Restaking: Principles and Applications

Course: COS 473 / ECE 473

Lab Materials: Week 10 - 4/8/2024

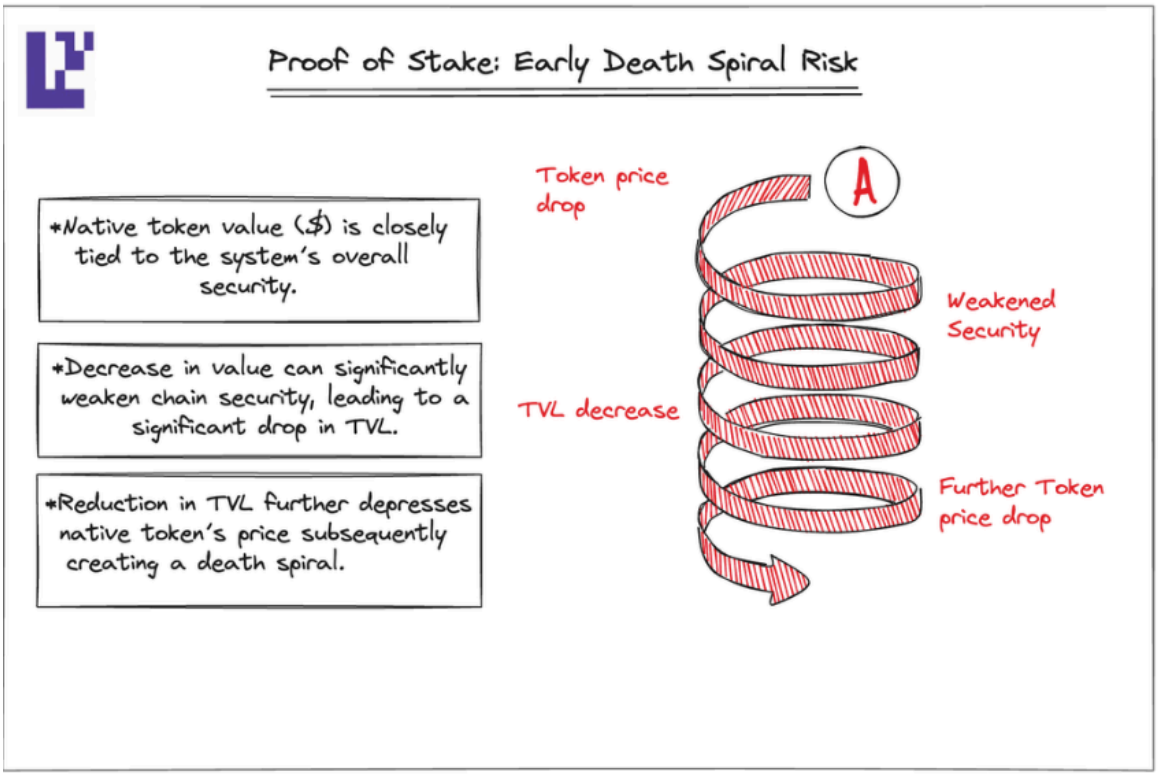
Email here if help needed:

Table of Contents

	Topic
1	Restaking Content Review
2	Creating and Deploying an AVS Middleware
3	Registering for Holesky Testnet and Mining Faucet
4	Data Availability Layers + Rollups Review
6	EigenDA AVS + Arbitrum Orbit Deploy
7	Putting It All Together
8	Submission
9	Extra Resources

Before EigenLayer

This section hopes to inspire the key issues that motivated the creation of EigenLayer. The cryptoeconomic security of a blockchain protocol is defined as leveraging both cryptographic tools and economic incentives in order to achieve information security objectives. A blockchain protocol relies on its cryptoeconomic security as a means of attracting consumers into its ecosystem: highly volatile, insecure tokens typically do not inspire confidence in consumers. It is easy to understand, then, that it is somewhat paradoxical for new, emerging blockchains to achieve high levels of cryptoeconomic security. See below:



If it is so difficult for new protocols to achieve cryptoeconomic security, why not integrate directly with Ethereum ecosystem?`

From EigenLayer blog:

`While Ethereum provides economic security for smart contract protocols, infrastructures like bridges or sequencers require their own economic security to enable a distributed network of nodes to reach consensus.`

Thus, it is important for new protocols to be able to define their own consensus algorithms, but especially in a Proof of Stake environment (as opposed to Proof of Work), it is difficult to find node operators that are willing to pay for stake in a new protocol's native, often volatile token. This must compete with benefits offered to operators for obtaining stake directly on Ethereum Network.

> After EigenLayer

Goal of EigenLayer: Goal: Creating a platform connecting stakers and infrastructure developers

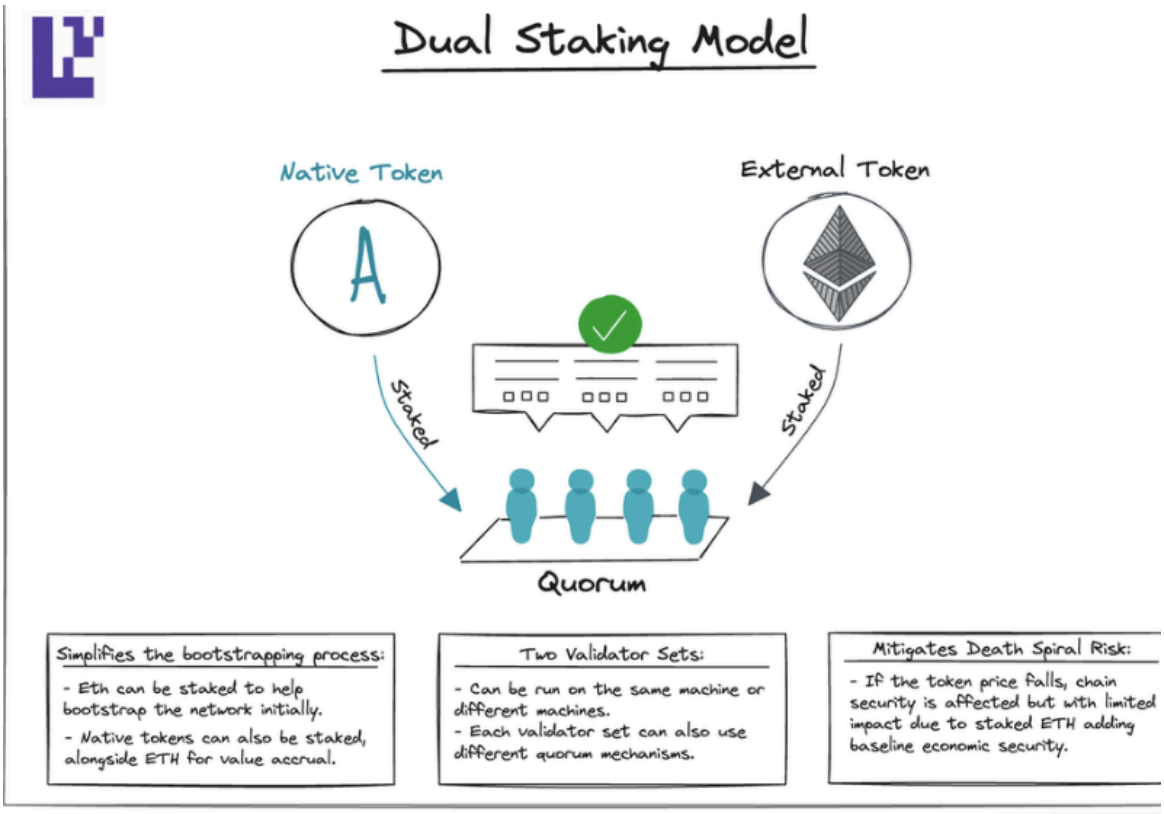
A reasonable analogy here is that of the relationship between states and countries. States leverage the enhanced economic output, security and ecosystem of their larger countries in exchange for conformity to certain rules and taxes. Similarly, think of Eigenlayer as a matching system between states (protocols, also known as AVS) and a country (Ethereum). Eigenlayer thus enables a pooled, strengthened cryptoeconomic security through the main Ethereum ecosystem, as opposed to resources being split across many different protocols all of reduced security.

Here's How it Works:

Let's say you want to deploy your own new protocol, i.e. your own "Chainlink" with unique consensus and governance mechanisms that enable some unique functionality. (You will deploy a toy AVS middleware below!) In order to deploy such a protocol, it must conform to some standardizations set by EigenLayer. See here: https://docs.eigenlayer.xyz/category/node-specification. Separately, operators determine which AVS's they would like to opt in to. Through EigenLayer in a PoS environment, these operators are allowed to stake ETH to operate that given AVS. From here, consumers are also allowed to stake ETH toward an AVS of their choice, with an operator of their choice. The result is an ecosystem where new protocols can leverage the existing ecosytem of Ethereum and its network of consumers/operators, while maintaining their own unique consensus mechanisms.

Note: Dual-Staking:

See EigenLayer blog here for more details: https://www.blog.eigenlayer.xyz/dual-staking/



Creating and Deploying an AVS Middleware

1. Clone the repo

```
# Recommended: clone in your PC's root directory
# otherwise may run into issues downloading all submodules due to
long file paths
cd ~
git clone https://github.com/Shivamkak19/
eigenlayer_middleware_starterkit
cd eigenlayer_middleware_starterkit
git submodule init
git submodule update --init --recursive
```

2. Note, above is forked from here. Check for sample solution:

```
git clone https://github.com/Layr-Labs/incredible-squaring-avs
```

3. Open Bash/PuTTY client. Download Dependencies

- Foundry (Rust Ethereum toolkit)

  ```
  curl -L https://foundry.paradigm.xyz | bash
  exit
  foundryup
  ```

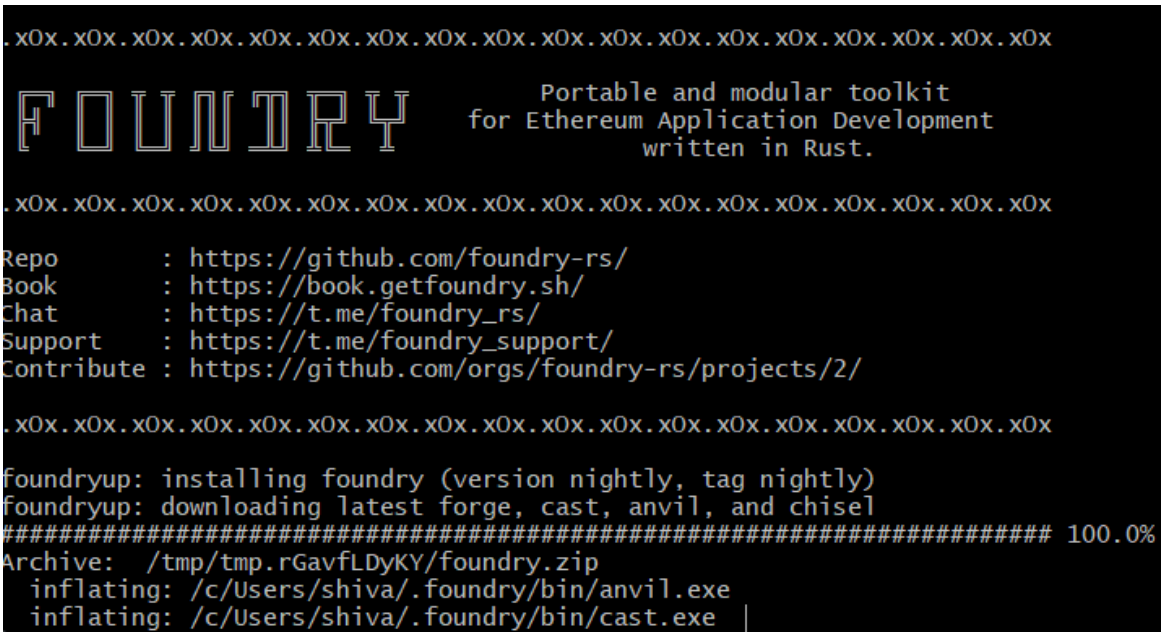
- log prettifier for Zap

  ```
  brew install maoueh/tap/zap-pretty
  go install github.com/maoueh/zap-pretty@latest
  ```

    - see here for troubleshooting with foundry: https://book.getfoundry.sh/getting-started/installation
    - Install go if needed here: https://go.dev/dl/
- Install docker (tool for running applications in a container): https://docs.docker.com/get-docker/
- After installing docker, build contracts:

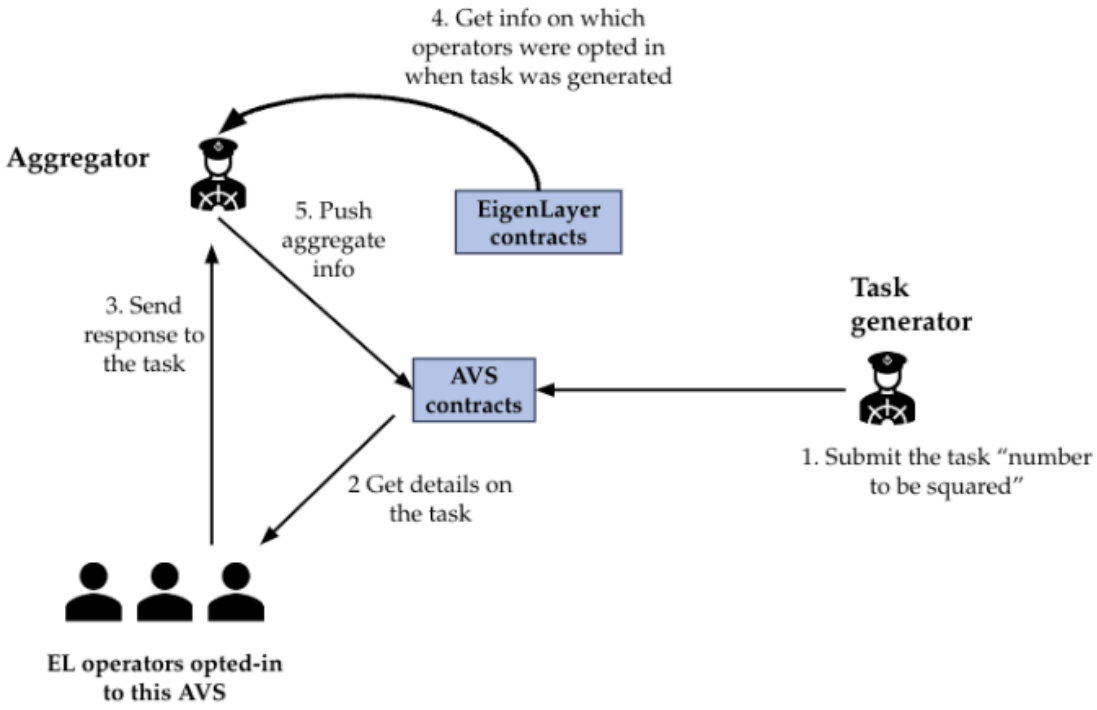
  ```
  make build-contracts
  ```

(Should see confirmation as below!)



Before proceeding further, it is important to understand the system design of an AVS. There are typically 3 separate parties involved: aggregators, operators, and generators. An AVS (Actively Validated Services) is essentially a blockchain protocol, aside from the central Ethereum protocol. Protocols are typically run on a decentralized network of nodes (think how Ethereum runs on a network of

separate node operators). Similarly, AVS operators are nodes that have elected to opt into a given AVS. Generators are the clients issuing instances of the task needed to be completed. Finally, Aggregators are algorithms that determine the aggregate response to deliver to the AVS contract, after having been sent all responses to tasks from node operators. See the diagram below for further clarification:



Activity Number #1

Take a close look at the Makefile below. Try to determine what Makefile commands may be used for each of the 3 tasks below. After you have given this some thought, the solutions are also attached below. They are also in the GitHub repo's README as well.

4. Start Operator (locally hosted Anvil chain)

```
make [...]
```

5. Start Aggregator

```
make [...]
```

6. Register Operator with EigenLayer and Opt-In to Participate in Sample AVS

```
make [...]
```

7. OPTIONAL: Create a Custom AVS with consensus mechanisms of your choice! Redeploy modified contracts to anvil saved state.

```
make [...]
```

Note: Full Makefile:

```
############################## HELP MESSAGE
##############################
# Make sure the help command stays first, so that it's printed by
default when `make` is called without arguments
.PHONY: help tests
help:
	@grep -E '^[a-zA-Z0-9_-]+:.*?## .*$$' $(MAKEFILE_LIST) | awk
'BEGIN {FS = ":.*?## "}; {printf "\033[36m%-30s\033[0m %s\n", $$1,
$$2}'

AGGREGATOR_ECDSA_PRIV_KEY=0x2a871d0798f97d79848a013d4936a73bf4cc922c825
CHALLENGER_ECDSA_PRIV_KEY=0x5de4111afa1a4b94908f83103eb1f1706367c2e68ca

CHAINID=31337
# Make sure to update this if the strategy address changes
# check in
contracts/script/output/${CHAINID}/credible_squaring_avs_deployment_out
STRATEGY_ADDRESS=0x7a2088a1bFc9d81c55368AE168C2C02570cB814F
DEPLOYMENT_FILES_DIR=contracts/script/output/${CHAINID}

----------------------------: ##

___CONTRACTS___: ##

build-contracts: ## builds all contracts
	cd contracts && forge build

deploy-eigenlayer-contracts-to-anvil-and-save-state: ## Deploy
eigenlayer
	./tests/anvil/deploy-eigenlayer-save-anvil-state.sh

deploy-incredible-squaring-contracts-to-anvil-and-save-state: ##
```

```
Deploy avs
	./tests/anvil/deploy-avs-save-anvil-state.sh

deploy-all-to-anvil-and-save-state:
deploy-eigenlayer-contracts-to-anvil-and-save-state
deploy-incredible-squaring-contracts-to-anvil-and-save-state ##
deploy eigenlayer, shared avs contracts, and inc-sq contracts

start-anvil-chain-with-el-and-avs-deployed: ## starts anvil from a
saved state file (with el and avs contracts deployed)
	./tests/anvil/start-anvil-chain-with-el-and-avs-deployed.sh

bindings: ## generates contract bindings
	cd contracts && ./generate-go-bindings.sh

___DOCKER___: ##
docker-build-and-publish-images: ## builds and publishes operator
and aggregator docker images using Ko
	KO_DOCKER_REPO=ghcr.io/layr-labs/incredible-squaring ko
build aggregator/cmd/main.go --preserve-import-paths
	KO_DOCKER_REPO=ghcr.io/layr-labs/incredible-squaring ko
build operator/cmd/main.go --preserve-import-paths
docker-start-everything: docker-build-and-publish-images ## starts
aggregator and operator docker containers
	docker compose pull && docker compose up

__CLI__: ##

cli-setup-operator: send-fund cli-register-operator-with-eigenlayer
cli-deposit-into-mocktoken-strategy cli-register-operator-with-avs
## registers operator with eigenlayer and avs

cli-register-operator-with-eigenlayer: ## registers operator with
delegationManager
	go run cli/main.go --config
config-files/operator.anvil.yaml register-operator-with-eigenlayer

cli-deposit-into-mocktoken-strategy: ##
	./scripts/deposit-into-mocktoken-strategy.sh

cli-register-operator-with-avs: ##
	go run cli/main.go --config
config-files/operator.anvil.yaml register-operator-with-avs

cli-deregister-operator-with-avs: ##
	go run cli/main.go --config
config-files/operator.anvil.yaml deregister-operator-with-avs

cli-print-operator-status: ##
```

```
        go run cli/main.go --config
config-files/operator.anvil.yaml print-operator-status

send-fund: ## sends fund to the operator saved in
tests/keys/test.ecdsa.key.json
        cast send 0x860B6912C2d0337ef05bbC89b0C2CB6CbAEAB4A5
--value 10ether --private-key
0x2a871d0798f97d79848a013d4936a73bf4cc922c825d33c1cf7073dff6d409c6

----------------------------: ##
# We pipe all zapper logs through https://github.com/maoueh/zap-
pretty so make sure to install it
# TODO: piping to zap-pretty only works when zapper environment is
set to production, unsure why
____OFFCHAIN_SOFTWARE___: ##
start-aggregator: ##
        go run aggregator/cmd/main.go --config
config-files/aggregator.yaml \
                --credible-squaring-deployment
${DEPLOYMENT_FILES_DIR}/credible_squaring_avs_deployment_output.json
\
                --ecdsa-private-key ${AGGREGATOR_ECDSA_PRIV_KEY} \
                2>&1 | zap-pretty

start-operator: ##
        go run operator/cmd/main.go --config
config-files/operator.anvil.yaml \
                2>&1 | zap-pretty

start-challenger: ##
        go run challenger/cmd/main.go --config
config-files/challenger.yaml \
                --credible-squaring-deployment
${DEPLOYMENT_FILES_DIR}/credible_squaring_avs_deployment_output.json
\
                --ecdsa-private-key ${CHALLENGER_ECDSA_PRIV_KEY} \
                2>&1 | zap-pretty

run-plugin: ##
        go run plugin/cmd/main.go --config
config-files/operator.anvil.yaml
----------------------------: ##
_____HELPER_____: ##
mocks: ## generates mocks for tests
        go install go.uber.org/mock/mockgen@v0.3.0
        go generate ./...

tests-unit: ## runs all unit tests
        go test $$(go list ./... | grep -v /integration)
```

```
-coverprofile=coverage.out -covermode=atomic --timeout 15s
	go tool cover -html=coverage.out -o coverage.html

tests-contract: ## runs all forge tests
	cd contracts && forge test

tests-integration: ## runs all integration tests
	go test ./tests/integration/... -v -count=1
```

Solutions:

(Do all of the following in root directory of the cloned GitHub repo)

4. Start Operator (locally hosted Anvil chain)

```
make start-anvil-chain-with-el-and-avs-deployed
```

5. Start Aggregator

```
make start-aggregator
```

6. Register Operator with EigenLayer and Opt-In to Participate in your AVS

```
make start-operator
```

7. OPTIONAL: Create a Custom AVS with consensus mechanisms of your choice! Redeploy modified contracts to anvil saved state.

```
# Make changes to files in ~/contracts/src/
# Specifically, IncredibleSquaringTaskManager.sol and its interface IIncredibleSquaringTaskManager.sol
# Redeploy all contracts to local anvil saved state
make deploy-all-to-anvil-and-save-state
```

You should now have an anvil operator running locally, that has opted in to participate with your AVS middleware via EigenLayer. The natural next step here is to create your own protocol for a given issue and register with EigenLayer! Note, however, that this example is for testnet only (anvil is local deploy).

Note: Example AVS (Squaring a Number) -

Take a look at the locations with a FLAG comment, and please stop to think about the questions attached to each. Solutions are also attached below.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.9;

import "@openzeppelin-upgrades/contracts/proxy/utils/
Initializable.sol";
import "@openzeppelin-upgrades/contracts/access/
OwnableUpgradeable.sol";
import "@eigenlayer/contracts/permissions/Pausable.sol";
import "@eigenlayer-middleware/src/interfaces/IServiceManager.sol";
import {BLSApkRegistry} from "@eigenlayer-middleware/src/
BLSApkRegistry.sol";
import {RegistryCoordinator} from "@eigenlayer-middleware/src/
RegistryCoordinator.sol";
import {BLSSignatureChecker, IRegistryCoordinator} from
"@eigenlayer-middleware/src/BLSSignatureChecker.sol";
import {OperatorStateRetriever} from "@eigenlayer-middleware/src/
OperatorStateRetriever.sol";
import "@eigenlayer-middleware/src/libraries/BN254.sol";
import "./IIncredibleSquaringTaskManager.sol";

contract IncredibleSquaringTaskManager is
    Initializable,
    OwnableUpgradeable,
    Pausable,
    BLSSignatureChecker,
    OperatorStateRetriever,
    IIncredibleSquaringTaskManager
{
    using BN254 for BN254.G1Point;

    /* CONSTANT */
    // The number of blocks from the task initialization within
which the aggregator has to respond to
    uint32 public immutable TASK_RESPONSE_WINDOW_BLOCK;
    uint32 public constant TASK_CHALLENGE_WINDOW_BLOCK = 100;
    uint256 internal constant _THRESHOLD_DENOMINATOR = 100;

    /* STORAGE */
    // The latest task index
    uint32 public latestTaskNum;

    // mapping of task indices to all tasks hashes
    // when a task is created, task hash is stored here,
```

```
    // and responses need to pass the actual task,
    // which is hashed onchain and checked against this mapping
    mapping(uint32 => bytes32) public allTaskHashes;

    // mapping of task indices to hash of abi.encode(taskResponse,
taskResponseMetadata)
    mapping(uint32 => bytes32) public allTaskResponses;

    mapping(uint32 => bool) public taskSuccesfullyChallenged;

    address public aggregator;
    address public generator;

    // FLAG 1: Why might this modifier be used?
    /* MODIFIERS */
    modifier onlyAggregator() {
        require(msg.sender == aggregator, "Aggregator must be the
caller");
        _;
    }

    // FLAG 2: What are the pros/cons of using this modifier vs.
making the function it modifies payable instead?
    modifier onlyTaskGenerator() {
        require(msg.sender == generator, "Task generator must be the
caller");
        _;
    }

    constructor(
        IRegistryCoordinator _registryCoordinator,
        uint32 _taskResponseWindowBlock
    ) BLSSignatureChecker(_registryCoordinator) {
        TASK_RESPONSE_WINDOW_BLOCK = _taskResponseWindowBlock;
    }

    function initialize(
        IPauserRegistry _pauserRegistry,
        address initialOwner,
        address _aggregator,
        address _generator
    ) public initializer {
        _initializePauser(_pauserRegistry, UNPAUSE_ALL);
        _transferOwnership(initialOwner);
        aggregator = _aggregator;
        generator = _generator;
    }

    /* FUNCTIONS */
```

```
    // NOTE: this function creates new task, assigns it a taskId
    function createNewTask(
        uint256 numberToBeSquared,
        uint32 quorumThresholdPercentage,
        bytes calldata quorumNumbers
    ) external onlyTaskGenerator {
        // create a new task struct
        Task memory newTask;
        newTask.numberToBeSquared = numberToBeSquared;
        newTask.taskCreatedBlock = uint32(block.number);
        newTask.quorumThresholdPercentage =
quorumThresholdPercentage;
        newTask.quorumNumbers = quorumNumbers;

        // store hash of task onchain, emit event, and increase
taskNum
        allTaskHashes[latestTaskNum] =
keccak256(abi.encode(newTask));
        emit NewTaskCreated(latestTaskNum, newTask);
        latestTaskNum = latestTaskNum + 1;
    }

    // NOTE: this function responds to existing tasks.
    function respondToTask(
        Task calldata task,
        TaskResponse calldata taskResponse,
        NonSignerStakesAndSignature memory
nonSignerStakesAndSignature
    ) external onlyAggregator {
        uint32 taskCreatedBlock = task.taskCreatedBlock;
        bytes calldata quorumNumbers = task.quorumNumbers;
        uint32 quorumThresholdPercentage =
task.quorumThresholdPercentage;

        // check that the task is valid, hasn't been responsed yet,
and is being responsed in time
        require(
            keccak256(abi.encode(task)) ==
                allTaskHashes[taskResponse.referenceTaskIndex],
            "supplied task does not match the one recorded in the
contract"
        );
        // some logical checks
        require(
            allTaskResponses[taskResponse.referenceTaskIndex] ==
bytes32(0),
            "Aggregator has already responded to the task"
        );
        require(
```

```
            uint32(block.number) <=
                taskCreatedBlock + TASK_RESPONSE_WINDOW_BLOCK,
            "Aggregator has responded to the task too late"
        );

        /* CHECKING SIGNATURES & WHETHER THRESHOLD IS MET OR NOT */
        // calculate message which operators signed
        bytes32 message = keccak256(abi.encode(taskResponse));

        // check the BLS signature
        (
            QuorumStakeTotals memory quorumStakeTotals,
            bytes32 hashOfNonSigners
        ) = checkSignatures(
                message,
                quorumNumbers,
                taskCreatedBlock,
                nonSignerStakesAndSignature
            );

        // FLAG 3: What is happening in this for loop?
        for (uint i = 0; i < quorumNumbers.length; i++) {
            // we don't check that the quorumThresholdPercentages
are not >100 because a greater value would trivially fail the check,
implying
            // signed stake > total stake
            require(
                quorumStakeTotals.signedStakeForQuorum[i] *
                    _THRESHOLD_DENOMINATOR >=
                    quorumStakeTotals.totalStakeForQuorum[i] *
                        uint8(quorumThresholdPercentage),
                "Signatories do not own at least threshold
percentage of a quorum"
            );
        }

        TaskResponseMetadata memory taskResponseMetadata =
TaskResponseMetadata(
            uint32(block.number),
            hashOfNonSigners
        );
        // updating the storage with task responsea
        allTaskResponses[taskResponse.referenceTaskIndex] =
keccak256(
            abi.encode(taskResponse, taskResponseMetadata)
        );

        // emitting event
        emit TaskResponded(taskResponse, taskResponseMetadata);
```

```
    }

    function taskNumber() external view returns (uint32) {
        return latestTaskNum;
    }

    // FLAG 4: What is the purpose of raising a challenge?
    // NOTE: this function enables a challenger to raise and resolve
a challenge.
    // TODO: require challenger to pay a bond for raising a
challenge
    // TODO(samlaf): should we check that quorumNumbers is same as
the one recorded in the task?
    function raiseAndResolveChallenge(
        Task calldata task,
        TaskResponse calldata taskResponse,
        TaskResponseMetadata calldata taskResponseMetadata,
        BN254.G1Point[] memory pubkeysOfNonSigningOperators
    ) external {
        uint32 referenceTaskIndex = taskResponse.referenceTaskIndex;
        uint256 numberToBeSquared = task.numberToBeSquared;
        // some logical checks
        require(
            allTaskResponses[referenceTaskIndex] != bytes32(0),
            "Task hasn't been responded to yet"
        );
        require(
            allTaskResponses[referenceTaskIndex] ==
                keccak256(abi.encode(taskResponse,
taskResponseMetadata)),
            "Task response does not match the one recorded in the
contract"
        );
        require(
            taskSuccesfullyChallenged[referenceTaskIndex] == false,
            "The response to this task has already been challenged
successfully."
        );

        require(
            uint32(block.number) <=
                taskResponseMetadata.taskResponsedBlock +
                    TASK_CHALLENGE_WINDOW_BLOCK,
            "The challenge period for this task has already
expired."
        );

        // logic for checking whether challenge is valid or not
        uint256 actualSquaredOutput = numberToBeSquared *
```

```
numberToBeSquared;
        bool isResponseCorrect = (actualSquaredOutput ==
            taskResponse.numberSquared);

        // FLAG 5: What would happen is isResponseCorrect == false?
        if (isResponseCorrect == true) {
            emit TaskChallengedUnsuccessfully(referenceTaskIndex,
msg.sender);
            return;
        }

        // get the list of hash of pubkeys of operators who weren't
part of the task response submitted by the aggregator
        bytes32[] memory hashesOfPubkeysOfNonSigningOperators = new
bytes32[](
            pubkeysOfNonSigningOperators.length
        );
        for (uint i = 0; i < pubkeysOfNonSigningOperators.length;
i++) {
            hashesOfPubkeysOfNonSigningOperators[
                i
            ] = pubkeysOfNonSigningOperators[i].hashG1Point();
        }

        // verify whether the pubkeys of "claimed" non-signers
supplied by challenger are actually non-signers as recorded before
        // when the aggregator responded to the task
        // currently inlined, as the
MiddlewareUtils.computeSignatoryRecordHash function was removed from
BLSSignatureChecker
        // in this PR: https://github.com/Layr-Labs/eigenlayer-
contracts/commit/
c836178bf57adaedff37262dff1def18310f3dce#diff-8ab29af002b60fc80e3d6564e
        // TODO(samlaf): contracts team will add this function back
in the BLSSignatureChecker, which we should use to prevent potential
bugs from code duplication
        bytes32 signatoryRecordHash = keccak256(
            abi.encodePacked(
                task.taskCreatedBlock,
                hashesOfPubkeysOfNonSigningOperators
            )
        );
        require(
            signatoryRecordHash ==
taskResponseMetadata.hashOfNonSigners,
            "The pubkeys of non-signing operators supplied by the
challenger are not correct."
        );
```

```
        // get the address of operators who didn't sign
        address[] memory addresssOfNonSigningOperators = new
address[](
            pubkeysOfNonSigningOperators.length
        );
        for (uint i = 0; i < pubkeysOfNonSigningOperators.length;
i++) {
            addresssOfNonSigningOperators[i] = BLSApkRegistry(
                address(blsApkRegistry)

).pubkeyHashToOperator(hashesOfPubkeysOfNonSigningOperators[i]);
        }
```

Solutions:

`1.`

The onlyAggregator() modifier is used to modify the respondToTask() function. Referring to the Operator/Generator/Aggregator diagram, it is seen that only the Aggregator should respond to tasks, and deliver the aggregated response to the AVS contracts.

`2.`

onlyTaskGenerator is used to restrict createNewTask from only being called by a permissioned entity. In a real world scenario, this would be removed by instead making createNewTask a payable function.

`3.`

This for loop checks that signatories own at least a threshold percentage of each quorum. This ensures voting power

`4.`

Raising challenges allows for verification of the operator's responses to the given task.

`5.`

If isResponseCorrect == false, then the operator did not answer the task truthfully with the correct squared number. As a punishment for acting untruthfully, the ETH our operator has staked on the AVS Number Squaring protocol is slashed (permanently lost)

Registering for Holesky Testnet and Mining from Faucet

Add new network manually to MetaMask Wallet

- Network name: Holesky Testnet
- Network URL: https://ethereum-holesky.publicnode.com
- Chain ID: 17000
- Currency symbol: ETH
- Block explorer URL: https://holesky.beaconcha.in

Mine approx. 1 Holesky ETH from faucet here: https://holesky-faucet.pk910.de/ (try not to get hacked)

Holešky PoW Faucet

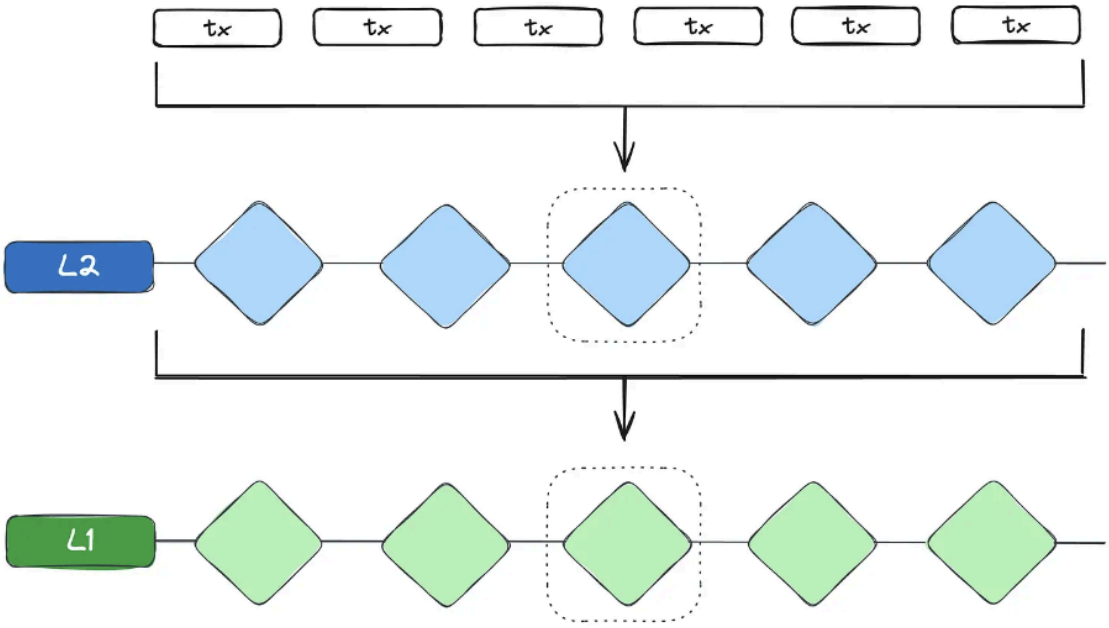
> ## Data Availability Layers & Rollups

`No problem remains the same at scale`

To understand data availability, which is the motivating factor of many prominent AVS registered with EigenLayer and thus an important part of the restaking economic paradigm, it is important to first understand **rollups**. As noted above, it is generally true of any system that its defining problems change almost entirely

when you scale by orders of magnitude. This is certainly true of Ethereum. The volume of transactions on Ethereum is growing rapidly every year; high traffic significantly increases gas fees given that the availability of operators is limited. Rollups have emerged as a solution to the scalability issue in Ethereum development. They facilitate processing of off-chain transactions whose logs are posted as a single transaction onto mainnet Ethereum, reducing traffing by several orders of magnitude.

Rollups outsource execution and transaction processing off of the main Ethereum (or L1) network, while still inheriting Ethereum security guarantees.



(Image from https://nader.substack.com/p/beyond-restaking-eigenlayer-for-developers)

There are two types of rollups, Zero-Knowledge (ZK) Rollups and Optimistic Rollups. Read more here:

https://www.immutable.com/blog/zero-knowledge-vs-optimistic-rollups-explained-which-one-is-better-for-blockchain-games

Data Availability layers host data on chain, and they enable both types of rollups access to this data for verification off-chain. In a centralized system, think of an analogy where a client needs to access data to verify transactions (rollup/Arbitrum) with a central database (some server running SQL database). DA Layers allow for this accessility to on-chain data, even while rollups operate off-chain for increased performance.

EigenDA is a data availability layer made by EigenLabs and built as an AVS compatible with EigenLayer, currently launched on the Holesky testnet and launching on mainnet in early Q2 2024.

> # Utilizing EigenDA (AVS registered with EigenLayer)
>
> # with Arbitrum for Efficient Data Access in Ethereum Development (Optional Materials)

Take at the look at the docs here:

https://docs.eigenlayer.xyz/eigenda/rollup-guides/orbit/

> # Putting it All Together

- Restaking allows operators to put down stake (typically ETH) for some given PoS protocol
- EigenLayer acts as an intermediate that matches AVS's (new protocols), operators interested putting stake on these protocols, and consumers interested in putting stake on these protocols via a trusted operator
- You have seen the basics of how to do the following:
- - deploy an operator

- design an AVS of your own
- register your operator with EigenLayer and opt-in to your protocol
- Understand several important types of AVS (bridges, oracles, data availability layers, L1 chains)
- Understand how to deploy an EigenDA AVS along with Arbitrum Orbit Rollup (launched April 2024!)

Natural follow up - where should I host my node operator if I don't have local compute?

- Check out Google Cloud Platform Blockchain Node Engine: https://cloud.google.com/blockchain-node-engine?hl=en
- Digital Ocean Cloud Blockchain Servers https://www.digitalocean.com/solutions/blockchain
- Many more

Submission

Form here: https://docs.google.com/forms/d/e/1FAIpQLSckhRdyL5PQMlFBbAtJKpzwKjI56ap6dsgI38F_zHvmgf21Ew/viewform?usp=sf_link

Sources - Great Resources for Further Learning

General Reads on Restaking:

https://medium.com/@prezzel/eigenlayer-d90e21ab4081

https://www.blog.eigenlayer.xyz/ycie/

https://nader.substack.com/p/beyond-restaking-eigenlayer-for-developers

AVS Middleware Toy Deployment Repo:

https://github.com/Layr-Labs/incredible-squaring-avs

modular blockchain thesis:

https://medium.com/@prezzel/the-modular-blockchain-thesis-bc7d11ed4e98

Dual Staking:

https://www.blog.eigenlayer.xyz/dual-staking/

Setting Up Holesky ETH:

https://www.coingecko.com/learn/holesky-testnet-eth

EigenDA + Arbitrum Orbit

https://www.blog.eigenlayer.xyz/eigenda-altlayer-arbitrum-orbit/

Optimistic Vs. ZK Rollups:

https://www.immutable.com/blog/zero-knowledge-vs-optimistic-rollups-explained-which-one-is-better-for-blockchain-games