Security Review Report
NM-0147 PRAGMA

NETHERMIND SECURITY

(Nov 16, 2023)

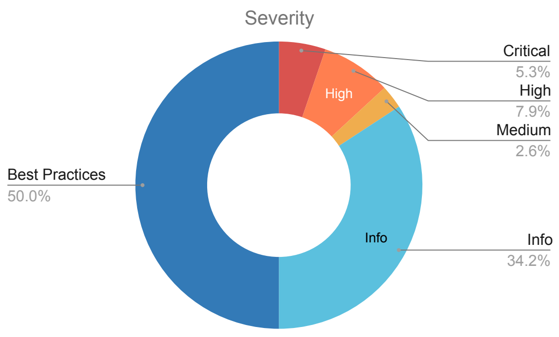
Contents

1 Executive Summary

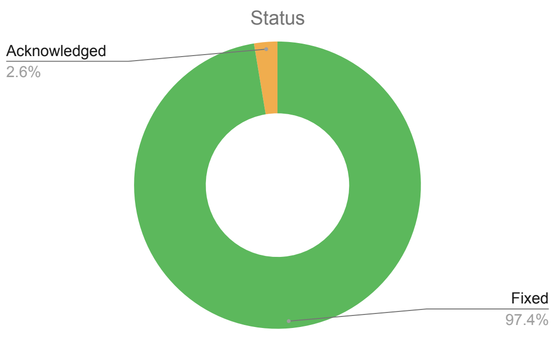
This document presents the security review performed by Nethermind on the Pragma Oracle protocol. Pragma is an oracle protocol designed to provide asset price feeds to other various protocols on the Starknet network. It is designed to reduce off-chain infrastructure dependencies by using trusted "publishers" to submit their pricing data directly to Oracle, where multiple submissions from different publishers can be aggregated, and the final results can be calculated on-chain. The data aggregation can be done as a mean or median; however, the Pragma team highly recommends protocols use the calculated median to prevent outlier price data from affecting the result. Publishers must complete an application process before being verified and whitelisted by the Pragma team. Each publisher is then manually given permission to submit to each price feed individually. The protocol also features more advanced computational feeds using historical price data. At the time of this report, the supported computational feeds are volatility and yield curves, with more to be added in the future.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contract, and (d) creation of test cases. Along with this document, we report 38 points of attention, where 2 are classified as `Critical`, 3 are classified as `High`, 1 is classified as `Medium`, 13 are classified as `Info` and 19 are classified as `Best Practices`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology adopted for this audit. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, and tests output. Section 9 concludes the document.



(a) distribution of issues according to the severity



(b) distribution of issues according to the status

Fig 1: (a) Distribution of issues: Critical (2), **High** (3), **Medium** (1), **Low** (0), **Undetermined** (0), **Informational** (13), **Best Practices** (19). **(b) Distribution of status: Fixed** (37), **Acknowledged** (1), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Nov 9, 2023
Response from Client	Nov. 14, 2023
Final Report	Nov. 16, 2023
Methods	Manual Review, Automated Analysis, Tests
Repository	astraly-labs/pragma-oracle
Commit Hash (Network-Contract	43ae793167c1d8f65ddd84ad1f89a00cb396a093
Documentation	Website documentation
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/lib.cairo	46	1	2.2%	1	48
2	src/account/account.cairo	75	8	10.7%	12	95
3	src/operations/time_series/metrics.cairo	275	32	11.6%	37	344
4	src/operations/time_series/scaler.cairo	109	11	10.1%	20	140
5	src/operations/time_series/convert.cairo	36	4	11.1%	14	54
6	src/operations/time_series/structs.cairo	19	1	5.3%	2	22
7	src/operations/sorting/merge_sort.cairo	120	25	20.8%	11	156
8	src/randomness/example_randomness.cairo	77	14	18.2%	14	105
9	src/randomness/randomness.cairo	333	12	3.6%	33	378
10	src/entry/entry.cairo	461	49	10.6%	42	552
11	src/entry/structs.cairo	121	31	25.6%	25	177
12	src/oracle/oracle.cairo	1720	217	12.6%	169	2106
13	src/upgradeable/upgradeable.cairo	25	6	24.0%	5	36
14	src/publisher_registry/publisher_registry.cairo	276	77	27.9%	68	421
15	src/compute_engines/summary_stats/summary_stats.cairo	210	44	21.0%	32	286
16	src/compute_engines/yield_curve/yield_curve.cairo	598	151	25.3%	62	811
17	src/admin/interface.cairo	6	0	0.0%	2	8
18	src/admin/admin.cairo	24	9	37.5%	11	44
	Total	**4531**	**692**	**15.3%**	**560**	**5783**

3 Summary of Issues

	Finding	Severity	Update
1	A publisher can reliably manipulate price feeds to any value	Critical	Fixed
2	Data aggregation includes stale entries	Critical	Fixed
3	Incorrect calculation of pair prices with USD hop	High	Fixed
4	Incorrect price comparison results in inaccurate yield points	High	Acknowledged
5	`StorePacking` implementation for `EntryStorage` struct is not correctly packed	High	Fixed
6	Lack of timestamp validation allows potential DOS in `publish_data(...)`	Medium	Fixed
7	A currency and pair with `id` of zero can be added	Info	Fixed
8	A new pair can be added with non-existing base and quote currencies	Info	Fixed
9	Cannot remove publisher with zero address	Info	Fixed
10	Empty arrays lead to revert in `_sum_volatility(...)`	Info	Fixed
11	Empty arrays lead to revert in `merge(...)` function	Info	Fixed
12	Incorrect sign determination in `pairwise_1D(...)` multiplication operation	Info	Fixed
13	Missing check in `calculate_volatility(...)`	Info	Fixed
14	Missing request cancellation logic in `cancel_random_request (...)`	Info	Fixed
15	Multiple publishers can share the same address	Info	Fixed
16	Necessary checks are inadequate	Info	Fixed
17	Request status handling issues in `randomness.cairo` contract	Info	Fixed
18	Undefined divisions may occur in several functions	Info	Fixed
19	`get_implementation_hash(...)` returns empty classhash until first upgrade	Info	Fixed
20	Control Coupling in function `pairwise_1D(...)`	Best Practices	Fixed
21	Dead code	Best Practices	Fixed
22	Duplicate code in `mul_decimals(...)` function	Best Practices	Fixed
23	Hash implementation `TupleSize4LegacyHash` has an incorrect name	Best Practices	Fixed
24	Inconsistent type casting approach for `u8` and `AggregationMode`	Best Practices	Fixed
25	Inconsistent use of `getter` function	Best Practices	Fixed
26	Incorrect comments	Best Practices	Fixed
27	Shadowed arrays are unused	Best Practices	Fixed
28	Unnecessary iteration counter	Best Practices	Fixed
29	Unnecessary local variable assignment	Best Practices	Fixed
30	Unnecessary storage read	Best Practices	Fixed
31	Unnecessary use of `into` in some functions	Best Practices	Fixed
32	Unnecessary variable redefinition inside a loop	Best Practices	Fixed
33	Unnecessary variable shadowing	Best Practices	Fixed
34	Unrequired overhead for computing the variance	Best Practices	Fixed
35	Unused function parameters	Best Practices	Fixed
36	Unused imports	Best Practices	Fixed
37	Unused parameters in function get_future_spot_pragma_source_key(...)	Best Practices	Fixed
38	Use the same value while converting types	Best Practices	Fixed

4 Protocol Overview

The Pragma protocol consists of the following components:

Oracle: Contains the primary logic for the protocol, including storing published data, aggregating data on-chain using median or mean, setting and querying checkpoints, getters for information related to pair and sources, and admin management.

PublisherRegistry: Tracks all publishers and the sources they have permission to publish data to. Once approved as a publisher, an admin specifies each source that the publisher can provide data for. Publishers are able to change their address which they use to submit data. Admins are also able to disable sources or remove publishers entirely.

Compute Engine: The compute engine represents a group of contracts that can return more complex data based on historical checkpoints from the oracle contract. At the time of this report, the supported compute engine contracts are the yield curve and summary stats. In the future, more compute engine contracts will be supported.

Publishers: Publishers are selected by the Pragma team and are permitted to submit price data to the oracle contract. There are multiple publishers per source, to ensure that price data updates regularly.

Admin: Controlled by the Pragma team, with the ability to manage publishers, change the registry contract, manage new pairs, currencies and sources and upgrade the oracle contract implementation.

Dependent Protocol: A protocol that depends on the oracle price feed data, such as lending/borrowing protocols or perpetual trading protocols. These contracts will use the price aggregation functions to get the median price for a given pair most frequently.

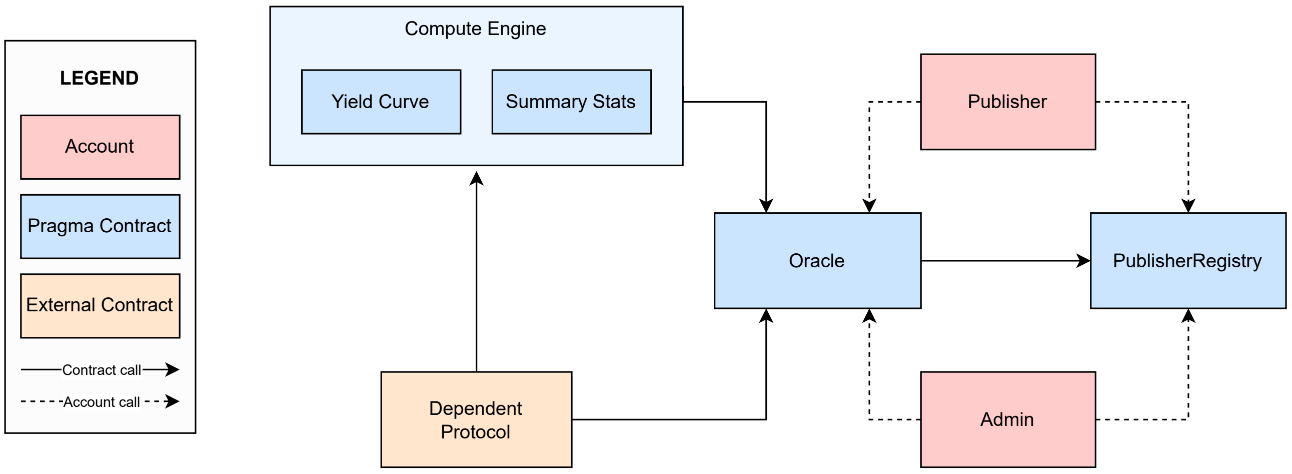


Fig. 2: Pragma - Structural Diagram of Contracts

4.1 Analysis of the source-publisher oracle structure

Pragma sources its pricing data through “publishers” and “sources”. A publisher is a trusted entity with permission to publish price data to a source. There can be multiple publishers per source. A source represents pricing data from one entity, and there can be many sources for a given pair. When protocols request pricing data on-chain, the mean or median of all source pricing is calculated and returned to the caller.

Each publisher must be manually approved for each source by the admin of the publisher registry contract, meaning that the publisher role is effectively whitelisted, but this does not guarantee that publishers will always behave correctly. Publishers will likely publish data through an automated program or script. It is possible that the private key for a given publisher could be compromised, or the script could be tampered with. It also may be possible that the script may have a bug or behave incorrectly for some unintentional reason.

The current data aggregation structure relies on the trusted publishers as a group to behave correctly, so we only wish to explore the impacts of a single compromised publisher.

The purpose of having multiple sources and multiple publishers is that if one publisher acts incorrectly, the remaining publishers will still be correct. Pragma highly recommends its clients use the median for their aggregation approach, so we will assume that all pricing calculations use the median for the rest of this analysis. With only one publisher providing incorrect data, it won’t be considered, and the median will still return an accurate price. This structure prevents one publisher from manipulating the price, but there are some constraints and nuances that should be considered, which we explore below:

The safety in numbers comes from how many sources you have, not how many publishers. Each source only tracks its most recent entry, so if you have a source with many honest publishers and one malicious publisher, we can consider the entire source compromised since the malicious publisher can repeatedly publish new incorrect data to overwrite the honest data. While the number of publishers doesn’t contribute to safer pricing calculations, the number of sources do. The more sources you have, the more prices you calculate a median from, making it harder for an incorrect price to impact the result. This distinction between publishers and sources is important:

- More publishers increase the rate at which a source is updated.
- More sources increase the safety and reliability of price data.

A minimum number of sources per pair is required. Having just one source is naturally a risk since pricing is effectively centralized to the most recent publisher. A risk still exists with two sources because the median calculation will attempt to average out the two source prices. If one of these sources is compromised (has a malicious publisher), the outcome of this average can be manipulated. With three sources, if one source is compromised, the median result will be chosen from a correct source. Therefore, at least three sources are needed to protect against a malicious publisher. This minimum threshold check already exists in the code. However, when setting this minimum using `set_sources_theshold`, there is no check to ensure the new threshold is three or more. Without this check, it’s possible to put the protocol in a state where price manipulation is possible. The threshold check also only applies to checkpoints, not live data, so price manipulation can still be done and may impact other protocols that depend on Pragma for pricing.

A publisher should not be able to publish to more than one source for a given pair. Returning to a point discussed earlier, if one publisher is compromised, we can consider its source compromised as well. If a publisher has permission to more than one source for the same pair, each of these sources are compromised, which breaks the idea that a minimum of three sources is needed to be safe from a malicious publisher. For example, with three sources, a malicious publisher could publish a zero price for both sources, and the median would choose the sorted middle price, which would be zero, meaning a publisher could have complete control of price data. For this reason, a publisher should only have permission to one source for each pair.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely an attacker will uncover and exploit the finding. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Critical] A publisher can reliably manipulate price feeds to any value

File(s): `src/oracle/oracle.cairo`

Description: Publishers use the `publish_data(...)` function to submit new price data to the `Oracle` contract. This function has two logic flows, depending on whether the given source has previously provided data.

If the source has not previously provided data, it is considered "new" for the given pair, and the source ID is added to the pair's list of valid sources. After this, the entry data is written to storage. If the source has already provided data, then the new data is validated and written to storage. A snippet demonstrating these logic flows is shown below:

```
let res = get_entry_storage(@self, spot_entry.pair_id, SPOT, spot_entry.base.source, 0);

// When a source has an existing entry
if (res.timestamp != 0) {
    let entry: PossibleEntries = IOracleABI::get_data_entry(
        @self, DataType::SpotEntry(spot_entry.pair_id), spot_entry.base.source
    );
    match entry {
        PossibleEntries::Spot(spot) => {
            validate_data_timestamp(ref self, new_entry, spot);
        },
        PossibleEntries::Future(_) => {},
        PossibleEntries::Generic(_) => {},
    }
// When a source doesn't have an entry
} else {
    let sources_len = self
        .oracle_sources_len_storage
        .read((spot_entry.pair_id, SPOT, 0));
    self
        .oracle_sources_storage
        .write(
            (spot_entry.pair_id, SPOT, sources_len, 0),
            spot_entry.get_base_entry().source
        );
    self
        .oracle_sources_len_storage
        .write((spot_entry.pair_id, SPOT, 0), sources_len + 1);
}
```

The `Oracle` contract determines if a source is "new" by checking if the stored entry's timestamp is zero. However, the only validation for a newly submitted timestamp is that it must be greater than or equal to the existing stored timestamp. A malicious publisher could provide a timestamp of zero to a new source for a given pair and repeatedly call `publish_data(...)`, each time creating an additional entry into the list of sources for the given pair, all duplicates of the same source.

```
fn build_sources_array(
    self: @ContractState, data_type: DataType, ref sources: Array<felt252>, sources_len: u64
) {
    let mut idx: u64 = 0;
    loop {
        if (idx == sources_len) {
            break ();
        }
        match data_type {
            DataType::SpotEntry(pair_id) => {
                let new_source = self
                    .oracle_sources_storage
                    .read((pair_id, SPOT, idx.into(), 0));
                ////////////////////////////////////////////////////////////////
                // @audit The sources array can contain duplicate source entries
                ////////////////////////////////////////////////////////////////
                sources.append(new_source);
            },
            // ...
        }
        // ...
    }
    // ...
}
```

All data aggregation methods use the list of sources. A mutated source list containing many entries for the same source ID can lead to the same price data being considered multiple times, heavily affecting the mean and median calculations.

This exploit can be combined with the finding related to entry storage packing to manipulate the sources list even if an existing entry exists. An attacker could provide a timestamp greater than 32 bits, which will pass input validations, but when written to storage, all bits over 32 are cut off. If the remaining 32 bits are zero, resetting the stored entry timestamp to zero is possible, allowing the exploit scenario discussed above.

This exploit allows any publisher to have their source data duplicated an arbitrary number of times, affecting the mean and median calculations such that their single price source can outweigh all other pricing, effectively taking complete control of Oracle pricing.

A malicious publisher can also leave the timestamp as zero and never set it to a non-zero value, causing the function `get_data_entry(...)` to revert due to the following line, which will cause any on-chain price query to revert:

```
assert(!_entry.timestamp.is_zero(), 'No data entry found');
```

Recommendation(s): Address the packing issue highlighted in the other finding and add a zero check to `validate_data_timestamp(...)` for the new entry timestamp.

Status: Fixed

Update from the client: The issue was resolved with the changes made in commit e46838774da325db1541d6dcc6d6b954fa1e9777. In this update, we introduced a check for zero timestamps and also reevaluated the overall structure of our data storage. The initial design is thoroughly explained in the Analysis section of our audit documentation. However, we ultimately chose not to adopt the approach where one source equates to one publisher. Instead, we opted for a strategy that categorizes data by source, providing us with more flexibility.

Rather than simply updating the latest price from a source, we have restructured our storage system to also consider the publisher's information. This means that when our aggregation functions are called, they will first aggregate data from different publishers within each source (noting that a single publisher may provide data for multiple sources), followed by an aggregation across various sources. This new method makes her less prone to manipulation.

6.2 [Critical] Data aggregation includes stale entries

File(s): src/oracle/oracle.cairo

Description: When getting price data, the internal function build_entries_array(...) is used to collect an array of PossibleEntries, which can then be aggregated to return a price. For an entry to be appended to the array, it should be initialized and within 2h10m from the most recent entry timestamp or the current block timestamp, whichever is lower. These requirements are designed to prevent stale data from being a part of the aggregate calculations. A snippet from the function is shown below:

```
PossibleEntries::Spot(spot_entry) => {
    let is_entry_not_initialized: bool = spot_entry.get_base_timestamp() == 0;
    let condition: bool = is_entry_not_initialized
        && (spot_entry
            .get_base_timestamp() < (latest_timestamp - BACKWARD_TIMESTAMP_BUFFER));
    if !condition {
        entries.append(PossibleEntries::Spot(spot_entry));
    }
},
```

The condition variable is determined by the AND operation, considering (both) whether the entry is initialized and the timestamp is stale. This logic uses many negatives both language-wise and with the NOT operator, which makes it hard to follow when reading the code, so the logic paths have been simplified below:

```
Condition:  entry is uninitialized AND entry is stale

Case 1:     entry is uninitialized AND entry is stale
            True AND True == True
            !True == False
            False so we skip

Case 2:     entry is initialized AND entry is recent
            False AND False == False
            !False == True
            True so we append

Case 3:     entry is initialized AND entry is stale
            False AND True == False
            !False == True
            True so we append
```

As shown in the logic paths above, an entry will be appended even if the entry timestamp for the given source is stale. A source may stop providing data for publishers to use for a number of reasons, such as de-listing assets, changing to a paid API that publishers don't want to use, or even the source shutting down.

The impact of stale source entries being part of the aggregate calculations differs depending on whether a Median or Mean aggregation method is used:

- Mean: The impact can be measured immediately, where the stale price is part of the calculated mean;
- Median: The impact cannot be measured until a minimum of 50% of the entries for a given source are stale. Another condition is that every stale price is lower than the current price or every stale price is higher than the current price. When these two conditions are met, the median price will begin to return arbitrary stale prices from retired sources;

Recommendation(s): Consider changing the condition from LOGICAL AND to LOGICAL OR, which will ensure that an entry can only be appended when it is both initialized and it's timestamp is recent. We also recommend changing how this logic is written to make it more readable, removing "negatives" where possible, and giving a more meaningful name to the condition variable.

Status: Fixed

Update from the client: The issue is resolved in commit 1a7b0ce207f90065797a45f79422d5f7cff78e73. We made the logic much clearer and, thus, more understandable.

6.3 [High] Incorrect calculation of pair prices with USD hop

File(s): src/operations/time_series/convert.cairo

Description: Function convert_via_usd(...) is used to calculate the base tokens price quoted by quote tokens price with USD prices of each token. However, the calculation inside convert_via_usd(...) function is incorrect. In cases like base tokens, decimals are higher than quote token decimals. The calculation will be like this:

```
BASE_TOKEN_PRICE_IN_USD * (OUTPUT_DECIMALS) / QUOTE_TOKENS_PRICE_IN_USD
```

The above calculation results in higher values than expected.

Recommendation(s): Consider converting each token price into the same decimals, then continue the calculation.

Status: Fixed

Update from the client: The issue is resolved in commit 2db748ed922f6a0b6d80dc83da54df456fb52646, following the recommendations.

Update from Nethermind: The fix resolves the issue of using prices in different decimal representations. However, we noticed that the variable decimals was changed from min(base_decimals, quote_decimals) to be the base_decimals. This makes the first call to normalize_to_decimals(..) unnecessary as it converts from base_decimals to base_decimals.

Additionally, if base_decimals is less than quote_decimals, we will have a precision loss while normalizing the quote price. We suggest a better approach to avoid this issue:

- Convert one of the two prices to the max decimals between base and quote;
- Compute rebased_value using the desired output decimals. Below we exemplify our suggestion;

```
if ( base_decimals < quote_decimals) {
   // convert base price to quote decimals
} else {
  // convert quote price to base decimals
}
// compute rebased value, `decimals` here is the expected output decimals
let rebased_value = convert_via_usd( normalised_basePPR_price, normalised_quotePPR_price, decimals);
```

Second update from the client: We implemented the following logic in our code:

```
let (rebased_value, decimals) = if (base_decimals < quote_decimals) {
let normalised_basePPR_price = normalize_to_decimals(
    basePPR.price, IOracleABI::get_decimals(self, base_data_type), quote_decimals
);
(
    convert_via_usd(normalised_basePPR_price, quotePPR.price, quote_decimals),
        quote_decimals)
} else {
    let normalised_quotePPR_price = normalize_to_decimals(
    quotePPR.price, IOracleABI::get_decimals(self, quote_data_type), base_decimals);
(convert_via_usd(basePPR.price, normalised_quotePPR_price, base_decimals),base_decimals)
            };
```

In the original implementation, we operated under the assumption that we would always deal with the same number of decimals, since the conversion was being realized through the USD currency. This assumption was based on the fact that the get_decimals function would always return the decimal count of USD. However, we recognized the need for greater flexibility to handle different decimal counts between base and quote currencies. Therefore, we introduced changes to normalize the price of the base and quote currencies to the higher decimal precision of the two. This approach ensures accurate conversions regardless of the decimal discrepancies between different currencies. You can check the commit here: 008c977b25b502ba5ec427158a98109d8935ed6a

6.4 [High] Incorrect price comparison results in inaccurate yield points

File(s): `yield_curve.cairo`

Description: The function `calculate_future_spot_yield_point(...)` calculates the yield point for a given future and spot entry. It takes as an input the future and spot entries: `future_entry`, `spot_entry`, along with their respective price decimals: `future_decimals` and `output_decimals`.

However, an issue arises as the function directly compares `future_entry.price` and `spot_entry.price` without converting them to a unified decimal representation. This can result in inaccurate outputs, possibly leading to erroneously returning a rate of 0 in many cases, particularly when `future_decimals` are less than `spot_decimals`.

Additionally, there's a potential risk of underflow in the `interest_ratio` computation due to the incorrect comparison of prices. The calculation assumes `shifted_ratio >= decimals_multiplier`. This is ensured when the future price is higher than the spot price when both are in the same decimal representation. However, the current comparison doesn't guarantee this, resulting in unexpected reverts in multiple cases.

This issue directly impacts the construction of yield curve points within the `get_yield_points(...)` function, leading to unexpected reverts or incorrect data returns.

```
fn calculate_future_spot_yield_point(...) -> YieldPoint {
        let mut time_scaled_value = 0;
        // @audit `future_entry.price` and `spot_entry.price` are in different decimals
        if (future_entry.price > spot_entry.price) {
            // ...
            let decimals_multiplier = pow(10, output_decimals.into());
           // ...
            if (future_decimals <= output_decimals
                + spot_decimals) {
                let exponent = output_decimals + spot_decimals - future_decimals;
                // ...
                let ratio_multiplier = pow(10, exponent.into());
                shifted_ratio = (future_entry.price * ratio_multiplier) / spot_entry.price;
            } else {
                let exponent = future_decimals - output_decimals - spot_decimals;
               // ...
                let ratio_multiplier = pow(10, exponent.into());
                shifted_ratio = (future_entry.price) / (spot_entry.price * ratio_multiplier);
            }
            // @audit the current check could lead to an underflow in `interest_ratio` computation
            let interest_ratio = shifted_ratio - decimals_multiplier;
           // ...
        }
        // ...
}
```

Recommendation(s): Ensure proper conversion of `future_entry.price` and `spot_entry.price` to a unified decimal representation before comparison. Implementing this fix will mitigate the risk of underflow in the `interest_ratio` as `shifted_ratio >= decimals_multiplier` will always be true.

Status: Acknowledged

Update from the client: Our system architecture is designed in such a way that it does not support different decimal places for a given currency pair. To elaborate, let's examine the get_decimals function in our code:

```
fn get_decimals(self: @ContractState, data_type: DataType) -> u32 {
            let base_currency = match data_type {
                DataType::SpotEntry(pair_id) => {
                    let pair = self.oracle_pairs_storage.read(pair_id);
                    assert(!pair.id.is_zero(), 'No pair found');
                    let base_cur = self.oracle_currencies_storage.read(pair.base_currency_id);
                    base_cur
                },
                DataType::FutureEntry((
                    pair_id, expiration_timestamp
                )) => {
                    let pair = self.oracle_pairs_storage.read(pair_id);
                    assert(!pair.id.is_zero(), 'No pair found');
                    let base_cur = self.oracle_currencies_storage.read(pair.base_currency_id);
                    base_cur
                },
                DataType::GenericEntry(key) => {
                    let pair = self.oracle_pairs_storage.read(key);
                    assert(!pair.id.is_zero(), 'No pair found');
                    let base_cur = self.oracle_currencies_storage.read(pair.base_currency_id);
                    base_cur
                }
            // DataType::OptionEntry((pair_id, expiration_timestamp)) => {}
            };
            base_currency.decimals
        }
```

In this implementation, the function retrieves the decimal count solely from the base currency of the pair. This approach is consistent across different types of data entries, whether they are SPOT, FUTURE, or OPTION. As a result, the final decimal value remains the same for any given currency pair, regardless of the type of financial instrument involved. This uniformity in handling decimals ensures consistency but also means that our system does not accommodate variations in decimal places between different pairs or types of entries.

6.5 [High] `StorePacking` implementation for `EntryStorage` struct is not correctly packed

File(s): `src/oracle/oracle.cairo`

Description: Entries have a custom `StorePacking` implementation to use storage efficiently by fitting all values of an entry within a single storage slot of size `felt252`. The custom pack function is shown below:

```
// EntryStorage.timestamp: u64
// EntryStorage.volume:    u128
// EntryStorage.price:     u128
//                          320 bits total

fn pack(value: EntryStorage) -> felt252 {
    value.timestamp.into()
        + value.volume.into() * TIMESTAMP_SHIFT_U32
        + value.price.into() * VOLUME_SHIFT_U132
}
```

Since, as integers, it is not possible to fit 320 bits into a 252 bits storage slot, the Pragma team treats the `timestamp` as 32 bits in storage, `volume` as 100 bits, and `price` as 120, totaling 252 bits. The pack function converts the integer to `felt252`, then multiplies by the amount to complete the equivalent of a bit shift action. These shifted values are then summed together to result in the final packed representation of the entry data.

There is no check to ensure that the integer data size will exceed the expected packed storage size. For example, `timestamp` is represented as 32 bits in storage, but when packing the `u64` value, there is no check to ensure that the size of `timestamp` is greater than 32 bits. In the case of `timestamp`, this can lead to the 32 most significant bits "bleeding" into the bits above that are meant to represent volume.

The following demonstrates each part of the packed data and which bits may bleed into other sections or overflow:

```
'B' = Expected storage bits
'.' = Bleed or overflow bits
'_' = Unaffected bits

timestamp: 0x________________________________________________........BBBBBBBB 32  bits, actual 64bits
volume:    0x_______________________.......BBBBBBBBBBBBBBBBBBBBBBBBBB________ 100 bits, actual 128bits
price:     0xBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB.................................. 120 bits, actual 128bits

// Price has the potential to overflow, changing all bits in the packed storage felt
```

As shown in the diagram above, a specially crafted `timestamp` can be used to manipulate bits representing `volume`, and a specially crafted `volume` can be used to manipulate bits representing `price`, and a specially crafted `price` can be used to overflow the felt, manipulating the lower `timestamp` bits or even causing the entire felt to be zero.

Recommendation(s): Consider adding a check when packing an `EntryStorage` to ensure each field fits within its expected storage size.

Status: Fixed

Update from the client: The issue was addressed in two commits, identified by their hash codes. In these updates, we implemented a masking technique for the data elements we intended to store. For instance, by applying a bitwise AND operation between the timestamp and a predefined mask, we can determine if the timestamp exceeds the mask's limit. If the result of this operation differs from the original timestamp, it indicates an overflow beyond the mask's capacity. This method allows us to effectively manage and limit the size of the elements we are working with.

- 9e721f45cea2bac7381ef191486d0b008de35bbc;
- 34805720f7b3aeac834f347bdcb64335ea02f60c;

Update from Nethermind: The fixes implemented are not enough. There are 2 issues with it:

Issue 1) Possible overflow in function `pack(...)` when price value is greater than or equal to `0x800000000000011000000000000000`: The function `pack(...)` packs three variables with 120 bits, 100 bits, and 32 bits respectively - a total of 252 bits, into a single `felt252` variable.

```
fn pack(value: EntryStorage) -> felt252 {
    value.timestamp.into()
        + value.volume.into() * TIMESTAMP_SHIFT_U32
        + value.price.into() * VOLUME_SHIFT_U132
}
```

However, a `felt252` variable cannot actually hold all values with 252 bits. The maximum value of a `felt252` is `P = ` $2^{251} + 17 * 2^{192} + 1$, which means it cannot store the value in the range $[P + 1; 2^{252})$.

```
MAX_FELT = 3618502788666131213697322783095070105623107215331596699973092056135872020480;
// If we convert it to Hex, it would be
MAX_FELT = 0x800000000000011000000000000000000000000000000000000000000000000;
```

As seen, `value.price` will occupy the first 120 bits during packing. So, if `value.price` is greater than or equal to `0x800000000000011000000000000000`, the resulting `pack_value` will overflow.

Issue 2) Unnecessary check for `pack_value` in the function `pack(...)`: The function `pack(...)` packs three variables with 120 bits, 100 bits, and 32 bits respectively - a total of 252 bits, into a single `felt252` variable.

```
fn pack(value: EntryStorage) -> felt252 {
    ...
    let pack_value: felt252 = value.timestamp.into()
        + value.volume.into() * TIMESTAMP_SHIFT_U32
        + value.price.into() * VOLUME_SHIFT_U132;
    assert(pack_value.into() < MAX_FELT, 'EntryStorePacking:tot too big');
    pack_value
}
```

After the calculation, `pack_value` is verified to be smaller than `MAX_FELT`. However, since `pack_value` is already a `felt252`, this check is always true, making it unnecessary.

Second update from the client The issue stemmed from our initial intention to define pack_value as a u256 rather than a felt252. By opting to return a u256, the subsequent assertion becomes logical, as it ensures there is no overflow before we convert pack_value into its final felt252 form. Here is the final commit: 4355f00d3c3728c254eb12150ca19e686261a529

```
fn pack(value: EntryStorage) -> felt252 {
    ...
    let pack_value: u256 = value.timestamp.into()
        + value.volume.into() * TIMESTAMP_SHIFT_U32
        + value.price.into() * VOLUME_SHIFT_U132;
    assert(pack_value.into() < MAX_FELT, 'EntryStorePacking:tot too big');
    pack_value
}
```

6.6 [Medium] Lack of timestamp validation allows potential DOS in `publish_data(...)`

File(s): `src/oracle/oracle.cairo`

Description: When submitting a new timestamp, the function `validate_data_timestamp(...)` is used to ensure that the new entry timestamp cannot be less than the existing entry timestamp. However, there are no restrictions on how far into the future the new entry timestamp may be.

With no restriction on the future timestamp, if a publisher publishes an entry with a timestamp far in the future (either maliciously or unintentionally, e.g., a bug on the publisher's automated script), assuming all other publishers continue to behave correctly and attempt to submit valid timestamps on their entries, they will revert, as the current entry with a timestamp far in the future prevents and new entries with a timestamp at the current time. This check is shown below:

```
assert(
    spot_entry.get_base_timestamp() >= last_entry.get_base_timestamp(),
    'Existing entry is more recent'
);
```

Recommendation(s): Consider adding a check in `validate_data_timestamp(...)` to ensure that the provided timestamp for the new entry is reasonable.

Status: Fixed

Update from the client: The issue is resolved in commits:

- 22d29699a94910a93592ef6da30eb84e7e89c0ba,;
- 780b2bd95a4853736977bad7d0fd398dc778fdb5;

We defined a FORWARD_TIMESTAMP_BUFFER of 2 minutes to guarantee future validation within block emission range.

6.7 [Info] A currency and pair with `id` of zero can be added

File(s): `src/oracle/oracle.cairo`

Description: When adding a currency or pair to the oracle, there is no check to ensure that its ID is not zero. Various parts of the protocol assume that an ID of zero means the given pair or currency does not exist. When an ID of zero is used, it can lead to unexpected behavior, such as early reverts and overwrites, which should normally not be possible. The following functions have unexpected behaviors for zero ID currencies:

```
Oracle::add_currency
    The same currency ID can be overwritten multiple times

Oracle::update_currency
    The currency cannot be updated
```

The following functions have unexpected behaviors for zero ID pairs:

```
Oracle::get_decimals
    Reverts on what is expected to be a valid pair

Oracle::add_pair
    The same currency ID can be overwritten multiple times
```

Recommendation(s): Consider adding a check to `add_currency(...)`, `add_pair(...)`, `_set_keys_currencies(...)`, and `_set_keys_-pairs(...)` to ensure that the ID of the new item cannot be zero.

Status: Fixed

Update from the client: This issue is resolved in the commits below, following the recommendations. We added zero-check for the required functions.

- f2b3d81a9d064928b9882736e1c5b2b374230146;
- 49e7aca4d75a42a82baae2f8a65987a70ee297b4;

6.8 [Info] A new pair can be added with non-existing base and quote currencies

File(s): `src/oracle/oracle.cairo`

Description: When adding a pair through `add_pair(...)` and `_set_keys_pairs(...)`, there is no input validation for the new pair's `quote_currency_id` and `base_currency_id`.

```
fn add_pair(ref self: ContractState, new_pair: Pair) {
    self.assert_only_admin();
    let check_pair = self.oracle_pairs_storage.read(new_pair.id);

    ///////////////////////////////////////////////////////////////////////
    // @audit Missing validations on new pair quote and base currency id
    //        Quote or base or both currencies may not exist
    //        Quote and base currencies may be the same
    ///////////////////////////////////////////////////////////////////////
    assert(check_pair.id == 0, 'Pair with this key registered');
    self.emit(Event::SubmittedPair(SubmittedPair { pair: new_pair }));
    self.oracle_pairs_storage.write(new_pair.id, new_pair);
    self
        .oracle_pair_id_storage
        .write((new_pair.quote_currency_id, new_pair.base_currency_id), new_pair.id);
    return ();
}
```

This can allow a pair to be created with invalid one or both currencies. Other functions in the code that use these currency IDs may assume that the currency exists, which may not be the case. This affects `get_decimals(...)`, for example, which does not verify that the base currency of a given pair exists, making it possible to read from an unwritten storage slot and return a decimals of `0`, as shown in the snippet below:

```
DataType::SpotEntry(pair_id) => {
    let pair = self.oracle_pairs_storage.read(pair_id);
    assert(!pair.id.is_zero(), 'No pair found');
    /////////////////////////////////////////////////////////////////
    // @audit Existence check is only for pair
    //        Base currency existence is not checked before reading
    /////////////////////////////////////////////////////////////////
    let base_cur = self.oracle_currencies_storage.read(pair.base_currency_id);
    base_cur
},
```

Recommendation(s): Consider adding checks to both `add_pair(...)` and `_set_keys_pairs(...)` to ensure that the base and quote currencies exist before writing the new pair to storage.

Status: Fixed

Update from the client: The issue has been resolved through two separate commits.

- 3bee9c955d96a093866b6bd057a8cc940fdcdd63;
- 2f877e8b94765d3f9299ab477d13ff7cbf7abb3c;

The commit 3bee9c addresses the add_pair function, while the commit 2f877 pertains to the _set_keys_pairs() function. In both instances, the solution involved accessing the storage slot associated with the base/quote currency and managing scenarios where the returned ID is null.

6.9 [Info] Cannot remove publisher with zero address

File(s): `src/publisher_registry/publisher_registry.cairo`

Description: The publisher registry contract checks if a given publisher exists by checking if the publisher address is non-zero. However, setting a publisher address to zero through the functions `add_publisher` and `update_publisher_address` is possible. A publisher can update their address to zero, which can prevent them from being removed by `removed_publisher` since the zero check will cause the call to revert, as shown below:

```
fn remove_publisher(...) {
    // ...
    // @audit Zero check will revert for a manually set zero address
    let not_exists: bool = self.publisher_address_storage.read(publisher).is_zero();
    assert(!not_exists, 'Publisher not found');
    // ...
}
```

Recommendation(s): Consider implementing a zero address check in the functions `update_publisher_address(...)` and `add_publisher(...)`.

Status: Fixed

Update from the client: This issue is resolved in commits

- c0e9c49d09b42f5331d5ccda8fc91d90c2f62e42;
- 6e45950699fd652ba7472da553e8dc8b83b81056;

Following the recommendation, we added zero-address checks in both functions.

6.10 [Info] Empty arrays lead to revert in `_sum_volatility(...)`

File(s): `metrics.cairo`

Description: In the function `_sum_volatility(...)`, when an empty array is passed, the stop condition for the loop will not be triggered. This happens because the `cur_idx` is set to `1`, while the array length would be zero. As a result, the function will revert as it attempts to access an index that does not exist.

```
fn _sum_volatility(arr: Span<TickElem>) -> ... {
    let mut cur_idx = 1;
    let mut sum = FixedTrait::new(0, false);
    loop {
        if (cur_idx == arr.len()) {
            break ();
        }
        // @audit This array read will fail on a zero-length array
        let cur_val = *arr.at(cur_idx);
        let prev_val = *arr.at(cur_idx - 1);
        // ...
    };
}
```

This function is called by the `volatility(...)` function, which has logic to handle an array size of zero, but this check is done after the call to `_sum_volatility(...)`, so execution will have reverted before this check can be reached.

```
fn volatility(arr: Span<TickElem>) -> ... {
    // @audit Array of size zero will revert before the length check
    let _volatility_sum = _sum_volatility(arr);
    if (arr.len() == 0) {
        return 0;
    }
    // ...
}
```

Recommendation(s): Consider adjusting the loop condition to the following to handle empty arrays:

```
if (cur_idx >= arr.len()) {
    break ();
}
```

Status: Fixed

Update from the client: This issue is resolved in commit 01d9a1bf1a9d80bc4215b0d6673e86910dae7dbd, following the recommendation.

6.11 [Info] Empty arrays lead to revert in merge(...) function

File(s): merge_sort.cairo

Description: The merge(...) function performs a merge sort on an input array. However, the current implementation does not account for the case where the provided array arr is empty. In that case, the execution will revert inside the else block when attempting to access a non-existent element at index 0.

```
fn merge(...) -> Span<T> {
    if arr.len() > 1_u32 {
        // ...
    } else {
        let mut result_arr = ArrayTrait::<T>::new();
        result_arr.append(*arr.at(0)); // @audit reverts if `arr` is an empty array
        result_arr.span()
    }
}
```

Recommendation(s): Consider implementing a check for empty arrays and return the appropriate result.

Status: Fixed

Update from the client: The issue is resolved in commit a08dc20d26618f87af8bde8776c90db6e62b8764. We handle the edge case by returning an empty span if the length is null.

6.12 [Info] Incorrect sign determination in `pairwise_1D(...)` multiplication operation

File(s): `metrics.cairo`

Description: The function `pairwise_1D(...)` is used to perform subtraction and multiplication on elements in two given arrays. When conducting a multiplication operation, the current implementation erroneously determines the output sign by comparing the magnitudes of the input values.

This approach will lead to the wrong sign being applied to the output value, where it will only be possible to have a `false` sign when both magnitudes are equal. Instead, the signs of `x1` and `y1` should be compared, resulting in the correct outputs.

```
fn pairwise_1D(...) -> ... {
    // ...
    Operations::MULTIPLICATION(()) => {
        // ...
        // @audit Resulting sign should be determined by the equality of sign, not magnitude
        if x1.mag == y1.mag {
            output.append(FixedTrait::new(mag: x1.mag * y1.mag, sign: false));
        } else {
            output.append(FixedTrait::new(mag: x1.mag * y1.mag, sign: true));
        }
    },
    // ...
}
```

Recommendation(s): Consider determining the resulting output sign by checking the signs of `x1` and `y1` rather than their magnitudes.

Status: Fixed

Update from the client: The isssue is resolved in commit 0476e45d41d1e89fbaf2810655308ea01da2cc49, considering the sign instead of the magnitude.

6.13 [Info] Missing check in `calculate_volatility(...)`

File(s): `summary_stats.cairo`

Description: The function `calculate_volatility(...)` has the following check to ensure that there is enough data available to compute the volatility, as shown below.

```
assert(start_index != latest_checkpoint_index, 'Not enough data');
```

However, another condition can lead to insufficient data to compute the volatility, which is when the `start_index` and `end_index` are the same.

Recommendation(s): Consider updating the check to cover the case when `start_index == end_index`. Alternatively, the existing check can be changed to `start_index < end_index` to cover both cases.

Status: Fixed

Update from the client: Resolved in commits d5273b673ecc894c80e3f73a369af54c839ec1dc and 287959d51e8107aaec01c09ee0d4b76cb8d128e0, the operation will be reverted if the `start_index` is higher than the `end_index`.

6.14 [Info] Missing request cancellation logic in `cancel_random_request (...)`

File(s): `randomness.cairo`

Description: The `cancel_random_request(...)` function is currently missing the logic for canceling a request. Specifically, it does not update the request status to `CANCELLED` as it is indicated in the `RandomnessStatusChange` event.

```
fn cancel_random_request(...) {
    ...
    assert(_hashed_value == stored_hash_, "invalid request owner");
    assert(requestor_address == caller_address, "invalid request owner");

    self
        .emit(
            Event::RandomnessStatusChange(
                RandomnessStatusChange {
                    requestor_address: requestor_address,
                    request_id: request_id,
                    status: RequestStatus::CANCELLED(()) // @audit the function is not updating the request status
                }
            )
        );
    ...
}
```

Recommendation(s): Consider updating the request status to `CANCELLED`, and implementing any other missing cancellation logic.

Status: Fixed

Update from the client: This issue is resolved on commits

- a690ee32ab3fe3c6f7f89f095a8037e0e9f427d7;
- 0ac4296564900bdb4dbd647e82c1844020751df1;

We implemented the missing logic for the cancellation and added tests.

6.15 [Info] Multiple publishers can share the same address

File(s): `src/publisher_registry/publisher_registry.cairo`

Description: When adding or updating a publisher address on the publisher registry, it is possible to use the same address for multiple publisher IDs. Given two publishers, Alice and Bob, it is possible for Alice to set their address to the same as Bob, which will allow Bob to publish data on behalf of Alice as well as his own data. Although this is unlikely to occur, it can potentially lead to hidden centralization risks where one publisher can submit data on behalf of others.

Recommendation(s): Consider adding input validation to `add_publisher(...)` and `update_publisher_address(...)` to ensure the given address is not already used.

Status: Fixed

Update from the client: The issue is resolved in the following commits. We added a function realizing the verification among the publishers' addresses. If the boolean returned by the function is true, the address is already registered, and the operation is reverted.

- b1349e2ae55e5f7752b43947945087b4e8efe736;
- 9cd8d8d23165aa18a452806994c75ef78218f702;

6.16 [Info] Necessary checks are inadequate

File(s): src/compute_engines/yield_curve/yield_curve.cairo

Description: The following functions do not check that their parameters pair_id and on_key already exist in storage. This can lead to non-existing pairs or keys being activated where it should otherwise not be possible.

```
fn set_pair_id_is_active(ref self: ContractState, pair_id: felt252, is_active: bool) {
    assert_only_admin(@self);
    self.pair_id_is_active_storage.write(pair_id, is_active); // @audit-issue: Can set active non-existent pair id
    return ();
}

fn set_on_key_is_active(ref self: ContractState, on_key: felt252, is_active: bool) {
    assert_only_admin(@self);
    self.on_key_is_active_storage.write(on_key, is_active); // @audit-issue: Can set active non-existent key
    return ();
}
```

Recommendation(s): We recommend checking if the relevant data exists in storage before setting it as active.

Status: Fixed

Update from the client: The issue has been addressed in commit afff37b5fdbf41450e03d51c306c274468583601. We have implemented existence checks for the functions set_pair_id_is_active, add_pair, set_on_key_is_active, and add_on_keys. Now, before adding a new pair or on_key, we first verify that it does not already exist. Similarly, when the administrator intends to activate a pair_id or an on_key, we ensure it already exists in the storage.

6.17 [Info] Request status handling issues in `randomness.cairo` contract

File(s): `randomness.cairo`

Description: The `randomness.cairo` contract defines the `RequestStatus` enum structure to represent the status of a user request.

```
enum RequestStatus {
    UNINITIALIZED: (),
    RECEIVED: (),
    FULFILLED: (),
    CANCELLED: (),
    EXCESSIVE_GAS_NEEDED: (),
    ERRORED: (),
}
```

However, there are several issues related to the proper handling of request statuses:

- A user is able to cancel a request that is in a state different from `RECEIVED`. This means the request could be already fulfilled or marked as `EXCESSIVE_GAS_NEEDED` or `ERRORED` by the admin, and the user will still be able to override this state. This is possible because `cancel_random_request(...)` function does not check the current request status;
- Admin can mistakenly fulfill a request that is canceled. That is because `submit_random(...)` function doesn't check for the current request status;
- Contract admin is allowed to update the request status to any value, regardless of the current status. This can lead to unexpected consequences;
- The admin can update the request status to `EXCESSIVE_GAS_NEEDED` or `ERRORED`. However, the code has no logic to handle these cases;

Recommendation(s): Consider revisiting the statuses implementation and flows within the `randomness.cairo` contract to ensure that only the expected flows are allowed. Remove any unused status for better code readability.

Status: Fixed

Update from the client: This issue is resolved in commits 4d056879c81d6d4e50df00ee473bd7f3a854cc0a. We added more logics regarding the status management. We removed the unused requests status. An admin is not able anymore to fulfill a canceled request, and is able to upgrade the request status only if it is not canceled or fulfilled (status we considered as definitive). The user cannot cancel a request already fulfilled.

6.18 [Info] Undefined divisions may occur in several functions

File(s): `scaler.cairo`, `convert.cairo`, `metrics.cairo`

Description: The functions below may face undefined divisions, i.e., divisions by zero:

- scaler.calculate_slope(...);
- scaler.scale_data(...);
- convert.div_decimals(...);
- metrics._sum_volatility(...);

Recommendation(s): Consider applying the following checks:

```
fn _sum_volatility(arr: Span<TickElem>) -> Fixed {
        ...
        ////////////////////////////////////////////////////
        // @audit check if prev_value is greater than zero
        ////////////////////////////////////////////////////
        let numerator_value = FixedTrait::ln(cur_value / prev_value);
        let numerator = numerator_value.pow(FixedTrait::new(2 * ONE_u128, false));
        let denominator = FixedTrait::new((cur_timestamp - prev_timestamp).into(), false)
            / FixedTrait::new(ONE_YEAR_IN_SECONDS, false);
        ////////////////////////////////////////////////////
        // @audit in case prev_timestamp == cur_timestamp,
        //        denominator is zero
        ////////////////////////////////////////////////////
        let fraction_ = numerator / denominator;
        ...
}
```

```
fn scale_data(
    start_tick: u64, end_tick: u64, tick_array: Span<TickElem>, num_intervals: u32
) -> Array<TickElem> {
    ////////////////////////////////////////////////////
    // @audit check if num_intervals > 1 AND
    //              if start_tick <= end_tick
    ////////////////////////////////////////////////////
    let interval = (end_tick - start_tick) / (num_intervals.into() - 1);
    ...
}
```

```
fn calculate_slope(x1: Fixed, x2: Fixed, y1: Fixed, y2: Fixed) -> Fixed {
    ////////////////////////////////////////////////////
    // @audit check if x2 > x1
    ////////////////////////////////////////////////////
    (y2 - y1) / (x2 - x1)
}
```

```
fn div_decimals(a_price: u128, b_price: u128, output_decimals: u128) -> u128 {
    let power = pow(10_u128, output_decimals);
   ...
    ////////////////////////////////////////////////////
    // @audit check if b_price > 0
    ////////////////////////////////////////////////////
    a_price * power / b_price
}
```

Status: Fixed

Update from the client: This issue is resolved in commit 5e6bd300af2b1b72287a98129da2d39389d48f30. We added assertions to avoid undefined operations.

6.19 [Info] `get_implementation_hash(...)` returns empty classhash until first upgrade

File(s): `src/upgradeable/upgradeable.cairo`

Description: The `Upgradeable` contract contains the storage variable `class_hash`, which stores the class hash of the current contract. This storage variable is updated whenever the `upgrade(...)` function is called. However, `class_hash` will remain unset between deployment and the first upgrade, causing `get_implementation_hash(...)` to return a classhash of zero until the first upgrade.

```
////////////////////////////////////////////////////////////////////////////
// @audit Storage variable `class_hash` is only written to after an upgrade
////////////////////////////////////////////////////////////////////////////
fn upgrade(ref self: ContractState, new_class_hash: ClassHash) {
    assert(!new_class_hash.is_zero(), 'Class hash cannot be zero');
    starknet::replace_class_syscall(new_class_hash).unwrap();
    self.class_hash.write(new_class_hash);
    self.emit(Upgraded { class_hash: new_class_hash });
}

//////////////////////////////////////////////////////////////////
// @audit After deployment this will return zero until upgrade
//////////////////////////////////////////////////////////////////
fn get_implementation_hash(self: @ContractState) -> ClassHash {
    self.class_hash.read()
}
```

Recommendation(s): Consider initializing the `Upgradeable` contract with its current class hash. Alternatively, since `get_implementation_hash(...)` is only used externally, it could be removed, and the class hash can be queried from a block explorer such as Voyager instead.

Status: Fixed

Update from the client: The issue is resolved in the commit 5242bbaa2a4c2b3c791ac566977bfd9ffccc0122. We decided to simply remove the function.

6.20 [Best Practices] Control Coupling in function `pairwise_1D(...)`

File(s): `src/operations/time_series/metrics.cairo`

Description: The function `pairwise_1D(...)` receives the operation as an argument (`SUBTRACTION` or `MULTIPLICATION`). Control coupling refers to a situation where one module (or component) controls the behavior of another module by passing information about what to do, typically in the form of control flags, parameters, or other control-related data. This form of coupling occurs when one module influences the execution flow or behavior of another module by providing information on how to perform a particular operation.

Control coupling can be contrasted with data coupling, where modules communicate primarily by passing data rather than control information. In well-designed software, minimizing control and data coupling is often considered a good practice. It enhances the modularity, reusability, and maintainability of the code.

Reducing control coupling helps to make modules more independent and less reliant on the internal details of each other. This independence makes modifying and maintaining individual modules easier without affecting the entire system. The code with audit comments is presented below.

```
fn pairwise_1D(operation: Operations, x_len: u32, x: Span<Fixed>, y: Span<Fixed>) -> Span<Fixed> {
    //We assume, for simplicity, that the input arrays (x & y) are arrays of positive elements
    let mut cur_idx: u32 = 0;
    let mut output = ArrayTrait::<Fixed>::new();
    match operation {
        Operations::SUBTRACTION(()) => {
            loop {
                if (cur_idx >= x_len) {
                    break ();
                }
                let x1 = *x.get(cur_idx).unwrap().unbox();
                let y1 = *y.get(cur_idx).unwrap().unbox();
                if x1 < y1 {
                    output.append(FixedTrait::new(mag: y1.mag - x1.mag, sign: true));
                } else {
                    output.append(FixedTrait::new(mag: x1.mag - y1.mag, sign: false));
                }
                cur_idx = cur_idx + 1;
            };
        },
        Operations::MULTIPLICATION(()) => {
            loop {
                if (cur_idx >= x_len) {
                    break ();
                }
                let x1 = *x.get(cur_idx).unwrap().unbox();
                let y1 = *y.get(cur_idx).unwrap().unbox();
                if x1.mag == y1.mag {
                    output.append(FixedTrait::new(mag: x1.mag * y1.mag, sign: false));
                } else {
                    output.append(FixedTrait::new(mag: x1.mag * y1.mag, sign: true));
                }
                cur_idx = cur_idx + 1;
            };
        },
    }
    output.span()
}
```

Recommendation(s): We recommend breaking the function into two separate functions, one for performing subtraction and the other for performing multiplication.

Status: Fixed

Update from the client: This issue is resolved in commit 79f96d520c371bfd5223fc50b35db3f89216a4ba. Since the execution is different according to the operation parameter, we just implemented two separate functions instead.

6.21 [Best Practices] Dead code

File(s): `src/*`

Description: The codebase contains unused code, affecting overall readability and quality. A list of files and their unused declarations are shown below:

```
src/entry/structs.cairo
    const MEDIAN
    const BOTH_TRUE
    enum PossibleEntryStorage
    struct GenericEntryStorage
    struct FetchCheckpoint

src/oracle/oracle.cairo
    struct SubmittedOptionEntry
    fn aggregation_into_u8
    fn add_pair (private function only)

src/compute_engines/summary_stats/summary_stats.cairo
    const SCALED_ARR_SIZE
    fn _make_scaled_array:
        test
        first
        first_t

src/operations/time_series/metrics.cairo
    fn extract_value:
        cur_idx
    fn sum_tick_array:
        cur_idx
    fn sum_array:
        cur_idx
```

Recommendation(s): Consider removing the unused code shown above.

Status: Fixed

Update from the client: This issue is resolved in commits (Part of the dead code was already handled while solving other audit findings):

- 6ce4288faa7a3c274a22517a66dee1c298be3998;
- 82cca31d15552b04eee039d5497861c6ff6bbcb6;
- 5242bbaa2a4c2b3c791ac566977bfd9ffccc0122;

6.22 [Best Practices] Duplicate code in `mul_decimals(...)` function

File(s): `convert.cairo`

Description: The function `mul_decimals(...)` contains duplicate `assert` statements, which are shown below:

```
fn mul_decimals(...) -> ... {
    ...
    assert(power <= MAX_POWER, 'Conversion overflow');
    assert(a_price <= MAX_POWER, 'Conversion overflow');

    assert(power <= MAX_POWER, 'Conversion overflow');
    assert(a_price <= MAX_POWER, 'Conversion overflow');
    ...
}
```

Recommendation(s): Consider removing the unnecessary code to improve readability.

Status: Fixed

Update from the client: This issue is resolved in commit 779c574d531fd404a6ca763bac54b8a2c0d0a029, by removing the duplicate code.

6.23 [Best Practices] Hash implementation `TupleSize4LegacyHash` has an incorrect name

File(s): `src/oracle/oracle.cairo`

Description: To support a mapping that takes a tuple of five elements as a key, a custom implementation for `LegacyHash` has been written, which can hash such a tuple into one `felt252`. This implementation has been incorrectly named `TupleSize4LegacyHash`. It should be named `TupleSize5LegacyHash` instead.

Recommendation(s): Consider correcting the name of the custom `LegacyHash` implementation to `TupleSize5LegacyHash`.

Status: Fixed

Update from the client: Resolved in commit c5141649d45264bdd7f8206ccab76634cad3d6fd. We changed the previous implementation for `TupleSize5LegacyHash`, since we implemented it for 5 elements.

6.24 [Best Practices] Inconsistent type casting approach for `u8` and `AggregationMode`

File(s): `src/oracle/oracle.cairo`

Description: Type casting between `AggregationMode` and `u8` is supported to allow packed checkpoint storage reads and writes. The type casting approach differs depending on whether the conversion is to or from an `AggregationMode` type. When converting `AggregationMode` to `u8`, the into trait is used:

```
impl AggregationModeIntoU8 of Into<AggregationMode, u8> {
    fn into(self: AggregationMode) -> u8 {
        match self {
            AggregationMode::Median(()) => 0_u8,
            AggregationMode::Mean(()) => 1_u8,
            AggregationMode::Error(()) => 150_u8,
        }
    }
}
```

However, when converting `u8` to `AggregationMode`, a regular function is used instead:

```
fn u8_into_AggregationMode(value: u8) -> AggregationMode {
    if value == 0_u8 {
        AggregationMode::Median(())
    } else if value == 1_u8 {
        AggregationMode::Mean(())
    } else {
        AggregationMode::Error(())
    }
}
```

Recommendation(s): Consider implementing the conversion from `u8` to `AggregationMode` as an `into` method to improve consistency and composability, should this type be used in other contracts.

Status: Fixed

Update from the client: Resolved in commit 72bd41086c2af243abc11c76bc67648d01ac4806. We implemented try_into for the aggregation mode into u8 conversion and the into for the u8 into aggregation mode.

6.25 [Best Practices] Inconsistent use of `getter` function

File(s): `src/oracle/oracle.cairo`

Description: The oracle contract contains the function `get_publisher_registry_address(...)`, which is used internally and externally. However, some functions use the getter, and others read directly from the storage variable.

```
// @audit Direct storage read
fn update_publisher_registry_address(...) {
    ...
    let old_publisher_registry_address = self
        .oracle_publisher_registry_address_storage
        .read();
    ...
}

// @audit Getter function used
fn validate_sender_for_source<...>(...) {
    let publisher_registry_address = IOracleABI::get_publisher_registry_address(self);
    // ...
}
```

Recommendation(s): Consider retrieving the publisher registry address in a consistent manner across all functions.

Status: Fixed

Update from the client: This issue is resolved in commit 5242bbaa2a4c2b3c791ac566977bfd9ffccc0122.

6.26 [Best Practices] Incorrect comments

File(s): `src/*`

Description: There are several comments which contain incorrect information or contain typos, which are listed below:

- `Oracle.get_all_sources` has a comment stating it returns a span of sources when an array is returned instead;
- `Oracle.get_all_entries` is missing an `@returns` NatSpec comment;
- `PublisherRegistry.get_publisher_address` has a typo for the word "publisher";
- `Randomness.cancel_random_request` has a revert message "invalid request owner", while it should be "request hash != stored hash";

Recommendation(s): Consider adding the missing comments and fixing the incorrect ones.

Status: Fixed

Update from the client: This issue is resolved in commit 6a0b2e3a4db9de42a75db71e521fb664e513fe6f. We updated and corrected the wrong comments.

6.27 [Best Practices] Shadowed arrays are unused

File(s): src/oracle/oracle.cairo

Description: The functions get_data_for_sources(...) and get_data_entries(...) define an array immediately shadowed by a span with the same name returned from an internal function call. In both of these cases, the original array is never used. A snippet from get_data_for_sources is shown below:

```
fn get_data_for_sources(...) -> PragmaPricesResponse {
    //////////////////////////////////////////////////////////
    // @audit `entries` is shadowed immediately and not used
    //////////////////////////////////////////////////////////
    let mut entries = ArrayTrait::<PossibleEntries>::new();

    let (entries, last_updated_timestamp) = IOracleABI::get_data_entries_for_sources(
        self, data_type, sources
    );
    // ...
}
```

Recommendation(s): Consider removing the shadowed arrays in get_data_for_sources and get_data_entries to reduce unused code and improve readability.

Status: Fixed

Update from the client: This issue is resolved in commit 8e1ff20847cdb1bb52758a43007a34b066b14ab2. We just removed the unused code.

6.28 [Best Practices] Unnecessary iteration counter

File(s): `src/operations/time_series/metrics.cairo`

Description: The functions `extract_value`, `sum_tick_array` and `sum_array` all define an unused variable `cur_idx`, as shown below:

```
/////////////////////////////////////////////////////////////////
// @audit Unused variable `cur_idx`
//        Same issue applies to `sum_tick_array` and `sum_array`
/////////////////////////////////////////////////////////////////
fn extract_value(mut tick_arr: Span<TickElem>) -> Array<Fixed> {
    let mut output = ArrayTrait::<Fixed>::new();
    let mut cur_idx = 0; // @audit-issue: Unnecessary variable.
    // ...
}
```

Recommendation(s): Consider removing the unused variables to improve code clarity.

Status: Fixed

Update from the client: This issue is resolved in commit 5242bbaa2a4c2b3c791ac566977bfd9ffccc0122. We deleted the unnecessary variable.

6.29 [Best Practices] Unnecessary local variable assignment

File(s): `src/oracle/oracle.cairo`

Description: The function `get_data_with_USD_hop(...)` declares local variables `base_dt` and `quote_dt` inside the `FutureEntry` branch of the match statement. These variable declarations are unnecessary as the assigned value can be returned directly, similar to what is done in the `SpotEntry` branch.

Recommendation(s): We recommend to directly return the values using idiomatic syntax to make the code more concise and readable.

```
Option::Some(expiration) => {
-     let base_dt = DataType::FutureEntry((base_pair_id, expiration));
-     let quote_dt = DataType::FutureEntry((quote_pair_id, expiration));
   (
-     base_dt,
-     quote_dt,
+     DataType::FutureEntry((base_pair_id, expiration)),
+     DataType::FutureEntry((quote_pair_id, expiration)),
      self.oracle_currencies_storage.read(quote_currency_id)
   )
},
```

Status: Fixed

Update from the client: This issue is resolved in commit 5242bbaa2a4c2b3c791ac566977bfd9ffccc0122, following the recommendation.

6.30 [Best Practices] Unnecessary storage read

File(s): src/publisher_registry/publisher_registry.cairo

Description: The function add_source_for_publisher(...) performs an unnecessary storage read by defining the cur_idx and then re-reading later in the function. This is shown in the code snippet below:

```
fn add_source_for_publisher(...) {
    ...
    let cur_idx = self.publishers_sources_idx.read(publisher); // @audit Defined here 1st
    if (cur_idx == 0) {
        // ...
    } else {
        // ...
        let cur_idx = self.publishers_sources_idx.read(publisher); // @audit Shadowed here 2nd
    ...
    }
}
```

Recommendation(s): Consider removing the unnecessary storage read for cur_idx.

Status: Fixed

Update from the client: This issue is resolved in commmit 6e45950699fd652ba7472da553e8dc8b83b81056, by deleting the unnecessary index definition.

6.31 [Best Practices] Unnecessary use of `into` in some functions

File(s): `src/oracle/oracle.cairo`

Description: The functions `publish_data(...)` and `build_sources_array(...)` unnecessarily use the `into` trait, where both the variable and the target type are the same. Snippets of both functions are shown below:

```
//////////////////////////////////////////////////
// @audit Function `publish_data`
//        All fields are already the correct type
//////////////////////////////////////////////////

let element = EntryStorage {
    timestamp: spot_entry.base.timestamp.into(),
    volume: spot_entry.volume.into(),
    price: spot_entry.price.into()
};
```

```
//////////////////////////////////////////////////
// @audit Function `build_sources_array`
//        `idx` is aready the correct type
//////////////////////////////////////////////////

let new_source = self
    .oracle_sources_storage
    .read((pair_id, SPOT, idx.into(), 0));
```

Recommendation(s): As shown above, consider removing the unnecessary `into` casts. Note that for each of these functions, there are three occurrences of the shown snippets. We have only shown `SPOT`, but there is `FUTURE` and `GENERIC` too.

Status: Fixed

Update from the client: This issue is resolved in commit 5242bbaa2a4c2b3c791ac566977bfd9ffccc0122, removing the unnecessary conversion.

6.32 [Best Practices] Unnecessary variable redefinition inside a loop

File(s): `summary_stats.cairo`

Description: The `oracle_dispatcher` and `offset` variables are unnecessarily redefined in each iteration of the loop, even though their values are constant during the execution.

```
fn _make_scaled_array(...) -> Array<TickElem> {
    let mut tick_arr = ArrayTrait::<TickElem>::new();
    let mut idx = 0;
    loop {
        // @audit `oracle_dispatcher` and `offset` are redefined in each iteration
        let oracle_dispatcher = IOracleABIDispatcher { contract_address: oracle_address };
        let offset = latest_checkpoint_index - num_datapoints;
        // ...
    };
    // ...
}
```

Recommendation(s): Consider moving the declaration of `oracle_dispatcher` and `offset` outside the loop.

Status: Fixed

Update from the client: This issue is resolved in commit f634c4c1e3aecdcad0a758351e1b992860abd027, following the recommendation.

6.33 [Best Practices] Unnecessary variable shadowing

File(s): src/compute_engines/yield_curve/yield_curve.cairo

Description: The function `set_future_expiry_timestamp_is_active(...)` defines the variable `old_expiry`, which is immediately shadowed by a duplicated line that declares the same data to the same variable name.

```
fn set_future_expiry_timestamp_is_active(...) {
    // ...
    let old_expiry = IYieldCurveABI::get_future_expiry_timestamp_expiry(
        @self, pair_id, future_expiry_timestamp
    );
    let old_expiry = IYieldCurveABI::get_future_expiry_timestamp_expiry(
        @self, pair_id, future_expiry_timestamp
    );
    // ...
}
```

Recommendation(s): Consider removing the duplicated line.

Status: Fixed

Update from the client: This issue is resolved in commit 5242bbaa2a4c2b3c791ac566977bfd9ffccc0122. We removed the duplicate.

6.34 [Best Practices] Unrequired overhead for computing the variance

File(s): `src/operations/time_series/metrics.cairo`

Description: The function `variance(...)` computes the variance of the dataset. To compute the variance of a dataset, first, we must find the mean by summing up all values and dividing by the number of data points. This is performed by the function `mean(...)`.

Next, we must calculate the squared differences between each data point and the mean. To achieve this, the code creates an array having the same length as the original dataset using the function `fill_1d(...)`, as shown below.

```
/// Fills an array with one `value`
fn fill_1d(arr_len: u32, value: u128) -> Array<Fixed> {
    let mut cur_idx = 0;
    let mut output = ArrayTrait::new();
    loop {
        if (cur_idx >= arr_len) {
            break ();
        }
        output.append(FixedTrait::new(mag: value, sign: false));
        cur_idx = cur_idx + 1;
    };
    output
}
```

The instantiation of a new array with identical values at each position is deemed an unnecessary overhead.

Recommendation(s): Compute the difference between each element of the dataset and the mean without instantiating this new array with identical values at each position.

Status: Fixed

Update from the client: The issue has been resolved in the commit 59a78563561e58bcecaab6053300a2b5c35a542b. We modified the second parameter of the `pairwise_1D_sub` function. Rather than accepting a Span of elements, it now takes a single element. This element is then subtracted from an array of elements `x`. By doing so, we were able to remove `fill_1D`.

6.35 [Best Practices] Unused function parameters

File(s): src/*

Description: The codebase contains some functions with unused parameters, which are listed below:

```
src/compute_engines/summary_stats/summary_stats.cairo
    _make_scaled_array -> start_tick

src/compute_engines/yield_curve/yield_curve.cairo
    assert_only_admin -> self
    change_decimals -> self

src/randomness/randomness.cairo
    assert_only_admin -> self
    submit_random -> block_hash
```

Recommendation(s): Consider removing these parameters from the above functions to improve code clarity and readability.

Status: Fixed

Update from the client: This issue is resolved in commits 7214f51f7b1801fedeadacbfafe874e3b30019c3 and d4e1fb90d7dcb5242c934770cc288cba5252
We removed all the unnecessary function parameters left after previous code changes.

6.36 [Best Practices] Unused imports

File(s): src/*

Description: The codebase contains unused imports affecting overall readability and code quality. A list of files with unused imports is shown below:

```
src/compute_engines/summary_stats/summary_stats.cairo
    cubit::f128::types::fixed::Fixed
    result::ResultTrait
    debug::PrintTrait
    starknet::get_caller_address
    zeroable::Zeroable
    option::OptionTrait
    result::ResultTrait
    traits::Into
    traits::TryInto
    pragma::operations::time_series::scaler::scale_data

src/operations/time_series/structs.cairo
    integer::u32
```

Recommendation(s): Consider removing the unused imports shown above.

Status: Fixed

Update from the client: This issue is resolved in commit 9c0b4aaefa2edd0ceee4f76214060528cd9fd963. We removed the unnecessary imports(the debug::PrintTrait was removed for all contracts in another commit).

6.37 [Best Practices] Unused parameters in function get_future_spot_pragma_source_-key(...)

File(s): `yield_curve.cairo`

Description: The function `get_future_spot_pragma_source_key(...)` does not use the parameters `pair_id` and `future_expiry_timestamp` in the contract `YieldCurve`.

```
fn get_future_spot_pragma_source_key(
    self: @ContractState, pair_id: felt252, future_expiry_timestamp: u64
) -> felt252 {
    let future_spot_pragma_source_key = self.future_spot_pragma_source_key_storage.read();
    return future_spot_pragma_source_key;
}
```

Recommendation(s): Consider removing them from the function.

Status: Fixed

Update from the client: The issue has been resolved with commit a6f92c2c87a7f84aaee26cbd4c1a503fd0c39634. In this update, we removed remnants of a previous implementation that were no longer needed.

6.38 [Best Practices] Use the same value while converting types

File(s): `src/oracle/oracle.cairo`

Description: The enum `AggregationMode` has three variants: `Median`, `Mean`, and `Error`. The `Error` variant represents failed conversions between `u8` where the provided integer is not associated with an aggregation variant, and if the aggregation mode is `Error`, then the protocol panics, as shown below:

```
fn aggregate_entries<...>(...) -> u128 {
    // ...
    match aggregation_mode {
        // ...
        AggregationMode::Error(()) => {
            panic_with_felt252('Wrong aggregation mode');
            0
        }
    }
}
```

Instead of having a native `Error` variant for `AggregationMode`, a better approach would be to use the `try_into` trait when converting from a `u8` to `AggregationMode`, where the `try_into` returns an `Option::None()` for an invalid `u8`. This more idiomatic approach suits the Cairo language and will ensure that conversion errors are handled at conversion rather than at some later point in the code.

Recommendation(s): Consider removing the `Error` variant from `AggregationMode` and using `try_into` to handle conversions from `u8` to `AggregationMode` instead.

Status: Fixed

Update from the client: This issue is resolved in commit 72bd41086c2af243abc11c76bc67648d01ac4806. We implemented try_into for the aggregation mode into u8 conversion and for the u8 into aggregation mode.

7 Documentation Evaluation

Software documentation refers to the written or visual information describing software's functionality, architecture, design, and implementation. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about the Pragma Oracle documentation

Pragma has sufficient documentation on the website https://docs.pragmaoracle.com/ and inline comments. Moreover, the client provided additional detailed documentation for the audit team. During the audit process, the audit team remained in constant communication with the client.

8 Test Suite Evaluation

8.1 Compilation Output

```
scarb build
Updating git repository https://github.com/keep-starknet-strange/alexandria
Updating git repository https://github.com/influenceth/cubit
Compiling lib(pragma) pragma v0.2.0 (C:\Users\computer\Desktop\Nethermind\pragma-oracle\Scarb.toml)
Compiling starknet-contract(pragma) pragma v0.2.0 (C:\Users\computer\Desktop\Nethermind\pragma-oracle\Scarb.toml)
Finished release target(s) in 15 seconds
```

8.2 Tests Output

```
scarb test
    Running cairo-test pragma
    Compiling test(pragma_unittest) pragma v0.2.0
    Finished release target(s) in 6 seconds

testing pragma ...
running 63 tests

test pragma::operations::time_series::scaler::test_scaler ... ok
test pragma::entry::entry::test_aggregate_entries_median ... ok
test pragma::operations::time_series::metrics::test_utils ... ok
test pragma::entry::entry::test_aggregate_entries_mean ... ok
test pragma::entry::entry::test_aggregate_timestamp_max ... ok
test pragma::operations::time_series::metrics::test_metrics ... ok
test pragma::entry::entry::test_empty_array ... ok
test pragma::operations::sorting::merge_sort::test_merge ... ok
test pragma::operations::sorting::merge_sort::test_slice ... ok
test pragma::operations::time_series::convert::test_convert_via_usd ... ok
test pragma::tests::test_publisher_registry::test_remove_publisher ... ok
test pragma::tests::test_publisher_registry::test_update_publisher_should_fail_if_not_publisher ... ok
test pragma::tests::test_publisher_registry::test_remove_publisher_should_fail_if_publisher_does_not_exist ... ok
test pragma::tests::test_publisher_registry::test_add_publisher_should_fail_if_not_admin ... ok
test pragma::tests::test_publisher_registry::test_add_source_should_fail_if_source_already_exists ... ok
test pragma::tests::test_publisher_registry::test_add_publisher_should_fail_if_publisher_already_exists ... ok
test pragma::tests::test_publisher_registry::test_add_source_should_fail_if_not_admin ... ok
test pragma::tests::test_publisher_registry::test_change_admin ... ok
test pragma::tests::test_publisher_registry::test_add_source ... ok
test pragma::tests::test_publisher_registry::test_change_admin_should_fail_if_not_admin ... ok
test pragma::tests::test_publisher_registry::test_remove_source_should_fail_if_not_admin ... ok
test pragma::tests::test_publisher_registry::test_change_admin_should_fail_if_admin_is_zero ... ok
test pragma::tests::test_oracle::test_data_entry_should_fail_if_not_found_2 ... ok
test pragma::tests::test_publisher_registry::test_remove_source_should_fail_if_source_does_not_exist ... ok
test pragma::tests::test_oracle::get_data_median_for_sources_should_fail_if_wrong_sources ... ok
test pragma::tests::test_publisher_registry::test_change_admin_should_fail_if_admin_is_same_as_current_admin ... ok
test pragma::tests::test_oracle::get_data_median ... ok
test pragma::tests::test_oracle::test_set_checkpoint ... ok
test pragma::tests::test_oracle::test_get_decimals ... ok
test pragma::tests::test_publisher_registry::test_remove_source ... ok
test pragma::tests::test_publisher_registry::test_change_admin_should_fail_if_admin_is_same_as_current_admin_2 ... ok
test pragma::tests::test_oracle::test_publish_multiple_entries ... ok
test pragma::tests::test_publisher_registry::test_remove_publisher_should_fail_if_not_admin ... ok
test pragma::tests::test_yield_curve::test_yield_curve_deploy ... ok
test pragma::tests::test_oracle::test_data_entry_should_fail_if_not_found_3 ... ok
test pragma::tests::test_oracle::get_data_for_sources ... ok
test pragma::tests::test_oracle::get_data_median_for_sources ... ok
test pragma::tests::test_oracle::test_get_decimals_should_fail_if_not_found ... ok
test pragma::tests::test_oracle::test_set_checkpoint_should_fail_if_wrong_data_type ... ok
test pragma::tests::test_yield_curve::test_yield_curve_empty ... ok
test pragma::tests::test_oracle::test_get_decimals_should_fail_if_not_found_2 ... ok
test pragma::tests::test_oracle::test_get_admin_address ... ok
test pragma::tests::test_yield_curve::test_yield_curve_computation ... ok
test pragma::tests::test_summary_stats::test_summary_stats_mean_median ... ok
test pragma::tests::test_oracle::test_get_data ... ok
test pragma::tests::test_oracle::test_get_last_checkpoint_before ... ok
test pragma::tests::test_randomness::test_randomness ... ok
test pragma::tests::test_oracle::test_max_publish_multiple_entries ... ok
test pragma::tests::test_summary_stats::test_summary_stats_mean_mean ... ok
test pragma::tests::test_oracle::test_data_median_multi_should_fail_if_no_expiration_time_associated ... ok
test pragma::tests::test_oracle::test_data_entry_should_fail_if_not_found ... ok
test pragma::tests::test_oracle::test_data_entry ... ok
test pragma::tests::test_oracle::test_get_data_median_multi ... ok
test pragma::tests::test_oracle::test_get_last_checkpoint_before_should_fail_if_wrong_data_type ... ok
test pragma::tests::test_oracle::test_get_data_with_usd_hop ... ok
test pragma::tests::test_publisher_registry::test_add_publisher ... ok
test pragma::tests::test_publisher_registry::test_update_publisher_address ... ok
test pragma::tests::test_publisher_registry::test_register_non_admin_fail ... ok
test pragma::tests::test_oracle::test_data_median_multi_should_fail_if_wrong_sources ... ok
test pragma::tests::test_oracle::test_data_median_multi_should_fail_if_wrong_data_types ... ok
test pragma::tests::test_oracle::test_get_last_checkpoint_before_should_fail_if_timestamp_too_old ... ok
test pragma::tests::test_oracle::test_get_data_with_USD_hop_should_fail_if_wrong_id ... ok
test pragma::tests::test_summary_stats::test_set_future_checkpoint ... ok
test result: ok. 63 passed; 0 failed; 0 ignored; 0 filtered out;
```

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;

- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;

- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

Disclaimer