MODULE *protocol*

NU7 memo bundles specification

This specification is a simplified version of the memo bundles protocol update to happen at NU7 with the introduction of V6 transactions.

The protocol is designed to demonstrate the functionality of the encryption and decryption process as described in ZIP − 231.It includes :
− A User process that encrypts a memo, constructs a transaction, and adds it to a transaction pool.
− A Node process that validates and commits transactions from the pool. To demostrate pruning, all chunks are eventually pruned from the bundle.
− A Scanner process that scans the blockchain, decrypts memo data, and verifies correctness.

The module uses helper operators which are defined in the Operators module.

Note : The cryptographic functions(e.g., EncryptionKey, EncryptMemo, DecryptMemo)are abstracted for modeling purposes and do not reflect the full complexity of the real protocol.

EXTENDS *FiniteSets*, *Naturals*, *TLC*, *Sequences*, *Operators*

CONSTANTS

Full message(as a sequence of characters)that will be encrypted.
$memo \triangleq \langle$"h", "e", "l", "l", "o", "w", "o", "r", "l", "d"$\rangle$
Defines the maximum allowed number of memo chunks in a transaction.
$memo\_chunk\_limit \triangleq 2$
The fixed size of each chunk after splitting(and padding, if necessary).
$memo\_chunk\_size \triangleq 6$
Representation of a pruned chunk.
$pruned\_chunk \triangleq \langle$"p", "r", "u", "n", "e", "d"$\rangle$

--algorithm *memo*

variables
Pool where transactions are stored before being validated.
$txPool = \{\}$;
The blockchain is a set of transactions.
$blockchain = \{\}$;
The memo key used for memo encryption.
$memo\_key = RandomHash(32)$;
Randomness salt used for key derivation.
$salt = RandomHash(32)$;
Decrypted memo after decryption.
$decrypted\_memo = \langle\rangle$;

define
At least in 1 behaviour, no pruning occurred, decrypted_memo is equal memo.
$DecrypedEqOrig \triangleq Cardinality(blockchain) > 0 \Rightarrow$

$\Diamond(decrypted\_memo = memo)$

At least in 1 behaviour, the first chunk is pruned.

$DecrypedEqPruned1 \triangleq Cardinality(blockchain) > 0 \Rightarrow$
$\Diamond(decrypted\_memo = pruned\_chunk \circ SubSeq(memo,\ memo\_chunk\_size + 1,\ Len(memo)))$

At least in 1 behaviour, the last chunk was pruned.

$DecrypedEqPruned2 \triangleq Cardinality(blockchain) > 0 \Rightarrow$
$\Diamond(decrypted\_memo = (SubSeq(memo,\ 1,\ memo\_chunk\_size)) \circ pruned\_chunk)$

At least in 1 behaviour, all chunks were pruned.

$DecrypedEqAllPruned \triangleq Cardinality(blockchain) > 0 \Rightarrow$
$\Diamond(decrypted\_memo = (pruned\_chunk \circ pruned\_chunk))$

end define ;

Encrypt the memo, build a transaction and add it to the pool.

fair process *User* = "USER"
variables
encryption_key,
plaintext_memo_chunks,
encrypted_memo_chunks,
tx_v6,
begin
Encrypt:
Derive the encryption key from the memo key and salt using a(simplified)key derivation function.
encryption_key := *EncryptionKey*(*memo_key*, *salt*) ;
Split the memo into fixed − size chunks(with padding on the final chunk).
plaintext_memo_chunks := *SplitAndPadMemo*(*memo*, *memo_chunk_size*) ;
Encrypt each chunk using the derived encryption key.
encrypted_memo_chunks := *EncryptMemo*(*encryption_key*, *plaintext_memo_chunks*) ;
BuildTx:
Construct the transaction
tx_v6 :=
[
No memo chunk is pruned at memo creation.
f_all_pruned ↦ FALSE,
Stores the salt used for key derivation
salt_or_hash ↦ *salt*,
The number of memo chunks in the encrypted bundle.
n_memo_chunks ↦ *Len*(*encrypted_memo_chunks*),
A sequence of 0 s indicating no chunk is pruned.
pruned ↦ [*_i* ∈ 1 .. *Len*(*encrypted_memo_chunks*) ↦ 0],
The encrypted memo chunks.
v_memo_chunks ↦ *encrypted_memo_chunks*,
actions ↦ {[
The receiver of the memo is the user itself.
receiver ↦ "USER",
The memo key used for encryption.

```
                memo_key ↦ memo_key,
                 The amount of the transaction is set to 0.
                amount   ↦ 0
            ]}
        ];
    PushTx:
        await txPool = {};
        txPool := {tx_v6} ;
end process ;

 Validates, prunes, and commits transactions
fair process Node = "NODE"
variables
    tx,
    new_tx,
    i = 1;
begin
    ValidateTx:
        await txPool ≠ {};
        tx := CHOOSE transaction ∈ txPool : TRUE;
        txPool := txPool \ {tx};

        assert Len(tx.v_memo_chunks) ≤ memo_chunk_limit;
        assert (CHOOSE a ∈ tx.actions : TRUE).memo_key ≠ ⟨⟩;
        assert VerifyTx(tx);

         Commit valid transactions
        blockchain := blockchain ∪ {tx};
    PruneChunks:
         Loop over each memo chunk in the transaction until all are pruned.
         This will produce a state for each chunk that is pruned.
        while i ≤ Len(tx.v_memo_chunks) do
            if tx.v_memo_chunks[i].chunk ≠ pruned_chunk then
                new_tx :=
                [tx EXCEPT
                   !.v_memo_chunks[i].chunk = pruned_chunk,   H(AEAD(MemoChunk, memo_key))
                   !.pruned[i] = 1];
            end if ;
            i := i + 1;
             Update the blockchain: replace the original transaction with the updated one.
            blockchain := (blockchain \ {tx}) ∪ {new_tx};
             Update the transaction variable to point to the new transaction.
            tx := new_tx;
        end while ;
    UpdateTx:
         When all chunks have been processed, update overall transaction fields:
```

```
        new_tx :=
            [tx EXCEPT
               !.f_all_pruned = TRUE,
               !.salt_or_hash = RandomHash(32)  memo_bundle_digest = H(concat(memo_chunk_digests))
            ];
         Update the blockchain : replace the original transaction with the updated one.
        blockchain := (blockchain \ {tx}) ∪ {new_tx};
        tx := new_tx;
end process ;

 Scans for transactions belonging to the user and decrypts them
fair process Scanner = "SCANNER"
variables
    tx;
begin
    Scan:
        await Cardinality(blockchain) > 0;
        tx := CHOOSE t ∈ blockchain : ∃ a ∈ t.actions : a.receiver = "USER";
    Decrypt:
         Decrypt the memo bundle using the memo key and salt stored in the transaction.
        decrypted_memo :=
            DecryptedMemoFinal(DecryptMemo(memo_key, tx.salt_or_hash, tx.v_memo_chunks));
         If all chunks were pruned in transaction, then decrypted_memo should be all pruned,
         and the salt_or_hash field should be a memo_bundle_digest.
        if tx.f_all_pruned = TRUE then
           assert decrypted_memo = (pruned_chunk ∘ pruned_chunk);
        end if ;
end process ;

end algorithm ;
 BEGIN TRANSLATION(chksum(pcal) = "4674289f" ∧ chksum(tla) = "ae4b9f5e")
 Process variable tx of process Node at line 111 col 5 changed to tx_
CONSTANT defaultInitValue
VARIABLES pc, txPool, blockchain, memo_key, salt, decrypted_memo

 define statement
DecrypedEqOrig ≜ Cardinality(blockchain) > 0 ⇒
    ◇(decrypted_memo = memo)

DecrypedEqPruned1 ≜ Cardinality(blockchain) > 0 ⇒
    ◇(decrypted_memo = pruned_chunk ∘ SubSeq(memo, memo_chunk_size + 1, Len(memo)))

DecrypedEqPruned2 ≜ Cardinality(blockchain) > 0 ⇒
    ◇(decrypted_memo = (SubSeq(memo, 1, memo_chunk_size)) ∘ pruned_chunk)

DecrypedEqAllPruned ≜ Cardinality(blockchain) > 0 ⇒
    ◇(decrypted_memo = (pruned_chunk ∘ pruned_chunk))
```

VARIABLES $encryption\_key$, $plaintext\_memo\_chunks$, $encrypted\_memo\_chunks$, tx_v6, $tx\_$, new_tx, i, tx

$vars \triangleq \langle pc,\ txPool,\ blockchain,\ memo\_key,\ salt,\ decrypted\_memo,$
$encryption\_key,\ plaintext\_memo\_chunks,\ encrypted\_memo\_chunks,$
$tx\_v6,\ tx\_,\ new\_tx,\ i,\ tx\rangle$

$ProcSet \triangleq \{\text{"USER"}\} \cup \{\text{"NODE"}\} \cup \{\text{"SCANNER"}\}$

$Init \triangleq$ *Global variables*
$\land\ txPool = \{\}$
$\land\ blockchain = \{\}$
$\land\ memo\_key = RandomHash(32)$
$\land\ salt = RandomHash(32)$
$\land\ decrypted\_memo = \langle\rangle$
Process User
$\land\ encryption\_key = defaultInitValue$
$\land\ plaintext\_memo\_chunks = defaultInitValue$
$\land\ encrypted\_memo\_chunks = defaultInitValue$
$\land\ tx\_v6 = defaultInitValue$
Process Node
$\land\ tx\_ = defaultInitValue$
$\land\ new\_tx = defaultInitValue$
$\land\ i = 1$
Process Scanner
$\land\ tx = defaultInitValue$
$\land\ pc = [self \in ProcSet \mapsto \text{CASE } self = \text{"USER"} \to \text{"Encrypt"}$
$\square\ self = \text{"NODE"} \to \text{"ValidateTx"}$
$\square\ self = \text{"SCANNER"} \to \text{"Scan"}]$

$Encrypt \triangleq \land\ pc[\text{"USER"}] = \text{"Encrypt"}$
$\land\ encryption\_key' = EncryptionKey(memo\_key,\ salt)$
$\land\ plaintext\_memo\_chunks' = SplitAndPadMemo(memo,\ memo\_chunk\_size)$
$\land\ encrypted\_memo\_chunks' = EncryptMemo(encryption\_key',\ plaintext\_memo\_chunks')$
$\land\ pc' = [pc \text{ EXCEPT } ![\text{"USER"}] = \text{"BuildTx"}]$
$\land\ \text{UNCHANGED } \langle txPool,\ blockchain,\ memo\_key,\ salt,\ decrypted\_memo,$
$tx\_v6,\ tx\_,\ new\_tx,\ i,\ tx\rangle$

$BuildTx \triangleq \land\ pc[\text{"USER"}] = \text{"BuildTx"}$
$\land\ tx\_v6' = [$

$f\_all\_pruned \mapsto \text{FALSE},$

$salt\_or\_hash \mapsto salt,$

$n\_memo\_chunks \mapsto Len(encrypted\_memo\_chunks),$

$pruned \mapsto [\_i \in 1\ ..\ Len(encrypted\_memo\_chunks) \mapsto 0],$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\quad v\_memo\_chunks \mapsto encrypted\_memo\_chunks, \\
\qquad\qquad\qquad\qquad\quad actions \qquad\quad \mapsto \{[\\
\qquad\qquad\qquad\qquad\qquad\quad receiver \quad \mapsto \text{“USER”}, \\
\qquad\qquad\qquad\qquad\qquad\quad memo\_key \mapsto memo\_key, \\
\qquad\qquad\qquad\qquad\qquad\quad amount \quad \mapsto 0 \\
\qquad\qquad\qquad\qquad\quad]\} \\
\qquad\qquad\qquad\quad] \\
\qquad\qquad \wedge\, pc' = [pc \text{ EXCEPT } ![\text{“USER”}] = \text{“PushTx”}] \\
\qquad\qquad \wedge \text{ UNCHANGED } \langle txPool,\ blockchain,\ memo\_key,\ salt,\ decrypted\_memo, \\
\qquad\qquad\qquad\qquad\quad encryption\_key,\ plaintext\_memo\_chunks, \\
\qquad\qquad\qquad\qquad\quad encrypted\_memo\_chunks,\ tx\_,\ new\_tx,\ i,\ tx\rangle
\end{array}
$$

$$
\begin{array}{lll}
PushTx & \triangleq & \wedge\, pc[\text{“USER”}] = \text{“PushTx”} \\
& & \wedge\, txPool = \{\} \\
& & \wedge\, txPool' = \{tx\_v6\} \\
& & \wedge\, pc' = [pc \text{ EXCEPT } ![\text{“USER”}] = \text{“Done”}] \\
& & \wedge \text{ UNCHANGED } \langle blockchain,\ memo\_key,\ salt,\ decrypted\_memo, \\
& & \qquad\quad encryption\_key,\ plaintext\_memo\_chunks, \\
& & \qquad\quad encrypted\_memo\_chunks,\ tx\_v6,\ tx\_,\ new\_tx,\ i,\ tx\rangle
\end{array}
$$

$$
User \triangleq Encrypt \vee BuildTx \vee PushTx
$$

$$
\begin{array}{lll}
ValidateTx & \triangleq & \wedge\, pc[\text{“NODE”}] = \text{“ValidateTx”} \\
& & \wedge\, txPool \neq \{\} \\
& & \wedge\, tx\_' = (\text{CHOOSE } transaction \in txPool : \text{TRUE}) \\
& & \wedge\, txPool' = txPool \setminus \{tx\_'\} \\
& & \wedge\, Assert(Len(tx\_'.v\_memo\_chunks) \leq memo\_chunk\_limit, \\
& & \qquad\quad \text{“Failure of assertion at line 120, column 9.”}) \\
& & \wedge\, Assert((\text{CHOOSE } a \in tx\_'.actions : \text{TRUE}).memo\_key \neq \langle\rangle, \\
& & \qquad\quad \text{“Failure of assertion at line 121, column 9.”}) \\
& & \wedge\, Assert(VerifyTx(tx\_'), \\
& & \qquad\quad \text{“Failure of assertion at line 122, column 9.”}) \\
& & \wedge\, blockchain' = (blockchain \cup \{tx\_'\}) \\
& & \wedge\, pc' = [pc \text{ EXCEPT } ![\text{“NODE”}] = \text{“PruneChunks”}] \\
& & \wedge \text{ UNCHANGED } \langle memo\_key,\ salt,\ decrypted\_memo,\ encryption\_key, \\
& & \qquad\quad plaintext\_memo\_chunks,\ encrypted\_memo\_chunks, \\
& & \qquad\quad tx\_v6,\ new\_tx,\ i,\ tx\rangle
\end{array}
$$

$$
\begin{array}{lll}
PruneChunks & \triangleq & \wedge\, pc[\text{“NODE”}] = \text{“PruneChunks”} \\
& & \wedge \text{ IF } i \leq Len(tx\_.v\_memo\_chunks) \\
& & \quad \text{THEN } \wedge \text{ IF } tx\_.v\_memo\_chunks[i].chunk \neq pruned\_chunk \\
& & \qquad\quad \text{THEN } \wedge\, new\_tx' = [tx\_ \text{ EXCEPT} \\
& & \qquad\qquad\qquad\qquad !.v\_memo\_chunks[i].chunk = pruned\_chunk, \\
& & \qquad\qquad\qquad\qquad !.pruned[i] = 1]
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\textsc{else}\ \ \land \textsc{true} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \ \land \textsc{unchanged}\ new\_tx \\
\qquad\qquad\qquad\qquad\qquad \land i' = i + 1 \\
\qquad\qquad\qquad\qquad\qquad \land blockchain' = ((blockchain \setminus \{tx\_\}) \cup \{new\_tx'\}) \\
\qquad\qquad\qquad\qquad\qquad \land tx\_' = new\_tx' \\
\qquad\qquad\qquad\qquad\qquad \land pc' = [pc\ \textsc{except}\ ![\text{"NODE"}] = \text{"PruneChunks"}] \\
\qquad\qquad\qquad\qquad \textsc{else}\ \ \land pc' = [pc\ \textsc{except}\ ![\text{"NODE"}] = \text{"UpdateTx"}] \\
\qquad\qquad\qquad\qquad\qquad \land \textsc{unchanged}\ \langle blockchain,\ tx\_,\ new\_tx,\ i\rangle \\
\qquad\qquad\qquad \land \textsc{unchanged}\ \langle txPool,\ memo\_key,\ salt,\ decrypted\_memo, \\
\qquad\qquad\qquad\qquad\qquad encryption\_key,\ plaintext\_memo\_chunks, \\
\qquad\qquad\qquad\qquad\qquad encrypted\_memo\_chunks,\ tx\_v6,\ tx\rangle
\end{array}
$$

$$
\begin{array}{l}
UpdateTx \triangleq\ \land pc[\text{"NODE"}] = \text{"UpdateTx"} \\
\qquad\qquad\quad \land new\_tx' = [tx\_\ \textsc{except} \\
\qquad\qquad\qquad\qquad\qquad !.f\_all\_pruned = \textsc{true}, \\
\qquad\qquad\qquad\qquad\qquad !.salt\_or\_hash = RandomHash(32) \\
\qquad\qquad\qquad\qquad\quad] \\
\qquad\qquad\quad \land blockchain' = ((blockchain \setminus \{tx\_\}) \cup \{new\_tx'\}) \\
\qquad\qquad\quad \land tx\_' = new\_tx' \\
\qquad\qquad\quad \land pc' = [pc\ \textsc{except}\ ![\text{"NODE"}] = \text{"Done"}] \\
\qquad\qquad\quad \land \textsc{unchanged}\ \langle txPool,\ memo\_key,\ salt,\ decrypted\_memo, \\
\qquad\qquad\qquad\qquad\qquad encryption\_key,\ plaintext\_memo\_chunks, \\
\qquad\qquad\qquad\qquad\qquad encrypted\_memo\_chunks,\ tx\_v6,\ i,\ tx\rangle
\end{array}
$$

$$
Node \triangleq\ ValidateTx \lor PruneChunks \lor UpdateTx
$$

$$
\begin{array}{l}
Scan \triangleq\ \land pc[\text{"SCANNER"}] = \text{"Scan"} \\
\qquad\quad \land Cardinality(blockchain) > 0 \\
\qquad\quad \land tx' = (\textsc{choose}\ t \in blockchain : \exists\, a \in t.actions : a.receiver = \text{"USER"}) \\
\qquad\quad \land pc' = [pc\ \textsc{except}\ ![\text{"SCANNER"}] = \text{"Decrypt"}] \\
\qquad\quad \land \textsc{unchanged}\ \langle txPool,\ blockchain,\ memo\_key,\ salt,\ decrypted\_memo, \\
\qquad\qquad\qquad\qquad encryption\_key,\ plaintext\_memo\_chunks, \\
\qquad\qquad\qquad\qquad encrypted\_memo\_chunks,\ tx\_v6,\ tx\_,\ new\_tx,\ i\rangle
\end{array}
$$

$$
\begin{array}{l}
Decrypt \triangleq\ \land pc[\text{"SCANNER"}] = \text{"Decrypt"} \\
\qquad\quad \land decrypted\_memo' = DecryptedMemoFinal(DecryptMemo(memo\_key,\ tx.salt\_or\_hash,\ tx.v\_me \\
\qquad\quad \land \textsc{if}\ tx.f\_all\_pruned = \textsc{true} \\
\qquad\qquad\quad \textsc{then}\ \land Assert(decrypted\_memo' = (pruned\_chunk \circ pruned\_chunk), \\
\qquad\qquad\qquad\qquad\qquad \text{"Failure of assertion at line 168, column 13."}) \\
\qquad\qquad\quad \textsc{else}\ \ \land \textsc{true} \\
\qquad\quad \land pc' = [pc\ \textsc{except}\ ![\text{"SCANNER"}] = \text{"Done"}] \\
\qquad\quad \land \textsc{unchanged}\ \langle txPool,\ blockchain,\ memo\_key,\ salt,\ encryption\_key, \\
\qquad\qquad\qquad\qquad plaintext\_memo\_chunks,\ encrypted\_memo\_chunks,\ tx\_v6, \\
\qquad\qquad\qquad\qquad tx\_,\ new\_tx,\ i,\ tx\rangle
\end{array}
$$

$$
Scanner \triangleq\ Scan \lor Decrypt
$$

Allow infinite stuttering to prevent deadlock on termination.

$$\begin{aligned} \mathit{Terminating} \triangleq\ & \land \forall\, \mathit{self} \in \mathit{ProcSet} : \mathit{pc}[\mathit{self}] = \text{“Done”} \\ & \land \text{UNCHANGED } \mathit{vars} \end{aligned}$$

$$\begin{aligned} \mathit{Next} \triangleq\ & \mathit{User} \lor \mathit{Node} \lor \mathit{Scanner} \\ & \lor \mathit{Terminating} \end{aligned}$$

$$\begin{aligned} \mathit{Spec} \triangleq\ & \land \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}} \\ & \land \mathrm{WF}_{\mathit{vars}}(\mathit{User}) \\ & \land \mathrm{WF}_{\mathit{vars}}(\mathit{Node}) \\ & \land \mathrm{WF}_{\mathit{vars}}(\mathit{Scanner}) \end{aligned}$$

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

END TRANSLATION