- “GC” is “general purpose” – runs Ubuntu
- Super simple to implement for basic instructions, lots of test cases and well written implementations
- Can re-use existing RISC-V compilers for development
- Can re-use existing test cases for the VM (much less time to production)

cartesi

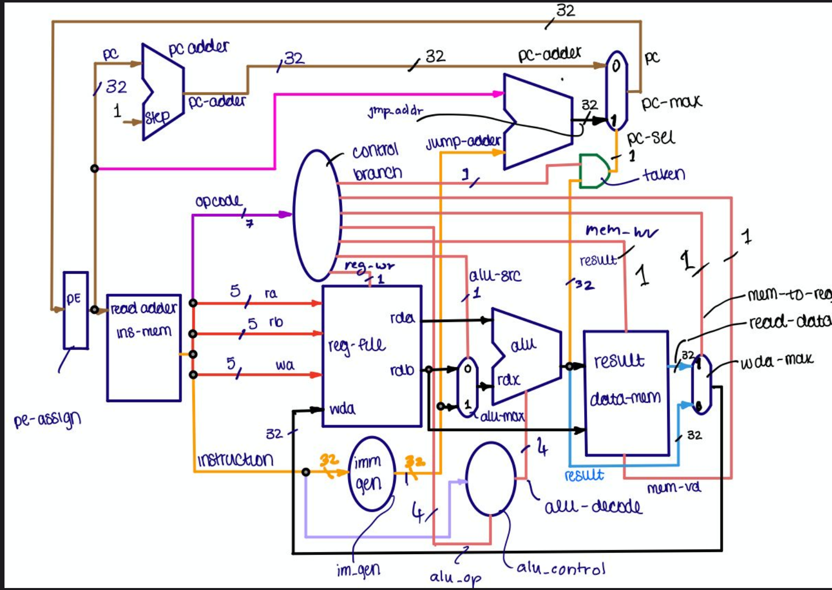
Why Cartesi and BitVM

- Typical reaction of communities in new computing environments is to implement custom VMs, tooling, compilers and it leaves a multi-year trail of dead projects and broken developer experience behind it
- Cartesi provides already now a general purpose RV64GC VM - it runs Ubuntu Linux in a deterministic manner
- Cartesi has a microarchitecture C++ code implementing RV64I that is suitable for circuit implementation
- All Cartesi state is in memory, making it ideal for bisection
- Been around for a few years already, not a new VM
- Bisection already done for Ethereum

cartesi

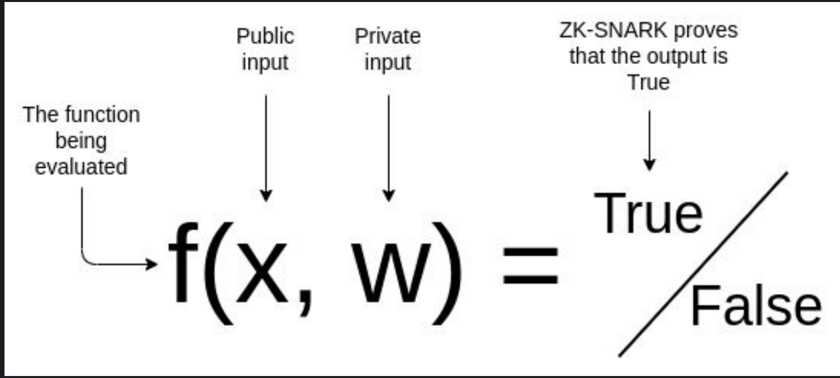
Making a RV64I circuit

- We took the microarch (uarch) implementation from Cartesi Machine and translated it to C & built it with HyCC
- Memory read/writes through access logs
- We managed to run a single RV64I step in 'gates: 74045, depth: 504'
- We then validated the circuit with the -existing- uarch test logs running RV64I tests; and made a script that converted from test JSON to HyCC circuit simulator input style
- **And all microarchitecture tests passed!**
- Next steps are dividing the circuit into fetch/decode/execute/memory access/register writeback steps that can be possibly bisected over on chain



Multiple ways to do a ZK verifier

- Hand-write a boolean circuit converted to bitcoin script; and linking to other bitcoin script 'circuits'
- Compile verifier in C to a boolean circuit a singular Bitcoin script that fits within multiple transactions (requires bit commitment challenge times)
- .. or single transaction (no challenge time), but large transaction
- Run it using BitVM's VM
- Implement a toy VM that emulates certain parts of the ZK verification process and bisect over that
- Base on a (near future?) optimistically proven RV64I VM and compile your ZK verifier to RV64I embedded environment (from Rust/C/C++/etc)
- Compile to RV64GC and run your verifier inside a Linux environment that bisects down to a RV64I step that is proven to (could literally build the RiscZero verifier as is in Rust to it)



cartesi

Things I didn't cover

- Exact structuring of Bitcoin transactions (I'm not an expert)
- How to extend this to multiple verifiers
- Catching lies not liars in optimistic proving: https://arxiv.org/abs/2212.12439
- OP_CAT discussion about potentially added as a softfork to Bitcoin
- Using taproot script as lookup tables

cartesi

More information

- Me: https://t.me/stskeeps / @stskeeps on X
- BitVM paper: https://bitvm.org/bitvm.pdf
- BitVM github: https://github.com/BitVM
- Cartesi Machine: https://docs.cartesi.io/cartesi-machine/
- RV64I circuit work: https://github.com/stskeeps/cartesi-circuit
- BitVM telegram: https://t.me/bitVM_chat