Continuing the discussion from Trustless, ZK-proof based Total Value Locked oracle:

I'm including context from the previous discussion (linked above) for completeness and convenience.

Summary:

Lido Oracle contract is trustful and relies on trusted third-parties and quorum mechanism to maintain its state. It is possible to make the contract trustless, allowing any third party to adjust the contract state, while mathematically securing the correctness and validity of the change via Zero Knowledge proofs.

Main idea at a glance

Lido oracle contract is trustful - it relies on external oracle(s) to honestly calculate the total ethereum value committed by Lido validators (aka TLV), and uses quorum mechanism and a list of trusted members to protect from malicious actors.

This leads to three consequences:

- Security: A dedicated and resourceful attacker can work towards acquiring control of the majority of oracle members - and compromising the oracle contract when it is achieved[1]

.

- Cost: Contract requires a considerable amount of expensive storage read/write operations to manage members, check if reports are coming from a trusted source, and keep track of reports while quorum is being accumulated.
- Scalability: With the network growing. the cost of quorum calculation grows linearly (O(N)) with the number of trusted oracle members

A trustless, ZK-proof secured approach can address these shortcomings in the following way:

- Security: With proper construction, zk-proof can ensure that only honest calculations produce input that would pass validation. This will make quorum and membership management unnecessary - any input that passes validation can be trusted to be coming from an accurate and honest calculation.
- Cost: As it will be shown in the "Implementation" section, input validation can be limited to a small number (3-4) checks against keccak hashes (stored in the contract or provided by the Execution Layer) + Execution Layer contract invocation to confirm ZK-proof validity.
- Scalability: since all honest and accurate calculations should produce the same outcome (there's only way to sum all staked balances), the contract can reject "duplicate" reports from multiple parties (e.g. based on eth epoch), essentially making it a constant (O(1)) and not grow with the network size/oracle operators.

Proof Construction

Lido Oracle contract essentially manages a single number - the TLV committed by Lido validators. The "smallest" payload we can pass to the contract to update the state is similarly a single number - new TLV value. However, with only this, the contract cannot verify the validity of the calculation in any way, except trusting the source (msg.sender). So we need to extend the payload to include data that the contract can independently validate and compare against something it already knows or trusts.

Essentially, computing Lido validators' TLV involves three pieces that can be independently checked by the contract:

- All Consensus Layer validators addresses and balances, as reported by BeaconState
- Lido validator addresses, as seen on the Lido Node Operators Registry
- A computation "program" itself: take the two above as an input, find balances of all Lido validators, sum the balances.

As such, an honest trustless oracle should perform the following operations:

- Obtain validators and balances from Consensus Layer BeaconState (or other means with identical result)
- Obtain Lido validators from a Lido Node Operators Registry (or other means with identical result - e.g. lido-sdk)
- Correctly compute the TLV according to the "program".
- Produce a ZK-proof of the computation, and pass TLV value and ZK-proof to the oracle contract

This opens the following attack vectors for a malicious actor:

1. Using an arbitrary/purposefully constructed list of public keys as list of Lido validators keys.
2. Using an arbitrary/purposefully constructed list of Consensus Layer validators and balances.
3. Using a different "program code" (e.g. using any arbitrary number, "fair random", etc.).
4. Honestly performing a calculation, and then replacing the result with an arbitrary value in a call to the contract, while still passing ZK-proof for the original result of the computation.

To protect from such attacks, the payload to "Modify TLV" contract call should contain proofs that the valid inputs were passed to a correct "computation program", and that the TLV value passed to the contract indeed comes from the output of that "computation program".

Trustless oracle contract should perform the following checks:

- "Program code" used to calculate the value corresponds to a valid TLV computation.
- At this stage, the contract ensures that a third party performs the correct computation, but not necessarily on the correct inputs - prevents attack vector 3.
- At this stage, the contract ensures that a third party performs the correct computation, but not necessarily on the correct inputs - prevents attack vector 3.
- Check that the input to the "program" is correct: actual Lido validators' keys and balances are used
- At this stage, the contract ensures that the correct inputs were passed to a correct computation, so the output of the computation is trustworthy - prevents attack vectors 1 and 2.
- At this stage, the contract ensures that the correct inputs were passed to a correct computation, so the output of the computation is trustworthy - prevents attack vectors 1 and 2.
- Check that the output of the "program" was used as the contract payload
- Through this check, the contract defends from malicious attackers to perform a trustworthy computation, and then replacing the output with arbitrary value - prevents attack vector 4.
- Through this check, the contract defends from malicious attackers to perform a trustworthy computation, and then replacing the output with arbitrary value - prevents attack vector 4.
- Verify validity of the ZK-proof produced by the "program":
- This ensures that the computation was indeed performed using the "program code" and inputs above, and can be cryptographically confirmed to be correct - secures the above checks via verifiable Zero Knowledge proof.
- This ensures that the computation was indeed performed using the "program code" and inputs above, and can be cryptographically confirmed to be correct - secures the above checks via verifiable Zero Knowledge proof.

Solution Architecture

Until this point I've intentionally put "program" in quotes - to emphasize that the argument above does not imply any particular implementation mechanism; as long as that mechanism can perform the necessary computation and generate a computation's ZK-proof, any mechanism will do. In this section, I'll focus on one particular implementation mechanism that supports both; however, other mechanisms could exist.

TL;DR:

- Trustless oracle obtain BeaconChain (or at least validator keys and balances) and Lido validators' keys, pass both to a Cairo program to compute 3 items: (1) Merkle tree root of BeaconChain, (2) Merkle tree root of Lido Validators and (3) total value locked; sends the program and input to SHARP; passes program output (Merkle Tree roots and TLV) to the contract.
- Trustless oracle contract compares Merkle trees to stored/independently computed values, generates a Fact ID from previously stored trusted program hash and program output, verifies the fact ID via STARK Fact Registry contract (Execution Layer contract), and if all checks pass updates the TLV value stored on the contract.
- The trust relationships are not completely removed, but shifted - instead of trusting the oracle(s), the contract should trust the StarkWare prover, verifier and fact registry contract. There is no trust between the contract and oracles though.

Overview

The first challenge is to make the oracle produce a verifiable ZK proof of the computation. At the moment, one option to do so is to utilize StarkWare's Cairo programming language and verifier (SHARP).

The best explanation of how it works is this quote fromCairo for Blockchain Developers post:

[With Cairo] you write your complex logic in Cairo, get it proved off-chain, and once that proof is validated on-chain, your smart contract application can use the result trustlessly – as if it executed that complex logic onchain, because that's what the proof asserts.

Generating ZK proof

Cairo ZK proof involves four parties: "computer", prover, verifier, and "user":

- Computer - performs a computation.
- Prover - proves that a computation was performed (produces a provable computation trace).
- Verifier - verifies the correctness of the proof.
- User - uses the result of the computation, but only if it is verified and trustworthy.

Computer and prover live off-chain, while verifier and user are on-chain. In our case, a trustless oracle becomes a "computer" and prover, and a trustless contract is a user; verifier role is fulfilled by the SHARP and other StarkWare infrastructure.

The trustless oracle essentially becomes a wrapper around a Cairo program: gathering inputs, invoking the program, submitting it to the SHARP prover, and sending the program output and ZK proof to the oracle contract. The computation of the TLV, as well as necessary supporting evidence happens in the Cairo program. Program execution trace (automatically generated by the Cairo runtime) is then submitted to SHARP, to prove and verify the computation and generate a Fact - testament that the computation was performed and verified to be correct.

Verifying ZK proof

In Cairo/StarkNet, ZK proof is represented by a Fact - a testament that a correct and sound computation was performed on some input. The Fact is stored on-chain, and can be checked by an on-chain contract by passing a Fact ID to the Fact Registry contract.

Verification happens simply via a call to theFact Registry contract (etherscan) isValid(FactID)

method - if it returns true, the computation is confirmed to be correct and trustworthy. Due to how the Fact ID is constructed (spoiler: hash of the program code and output), by verifying a Fact in the contract we can ensure two things:

- That a computation was performed with a correct Cairo program (attack vector 3)
- That the output of the program was passed to the contract without tampering or modifying (attack vector 4)

The contract would need to store a hash of the correct Cairo program

and take the entire program output

as part of the "Modify TLV" method input.

Verifying inputs

As mentioned in the previous section, verifying the fact only confirms that a certain computation was performed on some input

, to produce a certain result. Ensuring that a correct input was passed to the program requires performing some additional steps.

One way to achieve this is to include some information about the input that can be confirmed by the prover/verifier and checked by the contract. A straightforward way to do it is to include the inputs into the output - and let the contract check that the inputs indeed match the expected values. There's a challenge though - the inputs consist of public keys of all Consensus Layer validators, their balances, and Lido validator keys. Passing these data around would already be prohibitively expensive - there are >150K validators at the time of writing, so listing their keys alone would take 6Mb+ (150K+ validators x 40 bytes per validator).

However, since we only need to validate that the input was what we expect, it is enough to compute a cryptographically secure proof that a valid data was used - a widely used approach to do so is constructing a Merkle tree on the data, and passing around/comparing Merle tree root(s) (aka MTR, for brevity).

To make this happen, a Cairo program should, in addition to the TLV value, provide two additional “commitments”:

- MTR of Lido Validator keys (attack vector 1)
- MTR of Consensus Layer validator keys and balances (attack vector 2)

The contract then should obtain/compute MTRs

of Consensus Layer validators keys and balances, and Lido validator keys, and then compare these MTRs with the ones supplied in the “modify TLV” call

.

Note:

both operations are likely to be too expensive to implement naively in the contract. However, MTR of Lido validators can be trivially added to the Lido Node Operators Registry contract (or new contract), and validator keys+balances MTR can be “expanded” to be an MTR of a BeaconState that will become available to all L1 contracts when/ifEIP-4788 is implemented, or (worst case) be maintained by a separate mechanism (e.g. a first-party Lido oracle)

Summary

Trustless Oracle:

- Obtain Consensus Layer validator keys and balances
- Obtain Lido validator keys
- Execute Cairo program to compute MTRs of the above inputs + Lido TLV
- Submit the program execution trace to SHARP
- Send entire program output to a trustless oracle contract.

Trustless Oracle contract:

- Initialization:
- store a hash of correct Cairo program, address of a Fact Registry contract, and current TLV
- sets up additional access control/deployment/management data, as necessary
- store a hash of correct Cairo program, address of a Fact Registry contract, and current TLV
- sets up additional access control/deployment/management data, as necessary
- GetTLV method:
- simply return the stored TLV
- simply return the stored TLV
- ModifyTLV(uint256

programOutput): * Verify ZK proof: * Construct FactID: keccak(program_hash, programOutput)

- Verify Fact with a Fact Registry
- Construct FactID: keccak(program_hash, programOutput)
- Verify Fact with a Fact Registry
- Check Consensus Layer validators’ keys+balances:
- reconstruct MTR from program output
- obtain/compute BeaconChain MTR (directly or via separate contract)
- reconstruct MTR from program output
- obtain/compute BeaconChain MTR (directly or via separate contract)
- Check Lido validators keys:

- reconstruct MTR from program output
- obtain/compute Lido validator keys MTR (directly or via separate contract)
- reconstruct MTR from program output
- obtain/compute Lido validator keys MTR (directly or via separate contract)
- if all the above checks pass, replace the stored TLV with a new value
- Verify ZK proof:
- Construct FactID: keccak(program_hash, programOutput)
- Verify Fact with a Fact Registry
- Construct FactID: keccak(program_hash, programOutput)
- Verify Fact with a Fact Registry
- Check Consensus Layer validators' keys+balances:
- reconstruct MTR from program output
- obtain/compute BeaconChain MTR (directly or via separate contract)
- reconstruct MTR from program output
- obtain/compute BeaconChain MTR (directly or via separate contract)
- Check Lido validators keys:
- reconstruct MTR from program output
- obtain/compute Lido validator keys MTR (directly or via separate contract)
- reconstruct MTR from program output
- obtain/compute Lido validator keys MTR (directly or via separate contract)
- if all the above checks pass, replace the stored TLV with a new value

Practically, barring the access control, deployment, governance, modification, etc. machinery, the resulting contract would not be much more complex than a "toy" contract used in one of the Cairo tutorials - essentially just a few lines of code.

Implementation

A github repo is worth a thousand (or more) words, so check out GitHub - e-kolpakov/cairo-balance - it contains Cairo program, example contract and oracle "wiring" boilerplate (rest of the repo, and oracle.py in particular). Readme contains instructions on how to set up and interact with it, as well as some implementation details, assumptions and specifics.

Assumptions

The solution is based on a few assumptions on how the BeaconState hash will be calculated and exposed to Execution Layer contracts. Full details are listed in the corresponding readme section, but in short:

1. Assumption 1: BeaconState merkle tree hash will be available on chain (simply put, eip-4788 is implemented one way or the other)
2. If this assumption does not hold, the workaround would be to maintain the BeaconState hash separately, via a first-party contract/off-chain solution.
3. In fact this is what the example implementation is doing.
4. If this assumption does not hold, the workaround would be to maintain the BeaconState hash separately, via a first-party contract/off-chain solution.
5. In fact this is what the example implementation is doing.
6. Assumption 2: the algorithm to calculate merkle tree root exactly matches the one in the Beacon Deposit Contract - "progressive merkle tree" with 32 levels.

7. If this assumption does not hold, Cairo program will need to be updated to use an actual algorithm (merkle_tree.cairo)
8. If this assumption does not hold, Cairo program will need to be updated to use an actual algorithm (merkle_tree.cairo)
9. Assumption 3: Consensus Layer public keys are split into two 32-byte merkle tree leaves.
10. This is more of a shortcut, rather than an assumption - most likely it won't hold in reality and public keys will live in a single 64-byte leaf.
11. This can be supported in Cairo, but adds additional complexity that drives attention away from the important parts of the solution.
12. This is more of a shortcut, rather than an assumption - most likely it won't hold in reality and public keys will live in a single 64-byte leaf.
13. This can be supported in Cairo, but adds additional complexity that drives attention away from the important parts of the solution.

Key challenges

Computing Merkle Tree roots

The most non-trivial part of the oracle is making sure that MTRs computed by the Cairo program and oracle contract would match. There are two challenges with this:

- Cairo uses Pedersen hash by default, while Ethereum uses keccak. Keccak is available as a module in Cairo, but requires some non-trivial manipulations to make sure that the input to keccak calls match between Cairo and Ethereum (spoiler: need to align padding and breaking down into merkle tree leaves: Consensus Layer public keys being 40 bytes and keccak operating on 32 byte chunks)
- Calculating MTR on large inputs on the fly is expensive, so most solutions involving MTR (such as DepositContract) employ some optimizations allowing to "spread out" the calculation (aka "progressive merkle tree"). The optimizations might introduce non-trivial changes to the structure of the Merkle tree - e.g. DepositContract always computes an MTR of a 32-layer deep Merkle Tree, with actual values being leafs on the last layer (+ zeroes to fill the last layer to full).

Cairo specifics

Cairo language itself posed some unique challenges - in short, it feels a lot more like "assembly with syntactic sugar" rather than a high-level programming language (this is even stated in one of the official tutorials/docs). I'll omit excessive details (PM me if you're interested in more details), but at a high level the language features that left the most visible marks on the result are:

- No loop syntax - the loops can be achieved via recursion, or via explicit jumps.
- Peculiar memory model (non-deterministic read-only), references and erasure
- all "variables" are essentially address "offsets" from the memory pointer (aka ap

), and calling into a function (or builtin) moves the ap

, but not the offsets, revoking the references.

- writes are also assertions - if the "variable" was already written to, an equality check is performed instead.
- all "variables" are essentially address "offsets" from the memory pointer (aka ap

), and calling into a function (or builtin) moves the ap

, but not the offsets, revoking the references.

- writes are also assertions - if the "variable" was already written to, an equality check is performed instead.
- DictAccess - this library class "quacks like a dict", but is actually an append-only list, very similar to compacted topics in Kafka.
- Non-deterministic execution (aka "hints") and soundness - comprehensive explanation would require a deeper dive into the computation model, but in short, this feature allows computations "out-of-band" from the perspective of the Cairo program, magically arriving at a result:
- Example: find an element by key in a list in O(1) - the element index can be found in the hint and assigned to a Cairo variable (this is actually used in the oracle Cairo code))

- This might provide an opening for a malicious party to compromise soundness of the proof, so additional steps need to be made to ensure soundness (e.g. check if the found element is indeed in the list, and has the correct key)
- Example: find an element by key in a list in O(1) - the element index can be found in the hint and assigned to a Cairo variable (this is actually used in the oracle Cairo code))
- This might provide an opening for a malicious party to compromise soundness of the proof, so additional steps need to be made to ensure soundness (e.g. check if the found element is indeed in the list, and has the correct key)

It is worth noting that Cairo evolves rapidly, and some of the challenges above are already "marked for improvement" in the recently announced Cairo 1.0 release.

Conclusion

Summary:

TVL oracle contract can be made a lot simpler and cheaper - complete contract example is~100 lines long - by moving the computations off-chain, and securing them via ZK-based solution. Cairo and StarkWare provide one feasible solution to achieve it with production quality and reliability, but other solutions can exist. Cairo allows building arbitrary logic, and transparently (to the developer) converting it to a STARK proof. Biggest challenge is implementing and invoking merkle tree calculation in Cairo in the exact same way as it is done in Ethereum/oracle.

The most challenging part of the solution was to make merkle trees calculated in Cairo and in Ethereum/Oracle match - this can only happen if merkle tree algorithm and inputs are exactly the same between Cairo and Ethereum

, including byteorder, padding, splitting (or not) values into multiple leaves, order of the values, etc.

In addition, it is worth mentioning that in this case, the "knowledge" verified by the ZK-proof is actually publicly available. However, accessing and manipulating that data in the Execution Layer contract is almost prohibitively expensive. Essentially, we only need to ensure that the "correct" computation was performed over the "correct" data

(aka computational integrity) - without disclosing the data to the Execution Layer contract, but proving to it that we have it.

Cairo and StarkWare infrastructure provide a secure and (relatively) convenient way to do it, however similar results should be achievable using other solutions

(e.g. a smart contract on a different chain). Cairo has one additional advantage - the ability to implement arbitrary logic

(it is turing complete) and transparently turn it into a STARK proof to the developer.

1. Technically this is a 51% attack that even PoW blockchains do not try to address; however, at the moment there are only 5 trusted oracles (getOracleMembers()), so 51% attack essentially boils down to overtaking 3 entities.↩