Numerical Analysis for DeFi Audits: A TWAMM Case Study

Kurt Barry
https://twitter.com/Kurt_M_Barry

Analyzing DeFi Math

Order-of-magnitude reasoning for (over|under)flow checking

Testing the worst-case first

Safety in the presence of rounding error

SMT solver hax

TWAMM? (https://www.paradigm.xyz/2021/07/twamm)

Constant-product AMM with a “Time-Weighted Average Price” (TWAP) order type.

E.g. selling N tokens per block for 1000 blocks at the market price.

Spearbit audited an implementation by Cron Finance.

A desire for gas efficiency lead to a lot of optimizations that needed careful numerical analysis.

Double Overflow of Scaled Proceeds

Scaled Proceeds Accounting

Proceeds stored with only 128 bits so values for both tokens can be packed into a single storage slot.

tokens purchased per **order block interval**

tokens sold per **block**

```
scaledProceedsU128F64 = uint128(
  uint128(_scaledProceedsU128F64) + uint128((_tokenOutU112 << C.B64) / _salesRateU112)
);
```

Overflow Intended - one overflow is fine since only **differences** between scaled proceeds values matter.

If two or more occur between consecutive claims by a long-term trader, **funds will be lost**.

Order-of-Magnitude and Proportionality Analysis

~ price * OBI * N * 10^(decimals out)

$2^{64} \sim 10^{19}$

N * 10^(decimals in)

```
scaledProceedsU128F64 = uint128(
  uint128(_scaledProceedsU128F64) + uint128((_tokenOutU112 << C.B64) / _salesRateU112)
);
```

Since $2^{128} \sim 10^{38}$, (_tokenOutU112 / _salesRateU112) needs to be << $\sim 10^{19}$ for safety.

_tokenOutU112 / _salesRateU112 ~ price * OBI * 10^(decimals out) / 10^(decimals in)

Check the Worst-Case First

If an extreme example is fine, we don't have to worry anymore.

Extreme but realistic example: DAI (18 decimals) being bought with GUSD (2 decimals):
- price ~ 1 (i.e. # of DAI equal in value to 1 GUSD)
- OBI = 64 (stable-stable pool order block interval at audit commit)

$1 * 64 * 10^{18} / 10^{2} = 6.4 * 10^{17}$

Uh-oh…this is concerningly close to 10^{19}.

How long until an overflow?

of OBIs until overflow: $2^{128} / (2^{64} * 6.4 * 10^{17}) \sim 29$

Time to overflow: (29 * 64 * 12) = 22272 seconds ~ 6 hours (!!!)

This is well below the intended “safe” length of time for an order defined by the team (5 years). So, we have an issue.

Next step was to write a test to confirm; always test important findings!

Other Things to Note

Price differences are also dangerous: DAI-WBTC pool (18 vs 8 decimals, and WBTC is a factor of ~10^4 times more valuable) would overflow in less than two weeks.

Imbalances can partially cancel out–e.g. a USDC-ETH pool. A case like this caused the dev team to overlook the severity of the issue going into the audit.

Insofar as price stays constant or at least bounded, the time-to-overflow doesn't depend on the total sales rate, due to the ratio between tokens purchased and tokens being sold.

Solution: instead of hardcoding 2^64, use scaling factors based on decimals of the token in the denominator (extreme prices still a possible issue).

Block Number Subtraction Underflow

The things we do for gas...

Comment that that doesn't quite apply here (orders cancellable post-fill).

```
// #unchecked
//             This subtraction shouldn't underflow because expiry blocks are always
//             greater than the last virtual order block.
//
//             The multiplication is unchecked for overflow because the sales rate is
//             computed from the amount in, which is checked by Balancer (BAL#526
//             BALANCE_TOTAL_OVERFLOW) and known to be less than U112. The maximum
//             difference between the expiry block and last virtual order block is
//             constrained to be U18 (the MAX_ORDER_INTERVALS largest value,
//             STABLE_MAX_INTERVALS), the product of which cannot exceed U130.
refundU112 = (expiryBlock - virtualOrders.lastVirtualOrderBlock) * salesRateU112;
```

This can underflow, but is it exploitable?

Order-of-Magnitude Alone Is Too Simplistic

```
refundU112 = (expiryBlock - virtualOrders.lastVirtualOrderBlock) * salesRateU112;
```

~ 2^{256} ~ 10^{77}

<= ~ 10^{27}
(selling e.g. a billion 18-decimal tokens per-block)

Need to reason about the multiplication mod 2^{256}…

Rewrite to make “closeness” to 2^{256} more obvious

B := virtualOrder.lastVirtualOrderBlock - expiryBlock <= ~10^9
(several hundred years, definite overestimate)

```
refundU112 = (expiryBlock - virtualOrders.lastVirtualOrderBlock) * salesRateU112;
```

$$((2^{256} - 1 - B) * \text{salesRateU112}) \bmod 2^{256}$$

$$((2^{256} * \text{salesRateU122} - \underbrace{(1 + B) * \text{salesRateU112}}_{<= \sim 10^{36} << 2^{256}}) \bmod 2^{256}$$

$$\underbrace{2^{256} - (1 + B) * \text{salesRateU112}}_{\sim 10^{77} - 10^{36} \sim 10^{77}}$$

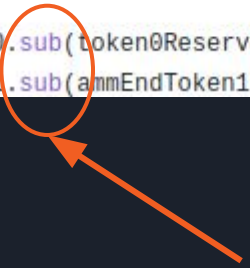
⇒ not actually exploitable

Refactoring For Gas and Safety

Core Invariant Calculation

```
uint256 sum0 = token0ReserveU112 + _token0InU112;
uint256 sum1 = token1ReserveU112 + _token1InU112;
uint256 ammEndToken1 = (token0ReserveU112 * sum1).divDown(sum0);
token0ReserveU112 = (token0ReserveU112 * token1ReserveU112).divDown(ammEndToken1);
token1ReserveU112 = ammEndToken1; // Updating here, otherwise would corrupt "k" value in line above.

token0OutU112 = sum0.sub(token0ReserveU112);
token1OutU112 = sum1.sub(ammEndToken1);
```



Potential concern: if an underflow does occur, the TWAMM “freezes up” and can’t process further trades, deposits, or withdrawals.

Core Invariant Calculation

```
uint256 sum0 = token0ReserveU112 + _token0InU112;
uint256 sum1 = token1ReserveU112 + _token1InU112;
uint256 ammEndToken1 = (token0ReserveU112 * sum1).divDown(sum0);
token0ReserveU112 = (token0ReserveU112 * token1ReserveU112).divDown(ammEndToken1);
token1ReserveU112 = ammEndToken1; // Updating here, otherwise would corrupt "k" value in line above.

token0OutU112 = sum0.sub(token0ReserveU112);
token1OutU112 = sum1.sub(ammEndToken1);
```

Correct-by-construction; of the form:

$y = x - (x*n)/d$ where $n <= d$

$x = sum1$

$n = token0ReserveU112$

$d = sum0 = token0ReserveU112 + \_token0InU112$

Core Invariant Calculation

```
uint256 sum0 = token0ReserveU112 + _token0InU112;
uint256 sum1 = token1ReserveU112 + _token1InU112;
uint256 ammEndToken1 = (token0ReserveU112 * sum1).divDown(sum0);
token0ReserveU112 = (token0ReserveU112 * token1ReserveU112).divDown(ammEndToken1);
token1ReserveU112 = ammEndToken1; // Updating here, otherwise would corrupt "k" value in line above.

token0OutU112 = sum0.sub(token0ReserveU112);
token1OutU112 = sum1.sub(ammEndToken1);
```

Same reasoning ***does not apply*** here.

B/c the division here decreases a denominator, potentially increasing the value subtracted via rounding error.

We can refactor to make both subtractions safe with no downsides!

```
uint256 sum0 = token0ReserveU112 + _token0InU112;
uint256 sum1 = token1ReserveU112 + _token1InU112;
```

```
uint256 ammEndToken0 = (token1ReserveU112 * sum0) / sum1;
uint256 ammEndToken1 = (token0ReserveU112 * sum1) / sum0;
token0ReserveU112 = ammEndToken0;
token1ReserveU112 = ammEndToken1;

token0OutU112 = sum0 - ammEndToken0;
token1OutU112 = sum1 - ammEndToken1;
```

No more freeze-up risk and less gas-intensive: better in every way!

Another nice benefit: manifest symmetry (good sanity check).

Moved on during the audit, as there was a lot of other stuff to check.

Could a freeze up have happened? Enter the SMT solver!

SMT : “Satisfiability Modulo Theories”

Satisfiability: problem of whether a boolean predicate has a satisfying assignment (values for its variables that make it evaluate to true)

Theories: mathematical objects like integers, rationals, reals, bit vectors, etc

SMT solver: tries to find satisfying assignments

E.g. find an assignment for (x > 3*y) && (x - y == 1) && (x > 0) where x, y are integers

- (x=1, y=0)

Using Z3's Python API

```
>>> from z3 import *
>>> t0In = Int("t0In")
>>> t1In = Int("t1In")
>>> t0R  = Int("t0R")
>>> t1R  = Int("t1R")
>>> s = Solver()
>>> s.add(t0In > 0)
>>> s.add(t1In > 0)
>>> s.add(t0R > 0)
>>> s.add(t1R > 0)
>>> sum0 = t0In + t0R
>>> sum1 = t1In + t1R
>>> ammEndToken1 = (t0R * sum1) / sum0;
>>> t0R_after = (t0R * t1R) / ammEndToken1;
>>> s.add(ammEndToken1 > 0)
>>> s.add(sum0 < t0R_after)
>>> s.check()
sat
>>> s.model()
[t0In = 5,
 t1In = 2,
 t0R = 1,
 t1R = 21,
 div0 = [(21, 3) -> 7, else -> 3],
 mod0 = [(21, 3) -> 0, else -> 5]]
```

concrete solution!

```
uint256 sum0 = token0ReserveU112 + _token0InU112;
uint256 sum1 = token1ReserveU112 + _token1InU112;
uint256 ammEndToken1 = (token0ReserveU112 * sum1).divDown(sum0);
token0ReserveU112 = (token0ReserveU112 * token1ReserveU112).divDown(ammEndToken1);
token1ReserveU112 = ammEndToken1; // Updating here, otherwise would corrupt "k" va

token0OutU112 = sum0.sub(token0ReserveU112);
token1OutU112 = sum1.sub(ammEndToken1);
```

underflow condition

Can add constraints to get more realistic assignments

very imbalanced sales rates seem to be required, would be hard to trigger in practice

```
>>> from z3 import *
>>> t0In = Int("t0In")
>>> t1In = Int("t1In")
>>> t0R  = Int("t0R")
>>> t1R  = Int("t1R")
>>> s = Solver()
>>> s.add(And(t0In > 0, t0In < 10**22))
>>> s.add(t1In > 0)
>>> s.add(t0R == 10**21)
>>> s.add(t1R == 76 * 10**21)
>>> sum0 = t0In + t0R
>>> sum1 = t1In + t1R
>>> ammEndToken1 = (t0R * sum1) / sum0;
>>> t0R_after = (t0R * t1R) / ammEndToken1;
>>> s.add(ammEndToken1 > 0)
>>> s.add(sum0 < t0R_after)
>>> s.check()
sat
>>> s.model()
[t0R = 100000000000000000000,
 t1R = 7600000000000000000000,
 t0In = 999999999999999999999,
 t1In = 1,
 div0 = [(760000000000000000000000000000000000000000,
          690909090909090909090) ->
         110000000000000000000,
         else -> 690909090909090909090],
 mod0 = [(760000000000000000000000000000000000000000,
          690909090909090909090) ->
         100000000000000000000,
         else -> 890909090909090909090]]
```

We can also check the refactoring! (don't really need to, but hey, it's fun)

```
>>> from z3 import *
>>> t0In = Int("t0In")
>>> t1In = Int("t1In")
>>> t0R  = Int("t0R")
>>> t1R  = Int("t1R")
>>> s = Solver()
>>> s.add(t0In > 0)
>>> s.add(t1In > 0)
>>> s.add(t0R > 0)
>>> s.add(t1R > 0)
>>> sum0 = t0In + t0R
>>> sum1 = t1In + t1R
>>> ammEndToken1 = (t0R * sum1) / sum0;
>>> ammEndToken0 = (t1R * sum0) / sum1;
>>> s.add(Or(sum0 < ammEndToken0, sum1 < ammEndToken1))
>>> s.check()
unsat
```

```
uint256 ammEndToken0 = (token1ReserveU112 * sum0) / sum1;
uint256 ammEndToken1 = (token0ReserveU112 * sum1) / sum0;
token0ReserveU112 = ammEndToken0;
token1ReserveU112 = ammEndToken1;

token0OutU112 = sum0 - ammEndToken0;
token1OutU112 = sum1 - ammEndToken1;
```

Questions?