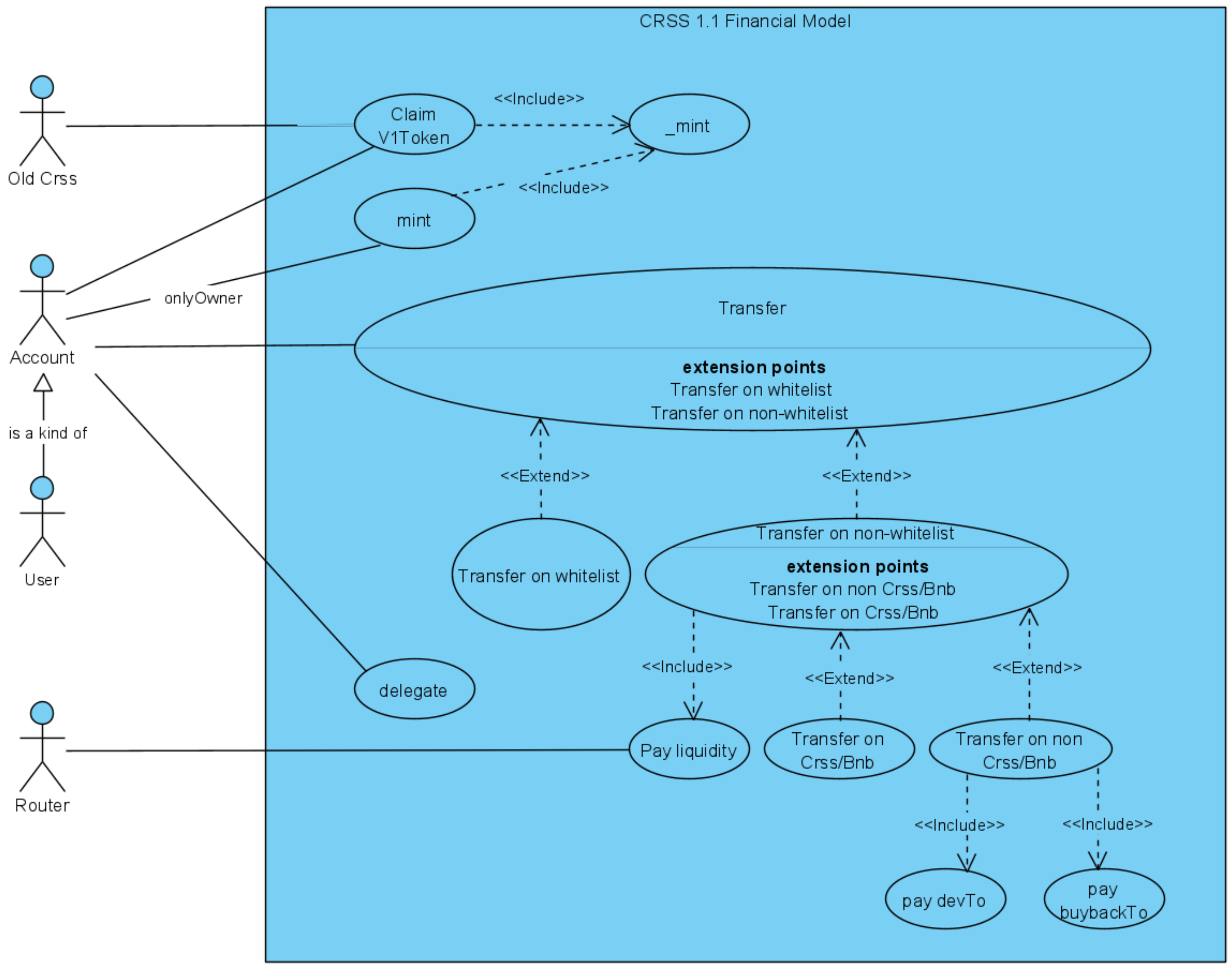
Source Core Review - CRSS Token

1. Finance

1.1 Functionalities

```
- The crss token contract provides the following use cases:
```



```
- **issue**: What's the use of the mint(.) function?
- **issue**: If the mint function is there, the burn function should also be there
for symmetricity.
```

1.2 A transferFrom(.) between any two accounts, (Sender, Recipient, Amount)

```
Note: this function is called by the router when, among others, **an account sells
or adds crss to the crss/bnb pair**.
```

```
From the contract's transferFrom(.) code, we can  is interpreted as follows:



- IF any of Sender and Recipient is whitelisted:
    - Amount is transferred, Sender -> Recipient
    - the same voting power is transferred, delegates[Sender] -> delegates[Recipient]
    - **issue**: Shouldn't whitelisted users pay fees?
    - CHECK: total transfers should be equal to Amount. **Correct**

- ELSE:

    - IF non-pool -> any:
        -  liquidityFee = Amount x liquidityFeeRate  is added to the crss/bnb pool
        - the resulting lp token is sent to the crss contract
        - the same voting power as the fee is transferred to the crss contract
        - **comment**: user-user, user-pool, user-nonPoolContract, nonPoolContract-user, nonPoolContract-pool, nonPoolContract-nonPoolContract.
        **LiquidityFee is paid** if only the sender is not the crss/bon pool.
        - This **effectively exempt, from paying liquidity fee, the transfer of buying crss and removing liquidity.**
        - **comment**: user-pool will not happen. nonPoolContract-pool should be either selling crss or adding liquidity.
        - **issue**: Should adding liquidity pay a fee?
    End IF



    - IF non-pool -> pool:
        -  liquidityFee  was already paid.
        - the transfer is made with  Amount - liquidityFee .
        - the same voting power is transferred, Sender -> Recipient
        - **No more fees, in addition to liquidity fee**
        - **comment**: user-pool will not happen. nonPoolContract-pool should be either swapping crss for bnb or adding liquidity.
        - **issue**: Shouldn't selling pay devFee and buybackFee?
        - CHECK: total transfers should be equal to Amount. **Correct**

    - ELSE IF pool -> pool:
        - this transfer does **not** happen
        - liquidityFee was **not** paid
        - the transfer is made with Amount
        - the same voting power is transferred, Sender -> Recipient
        - **This does not happen**
        - CHECK: total transfers should be equal to Amount
        - CHECK: total transfers should be equal to Amount. **Correct**

    - ELSE IF non-pool -> non-pool:
        - liquidityFee was already paid.

        -  (Amount - liquidityFee) x devFeeRate  is transferred, Sender -> devTo
        - the same voting power as the fee is transferred, delegates[Sender] ->
delegates[devTo]

        -  (Amount - liquidityFee) x buybackFeeRate  is transferred, Sender ->
buybackTo
        - the same voting power as the fee is transferred, delegates[Sender]  ->
delegates[buybackTo]

        -  Amount x ( 1 - devFeeRate - buybackFeeRate ) - Amount x
liquidityFeeRate , is transferred, Sender -> Recipient
        - the same voting power as the fee is transferred, delegates[Sender] ->
delegates[Recipient]

        - **comment**: user-user, nonPoolContract-user, user-NonPoolContract,
nonPoolContract-nonPoolContract.
        - **devFee and buybackFee are paid, in addition to liquidity fee**

        - CHECK: total transfers should be equal to Amount

            total transfers =
            Amount x liquidityFeeRate
            + (Amount - liquidityFee) x devFeeRate
            + (Amount - liquidityFee) x buybackFeeRate
            + Amount x ( 1 - devFeeRate - buybackFeeRate ) - Amount x
liquidityFeeRate
            = Amount - liquidityFee x (devFeeRate + buybackFeeRate)
            = Amount - Amount x liquidityFeeRate x (devFeeRate + buybackFeeRate)
            < Amount

            **Wrong. Recipient receives due amount but Sender pays devFee and
buybackFee less than expected.**
        - Fortunately, non-pool -> non-pool transfer does not take place by
CrossWise contract.
        Instead, 3rd party contracts that transfer crss tokens will see
inconsistency.

    - ELSE IF pool -> non-pool:
        - liquidityFee was **not** paid

        -  Amount x devFeeRate  is transferred, Sender -> devTo
        - the same voting power as the fee is transferred, delegates[Sender] ->
delegates[devTo]

        -  Amount x buybackFeeRate  is transferred, Sender -> buybackTo
        - the same voting power as the fee is transferred, delegates[Sender]  ->
delegates[buybackTo]

        -  Amount x ( 1 - devFeeRate - buybackFeeRate ) , is transferred, Sender -
> Recipient
        - the same voting power as the fee is transferred, delegates[Sender] ->
delegates[Recipient]
```

```
        - **comment**: pool-user is impossible. pool-nonPoolContract is either
swapping bnb for crss or removing liquidity.
        - **devFee and buybackFee are paid, with liquidity fee not paid**
        - **issue**: Should buying pay a fee?

        - CHECK: total transfers should be equal to Amount

            total transfers =
            Amount x devFeeRate
            + Amount x buybackFeeRate
            + Amount - Amount x devFeeRate - Amount x buybackFeeRate - Amount x
liquidityFeeRate
            = Amount x (devFeeRate + buybackFeeRate + 1 - devFeeRate -
buybackFeeRate )
            = Amount

            **Correct.**
    End IF
End IF
```

1.3 A transfer(.) between any two accounts, (Recipient, Amount)

Note: this function is called by the crss/bnb pair when, among others, **an account buys or remove crss from the crss/bnb pair**. This function has NO financial errors, unlike transferFrom(.).

1.4 swapAndLiquify() function

```
- **issue** This function copies conventional code, which produces a significant
residue of bnb in the crss contract.
- See the Section 3.1.5 for more.
```

1.4 swapAndLiquify() function

```
- Problem: swapAndLiquify() is gas-expensive, yet called for every transfer that
pays liquidity fee, however small the fee is.
- This problem was already listed in the Certik's audit report.
- Solution: collect liquidity fees to a given account until it accumulates over a
given threshold, before liquifying.
```

Summary:

```
- What's the use of the mint(.) function?
- If the mint function is there, the burn function should also be there for
symmetricity.
```

```
  - Shouldn't whitelisted users pay fees?
  - Should adding liquidity pay a fee?
  - Shouldn't selling pay devFee and buybackFee?
  - Should buying pay a fee?
  -
  - **Error**: nonPool-nonPool transfers, which covers user-user transfers,
    pay Amount x liquidityFeeRate x (devFeeRate + buybackFeeRate) less fees.
    This is a loss to the system, and may cause residue tokens in non-user accounts.
```

1.5 Discussion

1.5.1 How was the tokenomics calculated?

Throughout the industry, it still remains a wish to choose the best optimal combination of parameters of the operations: collecting fees, liquifying, farming, lending, voting power, minting, and burning. For the coming launch, we many need to keep it open for the next version to implement dynamic control of the parameters.

1.5.3 Is transfer call the only chance to collect fees from holders?

We should be able to collect fees from not only senders/recipients, but also from those who keep silent and hodle. Often, silent users benefit more from the deflation of crss. It would be better if you have the list of users and can enumerate each user to collect fees from them. Enumerating and iterating over all users is limited due to the block gas limit, if the number of users is large.

2. Security

2.1 antiWhale check can not resist a transitive whaling

```
- Logic: antiWhale check resists transfers, only if both the sender and recipient
are whales.
- Problem: A whale can transfer to a non-whale, and the non-whale can, in relay,
transfer to another whale.
- Solution: Stop transfers if any of the sender or recipient is a whale
- Note: All accounts are initially assumed to be a whale except two account.
```

2.2 whitelist check can not resist a transitive black transfer

```
- Logic: whitelist check resists transfers, only if both the sender and recipient
are non-whitelisted.
- Problem: A non-whilelisted can tarnsfer to a whitelisted, and the whitelisted
can, in relay, transfer to another non-whitelisted.
- Solution: Stop transfers if any of the sender or recipient is a not-
whiltelisted.
```

2.3 _mint(account, Amount) function

```
- Logic: they do mint minimum( maxSupply - totalSupply, Amount)
- Problem: The caller will wrongfully think the full Amount was minted.
- Solution: return how much was really minted.
```

3. Programming

3.1 Logic

3.1.1 Simple flaws

```
- _msgSender(), in place of msg.sender, will incur higher gas spending, while it
will not provide much value.
- There should be a check for the existence of crosswiseRouter value before
calling it. The calls will revert.
- ClaimV1Token(): the function has an implicit argument, claimer which is
msg.sender, degrading readability/productivity.
- ClaimV1Token(): oldCrss.transferFrom(_msgSender(), burnAddress, balance) is
impossible, before the msg.sender approves
  to itself the same amount.
```

3.1.2 ClaimV1Token() function

```
- Logic: Burn the v1 balance of the claimer, then mint v11 tokens, of the same
amount, to the claimer.
- Problem: When the v1 balance is burnt, a transfer() occurs, which collects fees
to other accounts.
  It means some of the burn amount remains unburnt.
  So the the total amount that moves to the new system is larger than the
claimer's v1 balance. This is consistency.
```

3.1.3 _moveDelegates(.) function

```
- Logic: A call of this function follows all _transfer functions.
  Its arguments can be calculated from the preceding _transfer function call. This
is an indirect redundancy.
- Solution: Create a new concept, like _jointTransfer, which calls both _transfer
and _moveDelegates.
  It's quite natural because a crss balance and and the voting power is coherent
and forms a new concept.
```

3.1.4 transfer(.) or transferFrom(.) functions

```
- Logic: The intended logic is explained in the Section 1.2.
- problem: The error is explained in the Section 1.2
- Solution: See the Section 3.2.1.
```

3.1.5 swapAndLiquify(.) function, internal.

```
- Logic: Split the given tokens into two haves. Swap one half for bnb.
  Add the remaining half, along with the swapped-for bnb, to the liquidity pool.
  Let the resulting LP tokens go to Crss contracts LP account.
- Problem: Splitting half-and-half is not precise. The residue bnb amount is
significant.
- Solution: We can predict the exact split ratio instead of 50-50, with some added
gas spending.
  The residue bnb will be reduced to a hundredth.
```

3.1.6 swapTokensForBN(.) function, internal.

```
- Logic: crosswiseRouter.swapExactTokensForETHSupportingFeeOnTransferTokens(.)
call is used to swap crss for bnb.
- Problem: This call is gas-expensive.
  This call is only suitable when the swap path includes more than one pairs and
the between-pair transfer is subject to fees.
  The swap this function aims to perform, is has a single pair path.
- Solution: use the simple crosswiseRouter.swapExactTokensForETH(.) call.
```

3.1.7 mint(.) function

```
- Logic: Stores a Mint event.
- Problem: This is redundant, as Transfer(zeroAddess, to, amount) is separately
stored.
- Solution: Remove.
```

3.2 Maintainability

3.2.1 transferFrom(.) function

- Logic: The code comes below:

  ```
      if (whitelist[recipient] || whitelist[sender]) {
          _transfer(sender, recipient, amount);
          _moveDelegates(_delegates[sender], _delegates[recipient], amount);
  ```

```
      emit WhitelistedTransfer(sender, recipient, amount);
  } else {

      uint256 transferAmount = amount.mul(10000 - devFee -
buybackFee).div(10000);

      if (
          !inSwapAndLiquify &&
          sender != crssBnbPair &&
          swapAndLiquifyEnabled
      ) {
          uint256 liquidityAmount = amount.mul(liquidityFee).div(10000);
          transferAmount = transferAmount.sub(liquidityAmount);

          // AUDIT : CTC-10 | Liquidity Fee Calculation
          amount = amount.sub(liquidityAmount);

          _transfer(sender, address(this), liquidityAmount);

          // AUDIT : CTC-09 | Voting Power Not Moved Along With Liquify
          _moveDelegates(_delegates[sender], _delegates[address(this)],
liquidityAmount);
          swapAndLiquify();
      }

      if(recipient == crssBnbPair) {
          _transfer(sender, recipient, amount);
          _moveDelegates(_delegates[sender], _delegates[recipient], amount);
      }
      else { // amount = amount - liquidity if sender is non-pool, else
amount
              // transferAmount = amount - liquidity - devFee - buybackFee
if sender is non-pool, else amount - liquidity - devFee.
          _transfer(sender, recipient, transferAmount);
          _moveDelegates(_delegates[sender], _delegates[recipient],
transferAmount);

          _transfer(sender, devTo, amount.mul(devFee).div(10000));
          _moveDelegates(_delegates[sender], _delegates[devTo],
amount.mul(devFee).div(10000));

          _transfer(sender, buybackTo, amount.mul(buybackFee).div(10000));
          _moveDelegates(_delegates[sender], _delegates[buybackTo],
amount.mul(buybackFee).div(10000));
      }
  }


  The intended logic is explained in the section 1.2. The actual code is as
follows:
  The function call comes along with its argument 'amount', which is the
```

```
amount to transfer.
  Later, the code introduces another variable 'transferAmount. Readers begin
to be confused, degrading productivity.
  Both the 'amount' and 'transfer' amount are subject to changes as the code
flows.
  Then, the following two factors are completely mingled, adding to the
complexity.
  - deciding what and how much fees the transfer has to pay
  - transferring the fees, along with the voting power.
  - **The code is so difficult to read that the coder himself was confused
and ended up with financial error.** See Section 1.2.
```

- Solution: (Sketched) amountAfterFee = amount; if(not isFeeFree(sender, recipient)) { amountAfterFee = payFees(sender, recipient, amount); } jointTransfer(sender, recipient, amountAfterFee);

  The above code factors the whole code into isFeeFree(.), payFees(.), and jointTransfer(.), removing the complexity and improving readability and productivity. The factor functions are focusing on a small, special topic and easier to implement. If gas spending matters, we can put the code in the place of the calls.

3.2.2 transfer(.) and transferFrom(.) functions

```
- Logic: the two functions, inevitably, has the almost the same code.
- Problems: As they essentially performs the same function, each time the two have
to change simultaneously.
  The error mentioned in the Section 1.2 came partly from the redundancy.

- Solution:
    transfer(recipient) is actually fransferFrom(me, recipient).
    Let tansfer(.) delegate to transferFrom with the sender being the msg.sender.
    The reason they couldn't to this, is that fransferFrom() has, unlike
transfer(), the allowance check.
    So, the industry convention is: (sketch only)

    transfer(recipient, amount) {
        _tansferFrom(me, recipient, mount);
    }
    transferFrom(sender, recipient, amount) {
        check if sender approved recipient to spend amount.
        _transfer(sender, recipient, amount);
    }
    _tansfer(sender, recipient, amount) {
        perform the actual transfer, **with all the complex fees/management
logic.**
    }

    The reason they couldn't do this, is they used out _transfer(.) function for
```

```
the core transfer.
    Introduce __transfer(.) function for the core transfer, which just moves
tokens between the two accounts.
```