Formal Methods at IOHK

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

\[(utxo, pending) \in \text{Wallet} = UTxO \times \text{Pending}\]

\[w_\emptyset \in \text{Wallet} = (\emptyset, \emptyset)\]

**Queries**

\[\text{availableBalance} = \text{balance} \circ \text{available}\]
\[\text{totalBalance} = \text{balance} \circ \text{total}\]

**Atomic updates**

\[\text{applyBlock} \ b \ (utxo, pending) = (\text{updateUTxO} \ b \ utxo, \ \text{updatePending} \ b \ pending)\]
\[\text{newPending} \ tx \ (utxo, pending) = (utxo, \ pending \cup \{tx\})\]

**Preconditions**

\[\text{newPending} \ (ins, outs) \ (utxo, pending)\]
\[\text{requires} \ ins \subseteq \text{dom}(\text{available} \ (utxo, pending))\]
\[\text{applyBlock} \ b \ (utxo, pending)\]
\[\text{requires} \ \text{dom}(\text{txouts} \ b) \cap \text{dom} \ utxo = \emptyset\]

**Auxiliary functions**

\[\text{available, total} \in \text{Wallet} \rightarrow UTxO\]
\[\text{available} \ (utxo, pending) = \text{txins} \ pending \not\in utxo\]
\[\text{total} \ (utxo, pending) = \text{available} \ (utxo, pending) \cup \text{change pending}\]

\[\text{change} \in \text{Pending} \rightarrow UTxO\]
\[\text{change} \ pending = \text{txouts} \ pending \Rightarrow \text{TxOutours}\]

\[\text{updateUTxO} \in \text{Block} \rightarrow UTxO \rightarrow UTxO\]
\[\text{updateUTxO} \ b \ utxo = \text{txins} \ b \not\in (utxo \cup (\text{txouts} \ b \Rightarrow \text{TxOutours}))\]

\[\text{updatePending} \in \text{Block} \rightarrow \text{Pending} \rightarrow \text{Pending}\]
\[\text{updatePending} \ b \ p = \{tx \mid tx \in p, (\text{inputs}, _) = tx, \text{inputs} \cap \text{txins} \ b = \emptyset\}\]

**Figure 3:** The basic model
Psi Calculus

Parametric family of process calculi

Specify terms, 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
```
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.