Overview

Topics:

- Smart Contracts
- State representation
- Expressivity (Turing Completeness, Undecidability of Termination)
- Fees for Smart Contract Processing
- Transaction Fee Economics

Making Smart Contracts More Expressive

One of the Ethereum goals is a more expressive smart contract language.

“As expressive as possible”

How expressive is that???

What can be computed?

Theoretical Computer Science has studied the following question:

What are the limits to what can be computed?

Versions of this question:

For which functions f: **N** → **N** does there exist a computer/program P such that for all inputs n in **N,** P(n) always halts and returns f(n)?

For which *partial* functions f: **N** → **N** does there exist a computer/program P such that for all inputs n in **N,** f(n) is defined iff P(n) halts and returns f(n)?

Diagonalisation

Theorem: For every type of program/computational device that can be finitely represented, there are functions f: **N** → **N** that cannot be computed.

Proof:

For any representation scheme, all finitely represented things can be put into 1-1 correspondence with **N.** (Example: lexicographic order on programs.)

List the computable functions f: **N** → **N** as $f_0, f_1, f_2, \ldots$

Define g: **N** → **N** by $g(n) = f_n(n)+1$

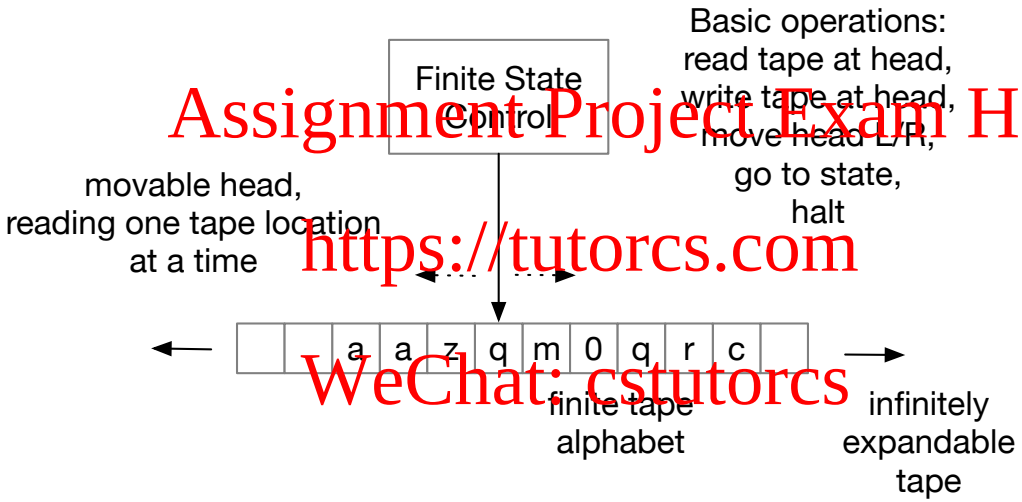
Then, for all n, the function g is not f_n, because $g(n) \neq f_n(n)$. So g is not computable.

QED.

Church-Turing Thesis

So which functions *can* be computed?

THESIS : the computable functions are those that can be computed by a Turing machine:



A programming language is *Turing Complete* if it can express all (partial) functions computable by a Turing machine.

Ethereum Expressivity Objective

Objective: for maximum flexibility of smart contract representation, the language for representing smart contracts should be Turing Complete.

Problem:

- Miners have to run transactions sent to them.

- Turing complete languages can express programs that do not terminate on some inputs.

- An attacker can run a DOS attack against miners by sending transactions that do not terminate, forcing miners to do an infinite amount of computation for no reward.

Question: Can miners defend themselves by testing whether a transaction will terminate, before running it?

Undecidability of the Halting Problem

Answer: No, not without abandoning the objective of supporting all programs in the Turing complete language.

Theorem:

If L is a Turing Complete programming language, then there is no program *Halts* (in L) that does the following:

Input: A program P and input x

Termination: Halts for every possible input

Output: *Yes* if running P(x) halts, *No* otherwise.

Proof: By contradiction, using a variant of diagonalisation. Suppose *Halts* does the job.

Consider program P, taking as input a program Q, defined by

P(Q) = if Halts(Q,Q) then loop-forever else halt

Consider running P with the input P:

Case 1: Assume P(P) halts.

Then by the property of *Halts*, Halts(P,P) returns True.
Hence P(P) loops forever.
Contradiction!

Case 2: Assume P(P) does not halt.

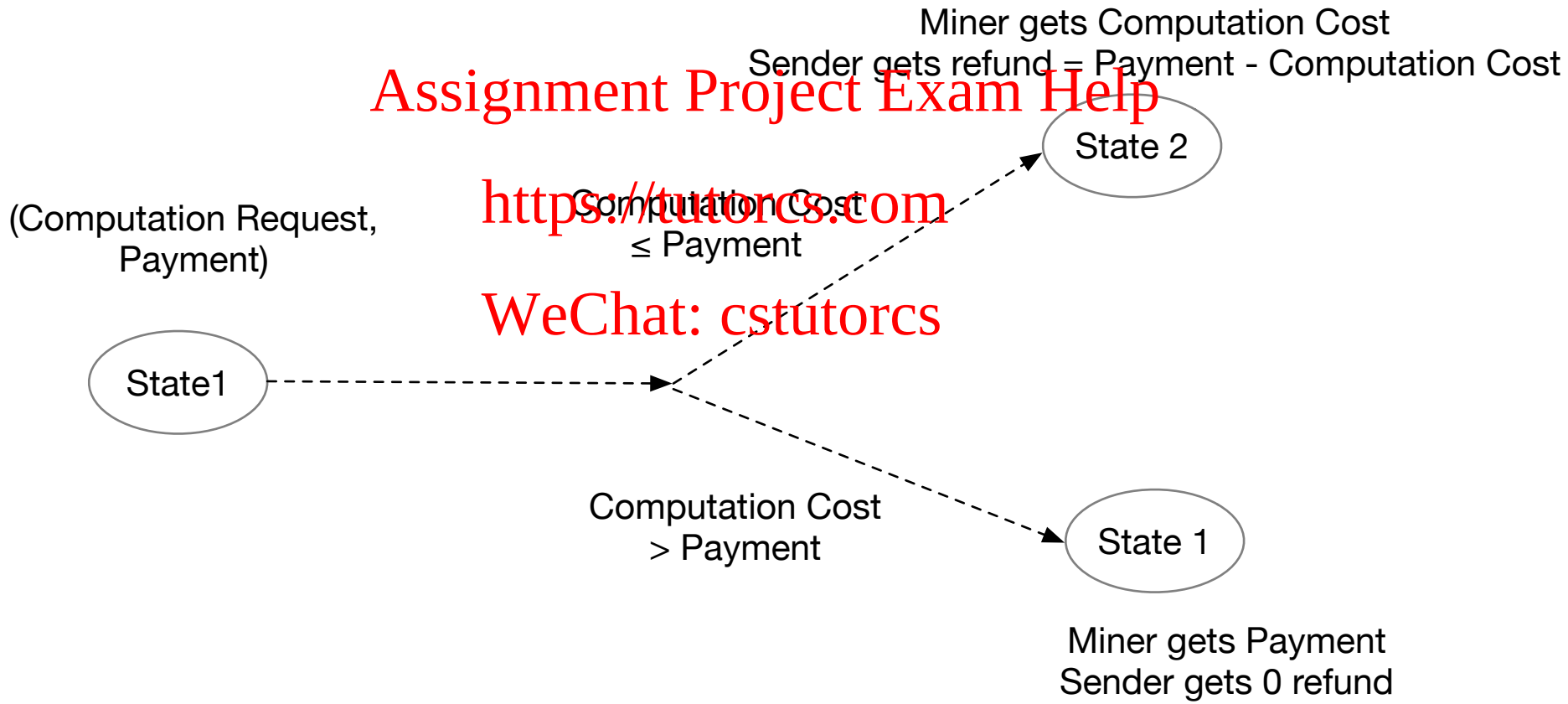
Then by the property of *Halts*, Halts(P,P) returns False.
Hence P(P) halts.
Contradiction!

A contradiction in either case, so the assumption that *Halts* does the job is false.

QED.

Ethereum's solution for infinite loops

Ethereum's solution to the risk of infinite loops and the halting problem is to make a transaction request specify a maximum amount of computation and pay a price per step (more details later).



Attack attempts on transaction processing

How does this scheme handle various types of attacks?

Attack: attacker tries to DOS a miner by sending a transaction with an infinite loop.

Behaviour: The transaction runs up to the computation bound, and is then aborted.

The state reverts to the starting state. But the attacker still has to pay the transaction fee.

(Attacker loses money)

Attack attempts on transaction processing

Attack: Attacker sends a transaction, that runs within the computation bound, but makes the running time long enough to make a miner miss an opportunity to include the transaction in a block because other blocks have come out in the meantime.

Behaviour: The miner can see that the running time could be large enough that it will miss blocks, and can decide not to process the transaction.

(Transaction is never processed.)

Attack attempts on transaction processing

Attack: Attacker tries to create “insecure” or “incorrect” states by sending transactions that terminate, but specifying computation bounds that are smaller than required by the transaction.

Behaviour: The transaction runs part-way, and then the state is reverted to the start state, and the miner collects the transaction fee.

(Attacker loses money).

Handling of Function calls

Turing complete programming languages tend to allow function calls and recursion.

How is the computation bound handled in this case?

Suppose T, called with a computation bound B, calls F at a time when T has already “spent” C.

Then F is run with computation bound B-C.

If F “spends” computation amount D.

On return from F, T continues running with remaining computation bound B-C-D.

Attack attempt: Transaction T runs code that at some stage calls a function F that is owned, or has been taken over, by the attacker. The attacker changes F so that it runs forever.

Behaviour: the original issuer of T specifies a total bound on the computation. F will cause the bound to be exceeded.The state reverts to the starting state, and the miner collects the fee. (Transaction issuer loses the transaction fee for no effect, but the state is not corrupted.)

Ethereum Computation Accounting

Ethereum measures amounts of computation in units called *gas*.

Each basic operation is assigned a cost (an amount of gas)

The total gas used by a computation:

gasUsed = the sum of the costs of each of the operations performed

The caller of a computation specifies:

- a maximum amount of gas they are prepared to pay for
- the price per unit of gas they are offering to pay

Arguments for Ethereum Economic Design

Going to a richer smart contract programming language imposes a cost on miners — they potentially need to do much more computation to verify transactions.

Some transactions require more computation than others.

=> (suggests) =>

We need a way to charge users for the costs of their transactions.

Bitcoin has a fees “market”: users offer a transaction fee when submitting the transaction, miners choose whether to accept the transaction or not.

This idea that there is a market is not quite right:

- The miner contributing the block collects the fee
- *All* miners who validate the block have to pay the cost of validating the transaction!

Transaction Fee Economics

The following analysis in the Ethereum White Paper suggests a lower bound on transaction fees :

Consider a transaction requiring k steps of computation.

Assume that k can be easily determined from the transaction (e.g., it only uses simple for loops).

Suppose there are N miners, and they all have equal processing power

Suppose the sender offers fee kR, where R = per operation payment

Suppose each operation costs a node C and that the cost is the same for all nodes

A miner will process the transaction if the expected reward is greater than the cost. (Profit!)

Cost = kC

Expected reward = fee * probability of winning the block race = kR * (1/N)

I.e., miners accept transaction if kR* (1/N) > kC, i.e., R > NC, i.e.

per operation payment > cost of one operation to the entire network.

This makes sense! (But does suggest that the cost will be higher than a centralised system!)

Limits to this argument

The white paper itself notes the following limitations to this argument :

1. miner of a transaction pays a higher computation cost via delay to block propagation time
2. non-mining nodes exist
3. miners do not actually have equal power
4. miners could create contracts that are cheap for them but impose a cost on others.

The White Paper states these are resolved by imposing a limit on the number of operations per block

Still, a deeper economic analysis of transaction cost charging seems necessary - this is a topic of current research.

Ethereum State Model

Ethereum aims to overcome limitations on Bitcoin that result from the UTXO model.

State = a (partial) map from **Addresses** to **Accounts**

An **Account** is an object with the following fields:

- **nonce***:* a counter used for *replay attack defence*
- **balance:** an amount of Ether (in Wei)
- **contract code:** optional
- **storage:** initially empty

different from the POW nonce!

Two types of accounts:

- **externally owned***:* no code - balance is controlled by a private key held by the account holder
- **contract accounts:** has code - balance controlled by the code reacting to messages sent to it

Transactions

Transactions are messages sent from an externally owned account

They containing the following fields:

- **nonce:** number of transactions sent from this account
- **to:** the *recipient* address
- ***signature:*** identifying the sender (by public key)
- **value:** an *amount* of Ether (in Wei) to transfer from sender to the recipient
- ***data:*** (optional)
- **gasLimit*:*** maximum amount of gas the code invoked may consume
- **gasPrice*:*** fee per unit of gas consumed offered by the sender.
- **init:** (optional) used in *contract creation* transactions to specify contract code

Effectively a transaction is a request from the *sender* to (any) *miner*.

Nonce matching

A transaction **T** can validly be executed only if

T.nonce = (T.sender).nonce

When processing the transaction **(T.sender.).nonce** is incremented

This ensures:

- transactions sent from an external account are processed in the order that they were sent
- no transaction can be executed more than once

The nonce of a contract account records the number of contract creation operations performed by the contract.

Contract creation

Contracts are created using transactions that are

- sent to a **null** address
- have **init** set to the contract code

The contract is stored at an address that depends on

- the address of the sender, and
- the **nonce** of the sender (which is ≥ the number of contracts created from the address)

Messages

Messages are like transactions, but sent from a contract account to another contract account

Fields:

- **sender:** address of contract account from which the call is made
- **transaction originator:** address of sender of the original transaction causing this call
- **to:** The recipient
- **value:** amount of Ether to transfer
- **data:** (optional)
- **startGas:** available gas
- **gasPrice**

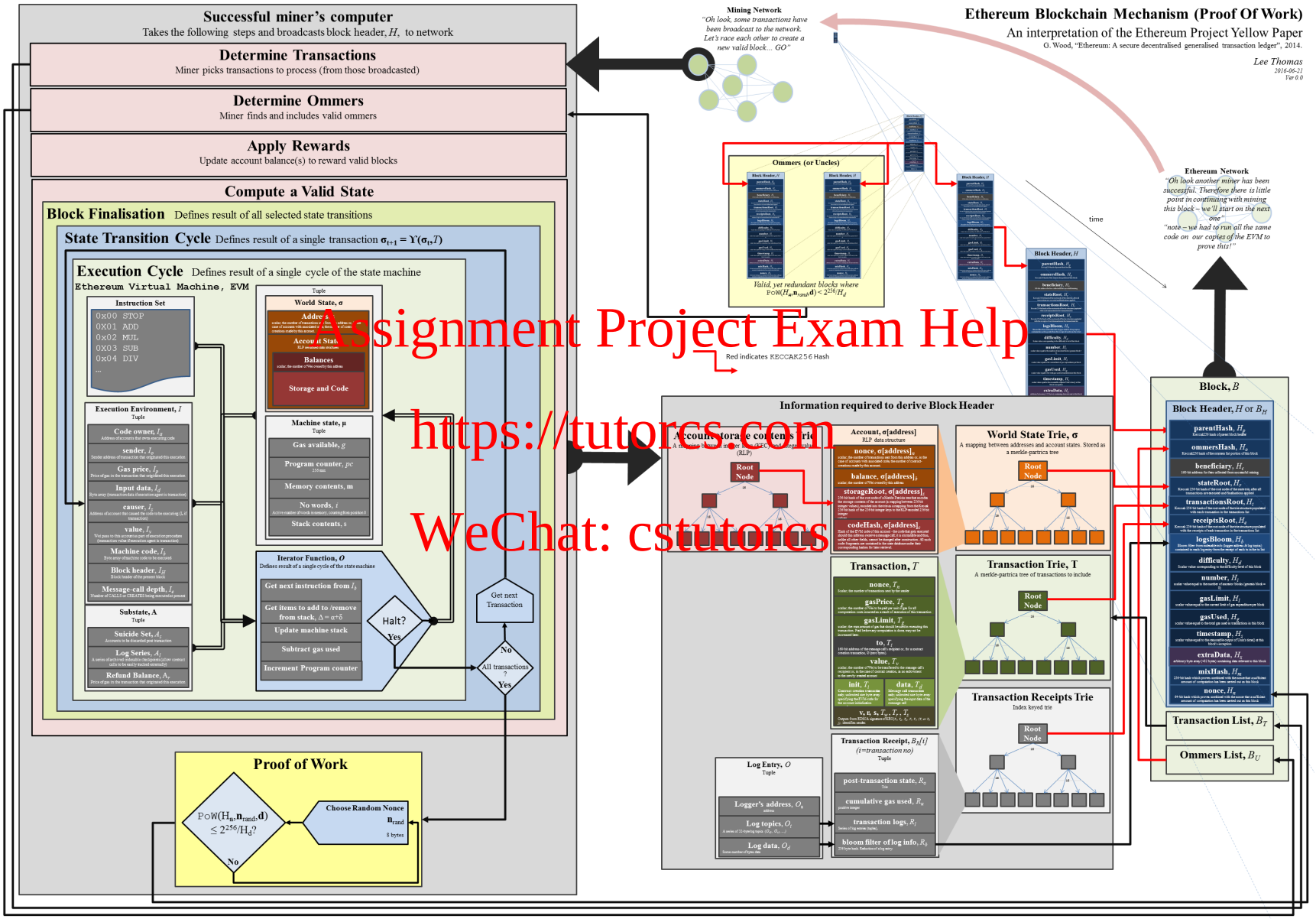
This is more like a subroutine call executed by a miner than a message from a user to a miner.

Messages calls can produce an output (Transactions do not)

Information in an Ethereum Block

- hash of parent block
- ommers included
- transactions included
- (hash of merkle-like tree of) world state:
  state of each active account after all transactions processed
- transaction receipts: outcome of each transaction
  (e.g., did it run out of gas, gas consumed, what information did it log)
- nonce
- block difficulty
- address of miner
- block gas limit
- amount of gas consumed in block (protects miners against computation DOS)

Picture of Ethereum miner process https://i.redd.it/vko4yn9gqopx.png



Ethereum Releases and Hard Forks

Ethereum Foundation has released a number of hard forks in order to improve the performance of the chain, add functionality and thwart attacks resulting from poor design choices.

Mostly the user community has universally adopted the hard forks, with one major exception

Olympic (May 2015) Developer testnet

Frontier (July 2015) First public release of Ethereum

Homestead (March 2016) First "stable" release

DAO Hack response (July 2016) - (details later)
this split Ethereum into two - the original chain, now called Ethereum Classic, still runs.

Tangerine Whistle (Oct 2016) in response to a DOS attack using faulty transaction cost accounting - change of accounting rule

Spurious Dragon (Nov 2016) to thwart an "empty transaction" memory bloat attack

Metropolis Byzantium (Oct 2017) various technical improvements, block reward reduced

Constantinople + St Petersburg (Feb 2019) new instructions, difficulty bomb adjustment

Istanbul (Dec 2019) accounting changes to enable zero-knowledge proofs, block spam attacks

Muir Glacier (Jan 2020) Difficulty Bomb delay

Berlin (Apr 2021) accounting changes, new transaction types

Summary

- Smart Contracts
- Expressivity (Turing Completeness, Undecidability of Termination)
- Fees for Smart Contract Processing
- Transaction Fee Economics
- Transactions and Messages
- State representation