Tackling Rounding Errors with Precision Analysis

Raoul Schaffranek

Formal Verification Engineer @ Runtime Verification Inc.

Overview

- ~~Quantifying Rounding Errors + Error Propagation~~
- Rounding directions
    - Real world examples
    - The two-way trading problem
    - Choosing a rounding function
    - Fuzzing with Foundry
    - Symbolic Execution with KEVM

Rounding direction errors

runtime verification

Uniswap V1[1]

Caught before deployment

Solana Token Lending Contract[2]

2.3B$ TVL at risk

Solana Token Stable Swap[3]

700M$ TVL at risk

1) https://github.com/runtimeverification/verified-smart-contracts/blob/master/uniswap/issues.md

2) https://blog.neodyme.io/posts/lending_disclosure/

3) https://osec.io/blog/reports/2022-04-26-spl-swap-rounding/

The two-way trading problem

Exchange rate: $r = 2\mathbb{G} / 1\$$

Exact arithmetic:

Sell $1\mathbb{G}$, get $1/2\$$ out: $1\mathbb{G} / r = 1\mathbb{G} / (2\mathbb{G} / 1\$) = 1/2\$$

Sell $1/2\$$, get $1\mathbb{G}$ out: $1/2\$ \cdot r = 1/2\$ \cdot (2\mathbb{G} / 1\$) = 1\mathbb{G}$

Rounding to nearest neighbor:

Sell $1\mathbb{G}$, get $1\$$ out: $[1\mathbb{G} / r] = [1\mathbb{G} / [2\mathbb{G} / 1\$]] = 1\$$

Sell $1\$$, get $2\mathbb{G}$ out: $[1\$ \cdot r] = [1\$ \cdot [2\mathbb{G} / 1\$]] = 2\mathbb{G}$

Trading-pair functions

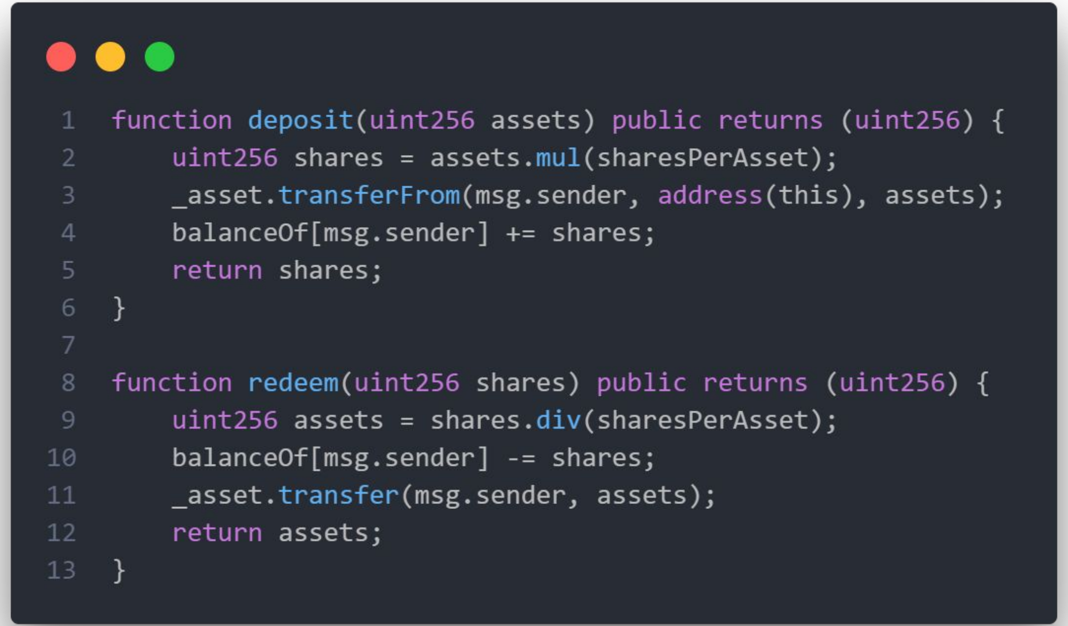
- deposit/redeem
- stake/unstake
- addLiquidity/removeLiquidty
- swap/swap

```
redeem(deposit(1𝔾))      = 2𝔾

redeem(deposit(1𝔾))      ≤ 1𝔾

redeem(deposit(amount)) ≤ amount
```

deposit/redeem



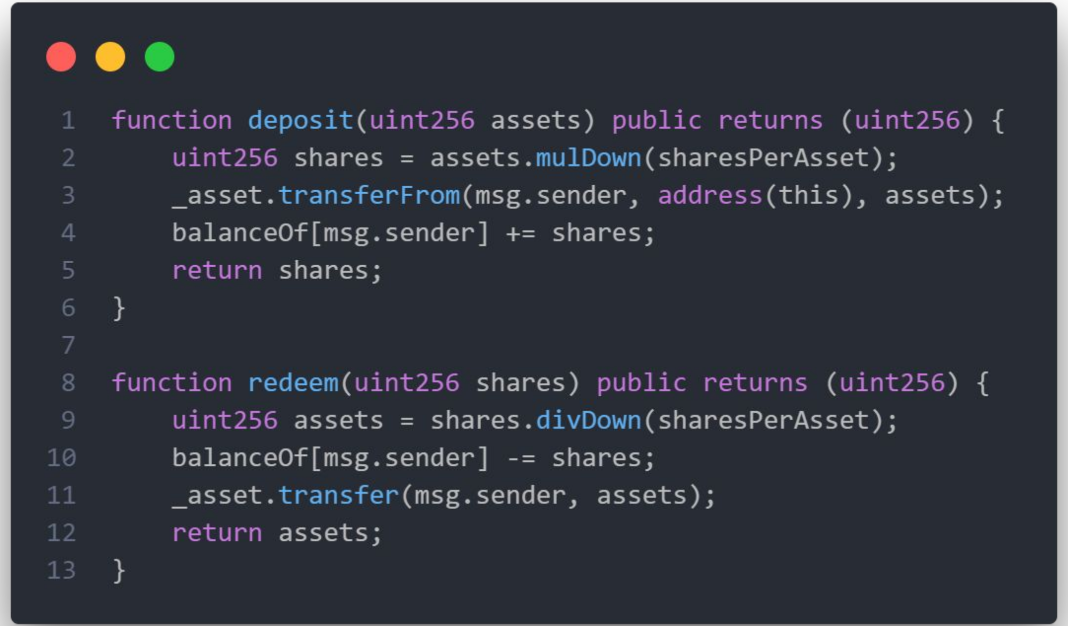
```
function deposit(uint256 assets) public returns (uint256) {
    uint256 shares = assets.mul(sharesPerAsset);
    _asset.transferFrom(msg.sender, address(this), assets);
    balanceOf[msg.sender] += shares;
    return shares;
}

function redeem(uint256 shares) public returns (uint256) {
    uint256 assets = shares.div(sharesPerAsset);
    balanceOf[msg.sender] -= shares;
    _asset.transfer(msg.sender, assets);
    return assets;
}
```

Mindful rounding

- Accept rounding errors
- Higher precision is not always better
- Direction matters
- Keep the change:
  - Round up for incoming assets
  - Round down for outgoing assets

deposit/redeem revisited

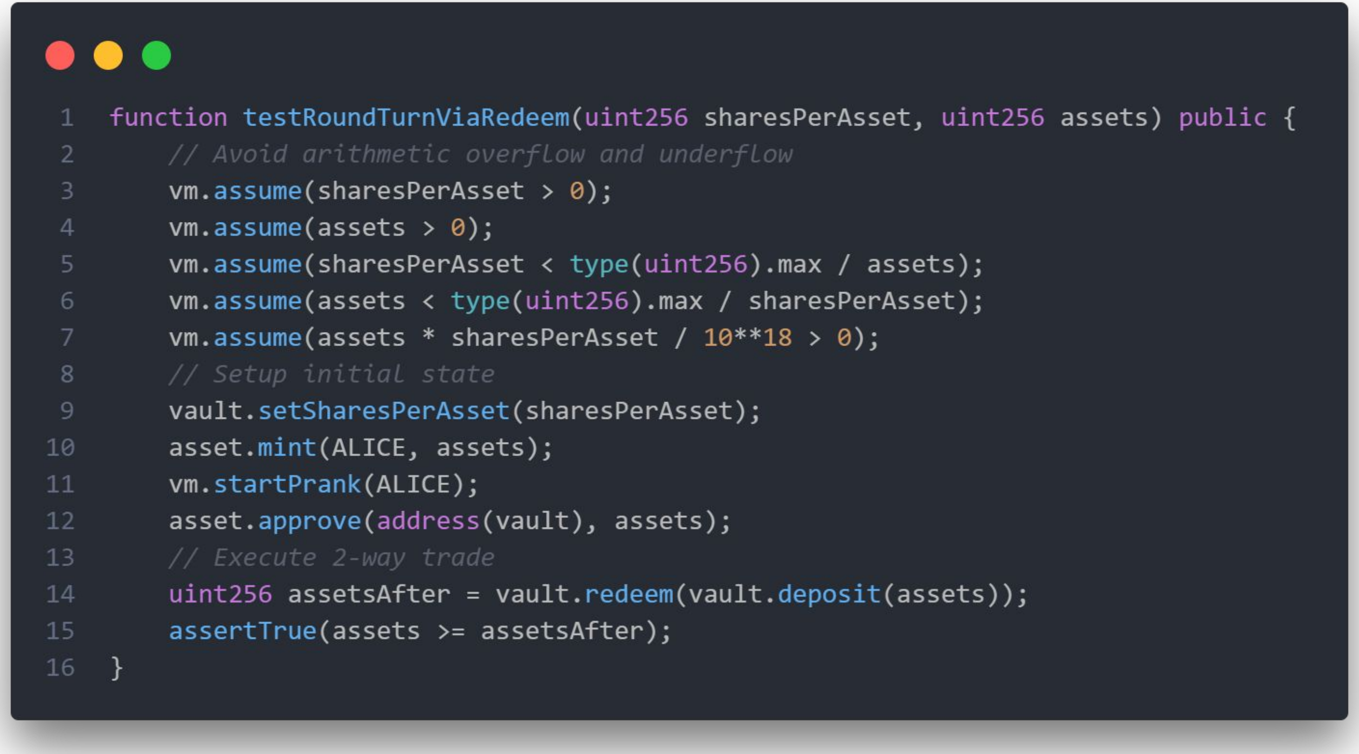


```
function deposit(uint256 assets) public returns (uint256) {
    uint256 shares = assets.mulDown(sharesPerAsset);
    _asset.transferFrom(msg.sender, address(this), assets);
    balanceOf[msg.sender] += shares;
    return shares;
}

function redeem(uint256 shares) public returns (uint256) {
    uint256 assets = shares.divDown(sharesPerAsset);
    balanceOf[msg.sender] -= shares;
    _asset.transfer(msg.sender, assets);
    return assets;
}
```

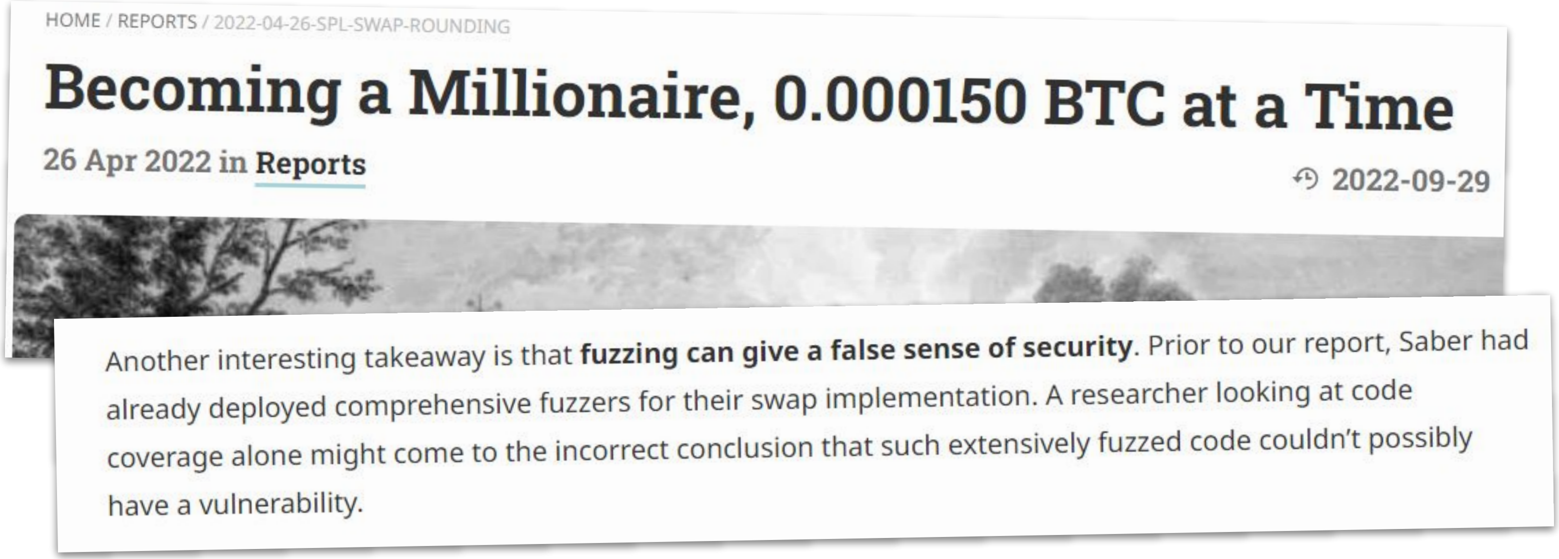
Fuzzing with Foundry

Reminder: redeem(deposit(amount)) ≤ amount



```
function testRoundTurnViaRedeem(uint256 sharesPerAsset, uint256 assets) public {
    // Avoid arithmetic overflow and underflow
    vm.assume(sharesPerAsset > 0);
    vm.assume(assets > 0);
    vm.assume(sharesPerAsset < type(uint256).max / assets);
    vm.assume(assets < type(uint256).max / sharesPerAsset);
    vm.assume(assets * sharesPerAsset / 10**18 > 0);
    // Setup initial state
    vault.setSharesPerAsset(sharesPerAsset);
    asset.mint(ALICE, assets);
    vm.startPrank(ALICE);
    asset.approve(address(vault), assets);
    // Execute 2-way trade
    uint256 assetsAfter = vault.redeem(vault.deposit(assets));
    assertTrue(assets >= assetsAfter);
}
```

When fuzzing is not enough



HOME / REPORTS / 2022-04-26-SPL-SWAP-ROUNDING

Becoming a Millionaire, 0.000150 BTC at a Time

26 Apr 2022 in Reports

2022-09-29

Another interesting takeaway is that **fuzzing can give a false sense of security**. Prior to our report, Saber had already deployed comprehensive fuzzers for their swap implementation. A researcher looking at code coverage alone might come to the incorrect conclusion that such extensively fuzzed code couldn't possibly have a vulnerability.

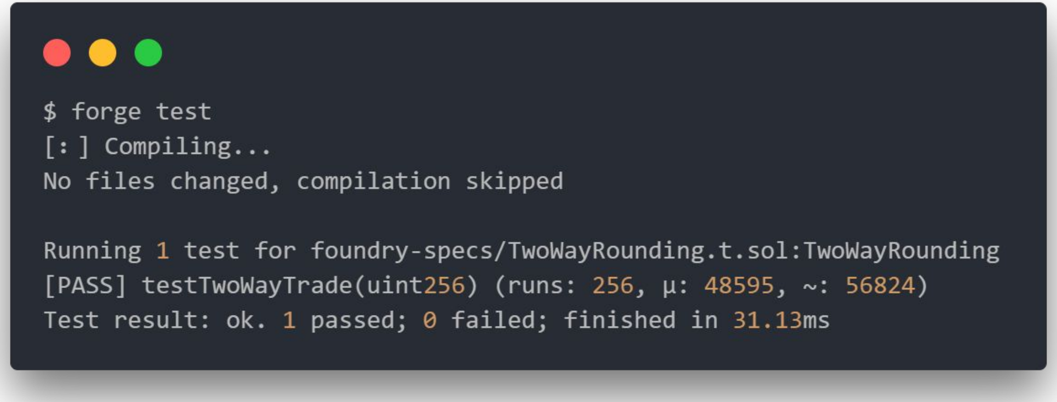
Source: https://osec.io/blog/reports/2022-04-26-spl-swap-rounding/

Fuzzing ♥ Symbolic Execution

runtime verification

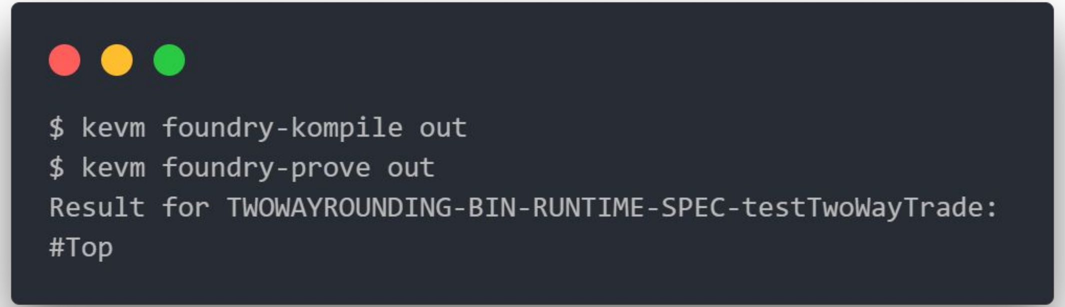
Fuzzing with Foundry	Symbolic Execution with KEVM
Write test using Solidity	Reuse Foundry tests
Expressiveness limited to Solidity	Enhanced expressiveness with K-language
Extremely fast	Slow
No human intervention required	Sometimes requires human intervention
Randomized inputs	Symbolic Inputs = 100% input coverage
No false positives	No false positives
False negatives	No false negatives
Easy to use	Easy to try, hard to master

Foundry ♥ KEVM



```
$ forge test
[:] Compiling...
No files changed, compilation skipped

Running 1 test for foundry-specs/TwoWayRounding.t.sol:TwoWayRounding
[PASS] testTwoWayTrade(uint256) (runs: 256, μ: 48595, ~: 56824)
Test result: ok. 1 passed; 0 failed; finished in 31.13ms
```



```
$ kevm foundry-kompile out
$ kevm foundry-prove out
Result for TWOWAYROUNDING-BIN-RUNTIME-SPEC-testTwoWayTrade:
#Top
```

runtime
verification
research

research.runtimeverification.com

runtime verification

Rikard Hjort: Formal Methods for the Working DeFi Dev
Oct 12th — 11:00 AM - 11:50 AM @ Workshop 3

Questions?

https://runtimeverification.com/

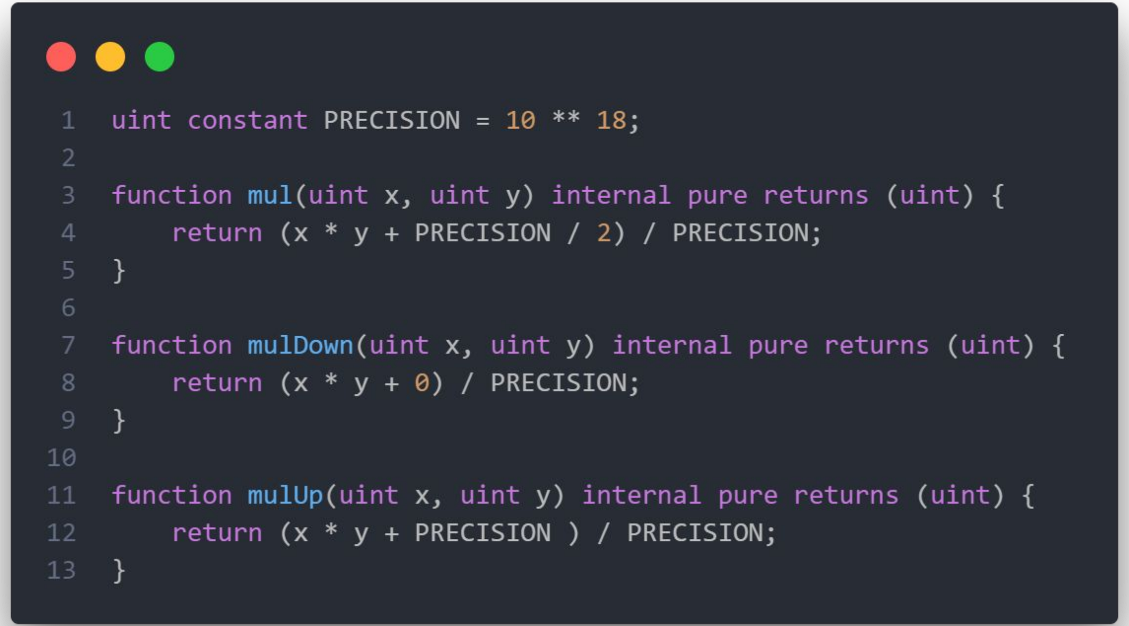
@rv_inc

https://discord.com/invite/CurfmXNtbN

contact@runtimeverification.com

Appendix.

Rounding multiplication



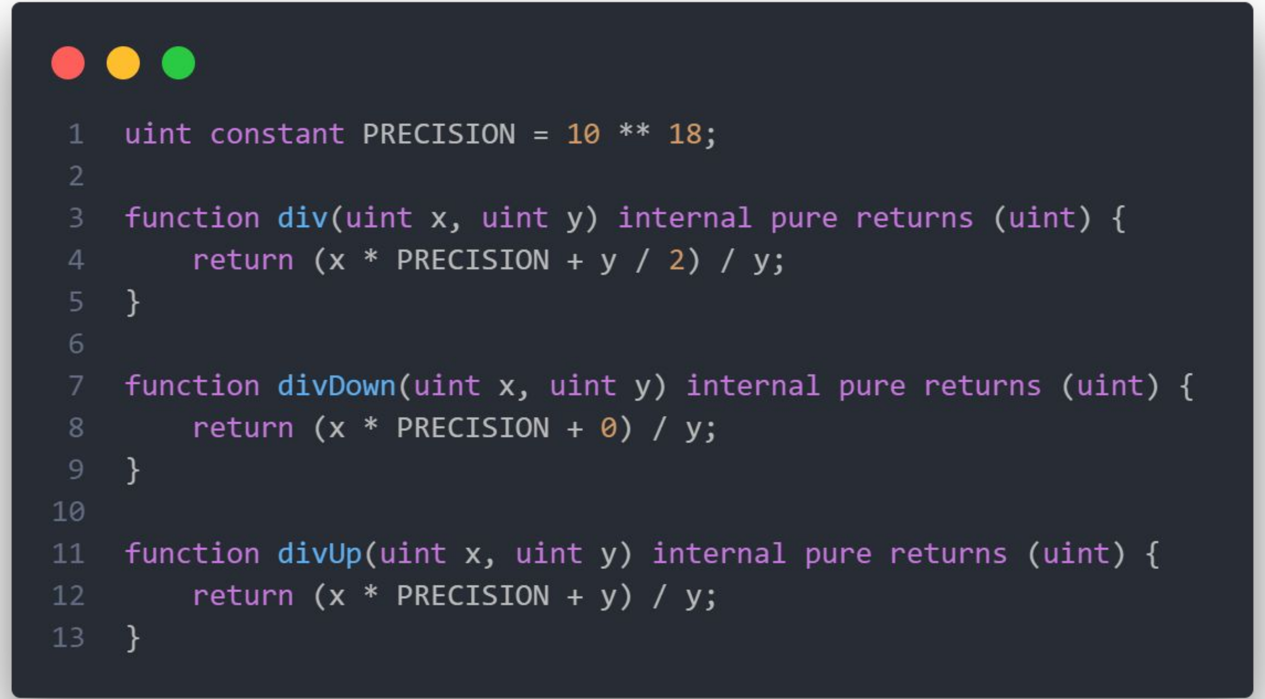
```
uint constant PRECISION = 10 ** 18;

function mul(uint x, uint y) internal pure returns (uint) {
    return (x * y + PRECISION / 2) / PRECISION;
}

function mulDown(uint x, uint y) internal pure returns (uint) {
    return (x * y + 0) / PRECISION;
}

function mulUp(uint x, uint y) internal pure returns (uint) {
    return (x * y + PRECISION ) / PRECISION;
}
```

Rounding division



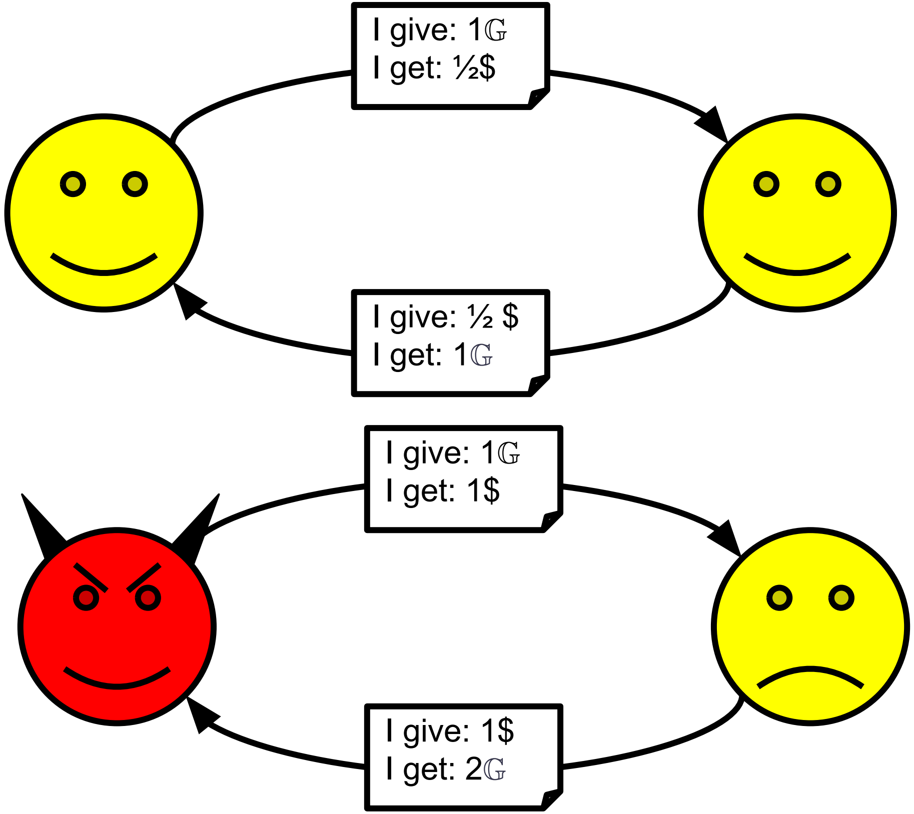
```
uint constant PRECISION = 10 ** 18;

function div(uint x, uint y) internal pure returns (uint) {
    return (x * PRECISION + y / 2) / y;
}

function divDown(uint x, uint y) internal pure returns (uint) {
    return (x * PRECISION + 0) / y;
}

function divUp(uint x, uint y) internal pure returns (uint) {
    return (x * PRECISION + y) / y;
}
```

The two-way trading problem



I give: 1𝔾
I get: ½$

I give: ½ $
I get: 1𝔾

I give: 1𝔾
I get: 1$

I give: 1$
I get: 2𝔾

Title

Text