Security Assessment & Formal Verification Report

Warp Exchange

May 2024

Prepared for
Warp Team

certora

Table of content

certora

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Warp Exchange	private		EVM/Yul (custom Yul)

Project Overview

This document describes the specification and verification of the Warp Exchange using the Certora Prover and manual code review findings. The work was undertaken from **April 10th** to **May 30th 2024**.

The Certora Prover demonstrated that the implementation of the **Yul** contracts above is correct with respect to the formal rules written by the Certora team. In addition, the team performed a manual audit of all the Yul contracts**.** During the verification process and the manual audit, the Certora team discovered issues in the Yul contracts code, as listed below.

Protocol Overview

The Warp exchange is a book-order model DEX.
The core part and innovation of this protocol is the order matching algorithm (and the data structure to keep it), which allows logarithmic execution and makes this model viable.
Warp Exchange was written in a custom YUL language, to further optimize the algorithm and gas usage.

Other features of the protocol:

- The protocol supports many trading pairs, each separated in a different book in its storage. Each trading pair doesn't affect the others.
- Orders could be matched in a batch, and users could claim later their own completed orders.
- Order matching conditions/types - GTC, IOC, FOK, POST
- Batching of orders and other interactions

- Users could deposit some native tokens (considered “fee”), to incentivize a third party (a “collector”) to claim a completed order for them.
- An order array, to keep track of what are the orders of a specific user

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	1		
High	0		
Medium	2		
Low	1		
Informational	9		
Total			

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

certora

Detailed Findings

ID	Title	Severity	Status
C-01	Overflowing a price tree can lead to a DOS on a specific matched pair	Critical	Not fixed yet
M-01	Bypassing codesize() check with a malicious contract could cause reverts and loss for Collector batches	Medium	Not fixed yet
M-02	Collector won't receive fee due to revert on a claim for order caused by token reverts	Medium	Not fixed yet
L-01	Rounding when matching can allow line cutting	Low	Not fixed yet
I-01	The "*fallback*" method is actually *multiview(bytes[])*	Informational	
I-02	The price decode is not always using the strict form	Informational	
I-03	Some multiplications don't use the safemul	Informational	
I-04	The 24h collector lock mitigation is counterproductive	Informational	

I-05	Some functions are commented as “internal use only” but are public functions.	Informational	
I-06	_replace() function lacks a pair_id check.	Informational	
I-07	codestyle suggestion - replace raw sstore() calls with dedicated functions/macros.	Informational	
I-08	codestyle suggestion - Macros and naming conventions	Informational	

Critical Severity Issues

C-01 Overflowing a price tree can lead to a DOS on a specific matched pair.

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: more_magic.yul#L77	Status: Acknowledged, Inserting to a tree would be slightly redesigned	

Description: When inserting a new order each parent has its total value and total quantity increased. However once we reach price subtree (depther 24+) we should check that the new price and quantity does not cause a node to hold more than the maximum allotted for each price. Currently the check is done on the child before the addition of the freshly added node.

This bug together with the fact that once we create a new branch we only check that the direct parent did not overflow, as opposed to checking that each paranet in that price did not overflow, this allows through clever insertions to overflow a price.

Together with the fact that if a tree only has one price, each insertion goes through that price before looking for a branching point, yields that "poisoning" one price (through overflow) causes each insertion to revert.

Once a tree has been poised in the above manner that matching pair cannot be used again.

Recommendations: We discussed several possible solutions, such as moving the check to be on the parent node and not the child node.

Customer's response: The price-subtree invariant checks have been moved to constrain the sum-bounds of the correct node within the `insert_order` traversal loop.

An investigation into the git commit history revealed that the misplaced check was present in the earliest versions of the tree. This was likely due to the nature of earlier designs of the tree

which were of greater code complexity, especially with regards to the additions performed on the node whose sums were being updated.

Medium Severity Issues

L-01 Bypassing codesize() check with a malicious contract could cause reverts and loss for Collector batches

Severity:**Low**	Impact: **Low**	Likelihood: **Medium**
Files: Exchange.yul#L742	Status: Acknowledged, collection mechanism would be redesigned	

Description: Any order with fees and a native token had a mitigation check that the user isn't a contract. This was done to prevent a situation where a contract would refuse a claimed order, causing its collector to lose the gas cost for nothing in return (instead of the expected fee).
The said check is done via codesize(), which could be bypassed by a contract during its construction time.
This could impact any "large" collector, even if it's trying to collect other trading pairs, a malicious user can add them as their collector and use a malicious trading pair in order to cause a revert to a whole batch.

Recommendations: We discussed several possible solutions, such as documenting this case so the collector could do a check off chain (or on chain), or possibly adding functionality for refused/failed claims (such as an external vault).

Customer's response: In the case that a courier (collector) is required to deliver native currency to a user, there is a possibility of the `CALL` opcode reverting or consuming an exorbitant amount of gas.

While not a perfect mitigation, the check in question decreased the probability of couriers encountering such a case. When implementing this check, we were fully aware of the ability for contract constructors to bypass this check as well as its incompatibility with ERC-4337.

After much discussion, we have decided to remove this check and push the responsibility of gas and reversion checks onto the couriers themselves. Documentation will be added to emphasize the importance of off-chain checks for couriers' batch deliveries.

L-02 Collector won't receive fee due to revert on a claim for order caused by token reverts

Severity: **Low**	Impact: **Low**	Likelihood: **Medium**
Files: Exchange.yul#L1679	Status: Acknowledged, collection mechanism would be redesigned	

Description: A similar idea to M-01 - in which an intentional but a revert during claim() would cause an entire batch to revert and gas waste (and fees denied) for the collector.
In this case, a malicious user could trigger a revert due to token's behavior: Which could happen due to blacklisting of the user, using an ERC777 token and a malicious callback, or using a malicious token altogether.
As in M-01, even though the trading pairs that are "susceptible" to this would be rare, a malicious user could intentionally pick a big collector and try to get added to big batches, and deny them.

Recommendations: We discussed several possible solutions, such as documenting this case so the collector could do a check off chain (or on chain), or possibly adding functionality for refused/failed claims (such as an external vault).

Customer's response: As mentioned previously, documentation will be added to emphasize the importance of off-chain checks before submitting a batch claim. While internal accounting could be used as a fallback in the case of a failed ERC20 transfer, we are currently hesitant to implement this as a solution due to its perceived fragility and difficulty of testing.

Low Severity Issues

L-03 Rounding when matching can allow line cutting

Severity: **Low**	Impact: **Low**	Likelihood: **Low**

Description: During the matching process when encountering a final leaf that might need to be split in order to fully match. The value to be reduced is rounded down for BID up for ASK. This means you can have the same effective ration as another seller, but for a better price. Less impactful but you can end up in a situation where a user could have had a better match by scanning the tree for rounding error orders that had their total_value decreased.

Recommendations: We discussed several possible solutions, showing the best price for users and possible solutions to maintain the ratio between quantity and value.

Customer's response: We conclude that this type of line-cutting would be more prevalent in a system with lower price precision. To some degree, the rounding scheme in our system is necessary as a consequence of its high price precision. Furthermore, as a result of this high precision, we expect the number of orders at a single price to be less than that of a lower precision system.

As such, we do not consider this issue consequential enough to merit any contract-level changes. However, the issue can be partially mitigated client-side by always picking the best price relative to the rounding properties of an order's value.

Informational Severity Issues

I-01. The "*fallback*" method is actually *multiview(bytes[])*.

Description: the fallback function is actually a function named multiview(bytes[]). We assume the fallback is used to optimize gas usage, but this function was not documented in the Exchange.sol interface.

Recommendation: Document this functionality as well; Add it to the Exchange.sol interface

Customer's response: Documentation and ABI declarations will be added.

I-02. The price decode is not always using the strict form.

Note: currently safe.
Description: This is mainly a codestyle / documentation note - there could be several representations of a specific price with the floating point encoding. Currently all user input is always checked against the strict method, and in other cases the values which are loaded and decoded are only derived from that (i.e. there's no way to exploit that, but it's a dangerous pitfall).

Recommendation: Future versions of the code should be careful with this nuance, possibly document near each decode why it is safe, or in the dev's docs

Customer's response: Code comments will be added explaining the rationale for using the non-strict decoding functions.

I-03. Some multiplications don't use the safemul.

Note: currently safe
Description: This is mainly a codestyle / documentation note - There are several cases where values are multiplied and said values have constraints that prevent them from being able to overflow.
Still, future versions or updates to the code should keep this in mind, so it might be valuable to document those.

The calculation of the value that is fulfilled from a leaf.
Any place that does not rely on the code but on the integrity of the storage is dangerous.

Recommendation: Possibly document near each mul why it is safe, or doc the constraints to preserve in the dev's docs

Customer's response: Code comments will be added explaining the bounds of each multiplication and the impossibility of overflows in such cases.

I-04. The 24h collector lock mitigation is counterproductive.

Description: The current mitigation gives a collector that was set a period of 24h where it's guaranteed that it won't be changed (and thus their efforts to claim won't go unpaid). If such a trust is needed between the user and the collector, it's counterproductive for the user to keep setting the collector every day if the entire point of a collector is to reduce the amount of interactions needed from the user.

Recommendation: A different approach such as the one that you're considering for per-order collector could work. Another idea could be using permits signed off-chain.

Customer's response: The courier (collector) system has been rewritten as a per-order system instead. This removes the need for any kind of global storage record tied to a user's account, and subsequently removes the need for a 24-hour lock.

I-05. Some functions are commented as "internal use only" but are public functions.

Description: As the title suggests. There's no way to exploit this, but if those are just debug functions then we recommend to either remove them from the production deployment, or update the comment/docs accordingly

Customer's response: The functions marked as "internal use only" fall into two categories:

1. View functions which do not follow the conventional EVM ABI encoding and are meant as a more optimal means of retrieving data from the contract due to their compact output.

2. View functions meant to be used with the `multiview` function. Usage in this manner reduces round trips for RPC servers which do not support batched JSON-RPC calls. An example includes retrieving one's ERC20 balance for an arbitrary token. Such calls could be utilized by a frontend to efficiently retrieve UI information for the user.

These functions are not meant for debugging and will be included in the final production deployment of the contract. They will be more explicitly documented in case users chose to utilize them for a UI frontend.

I-06. _replace() function lacks a pair_id check.

Description: Codestyle note - the _replace() function removes an order and inserts another. There are some checks at the beginning of the function, but pair_id is not checked until later on in the _remove() function.
Maybe it is done for gas reasons, but we recommend to either add a comment about that or add that check.

Customer's response: While not explicitly checked, `pair_id` is implicitly checked by the first `require` call following the retrieval of the order storage record. As such, an earlier explicit `pair_id` check is deliberately absent.

Code comments can be added to explain this.

I-07. codestyle suggestion - replace raw sstore() calls with dedicated functions/macros.

Description: There are several different data structures in the storage space, so it might be better to use a specific function that writes a particular data type to restrict the possibility of overwrites and errors.

Customer's response: While we find this to be a reasonable suggestion, we have opted to keep the `sstore` calls as they are. A refactor of this magnitude at this point is more likely to introduce bugs than to avoid them.

For future endeavors, an explicit form of "key syntax" and subsequent handling of keys/sstores could be implemented in the Jul language at the compiler level.

I-08. codestyle suggestion - Macros and naming conventions.

Description: This is about code duplication which makes the maintenance and audit more difficult.

Recommendation: Where there is similar logic (like removing nodes from the tree or updating the tree) it is safer to have common functions that do that in one place. Same for macros define them in one place (without local variables names) and return the changed values so it is clear what has been updated in the macro.

Customer's response: Certain functionality, like child promotion, could be de-duplicated in both `match_orders` and `remove_order`. However, we find that the two functions are sufficiently different to the point where code de-duplication may actually introduce some confusion.

With regards to macros, we hold the opinion that macros in programming languages serve three primary purposes:

1. Forceful inlining of code.

2. Simplification of otherwise cumbersome/repetitive syntax.

3. Access to local scope in order to minimize repetition and visual pollution.

For low-level languages, we consider these features to be especially important. In particular, the third feature is a widely used strategy leveraged in many codebases.

certora

Formal Verification

Verification Notations

Formally Verified	The rule is verified for every state of the contract(s), under the assumptions of the scope/requirements in the rule.
Formally Verified After Fix	The rule was violated due to an issue in the code and was successfully verified after fixing the issue
Violated	A counter-example exists that violates one of the assertions of the rule.

Formal Verification Properties

Exchange

Module General Assumptions

We assumed that loops are iterated through at most two times and for every trading pair, the number of decimals of corresponding tokens are either 12 or 18. We believe that if any bugs were present, they would already manifest under these conditions.

Contract Properties

P-01. Reentrancy Lock Liveness (Won't get stuck)

Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions	Link to rule report
mutexStaysUnlocked	Verified	*It was proved that there couldn't be a state where the reentrancy lock (mutex) would stay locked after a function call. (i.e. it's always released).*		*{https://prover.certora.com/output/6893/2217a8bc481a436f95c5578b074f71b4/?anonymousKey=5892b12c5e460098c834021d02bb69c3e9f173a2}*

P-02. integrity of addPair()

Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions	Link to rule report

integrityOfAddPair	Verified	*A trading pair is indeed added after addPair is called.*		*{https://prover.certora.com/output/6893/7f7554fb874848e280ec5a7096dae000/?anonymousKey=354270dd95f8214bab53ee7e55385bd910354e2d}*

P-03. integrity of cancel()

Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions	Link to rule report
integrityOfCancel	Verified	*An order is indeed removed after a cancel() call. (implies can't cancel twice, can't match against it, etc).*		*{https://prover.certora.com/output/6893/beff6ebaf23e4d7d95617a626a2e0a46/?anonymousKey=c64d8421c4360b7a5ba83c2c55ec32c01128af3f}*

P-04. integrity of order()

Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions	Link to rule report
integrityOfOrder	Verified	*An order is indeed inserted into the book*		*{https://prover.certora.com/output/31688/8*

				d4387f16231412ebec46dc38394b647/?anonymousKey=1a25e8dcc8193d68264f8ca1eee119e05f1c48c1}

P-05. Cancel() doesn't add the orders to other trees

Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions	Link to rule report
cancelDoesntAddToOtherSide	Verified	*A cancel order doesn't affect the tree of the other side (e.g. canceling an order on the ASK side won't affect an order in the BID side)*	*The other side is empty. I.e., we prove that it stays empty.*	*{https://prover.certora.com/output/6893/e10887c412e946ea941f1cdf71c1e74f/?anonymousKey=86b053b38b73f66a38fe219fe742331969de5cdd}*
cancelDoesntAddToOtherTIDs	Verified	*The same as the above, but proved that it doesn't add against any other tradingPair as well*		*{https://prover.certora.com/output/6893/19c5848c151c460f9544e85b0de932c4/?anonymousKey=499b5a3121ccdda21c7911dc1a16cde10c65c9a5}*

P-06. Integrity of get_leaf_path + get_side

Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions	Link to rule report
sideFromPathIntegrity	Verified	*For all TID, SIDE, PRICE:* *If PATH==get_leaf_path(TID, SIDE, PRICE)* *then* *get_side(PATH) == SIDE*		*{https://prover.certora.com/output/6893/3ec0c583a6d24b3f91879ea1fef6c7aa/?anonymousKey=70e5cbd0d301cd41398daa53d412eec071e07a73}*

P-07. Order price encoding (floating points) are monotonically increasing to their encoded value, given the strict encoding

Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions	Link to rule report
strictDecodeIsMonotonic	Verified	*We proved that given a number A that is larger than B (A>B), then also their encoded (bit) values correspond to this inequation:* *encoded(A) > encoded(B)* *This only holds assuming the strict encoding patterns.* *This property is an important assumption in the tree's matching algorithm, where the price leafs are sorted by their bitwise path which*		*{https://prover.certora.com/output/6893/7c8aac3a6d5845e9805352604bdfa1d8/?anonymousKey=b01746884a8a380e01fe0309e680eacd336c9b12}*

		represents an encoded price.		
encodeIsMonotonic	Verified	*As above*		*As above*

Conclusion

Wrap codebase was written with a lot of thought put into writing secure code. The code has sufficient tests and the audit showed no significant bugs through manual audit as well as using the Prover.
With that in mind, the code complexity would make maintaining, and auditing the code a hard effort. A significant number of redundant input checks have been removed to save on gas, but the code lacks relevant comments. It is likely that future code updates might change those assumptions.

Recommendations

- Simplify the code by merging code duplications into functions or macros.
    - Removing a node and reconnecting its parent to its child for example.
- Don’t call out of scope variables in macros.
- Document and comment input checks that were removed for gas savings.
- Simplify storage configuration by having a single document with all the relevant segments.
    - Such as order_array and pairIDs
- Separate public methods and API’s from private internal implementation.
- Add documentation to the Jul language.

Disclaimer

The Certora Prover takes a contract and a specification as input and formally proves that the contract satisfies the specification in all scenarios. Notably, the guarantees of the Certora Prover are scoped to the provided specification and the Certora Prover does not check any cases not covered by the specification.

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.

Appendix 1 : Architecture

This document holds a high level overview of the architecture of warp, the full technical explanations as well as design choices can be found in under book.md.

Memory

Warp's architecture has split the full 2^256 possible storage locations into various slots, defined by a unique key that is the first byte of the address. In addition to that Jul (warp's implantation of Yul) also has some addresses that have special meaning to them.

Segment Name	Hexadecimal Prefix	Explanation
PAIR_PREFIX	0x01	Stores the information for that matching pair
LHS_PREFIX	0x02	Stores the left-hand side address
RHS_PREFIX	0x03	Stores the right-hand side address
ORDER_PREFIX	0x04	Stores the orders
ORDER_ARRAY	0x05	Stores the information of the orders (UI)
COLLECTOR_PREFIX	0x06	Stores the addresses of the collectors
CTR_PREFIX	0xff	Stores the counter
NODE_PREFIX	0xfe	Stores the matching tree

PRICELOG_PREFIX	0xf1	Stores the price log information (UI)
CANDLE_PREFIX	0xf0	Stores candle information (UI)

In addition to these segments there is also the additional special location, used by JUL for mutex.

Segment Name	Location	Explanation
Mutex	0x0	Stores the mutex for reentrancy guard.

pairID

After initializing a pair for matching, the configurations for that pair are hashed then are stored in the correct segment under that same hash. Any future memory access for operations on that pair uses that same hash to further split each segment. This in theory helps against corruption from other matching pairs.
For example if you want to access a given order you first append the order prefix, then the pairID then the order.
Note: collector addresses are unique in that they are not tied to one specific pair. pairID is the only value that is hashed, each other part of the address used as plain text meaning.

Matching Tree

The matching tree is the most complicated data structure in Warp, the tree allows insertion of orders in o(logn) and matching of o(logn).
Bid trees and Ask trees are stored in the storage and are tied to a specific pairID. The difference between bid and ask in terms of memory location is side byte that is part of the address and further split the memory.
So a matching tree is stored in the matching tree segment, in the relevant side byte in that segment, then in the relevant pairID.
Each node in that tree is stored within the same memory subsegment, and the exact location can be derived from two variables: path and depth.

It is defined that the root of each tree is in depth 0 and path 0.
When traversing the tree, each node can have one of three states.

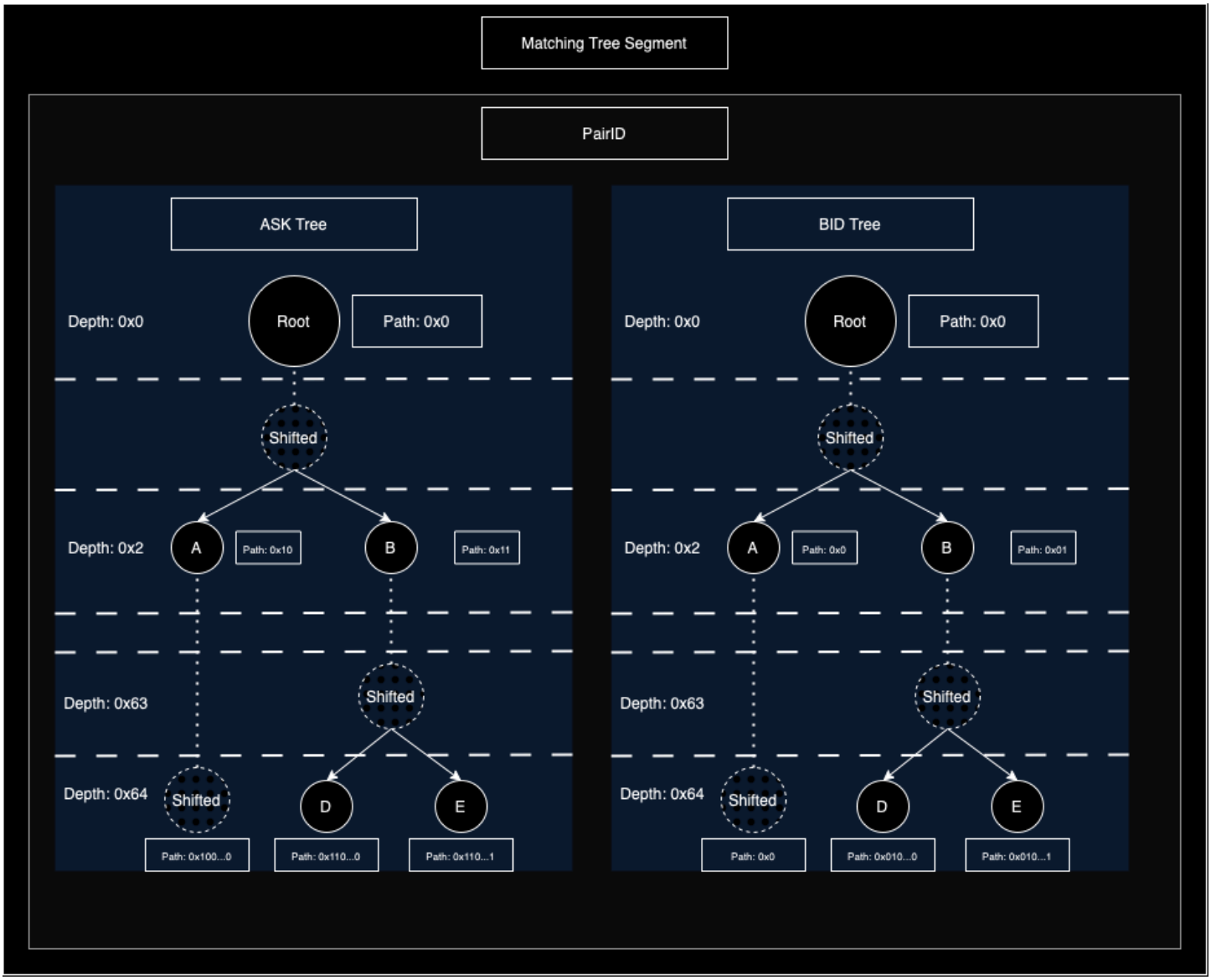
- Two successors
- One successor
- No successor (leaf)

Traversing rules:

1. **Two successors:**
    a. This is handled like a full binary tree. Each child is on the next layer and can be accessed by adding a 0 bit or a 1 bit to the path and increasing the depth by one to reach the next layer.
2. **One successor**
    a. There are a few assumptions on this scenario.
        i. The node (parent) has a brother node.
        ii. The successor node also has a brother node.
    b. This case uses the skip depth and skip bits parameters, using those parameters, the node is “shifted” to be right above the two successors, then traversing rule 1 is used.
    c. To “shift” the node we shift left by the skip depth, and add the skip bits.
3. **No successor (leaf)**
    a. If we have reached maximum depth, there are no successors and we can fetch the relevant order from the order segment of the storage.
    b. The leaf and the order have the same address but one is in the matching tree segment and the other is the orders segment.

Addresses:

- Each node path is derived from the prices of the leaves beneath it.
- The nodes are always traversed from largest price to smallest.
- On the ASK tree, because the traverse should be reversed, instead all the prices are reversed using a simple XOR (or one complementary)



Matching

The tree is traversed based on the traversing rules above.

1. If a sub-tree maximum price, is higher than the requested price, (notice its reversed to "lower" in the ASK tree) then:
    - If this subtree is not big enough to satisfy the order. The entire tree is matched, and matching continues.
    - If this sub-tree is too big, find a smaller sub-tree.
2. If this sub-tree has **some** leaves that have satisfactory price:
    - Continue to find the nodes with good prices.

3. If this sub-tree **minimum** price is too low. That means that every leaf within this tree would not satisfy. Therefore finish the matching process.

Compression

In order to save gas, price numbers have been compressed to 23 bit numbers. This compression is done like float, where there is an exponent representation and a remainder.
The 24th bit is assumed to be reserved for the “sign” or side. To know if the price is in the ASK tree or BID tree.
The base that is used for the compression is base10. Base 4 would yield greater precision for general prices. However for the most likely price numbers, base10 gives better representation.

Based on the precision of the coin, a somewhat different compression method is used. However due to the use of strict (where each number is decompressed then compressed again to make sure its a valid number) no problems were found with this approach.