Security Review

2025-06-05 by SECBIT Labs

At the request of the AladdinDAO team, we conducted a security review of recent modifications to the f(x) protocol's codebase.

This review covers the following change:

- https://github.com/AladdinDAO/fx-protocol-contracts-internal/pull/1 f(x): refactor `TickMath`, commit `5580e5259232361867df51491aa2a2e1c7620192`

The following report introduces the background, analyzes contract functions, details the changes, presents the security review results, and discusses the current status of this modification.

PR#1 f(x): Refactor TickMath

Change Background

The AladdinDAO team has optimized and adjusted specific parameters and related computational logic within the `TickMath` contract to better align with application requirements.

Contract Functions Analysis

The `TickMath` library contract primarily implements two core functions:

(1). The `getRatioAtTick()` function calculates the `ratioX96` value corresponding to a given tick. This value can be derived in two ways, establishing a mapping between `tick` and the `debtsShares / collsShares` ratio. This assumes no bad debt in the system and that funding costs have not yet accrued (i.e., `rawDebts = debtsShares` and `rawColls = collsShares`).

$$ratioX96 = 2^{96} \times 1.0015^{tick}$$

$$ratioX96 = \frac{debts \times 2^{96}}{colls} \tag{a}$$

(2). The `getTickAtRatio()` function performs the inverse operation, deriving the corresponding `tick` value from a given `ratioX96`. It is important to note that due to precision limitations, certain adjacent or closely-spaced `ratioX96` values may map to the same tick value.

Theoretically, the valid range for `tick` values in the `TickMath` library is [−32767, 32767]. Using the aforementioned functions and formulas, one can further derive the corresponding range of `ratioX96` values. Combined with the upper and lower bounds of the debt ratio defined in the protocol, this enables an estimation of the theoretical price range of collateral assets under ideal conditions.

In the f(x) protocol design, a minimum debt ratio is enforced for user positions. Based on the current parameter configuration, and assuming the absence of bad debt and funding costs (i.e., `rawDebts = debtsShares` and `rawColls = collsShares`), the minimum debt ratio must satisfy the following condition:

$$debtsRatio = \frac{debts \times 10^{18} \times 10^{18}}{colls \times collsPrice} \geq 0.091 \times 10^{18} \tag{1}$$

We can conclude that:

$$\frac{debts}{colls} \geq 0.091 \times collsPrice \times 10^{-18} \tag{b}$$

Assuming a position's `tick` reaches the minimum value defined in the `TickMath` contract (i.e., -32,767), the corresponding `ratioX96` would be `37,075,071`. (Note that an earlier version of the code returned `37,075,072`; the implications of this discrepancy will be discussed later). Based on this value, the theoretical lower bound for the debt-to-collateral ratio can be calculated. For simplicity, we again assume no bad debt in the system and no accrued funding costs, such that `rawDebts = debtsShares` and `rawColls = collsShares`:

$$\begin{gathered} RatioX96 = \frac{debts \times 2^{96}}{colls} = 37075071 \\ \\ \frac{debts}{colls} = 4.6795 \times 10^{-22} \end{gathered} \tag{c}$$

To meet the protocol's minimum collateral ratio requirement, the following inequality must hold:

$$\begin{gathered} \frac{debts}{colls} = 4.6795 \times 10^{-22} \leq 0.091 \times collsPrice \times 10^{-18} \\ \\ collsPrice \geq 5.1423 \times 10^{-3} \quad (\text{where the precision of collsPrice is } 10^{18}) \end{gathered} \tag{d}$$

This indicates that unless the price of the collateral asset falls below 5.1423×10^{-21} (essentially approaching 0), a user's position is unlikely to reach the minimum tick value of -32,767.

Similarly, the f(x) protocol imposes an upper bound on the debt ratio for user positions. Under the current parameter configuration, this constraint is defined as:

$$debtsRatio = \frac{debts \times 10^{18} \times 10^{18}}{colls \times collsPrice} \leq 0.865 \times 10^{18} \tag{2}$$

We can conclude that:

$$\frac{debts}{colls} \leq 0.865 \times 10^{-18} \times collsPrice \tag{e}$$

Assuming the tick reaches the maximum value of `32,767`, the corresponding `ratioX96` is `1693078772645279728478019290858414490958389225445 95`. From this, the debt-to-collateral ratio can be derived:

$$\begin{gathered} RatioX96 = \frac{debts \times 2^{96}}{colls} = 1693078772645279728478019290858414490958389225445 95 \\ \\ \frac{debts}{colls} = 2.13697 \times 10^{21} \end{gathered} \tag{f}$$

To satisfy the maximum collateral ratio constraint:

$$\frac{debts}{colls} = 2.13697 \times 10^{21} \geq 0.865 \times 10^{-18} \times collsPrice \tag{g}$$

$$collsPrice \leq 2.47 \times 10^{21} \times 10^{18} (\text{where the precision of collsPrice is } 10^{18})$$

This suggests that unless the price of the collateral asset exceeds 2.47×10^{21}—which is practically impossible under normal market conditions—a user's position is unlikely to exceed the maximum tick value of 32,767.

Given the current parameter settings and realistic price bounds, the tick range [−32767, 32767] defined in the `TickMath` contract is sufficiently wide to cover all required price intervals for opening user positions in the f(x) protocol.

Change Details and Security Review Results

This round of code modifications primarily focuses on the following three aspects:

(1). Refinement of boundary conditions in the `getTickAtRatio()` function. The boundary values for each `if` branch in the original version were adjusted. These values can be derived and verified using the following Python script:

```
from mpmath import mp, power,fmul,fdiv
mp.dps = 150  # The data accuracy was set to 150-bits

s1 = 16384
for i in range(14,-1,-1):
    print(f"--- the tick is 2^{i}")
    s2 = fmul(power(2,96),power(mp.mpf('1.0015'),s1))
    print("ratio x96 is:",s2)

    s3 = fdiv(fmul(s2,power(10,26)),power(2,96))
    print("the result is:",s3)

    s1 = s1 / 2
    print("--------------")
```

The corresponding output is as follows:

```
--- the tick is 2^14
ratio x96 is:
366525209813478329772199588853707735173 5.128471965239062987064016321689994700394942770686626149667769332182674067464165918688403114010266035028158861 96
the result is:
462619854079650871634840430834525598 5.061319646394894346557212247337553521029446055640079269039623054371570405312174247524538819148469421699688607657 83
--------------
--- the tick is 2^13
ratio x96 is:
170408681963910204790627764665098 65.905345717930764040238504293001938778243843772012146441663864853371976499792370715490426550620537290494347740115710 1
```

```
the result is:
215085995378511539117674904491620.30376486421538983776555051728425387670908622730714050251468837271579188535171818217149911476778697158010667555370502561
--------------
--- the tick is 2^12
ratio x96 is:
367439338510158215326118318511504213120066246086971683532983041594871701664198766950664618373069928727848057651357652640691943663866265130642879087283
the result is:
463773646705493108830028666480.977760764974262617364871694138592888132757426540714702810689409028457911020654758693333076085845595927455180215122305381l
--------------
--- the tick is 2^11
ratio x96 is:
170621052703400589920910445233503519262163358874783138568566468823436292071009073771349988411134639000987113562972079112192497809962259135289161441516
the result is:
215354044936586484546834476021506371084840960467433004203193225512532582780218429189136248383431615181980210344017446379735439884847276434605579663863071
--------------
--- the tick is 2^10
ratio x96 is:
367668226692760093024536487236.011626379946504039565787672605563945926327695766721651637472592778476585660882481925062666858673143611334118194345440381
the result is:
464062544207767844008185024.950588990554136265212906454481127887567666249440695471536249718544959732245130996295346791963771711845285893709191398299613
--------------
--- the tick is 2^9
ratio x96 is:
170674186729409605620119663668.182151611190328834819229292015986575671781216384852128541142160655325460419014089298336910815728616067077431240692961440
the result is:
215421109505955298802281577.031879604792139232258508172947569885727787941858721117750794086960165219397623484812189417015806012207670610027253688900330
--------------
--- the tick is 2^8
ratio x96 is:
116285004205991934861656513301.833432018437850265806453068327904879778042952993733740714912806629407727067671142322316189307646226913376435822983468216
the result is:
146772309890508740607270614.667650899656438875541505058062410944410634845437054956021802494916190787671921973286305280575274377236404667298852135212905
--------------
--- the tick is 2^7
ratio x96 is:
959846196596321417437470995901424102099314637014811782503846256133796546770753522027594668373323265383749991939009110274799153384211650585144812113006
the result is:
121149622323187099817270416.157248837742741760456796835775887150509830704599017771595808428731567546079666159303227322900645454896734096432277997443880
--------------
```

```
--- the tick is 2^6
ratio x96 is:
872048453084069580067178911 24.140860391256397280507046297160571673065782517807535178691299
663444634969813420529343756661284269265309422297403536062837 7
the result is:
110067989135437147685980801.568068573422377364214113968609839658296956701413661890105859 78
992008807330251625834132442513928146137728928391427466867724 7
--------------
--- the tick is 2^5
ratio x96 is:
831208737690223540299163744 75.945630787221705581412096057001439296598206116433746013517256
381330774749905822414046282756653055999999999999999999999999 9
the result is:
104913292358707887270979599.831816586773651266562785765558183105367479547827468106178829 61
692622876673610510770231485366821289062 5
--------------
--- the tick is 2^4
ratio x96 is:
811511804923363683271847161 76.963437239733905878684935810083804102514123472896
the result is: 10242718992470109119184092 7.762844039579442328381455567932128906 25
--------------
--- the tick is 2^3
ratio x96 is: 80183906840906820640659903620.983689199937486288060416
the result is: 10120631893548005690742131 2.890625
--------------
--- the tick is 2^2
ratio x96 is: 797046021395251527029597476 03.256181784576
the result is: 10060135135050625000000000 0.0
--------------
--- the tick is 2^1
ratio x96 is: 794660252651727877010841676 60.896256
the result is: 10030022500000000000000000 00.0
--------------
--- the tick is 2^0
ratio x96 is: 793470047580357340999342662 61.504
the result is: 10015000000000000000000000 00.0
--------------
```

In the previous code version, the boundary values for each `if` branch were strictly derived based on the results of the aforementioned Python script. In contrast, the latest implementation does not fully adopt the results computed by Python or WolframAlpha. Instead, the boundary values were redefined based on the following process:

The `getRatioAtTick()` function in the `TickMath` contract is used to compute the `ratioX96` values for specific tick inputs (e.g., $2^0, 2^1, 2^2, \ldots, 2^{14}$). These values are then normalized using the following formula in a Python script:

$$\text{getRatioAtTick}(\text{tick}) \times \frac{10^{26}}{79228162514264337593543950336} \tag{h}$$

The discrepancies in results are likely due to inherent limitations in numerical precision when performing calculations in Solidity compared to Python or WolframAlpha. While it remains uncertain whether using `getRatioAtTick()` as a reference provides a more rigorous approach from a theoretical standpoint, we have conducted tests around several critical values near the `if` branch thresholds and have not observed any anomalies or errors. Overall, the revised implementation demonstrates stable and reliable behavior.

(2). In the updated `_getTick()` function, the AladdinDAO team revised the `ratioX96` value corresponding to the minimum tick value (-32,767).

```
// new version
/// The minimum value that can be returned from getRatioAtTick. Equivalent to
getRatioAtTick(MIN_TICK). ~ Equivalent to `(1 << 96) * (1.0015**-32767)`
uint256 internal constant MIN_TICK_RATIO = 37075071;

// old version
/// The minimum value that can be returned from getRatioAtTick. Equivalent to
getRatioAtTick(MIN_TICK). ~ Equivalent to `(1 << 96) * (1.0015**-32767)`
uint256 internal constant MIN_RATIOX96 = 37075072;
```

According to the code comments, `MIN_TICK_RATIO` is calculated using the following formula:

$$\text{MIN\_RATIOX96} = 2^{96} \times 1.0015^{-32767} \tag{i}$$

Using a Python script for derivation, the resulting value is approximately 37,075,071.974231223.

```
from mpmath import mp, power,fmul,fdiv
mp.dps = 150  # The data accuracy was set to 150-bits
# min ratio (tick = -32767)
s = fmul(power(2,96),power(mp.mpf('1.0015'),-32767))
print("min ratio is(tick=-32767): ", s)
```

```
# The output is:
min ratio is(tick=-32767):
 37075071.974231223390586384854541966953501229994260118786731052264663347841212662569904591371463084767476608400842467415700869023102123530083398187484 7
```

In contrast to the previous approach, the new implementation adopts a floor operation, truncating the result to the integer value `37,075,071`. This value is then defined as the new `MIN_TICK_RATIO`. Under the current logic of the `_getTick()` function, when the input `ratio` equals `37,075,071`, the computed tick value is exactly -32,767, thereby satisfying the `TickMath` contract's boundary condition requirements.

It is important to note that `MIN_TICK_RATIO` and `MAX_TICK_RATIO` merely represent the theoretical bounds of the tick values supported by the `TickMath` contract. As previously analyzed in the context of extreme tick values, these boundary conditions are unlikely to be encountered in practice under normal market conditions when calculating user positions.

```
function _getTick(uint256 colls, uint256 debts) internal pure returns (int256 tick) {
    // @audit assume that the ratio is 37075071
    uint256 ratio = (debts * TickMath.ZERO_TICK_SCALED_RATIO) / colls;
```

```
    // @audit corresponding tick is -32768
    (tick, ) = TickMath.getTickAtRatio(ratio);
    // @audit the tick will reset to -32767
    if (tick < TickMath.MIN_TICK) {
      tick = TickMath.MIN_TICK;
    }

    // @audit the corresponding ratio of 37075071 was calculated for tick= -32767
    uint256 ratioAtTick = TickMath.getRatioAtTick(tick);
    unchecked {
      // @audit the if condition is not met, and the final tick value is -32767
      if (ratioAtTick < ratio) tick++;
    }
    if (tick > TickMath.MAX_TICK) {
      tick = TickMath.MAX_TICK;
    }
  }
```

(3). Corresponding adjustments have also been made to the implementation logic of the `_getTick()` function.

Since the new version of the code will be deployed as an upgrade to the existing contract, ensuring compatibility is critical. In particular, special attention must be paid to whether the same `ratioX96` value maps to different tick values under the old and new implementations.

Using the test input `ratioX96 = 6376778932548287112687935746710 80`, we compared the return values of the `_getTick()` function under both the old and new logic:

In the previous code version, the function returns a tick value of 6,000 for this input ratio.

```
// @audit old version
function _getTick(uint256 colls, uint256 debts) internal pure returns (int256 tick) {
   // @audit ratio = 6376778932548287112687935746710 80
   uint256 ratio = (debts * TickMath.ZERO_TICK_SCALED_RATIO) / colls;
   uint256 ratioAtTick;
   // @audit tick = 5999
   // ratioAtTick = 6367228090412668110522152581510 81
   (tick, ratioAtTick) = TickMath.getTickAtRatio(ratio);
   // @audit if condition is met
   if (ratio != ratioAtTick) {
     // @audit tick = 6000
     tick++;
     ratio = (ratioAtTick * 10015) / 10000;
   }
 }
```

Under the updated logic, when `ratioX96` is set to `6376778932548287112687935746710 80`, the function returns a tick value of 6,001.

```
function _getTick(uint256 colls, uint256 debts) internal pure returns (int256 tick) {
   // @audit ratio = 6376778932548287112687935746710 80
```

```
    uint256 ratio = (debts * TickMath.ZERO_TICK_SCALED_RATIO) / colls;
    // @audit tick = 6000
    (tick, ) = TickMath.getTickAtRatio(ratio);
    // @audit skipped
    if (tick < TickMath.MIN_TICK) {
      tick = TickMath.MIN_TICK;
    }

    // @audit tick = 6000, ratioAtTick = 637677893254828711268793574670086
    uint256 ratioAtTick = TickMath.getRatioAtTick(tick);
    unchecked {
      // @audit if condition is met, tick is updated to 6001
      if (ratioAtTick < ratio) tick++;
    }
    if (tick > TickMath.MAX_TICK) {
      tick = TickMath.MAX_TICK;
    }
  }
```

This observation reveals that the tick value calculated by the updated implementation may be greater, under certain conditions, than that returned by the previous version. It is worth noting that during `rebalance()` or `liquidate()` operations, the contract first shifts the affected position to the new tick and synchronizes the `topTick` value accordingly. As such, even if the liquidation process temporarily results in a larger `topTick` value, it does not adversely affect the protocol's normal logic. Users can continue to liquidate positions starting from the updated tick value, confirming that the code modification behaves as expected.

Status

A detailed analysis of the modified code indicates that the recent updates to the `TickMath` contract and its associated logic do not affect the normal functionality of the f(x) protocol and do not introduce new security risks.