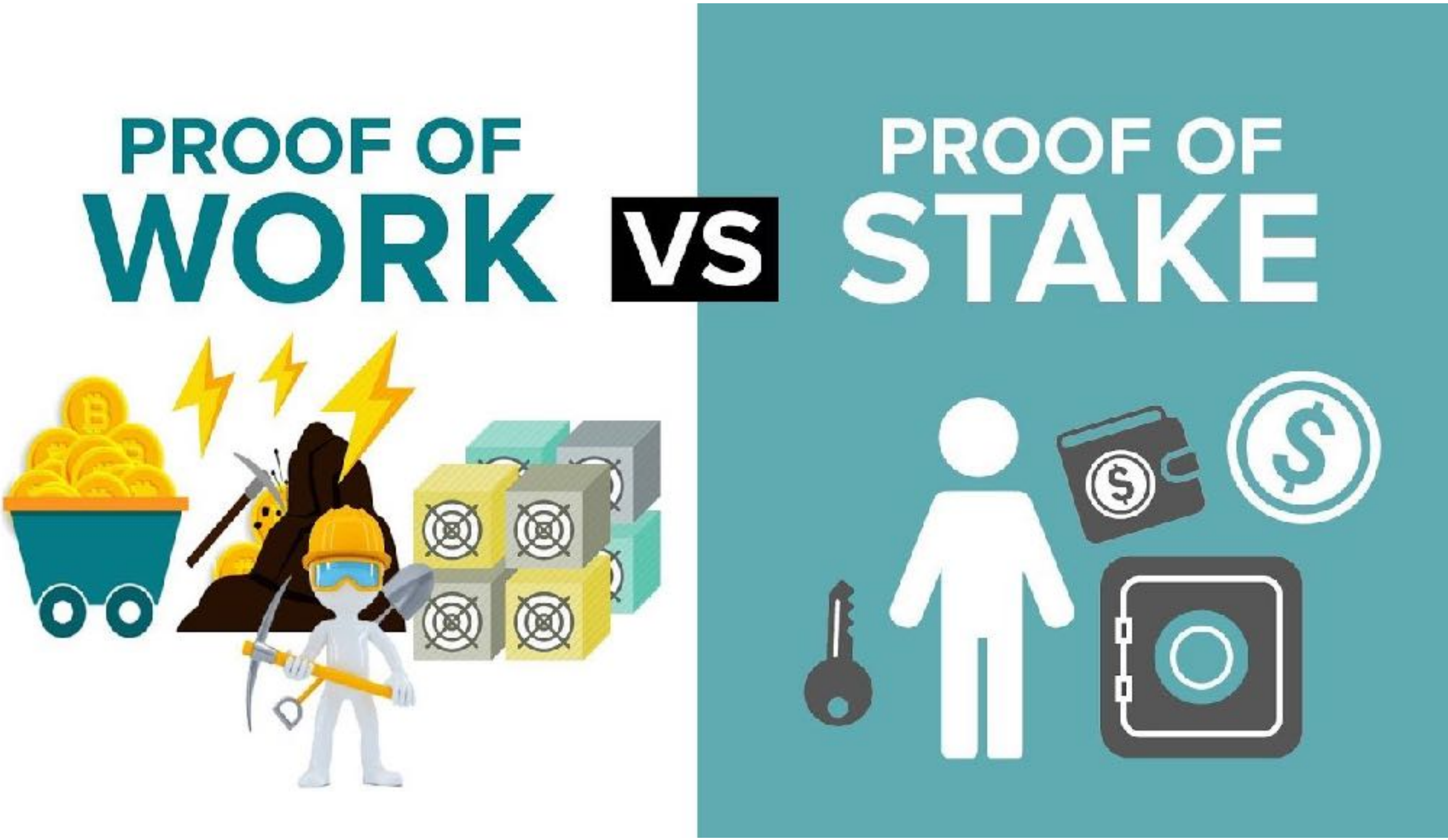
Formal Methods at IOHK

Philip Wadler
University of Edinburgh / IOHK
Edinburgh / Rio de Janeiro
SBMF, 28 November 2018

INPUT | OUTPUT


CARDANO
ADA


PROOF OF
WORK
VS
PROOF OF
STAKE

Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol

Aggelos Kiayias* Alexander Russell† Bernardo David‡ Roman Oliynykov§

August 21, 2017

Abstract

We present "Ouroboros", the first blockchain protocol based on *proof of stake* with rigorous security guarantees. We establish security properties for the protocol comparable to those achieved by the bitcoin blockchain protocol. As the protocol provides a "proof of stake" blockchain discipline, it offers qualitative efficiency advantages over blockchains based on proof of physical resources (e.g., proof of work). We also present a novel reward mechanism for incentivizing Proof of Stake protocols and we prove that, given this mechanism, honest behavior is an approximate Nash equilibrium, thus neutralizing attacks such as selfish mining. We also present initial evidence of the practicality of our protocol in real world settings by providing experimental results on transaction confirmation and processing.

1 Introduction

A primary consideration regarding the operation of blockchain protocols based on proof of work (PoW)—such as bitcoin [30]—is the energy required for their execution. At the time of this writing, generating a single block on the bitcoin blockchain requires a number of hashing operations exceeding 2^{60}, which results in striking energy demands. Indeed, early calculations indicated that the energy requirements of the protocol were comparable to that of a small country [32].

IOHK

Cryptocurrency firm

Product: Ethereum Classic (ETC, Proof of Work)

Product: Cardano (ADA & multicurrency, Proof of Stake)

Committed to peer-reviewed research

Committed to Haskell

Two approaches to formal methods

Previously: Write code, capture its behaviour with specification

Now: Write specification, then implement with code

- Some proofs sketched by hand

- One implementation adheres closely to spec, used for testing

- One implementation designed to be efficient

Wallet state

$$(utxo, pending) \in \mathsf{Wallet} = \mathsf{UTxO} \times \mathsf{Pending}$$
$$w_\varnothing \in \mathsf{Wallet} = (\varnothing, \varnothing)$$

Queries

$$\mathsf{availableBalance} = \mathsf{balance} \circ \mathsf{available}$$
$$\mathsf{totalBalance} = \mathsf{balance} \circ \mathsf{total}$$

Atomic updates

$$\mathsf{applyBlock}\ b\ (utxo, pending) = (\mathsf{updateUTxO}\ b\ utxo,\ \mathsf{updatePending}\ b\ pending)$$
$$\mathsf{newPending}\ tx\ (utxo, pending) = (utxo,\ pending \cup \{tx\})$$

Preconditions

$$\begin{aligned}
&\mathsf{newPending}\ (ins, outs)\ (utxo, pending) \\
&\quad \textbf{requires}\ ins \subseteq \mathrm{dom}(\mathsf{available}\ (utxo, pending)) \\
&\mathsf{applyBlock}\ b\ (utxo, pending) \\
&\quad \textbf{requires}\ \mathrm{dom}(\mathsf{txouts}\ b) \cap \mathrm{dom}\, utxo = \varnothing
\end{aligned}$$

Auxiliary functions

$$\begin{aligned}
&\mathsf{available}, \mathsf{total} \in \mathsf{Wallet} \to \mathsf{UTxO} \\
&\mathsf{available}\ (utxo, pending) = \mathsf{txins}\ pending \not\lhd utxo \\
&\mathsf{total}\ (utxo, pending) = \mathsf{available}\ (utxo, pending) \cup \mathsf{change}\ pending
\end{aligned}$$

$$\begin{aligned}
&\mathsf{change} \in \mathsf{Pending} \to \mathsf{UTxO} \\
&\mathsf{change}\ pending = \mathsf{txouts}\ pending \rhd \mathsf{TxOut}_{\mathsf{ours}}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{updateUTxO} \in \mathsf{Block} \to \mathsf{UTxO} \to \mathsf{UTxO} \\
&\mathsf{updateUTxO}\ b\ utxo = \mathsf{txins}\ b \not\lhd (utxo \cup (\mathsf{txouts}\ b \rhd \mathsf{TxOut}_{\mathsf{ours}}))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{updatePending} \in \mathsf{Block} \to \mathsf{Pending} \to \mathsf{Pending} \\
&\mathsf{updatePending}\ b\ p = \{tx \mid tx \in p, (inputs, _) = tx, inputs \cap \mathsf{txins}\ b = \varnothing\}
\end{aligned}$$

Figure 3: The basic model

Psi Calculus

$\mathbf{0}$	Nil
$\overline{M}N\,.\,P$	Output
$\underline{M}(\lambda\widetilde{x})N\,.\,P$	Input
$\mathbf{case}\ \varphi_1 : P_1 \ [\!] \cdots [\!]\ \varphi_n : P_n$	Case
$(\nu a)P$	Restriction
$P \mid Q$	Parallel
$!P$	Replication
$(\!\mid \Psi \mid\!)$	Assertion

T	the (data) terms, ranged over by M, N
C	the conditions, ranged over by φ
A	the assertions, ranged over by Ψ

Parametric family of process calculi

Specify tems, conditions, assertions

Established theory and tooling: Psi Calculi Workbench

Psi in Haskell

```
data Psi where
  Done     :: Psi                             -- completed process
  New      :: (Channel a → Psi) → Psi         -- create new unicast channel
  Inp      :: Channel a → (a → Psi) → Psi     -- unicast input
  Out      :: Channel a → a → Psi → Psi       -- unicast output
  Log      :: String → Psi bs → Psi bs        -- logging

-- interpreters
simulatePsi :: [Psi] → [String] -- simulate, print logs
exportPsi   :: [Psi] → [String] -- export to, say, Isabelle
runPsi      :: [Psi] → IO ∅     -- run concurrent processes
```

Add broadcast channels and subprocesses

```
data Psi :: [Type] → Type where
  Done     :: Psi bs                                   -- completed process
  New      :: Summarize a ⇒ (Unicast a → Psi bs) → Psi bs  -- create new unicast channel
  UInp     :: Unicast a → (a → Psi bs) → Psi bs        -- unicast input
  UOut     :: Unicast a → a → Psi bs → Psi bs          -- unicast output
  BInp     :: Broadcast bs a → (a → Psi bs) → Psi bs   -- broadcast input
  BOut     :: Broadcast bs a → a → Psi bs → Psi bs     -- broadcast output
  Fork     :: ProcId → Psi ^[] → Psi bs → Psi bs       -- fork new process
  Log      :: String → Psi bs → Psi bs                 -- logging
```

MARLOWE

PLUTUS

IELE

Reasoning about Smart Contracts

PROGRESS: If a contract is not quiescent, there is always a finite time by which each of the participants can take a step to progress the contract.

PRESERVATION: If a contract is quiescent, the sum of money paid into the smart contract is equal to the sum of money paid out of the smart contract.