Tackling Rounding Errors with Precision Analysis
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

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/
The two-way trading problem

Exchange rate: \( r = 2\text{G} / 1\$ \)

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

Rounding to nearest neighbor:
Sell \( 1\text{G} \), get \( 1\$ \) out: \[ 1\text{G} / r = [1\text{G} / [2\text{G} / 1\$]] = 1\$ \]
Sell \( 1\$ \), get \( 2\text{G} \) out: \[ 1\$ \cdot r = [1\$ \cdot [2\text{G} / 1\$]] = 2\text{G} \]
Trading-pair functions

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

redeem(deposit(1G)) = 2G
redeem(deposit(1G)) ≤ 1G
redeem(deposit(amount)) ≤ amount
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

```
1 function testRoundTurnViaRedeem(uint256 sharesPerAsset, uint256 assets) public {
2     // Avoid arithmetic overflow and underflow
3     vm.assume(sharesPerAsset > 0);
4     vm.assume(assets > 0);
5     vm.assume(sharesPerAsset < type(uint256).max / assets);
6     vm.assume(assets < type(uint256).max / sharesPerAsset);
7     vm.assume(assets * sharesPerAsset / 10**18 > 0);
8     // Setup initial state
9     vault.setSharesPerAsset(sharesPerAsset);
10    asset.mint(ALICE, assets);
11    vm.startPrank(ALICE);
12    asset.approve(address(vault), assets);
13    // Execute 2-way trade
14    uint256 assetsAfter = vault.redeem(vault.deposit(assets));
15    assertTrue(assets >= assetsAfter);
16 }
```
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

<table>
<thead>
<tr>
<th>Fuzzing with Foundry</th>
<th>Symbolic Execution with KEVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write test using Solidity</td>
<td>Reuse Foundry tests</td>
</tr>
<tr>
<td>Expressiveness limited to Solidity</td>
<td>Enhanced expressiveness with K-language</td>
</tr>
<tr>
<td>Extremely fast</td>
<td>Slow</td>
</tr>
<tr>
<td>No human intervention required</td>
<td>Sometimes requires human intervention</td>
</tr>
<tr>
<td>Randomized inputs</td>
<td>Symbolic Inputs = 100% input coverage</td>
</tr>
<tr>
<td>No false positives</td>
<td>No false positives</td>
</tr>
<tr>
<td>False negatives</td>
<td>No false negatives</td>
</tr>
<tr>
<td>Easy to use</td>
<td>Easy to try, hard to master</td>
</tr>
</tbody>
</table>
$ forge test
[.] Compiling...
No files changed, compilation skipped

Running 1 test for foundry-specs/TwoWayRounding.t.sol:TwoWayRounding
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
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

```solidity
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

```solidity
1  uint constant PRECISION = 10 ** 18;
2  
3  function div(uint x, uint y) internal pure returns (uint) {
4      return (x * PRECISION + y / 2) / y;
5  }
6  
7  function divDown(uint x, uint y) internal pure returns (uint) {
8      return (x * PRECISION + 0) / y;
9  }
10 
11 function divUp(uint x, uint y) internal pure returns (uint) {
12      return (x * PRECISION + y) / y;
13  }
```
The two-way trading problem

I give: $1$
I get: $\frac{1}{2}G$

I give: $\frac{1}{2}G$
I get: $1$

I give: $1$
I get: $1$

I give: $1$
I get: $2$

I give: $2$
I get: $1$

I give: $1$
I get: $1$

I give: $1G$
I get: $\frac{1}{2}G$

I give: $\frac{1}{2}G$
I get: $1G$

I give: $1G$
I get: $1$

I give: $1$
I get: $1$

I give: $1$
I get: $2G$

I give: $2G$
I get: $1$