MODULE *protocol*

EXTENDS *TLC*, *Naturals*, *FiniteSets*, *Sequences*, *definitions*

--algorithm *protocol*

variables

A compact zk-SNARK proof that summarizes the *Merkle* tree root of the current note commitments.

$noteCommitmentRoot = \langle\rangle$;

A compact zk-SNARK proof that summarizes the *Merkle* tree root of the current nullifiers.

$nullifierRoot = \langle\rangle$;

The blockchain last accepted block.

$tip\_block = [height \mapsto 1,\ transactions \mapsto \langle\rangle]$;

Transaction pool for producers to build blocks.

$txPool = \{\}$;

The proposed block from a miner.

$proposed\_block$;

define

The height of the blockchain always increases

$HeightAlwaysIncreases \triangleq \Box[tip\_block'.height > tip\_block.height]_{tip\_block}$

Transactions in the transaction pool are eventually processed

$TransactionsEventuallyProcessed \triangleq$
$(Cardinality(txPool) > 0) \Rightarrow \Diamond(Cardinality(txPool) = 0)$

For each transaction in the transaction pool, the *nullifier* is unique

$NoDoubleSpending \triangleq$
$\forall\, tx \in txPool :$
$\forall\, action1,\ action2 \in tx.actions :$
$action1 \neq action2 \Rightarrow action1.nullifier \neq action2.nullifier$

end define ;

User process: *User* creates actions and a proof, use that to build a transaction and add it to the pool.

fair process $User =$ "User"

variables

$tx\_$,
$actions$,
$nullifier$,
$commitment$;

begin

$CreateTx$:

Wait until the transaction pool is empty.

await $txPool = \{\}$;

Prepare the transaction actions: generate a new *nullifier* and commitment (each 32 bytes), along with a fixed value and a receiver.

```
        actions :=
        {[
            nullifier       ↦ RandomBytes(32),
            commitment      ↦ RandomBytes(32),
            value           ↦ 10,
            receiver        ↦ "receiver"
        ]};
        Create a new transaction with the actions and a zk-SNARK proof and add it to the pool.
        txPool :=
        {[
            actions ↦ actions,
            proof   ↦ GenerateZKProof(actions)
        ]};
end process ;

Producer process: assembles transactions into a block and computes updated state commitments and nullifiers.
fair process Producer = "Producer"
begin
    Produce:
        Wait until there is at least one transaction in the pool.
        await Cardinality(txPool) > 0;
        Create a new block with an incremented height and the transactions from the pool.
        proposed_block :=
        [
            height  ↦ tip_block.height + 1,
            txs     ↦ txPool
        ];
        Clear the transaction pool after block creation.
        txPool := {};
end process ;

Node process: verifies the proposed block and updates the state.
fair process Node = "Node"
begin
    Verify:
        Wait for a proposed block.
        await proposed_block ≠ defaultInitValue;

        Panics if the proposed block is invalid.
        assert VerifyBlockHeader(proposed_block, tip_block) = TRUE;
        assert VerifyBlockTransactions(proposed_block.txs) = TRUE;

        For each transaction in the proposed block.
        with tx ∈ proposed_block.txs do
            Verify the transaction zk-SNARK proof, panick if invalid.
            assert VerifyZKProof(tx.proof, noteCommitmentRoot, nullifierRoot) = TRUE;
```

Update the note commitment and *nullifier* roots.

noteCommitmentRoot := *ComputeNewNoteRoot*(*noteCommitmentRoot*, *tx*) ;

nullifierRoot := *ComputeNewNullifierRoot*(*nullifierRoot*, *tx*) ;

end with ;

Update the blockchain's tip block.

tip_block := [*height* ↦ *proposed_block.height*, *transactions* ↦ *proposed_block.txs*] ;

Regardless of validity, discard the proposed block after verification.

proposed_block := *defaultInitValue* ;

end process ;

end algorithm ;

BEGIN TRANSLATION (*chksum*(*pcal*) = "29f1a93a" ∧ *chksum*(*tla*) = "5b0dd1a1")

CONSTANT *defaultInitValue*

VARIABLES *pc*, *noteCommitmentRoot*, *nullifierRoot*, *tip_block*, *txPool*, *proposed_block*

define statement

$$HeightAlwaysIncreases \triangleq \Box[tip\_block'.height > tip\_block.height]_{tip\_block}$$

$$TransactionsEventuallyProcessed \triangleq (Cardinality(txPool) > 0) \Rightarrow \Diamond(Cardinality(txPool) = 0)$$

$$NoDoubleSpending \triangleq \forall\, tx \in txPool : \forall\, action1,\, action2 \in tx.actions : action1 \neq action2 \Rightarrow action1.nullifier \neq action2.nullifier$$

VARIABLES *tx_*, *actions*, *nullifier*, *commitment*

$$vars \triangleq \langle pc,\, noteCommitmentRoot,\, nullifierRoot,\, tip\_block,\, txPool,\, proposed\_block,\, tx\_,\, actions,\, nullifier,\, commitment\rangle$$

$$ProcSet \triangleq \{\text{"User"}\} \cup \{\text{"Producer"}\} \cup \{\text{"Node"}\}$$

$$\begin{aligned} Init \triangleq\ & \text{Global variables} \\ & \wedge\, noteCommitmentRoot = \langle\rangle \\ & \wedge\, nullifierRoot = \langle\rangle \\ & \wedge\, tip\_block = [height \mapsto 1,\, transactions \mapsto \langle\rangle] \\ & \wedge\, txPool = \{\} \\ & \wedge\, proposed\_block = defaultInitValue \\ & \text{Process } User \\ & \wedge\, tx\_ = defaultInitValue \\ & \wedge\, actions = defaultInitValue \\ & \wedge\, nullifier = defaultInitValue \\ & \wedge\, commitment = defaultInitValue \\ & \wedge\, pc = [self \in ProcSet \mapsto \text{CASE } self = \text{"User"} \rightarrow \text{"CreateTx"} \end{aligned}$$

$$\Box \quad self = \text{"Producer"} \rightarrow \text{"Produce"}$$
$$\Box \quad self = \text{"Node"} \rightarrow \text{"Verify"}]$$

$$
\begin{aligned}
CreateTx \triangleq\ & \land\ pc[\text{"User"}] = \text{"CreateTx"} \\
& \land\ txPool = \{\} \\
& \land\ actions' = \{[\\
& \qquad nullifier \mapsto RandomBytes(32), \\
& \qquad commitment \mapsto RandomBytes(32), \\
& \qquad value \mapsto 10, \\
& \qquad receiver \mapsto \text{"receiver"} \\
& \quad]\} \\
& \land\ txPool' = \{[\\
& \qquad actions \mapsto actions', \\
& \qquad proof \mapsto GenerateZKProof(actions') \\
& \quad]\} \\
& \land\ pc' = [pc \ \text{EXCEPT} \ ![\text{"User"}] = \text{"Done"}] \\
& \land\ \text{UNCHANGED} \ \langle noteCommitmentRoot, nullifierRoot, tip\_block, \\
& \qquad proposed\_block, tx\_, nullifier, commitment \rangle
\end{aligned}
$$

$$User \triangleq CreateTx$$

$$
\begin{aligned}
Produce \triangleq\ & \land\ pc[\text{"Producer"}] = \text{"Produce"} \\
& \land\ Cardinality(txPool) > 0 \\
& \land\ proposed\_block' = [\\
& \qquad height \mapsto tip\_block.height + 1, \\
& \qquad txs \mapsto txPool \\
& \quad] \\
& \land\ txPool' = \{\} \\
& \land\ pc' = [pc \ \text{EXCEPT} \ ![\text{"Producer"}] = \text{"Done"}] \\
& \land\ \text{UNCHANGED} \ \langle noteCommitmentRoot, nullifierRoot, tip\_block, tx\_, \\
& \qquad actions, nullifier, commitment \rangle
\end{aligned}
$$

$$Producer \triangleq Produce$$

$$
\begin{aligned}
Verify \triangleq\ & \land\ pc[\text{"Node"}] = \text{"Verify"} \\
& \land\ proposed\_block \neq defaultInitValue \\
& \land\ Assert(VerifyBlockHeader(proposed\_block, tip\_block) = \text{TRUE}, \\
& \qquad \text{"Failure of assertion at line 85, column 9."}) \\
& \land\ Assert(VerifyBlockTransactions(proposed\_block.txs) = \text{TRUE}, \\
& \qquad \text{"Failure of assertion at line 86, column 9."}) \\
& \land\ \exists\, tx \in proposed\_block.txs : \\
& \qquad \land\ Assert(VerifyZKProof(tx.proof, noteCommitmentRoot, nullifierRoot) = \text{TRUE}, \\
& \qquad\qquad \text{"Failure of assertion at line 91, column 13."}) \\
& \qquad \land\ noteCommitmentRoot' = ComputeNewNoteRoot(noteCommitmentRoot, tx) \\
& \qquad \land\ nullifierRoot' = ComputeNewNullifierRoot(nullifierRoot, tx) \\
& \land\ tip\_block' = [height \mapsto proposed\_block.height, transactions \mapsto proposed\_block.txs]
\end{aligned}
$$

$\land\ proposed\_block' = defaultInitValue$
$\land\ pc' = [pc \text{ EXCEPT } ![\text{"Node"}] = \text{"Done"}]$
$\land$ UNCHANGED $\langle txPool,\ tx\_,\ actions,\ nullifier,\ commitment\rangle$

$Node \triangleq Verify$

Allow infinite stuttering to prevent deadlock on termination.

$Terminating \triangleq \land \forall\, self \in ProcSet : pc[self] = \text{"Done"}$
$\land$ UNCHANGED $vars$

$Next \triangleq User \lor Producer \lor Node$
$\lor\ Terminating$

$Spec \triangleq \land\ Init \land \Box[Next]_{vars}$
$\land\ \mathrm{WF}_{vars}(User)$
$\land\ \mathrm{WF}_{vars}(Producer)$
$\land\ \mathrm{WF}_{vars}(Node)$

$Termination \triangleq \Diamond(\forall\, self \in ProcSet : pc[self] = \text{"Done"})$

END TRANSLATION