withdrawTo() documentation

Yao Sun

September 5th 2017

Documentation of withdraw functionality (mostly just for my notekeeping). The contract state is given as follows.

$$X_{256} := \{x \in \mathbb{R} | 0 \leq x < 2^{256}\} \tag{1}$$

X being the set of all real numbers within the range $[0, 2^256]$

$$X_8 := \{x \in \mathbb{R} | 0 \leq x < 2^8\} \tag{2}$$

$$A := \{0, 1\}^{20} \tag{3}$$

A being the set of all bytes of length 20 (this is non-standard notation sorry).

Furthermore, we define some constants to be used and place restrictions onto them:

$$\alpha := x \in X_{256} = 4 \tag{4}$$

α, reprsents total number of periods. We assume this number to be equal to 4.

$$\beta := x \in X_{256} := [0, 10000) \tag{5}$$

β, represents interval in which periods take place (with the final period taking forever). Assuming

Then we can define the following mappings.

$$dep(x) := A \rightarrow X_{256} \tag{6}$$

$$wit(x) := A \rightarrow X_{\alpha+1} \tag{7}$$

period()

The period function is responsible for calculating the current period the function is in, the source code is defined as

```
// what withdrawal period are we in?
// returns the period number from [0, periods)
function period() constant returns(uint) {
    require(startblock != 0);
    uint p = (block.number - startblock) / interval;
    if (p >= periods)
        p = periods-1;
    return p;
}
```

Therefore we can define a new set X_{period} that is responsible for mapping the block number to a period.

$$X_\alpha := \{x \in \mathbb{R} | 0 \leq x < \alpha\} \tag{8}$$

$$X_\alpha \subset X_8 \tag{9}$$

$$X_{\alpha ?} := X_{period} \cup E \tag{10}$$

Where the set E just represents an error that can be emitted from the function. In our scenario errors imply state rollback (no change). Then the period function is simply a function that maps from one set to another.

$$p(x) := X_{256} \rightarrow X_{period?} \tag{11}$$

Note that we implicitly assume if an element's set is subset of another that it can be cast to the other type without loss of information.

withdrawTo()

```
// trigger withdrawal to a particular address
// (allow anyone to force disbursement)
function withdrawTo(address addr) returns(bool) {
    if (!locked || startblock == 0)
        return false; // can't withdraw before locking
    uint b = total;
    uint d = totalfv;
    uint p = period();
```

```
        uint8 w = withdrawals[addr];

        // the sender can only withdraw once per period,
        // and only 'periods' times
        if (w > p || w >= periods)
            return false;

        // total amount owed, including bonus:
        // (deposited[addr] / d) is the fraction of deposited tokens
        // b is bonus plus total deposited
        //
        // since sum(deposited) = d, you can prove trivially that
        // sum((deposited * b) / d) == b, modulo any roundoff error
        assert(b >= d);
        uint owed = (deposited[addr] * b) / d;

        // distribute all the unclaimed periods
        uint ps = 1 + p - w;

        // (ps / periods) is fraction of total to be distributed;
        // (owed + bonus) is amount to be distributed;
        // (ps / periods) * (owed + bonus) =
        uint amount = (ps * owed) / periods;

        // deduct the face value from total deposits,
        // and account for the new withdrawal(s)
        withdrawals[addr] = w + uint8(ps);
        require(token.transfer(addr, amount));
        return true;
    }
```

Finally, `withdrawTo(addr)` can be verified through checking if the sets are correct (again, hand-waving). The function signature can be defined as:

$$wto(INPUT) := A \to \{true, false\} \cup E \tag{12}$$

But we're interested in the side-effects taking place within the function. We'll define all possible paths (sorry this is a little rough). For convenience, some global variables are defined here.

$$S := (b, d, p, w) \tag{13}$$

This tuple represents the input state, where:

$$d := x \in X_{256} \tag{14}$$

$$b := \{x \in X_{256} | x \geq d\} \tag{15}$$

$$p := p(INPUT) \tag{16}$$

$$w := wit(x) \tag{17}$$

The function fails immediately if `locked()` or `startBlock == 0`, that is our first exit condition. Secondly, if $p \in E$ then this function also returns $x \in E$ (throws).

Continuing, we arrive at a check that $w > p \land w \geq \alpha$, meaning that the user has used up all their withdraws. This is possible since there are elements in $X_{\alpha+1} > X_{\alpha}$. This is our next exit condition, if evaluated to true. Past this point, we define:

$$w_{checked} := \{x \in X_8 | 0 \leq x \leq p\} \tag{18}$$

Our next exit condition is an assertion that $b \geq d$, this should always be true since everytime d is incremented so is b. We can conclude that this should *never* throw.

$$\begin{aligned} o_{inner} &:= dep(addr) \cdot b \\ o &:= \lfloor (o_{inner})/d \rfloor \end{aligned} \tag{19}$$

Where / refers to integer division. Here we focus on whether the statement o_{inner} violates X_{256}. Our upper bound for our totalSupply is defined to be 2^{96}, therefore the upper limit case is:

$$o_{inner} = 2^{96} * 2^{96} = 2^{192} \tag{20}$$

Meaning that o_{inner} will never overflow. There is however the matter of round-off error to consider. $o \in X_{256} | o >= dep(addr)$ Then represents our *total* payout over the number of periods. To calculate the current payout we first calculate ps.

$$ps := 1 + p - w_{checked} \tag{21}$$

From our definition of $w_{checked}$, we can assert that this variable ranges from $[1, \alpha]$.

$$amt := \lfloor (ps * o)/\alpha \rfloor \tag{22}$$

$$amt = 2^8 * 2^{192} = 2^{200} \tag{23}$$

The upper bound calculation shows us we are still safe (but still the modulo stuff). Finally, lets define a function called $wit(x, v)$ that stores the input $v \in X_8$ input mapping with key x. Where

$$\begin{aligned} v &:= w + uint8(ps) \\ v &:= w + 1 + p - w \\ v &:= p + 1 \end{aligned} \tag{24}$$

Basically meaning that the entry gets updated to the latest period.

Round-off Loss