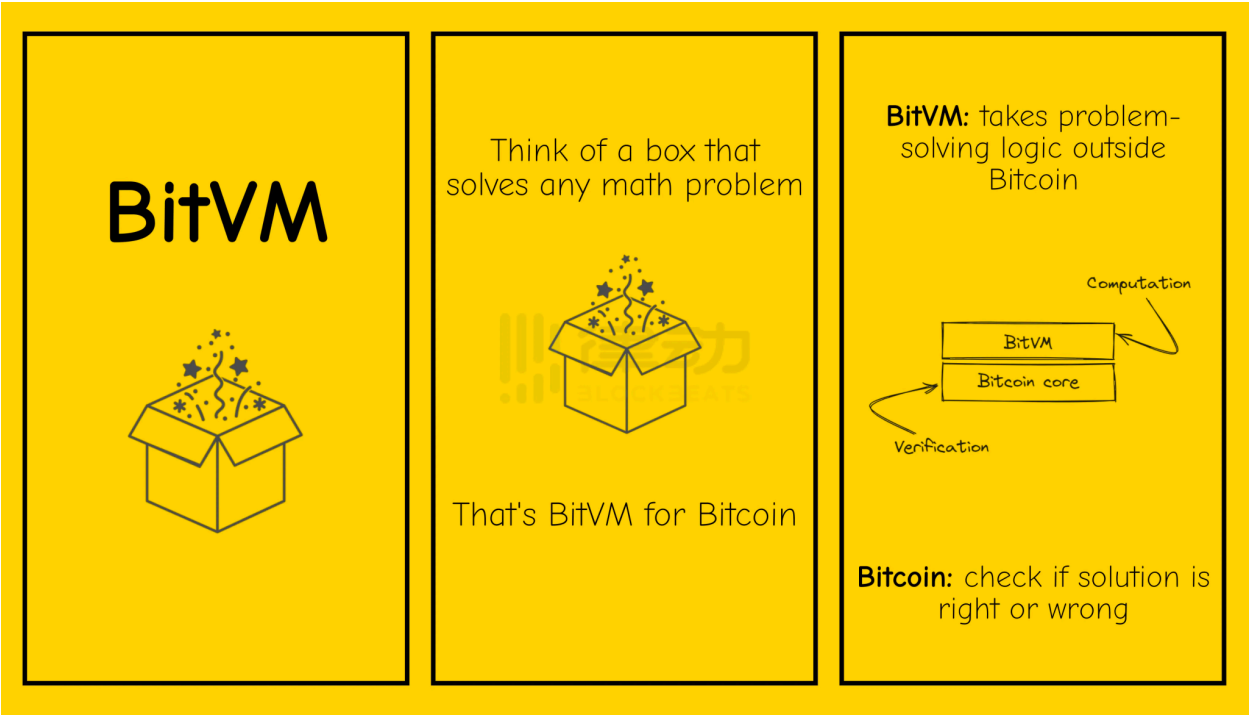
BitVM介绍

2023年10月，BitVM結合了 Optimistic Roll Up、Fraud Proof（欺诈证明）、Taproot Leaf 和 Bitcoin Script 等技术，发布了白皮书，认为比特币也是图灵完备的，具备合约能力。

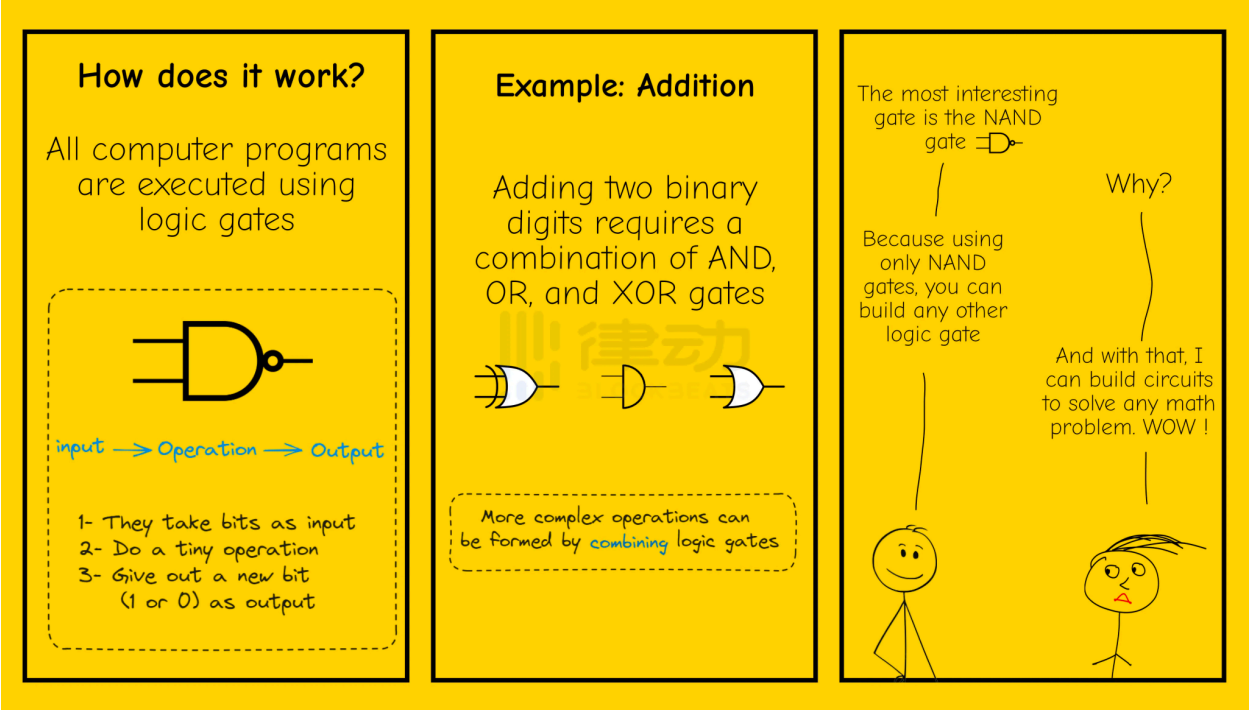
BitVM的比特币扩容方案将计算在链下执行，然后在链上验证，类似于**Optimism**的**Rollup**机制。

白皮书摘要：
BitVM 是一种表达图灵完备比特币合约的计算范式。它们只是进行验证，而不是在比特币上执行计算，类似于乐观汇总（**optimistic rollups**）。证明者（**Prover**）声称给定函数评估某些特定输入到某些特定输出。如果该声明是错误的，那么验证者（**Verifier**）可以进行简洁的欺诈证明并惩罚证明者。使用这种机制，任何可计算的函数都可以在比特币上进行验证。

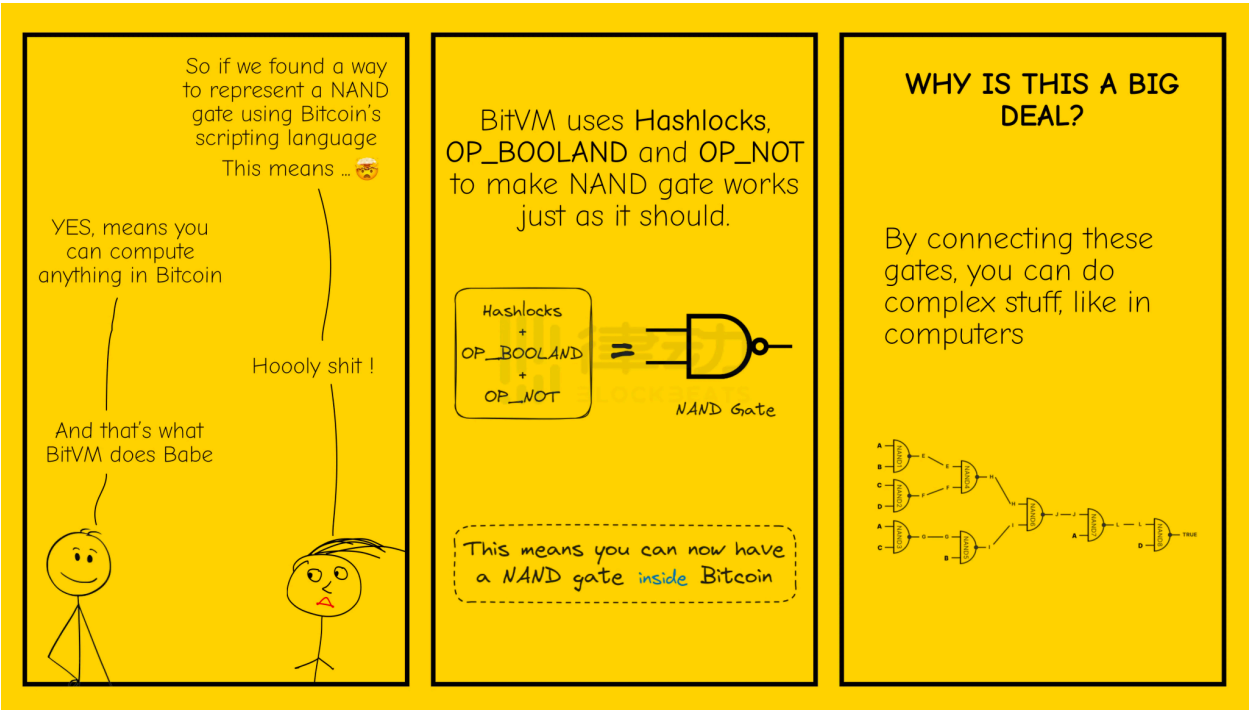
技术概述



设想一个数学问题，BitVM负责运算，bitcoin只负责验证结果是否正确。

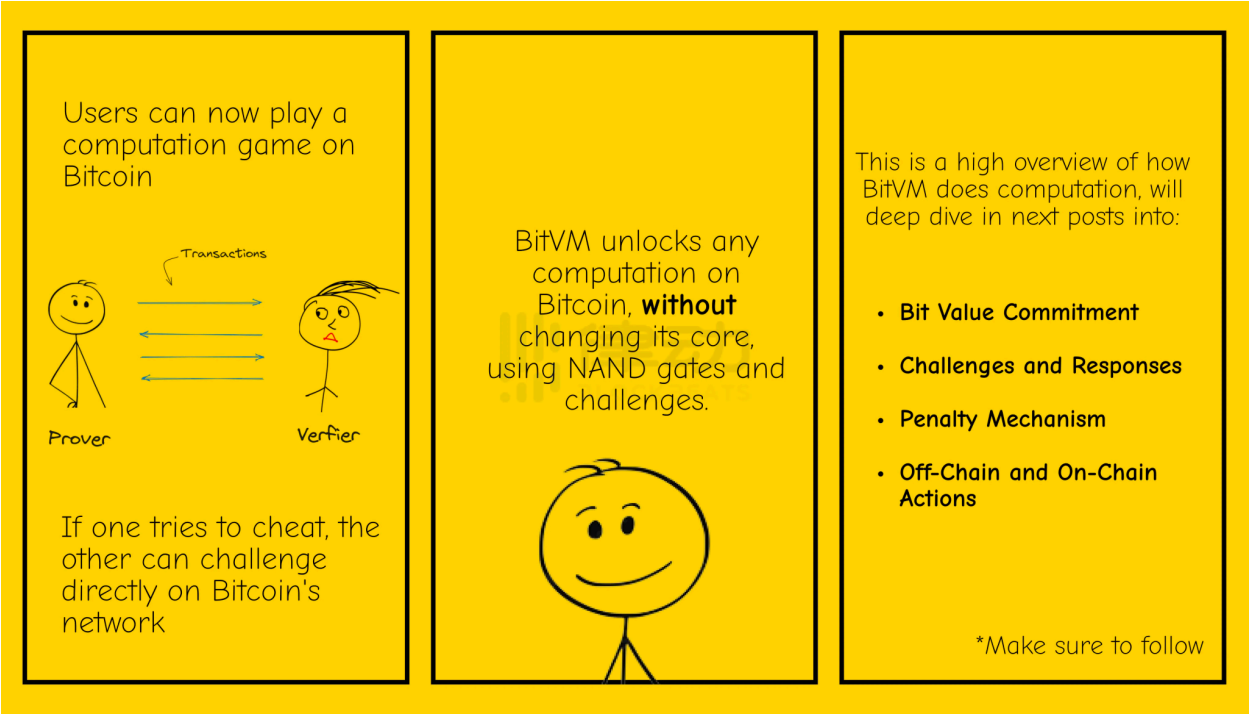


所有的计算机程序运算，都是通过门电路实现的。门包括：与门、或门、非门、异或门、与非门。与非门**NAND**最为有用，因为他可以实现所有其他的门电路。



如果找到一种方式，能够让Bitcoin script language来表达NAND，那么理论上讲，就可以使用Bitcoin实现所有数学运算。

但是，往往一个复杂运算，会用到非常多的门电路，在bitcoin script language上来实现的话，这是一个非常复杂和昂贵的方式。

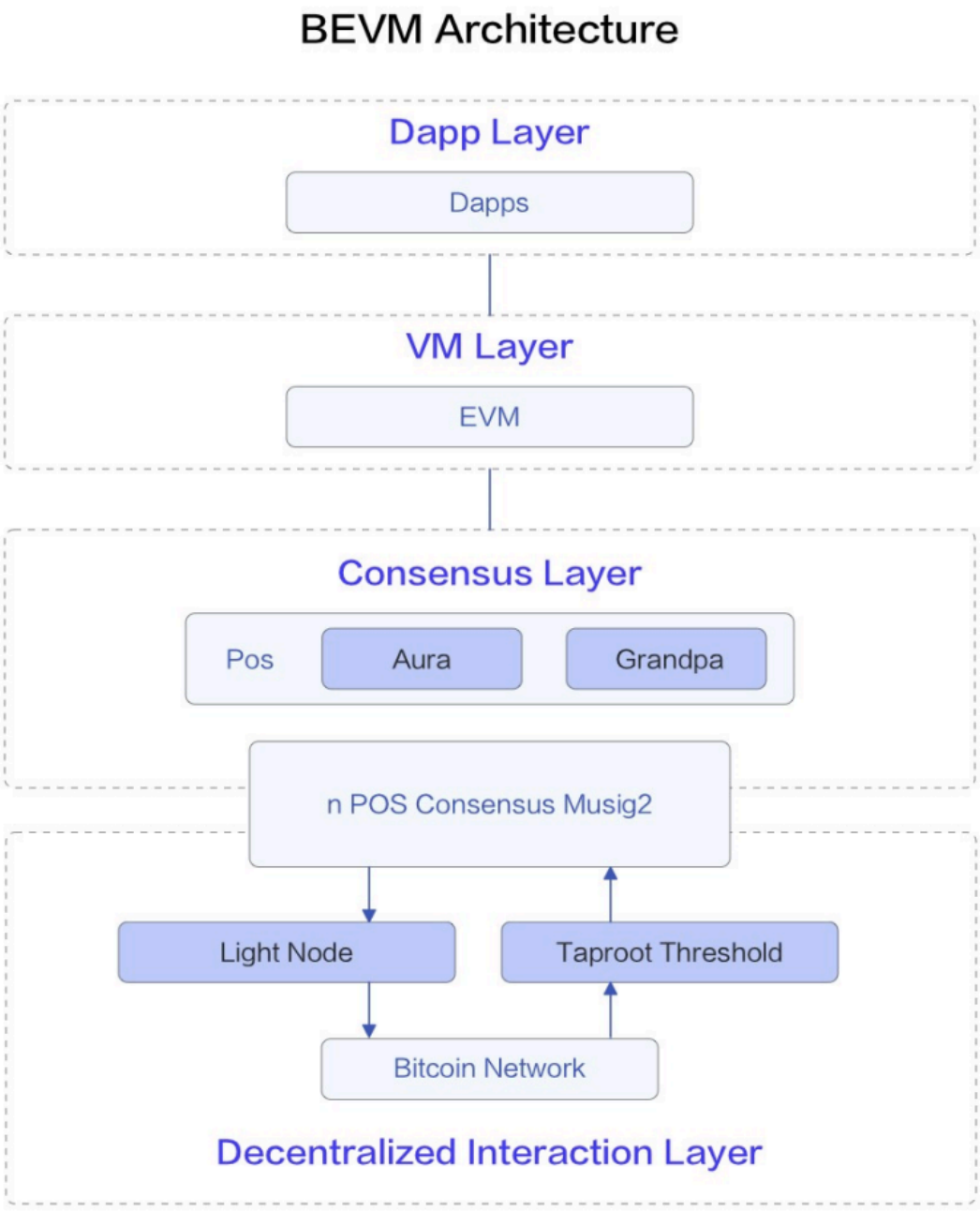


那么用户可以在bitcoin上玩一个计算游戏，证明者和验证者预先签订一系列交易，证明者发布计算结果，验证者验证结果，当证明者发布的数据有误（试图欺诈）时，验证者发起挑战，在bitcoin上进行回放验证，失败者将损失质押的资金。

技术细节

BTC扩容方案

大多数比特币Layer2链都支持智能合约语言（Solidity或Move），可以实现很多的Dapp，产生大量交易。



L1和L2之间，也存在大量交易跨链的需求，需要互相验证交易的真实可靠性。

L2验证L1的交易，通常比较好办，因为L2上的智能合约语言Solidity和Move等智能合约语言功能强大，容易在L2链上验证BTC L1的交易真实性。

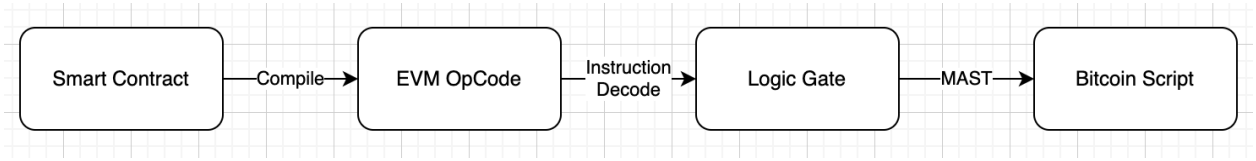
而反之，因为Bitcoin script比较简陋，很难在链上验证L2跨链来的交易真实性，这个也是目前对比特币跨链和扩容最大的掣肘。

BitVM L1验证方案

BTC扩容可以借鉴以太坊的扩容方案，比如Arbitrum。

如果比特币Layer2打算像Arbitrum等以太坊Layer2一样，在Layer1上验证欺诈证明，需要在**BTC**链上直接验证“某笔有争议的交易”或“某个有争议的操作码”。如此一来，就要把Layer2采用的Solidity语言 / EVM对应的操作码，放在比特币链上重新跑一遍。
问题归结为：用**Bitcoin Script**这种比特币**native**的简陋编程语言，实现出**EVM**或其他虚拟机的效果。

所以，从编译原理的角度去理解BitVM方案，它是把EVM / WASM / Javascript操作码，转译为Bitcoin Script的操作码，逻辑门电路作为**“ EVM** 操作码 **——> Bitcoin Script**操作码**”**两者之间的一种中间形态（**IR**）。



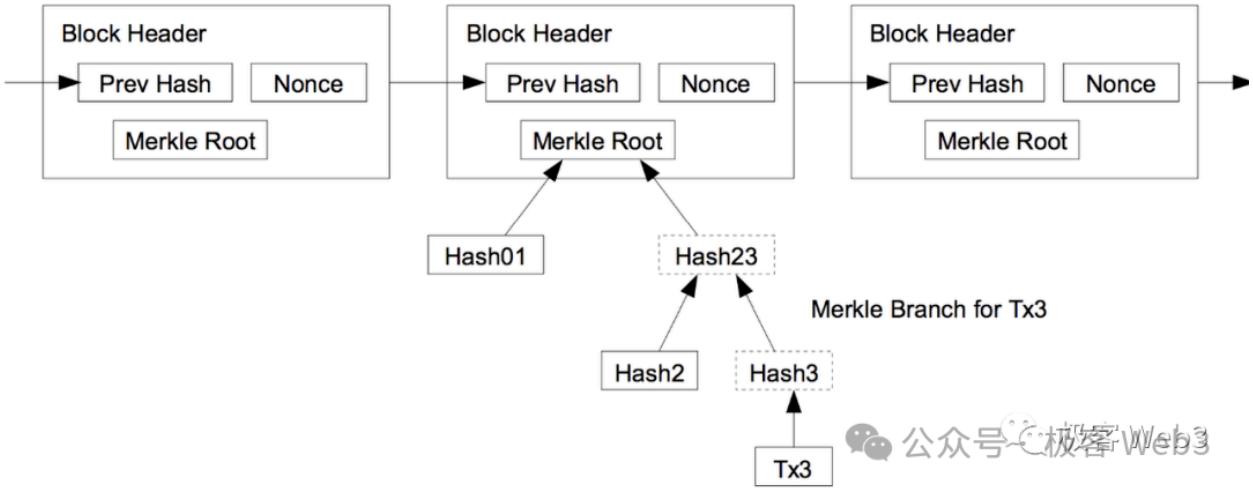
数据上链：

BitVM方案中，链上只存放**Commitment**（承诺 ），不会将Layer2在链下处理的大量交易数据/涉及的巨量逻辑门电路 直接on chain，只在必要时刻将极少数据/逻辑门电路**on chain**。

我们需要某种方式，证明这些“原本在链下，现在要on chain”的数据，不是随手捏造的，这就是密码学中常提到的Commitment。**Merkle Proof**就是**Commitment**的一种。

在Bitcoin中，**MAST**是结合了**Merkle**和**AST**的一种树，可用来实现此目的。

BitVM的方案，尝试把所有的逻辑门电路用比特币脚本表达出来，再组织成一个巨大的**MAST**树，这棵树最底下的叶子**leaf**（**Tx3**），就对应着用比特币脚本实现的逻辑门电路。



所以，无需将完整的MAST树存放在BTC链上，只需要提前披露其Root充当Commitment，在必要时(发送欺诈证明挑战时)出示 数据片段 + Merkle Proof /Branch即可。

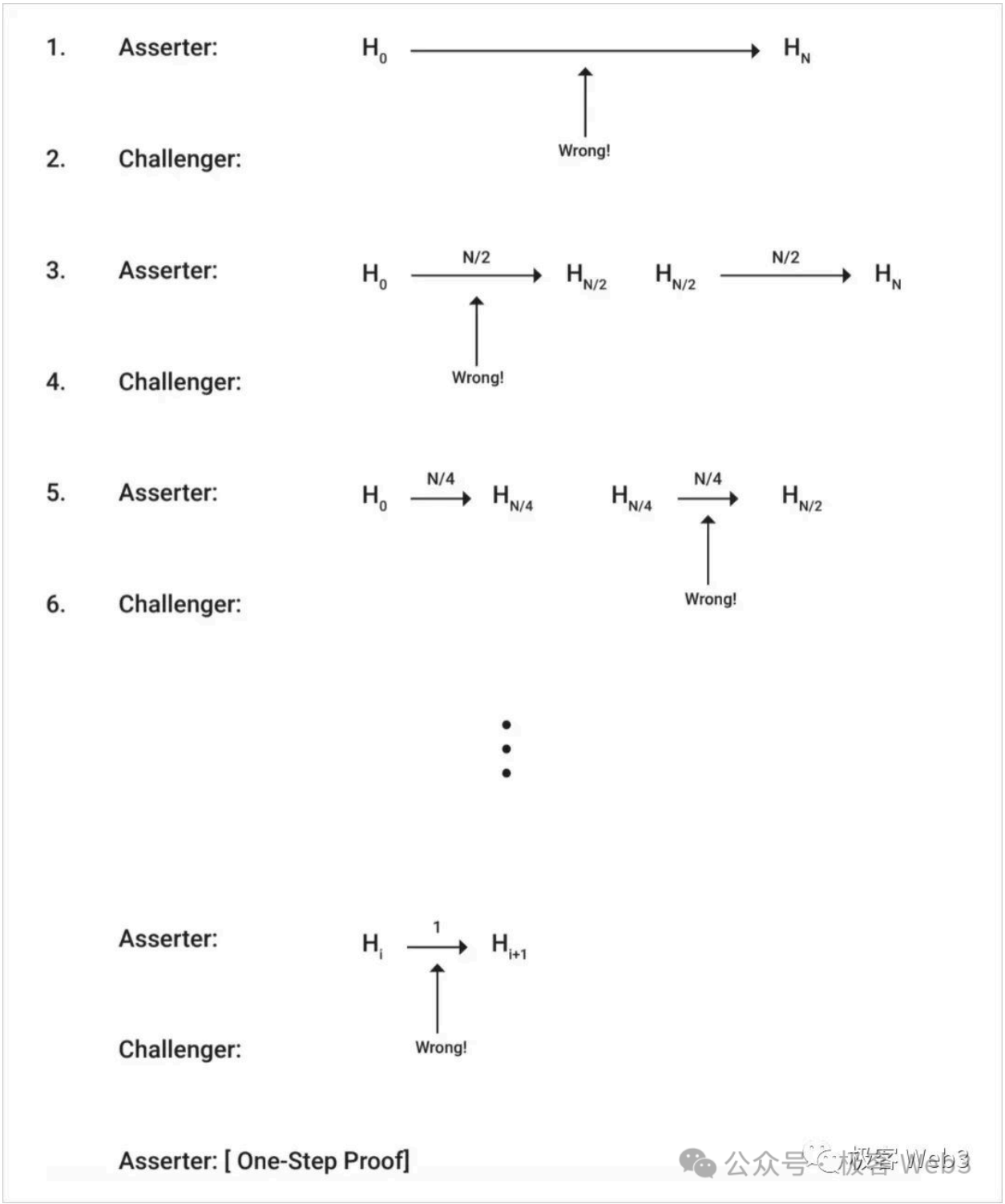
欺诈证明流程：

BitVM白皮书中提到的另一个核心，也就是与Arbitrum高度相似的“交互式欺诈证明”。

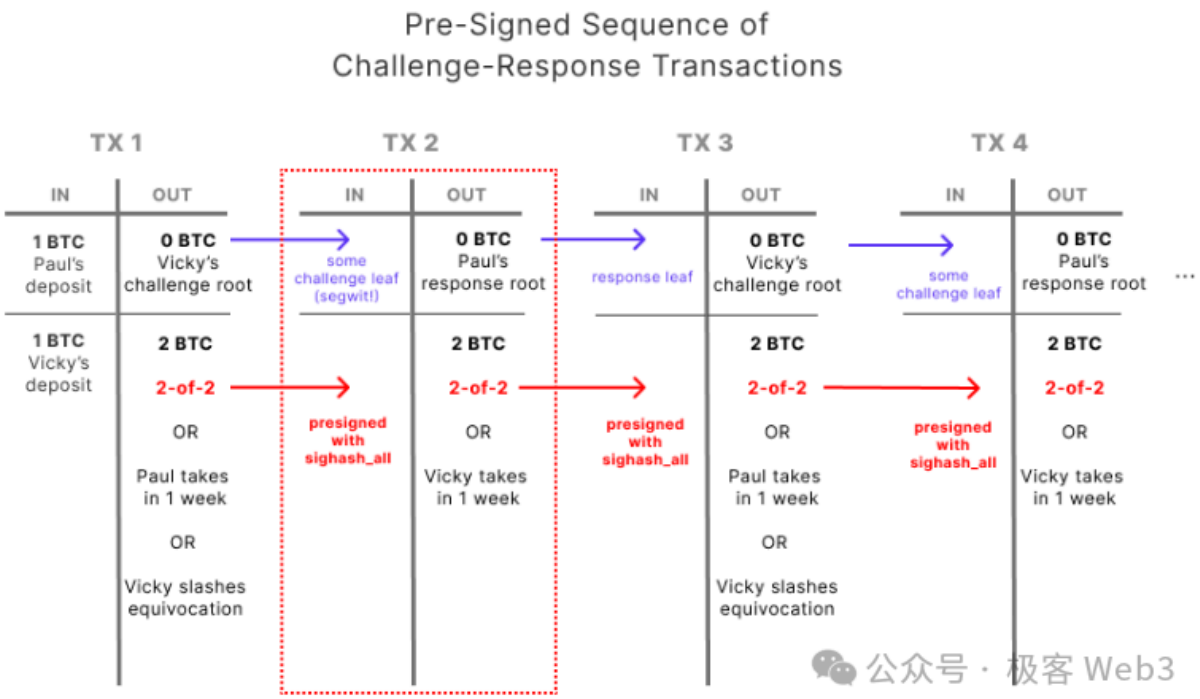
Prover（**Asserter**）会在**Layer1**上发布**assert**断言，声明某些交易数据、状态转换结果，是有效无误的。

如果有人认为Proposer提交的assert断言有问题（关联的数据有误），就会发生争议。此时，Proposer和Challenger会回合式的交换信息，并对有争议的数据进行二分法查找，快速定位到某个粒度极细的操作指令(OP Code)，及其关联的数据片段。

对这个有争议的操作指令（OP Code），需要连带其输入参数在**Layer1**上直接执行，并对输出结果作出验证。在**Arbitrum**里，这被称为**“单步欺诈证明”**。

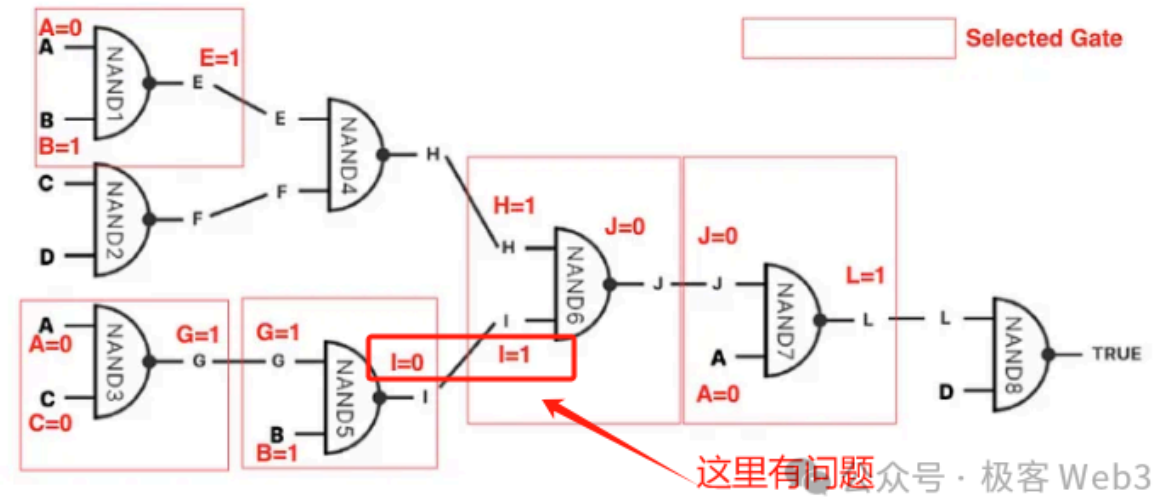


Arbitrum通过以太坊上的合约来实现上述效果，**BitVM**则要借助**Bitcoin Script**实现时间锁、多签等功能。



说明：Paul是Asserter，Vicky是Challenger，TX1 3是Vicky提出挑战，TX2 4是Paul响应，一直循环直到确认最终结果。

当争议发生时，挑战者在BTC链上声明，自己要挑战Proposer发布的哪个Root，然后要求Proposer揭示Root对应的某段数据。之后，Proposer出示默克尔证明，反复在链上披露MAST树的小部分数据片段，直到和挑战者共同定位到有争议的逻辑门电路。之后就可以执行Slash。



优点

1. 不需要bitcoin升级，现有的bitcon就可以支持。

缺点

1. BitVM不如EVM好，BitVM更慢、更昂贵、更复杂。
2. BitVM非常复杂，难以理解和实现。使用一个操作码升级比特币可能会达到相同的效果。
3. BitVM并未解决侧链的无需信任桥接问题（可能需要契约—Covenant）。
4. 应用场景受限，目前参与方只有Prover和Verifier。
5. Optimism 的 Rollup 机制，是通过一个taproot地址控制上百个逻辑门，组合大量地址构建taptree，需要预签名，非常复杂。

目前进度

Things bitvm needs

A todo list for making bitvm great

Implementations

- [ ] Create github repos for more implementations of bitvm
- [ ] E.g. one in rust
- [ ] E.g. one in python
- [ ] E.g. one in typescript
- [ ] Improve my javascript implementation
- [ ] Prototype a bisection protocol -- there are no implementations yet!
- [ ] "Translate" more circuit description formats so that bitvm isn't limited to "only" using bristol formatted circuits
- [ ] E.g. here are some other circuit description formats:
- [ ] “ABY format” (described here, examples here)
- [ ] “Fairplay’s Secure Hardware Definition Language (SHDL)” (examples here)
- [ ] “Simple Circuit Description (SCD)” used by the TinyGarble compiler (described here, I can’t find examples, but according to this document you can compile TinyGarble to find examples in bin/scd/netlists/).
- [ ] "Verilog Netlists" (pictured here)

Programming

- [ ] Develop and document a toolchain for making bitvm applications
- [ ] One possible toolchain: write app in python, export for bitvm using "circuit" and "bfcl" libraries, test/debug/deploy
- [ ] Another possible toolchain: write app in C, export for bitvm using HyCC, test/debug/deploy
- [ ] My current toolchain: find a bristol circuit, test it using a tester I wrote in js, give up if I can't make it work as expected, otherwise convert it to a tapleaf circuit using the procedure outlined here, do a final test, and add a button for it on this site
- [ ] Make a high level programming language that compiles programs down to the "bristol formatted" logical circuits that bitvm currently uses
- [ ] Make an IDE to help developers design apps by "hooking together" logic circuits that already work in bitvm
- [ ] Contribute to the chess app
- [ ] Contribute to the python toolchain