EIGEN: The Universal Intersubjective Work Token

Towards the Open Verifiable Digital Commons

Eigen Labs Team

Abstract

We introduce the structure of the EIGEN token, which can function as a universal intersubjective work token. We unpack this term here: work token refers to a token that needs to be staked (i.e., locked up), to perform some work. Work tokens are employed by many existing proof-of-stake blockchain platforms to perform validation in the blockchain using a mechanism called staking. Work tokens are used not only as entry conditions for performing digital work but also for punishing non-compliant workers through a cryptoeconomic mechanism called slashing (i.e., when digital workers break the covenants inherent in the platform, they risk losing their work tokens). Existing work tokens have at least one of two limitations: (a) they are special-purpose: created specifically for a particular enshrined digital task (e.g., the ETH token is used for validating Ethereum blocks), and/or (b) they are objective: enforceable only when the violations of the digital worker are clearly attributable onchain (via a programmatic proof). EigenLayer, a restaking (or more accurately, permissionless programmable staking) platform, removes the "special-purpose" limitation from ETH. Restaking expands the scope of ETH to become a "universal" objective work token so that it can be staked to participate in all kinds of tasks. However, the "objective" limitation remains, as punitive slashing can only be applied to objectively verifiable tasks on Ethereum.

In this paper, we build a new staking model that breaks both limitations simultaneously: (a) a universal work token that can be used across all digital tasks, which are (b) intersubjectively verifiable, i.e., any task for which any two reasonable, observant humans will agree whether a digital worker failed to carry out the task correctly. The core idea is to create a forking token at the application layer, building on a line of pioneering prior work. This paper expands the utility of the forking token significantly in four dimensions: a) universality: allowing it to be used universally across various tasks, b) isolation: endowing the ability to use the forking token in fork-unaware applications, c) metering: the cost of social consensus is metered and enforced, and d) compensation: enabling users to be compensated for faults affecting them.

This token significantly expands the types of credible commitments that digital workers can make with cryptoeconomic security. It significantly expands the scope of digital tasks that blockchains can offer their users securely. In particular, this token enhances the set of Actively Validated Services (AVS) that can be secured through EigenLayer. Furthermore, EIGEN serves a fully complementary role to ETH with ETH staking powering objective fault slashing and EIGEN staking powering slashing for intersubjective faults. We view this intersubjective work token as taking the Ethereum ecosystem one step closer to the goal of building the Open Verifiable Digital Commons.

1 Introduction

1.1 Background

The significant evolutionary advantage of humans rests in our ability to cooperate in large numbers. Large-scale cooperation requires trust, which is usually provided by intermediating institutions. The modern economy has seen the rise of digital platforms such as Facebook, Uber, Airbnb, Visa, iTunes, eBay, and Amazon marketplace, which allow global communication, efficient markets, and cooperation at scale. One might say that these digital platforms form a system of digital commons, somewhat akin to the physical commons, such as forests, fisheries, national parks, and oceans, that we have in the real world.

While existing digital platforms rival and in many cases out-strip physical commons in access, importance and reach, they do not fully qualify as digital commons as they are controlled without democratic governance, leading to significant frictions, such as (i) centralized information and control silos, where users or innovative solutions can be de-platformed any time, (ii) platforms enshrine somewhat subjective positions, such as the specific recommendation engine, moderation policy, and mechanisms to evaluate driver behavior, credit or renter behavior, (iii) innovations on top of these platforms require the permission of the platform creators,

and this significantly constrains the innovation velocity, and (iv) platforms can unilaterally alter policies that affect users without seeking their inputs.

Blockchains offer an alternative paradigm for building the global digital commons where it is possible to have (i) permissionless user access, (ii) native user governance, (iii) programmatic incentives to share value across the various participants, and (iv) permissionless innovation. All of these properties arise because blockchains have two main properties: (a) openness, which comprises censorship resistance and permissionlessness, and (b) verifiability, which ensures that the nodes participating in the blockchain are performing their duties correctly. Thus, blockchains offer a new paradigm for building the open, verifiable digital commons.

Staking via a work token. This brings us to the question of how one can create a digital commons platform, which is at the heart of blockchain architecture design. Instead of a legal solution of ensuring openness and correctness via laws, blockchains strive to create a neutral digital platform by ensuring that no single party is responsible for creating and maintaining the platform. That is, blockchains decentralize the ability to participate in the upkeep of the platform, by which we can achieve credible neutrality. A popular mechanism that blockchains employ to achieve this decentralization is using proof-of-stake protocols: creating a token that anyone can use to participate in the digital work required for the upkeep of the blockchain [1]. One can think of these staking tokens as work tokens, designed specifically for the particular digital task of participating in the operations of running that particular blockchain. Similar to how a taxi medallion is required to drive a taxi in certain cities, these work tokens are required to participate in the digital work associated with the upkeep of the blockchains.

Slashing. A major advantage of proof-of-stake protocols is that if the participating stakers violate the commitments required for the digital work, then they can lose their staking tokens using a mechanism called slashing. Slashing ensures that even if all the validators try to collude to break the commitments inherent in running the blockchain, they will lose their staking tokens. This creates a system of *Karma*: the law of attribution and enforcement. Ethereum, again, pioneered this structure of "cryptoeconomic security" [2]. Crypto - alludes to the attributability of the fault to agents using cryptographic methods like signatures, and economic - alludes to the loss of economic value associated with stake for agents who commit faults. Cryptoeconomic security is a significant advance in the ability of individual agents to cooperate to perform digital work, thus creating an emergent digital platform that can make credible commitments of neutrality and correctness.

EigenLayer turns `ETH` into a *Universal Objective Work Token*. Restaking in EigenLayer enables a universal mechanism to share the same stake across a variety of different digital tasks, thus making `ETH` a "universal work token". While prior staking mechanisms coupled a given type of stake/token with an enshrined set of digital tasks in the platform, restaking enables the same token to participate in a variety of digital tasks including new consensus mechanisms, optimistic rollups, bridges and MEV management solutions, which third parties can permissionlessly create and innovate upon. Thus EigenLayer expands the scope of staking `ETH` from a single enshrined task of participating in consensus to a variety of digital tasks that can be built permissionlessly on top of the common blockchain. However, restaking provides cryptoeconomic security via slashing only for digital tasks whose faults are objectively attributable, i.e., clearly verifiable onchain. Examples of such onchain attributable faults include double spending (the same coin should never be spent twice in a payment chain), double signing (there can be no two blocks with the same block index), invalidity (creating a new state of the system which is not correctly transitioned from the previous state given the intervening transactions). Thus we can say that EigenLayer converts `ETH` into a Universal Objective Work Token.

Limitations of restaking: non-objectively attributable tasks exist. There are many categories of faults that digital workers can commit that do not have objective attribution on the chain. For example, let us take the example of a price oracle required onchain, which brings the market price of a digital asset onchain. Price oracles are famously difficult to build slashing protocols for: if a digital worker performing an oracle task reports that $1\texttt{BTC} = 1\texttt{USD}$ (which is wrong by several orders of magnitude), the blockchain by itself has no locus to verify the correctness or otherwise of this assertion. This is because blockchains are purely deterministic state machines, and to preserve determinism and objectivity, these subjective claims are excluded from intrinsic verification in the blockchain. There are other examples of claims that are not verifiable objectively onchain: bridges (claims about the state of a different blockchain on another blockchain), data availability (i.e., verifying whether a data item has been published in public), and censorship (not incorporating some transactions into a ledger). Another major problem with enforcing slashing onchain is that immutable code for slashing is difficult to get right - if there is any error in the code, then honest nodes may get slashed.

Current state-of-the-art mechanisms to resolve non-objectively attributable frauds. There are two classes of mechanisms that have been proposed for this purpose of non-objectively attributable tasks:

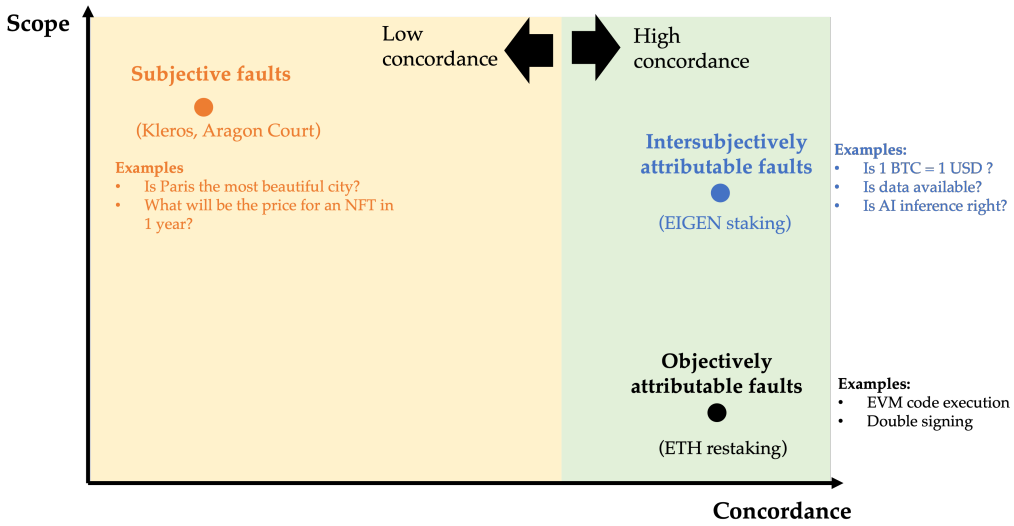


Figure 1: The $x-$axis represents the amount of concordance among the observers for the attribution. The $y-$axis represents the scope of faults attributable. We focus in this paper on faults attributable with high concordance.

1. Penalize (i.e., slash) divergent claims of a node relative to the majority of the nodes. However, such a mechanism is subject to the *tyranny-of-majority*. If the majority of workers are dishonest, then a minority of honest agents can be slashed. This is clearly unacceptable.
2. Utilize an entrenched committee to verify the veracity of a node's claims and issue slashing authorizations. The problem with this model is that if the committee is malicious, they can slash honest workers. The crux of the problem here is the *tyranny-of-majority* (of token holders or an entrenched committee).

Both approaches are vulnerable to tyranny-of-majority, and, even more fundamentally, it boils down to the question of who watches the watchers/committee.

1.2 Categorization of Faults in Digital Tasks

We can categorize faults made while performing digital tasks into three categories based on the ability to attribute them: (i) objectively attributable faults, (ii) intersubjectively attributable faults, and (iii) non-attributable faults.

1.2.1 Objectively Attributable Faults

These are faults that are attributable purely mathematically and cryptographically without the need to ask the opinions of others. More explicitly, any honest user can be convinced of the objectively attributable fault unilaterally with just local information. A downstream benefit of this feature of objectively attributable fault is that it provides the ability to resolve objectively attributable faults onchain by specifying their dispute resolution mechanism[1] in smart contracts. A clear example is execution validity: if a node runs a piece of code on a deterministic virtual machine and certifies the input and output, it can be verified onchain zk proof or optimistic fraud proofs. A second example of this is double-signing: in a consensus protocol, if a node is not supposed to sign two conflicting blocks but it does, then it can be proven onchain that the node has committed an onchain attributable fault. We furthermore note that the former fault (validity) is *proactively observable* - for example, the chain can run a verifier for a ZK proof and *proactively* ensure the correctness, whereas the latter (double-signing) is only *retroactively observable*: you can prove that a node committed the fault only *after* the signature on the second conflicting block is received.

1.2.2 Intersubjectively Attributable Faults

These are a set of faults where there is a broad-based agreement among all active observers of the system. We call such faults intersubjectively attributable faults. We can further subdivide this class into (i) retroactively observable, which can be attributed at any point in time, vs (ii) *concurrently observable*, which needs

[1]This dispute resolution mechanism could be a ZK verifier or a fraud proof mechanism.

Faults in digital tasks	Observability	Examples	Resolved via ZK / FHE?	Resolved via TEE?	Attributable via restaking?	Attributable via intersubjective staking?
Objectively attributable faults	Proactively observable	Deterministic Validity	✓	✓	✓	✓
	Retroactively observable	Reorgs	✗	✗	✓	✓
Intersubjectively attributable faults	Retroactively observable	Oracle price feed, validity (without slashing contracts)	✗	✓	✗	✓
	Concurrently observable	Data withholding, Censorship	✗	✗	✗	✓
Non-attributable faults	None	Privacy	✓	✓	✗	✗

Figure 2: This table illustrates the categorization of faults in digital tasks.

the observer to be online and observing at the time of the fault in order to be attributed. Some examples of retroactively observable intersubjective faults: (a) oracle price inputs - if the price input by an oracle node is clearly different from the actual price by a significant margin, all outside observers can come to a consensus on whether the price reported was correct or not, and whether the oracle should be slashed, (b) objectively attributable faults for which it is hard to write slashing conditions - as an example, running a specific instance of an AI / ML code. Some examples of concurrently observable intersubjective faults are: (c) Data availability - if a data item is published transparently into a peer-to-peer network or not can be attributed by all those observing the peer-to-peer network, (d) censorship - if a transaction that has been propagated in the peer to peer network but has not been included in the newly proposed blocks for sufficient time due to a malicious collusion, then in a synchronous network, it can be attributed by online observers watching the mempool.

The fault observability in concurrently observable faults can be enhanced by reducing the monitoring cost of the observer so many more nodes can observe the faults. We note that this phenomenon is not unique to digital platforms. In fact, in the breakthrough work [3] on "Governing the Commons" by Elinor Ostrom, winner of the Nobel Memorial Prize in Economics, it is observed that reducing the monitoring cost of violations is a prerequisite to a well-functioning commons.

For example, light nodes can sample portions of the data claimed available using a method called Data Availability Sampling. Light nodes can randomly sample the transactions in the p2p network (rather than monitoring all the transactions) to attribute censored transactions. Another factor that significantly boosts attributability is circumstantial evidence downstream of the digital task manipulation: for example, was an oracle manipulation then used downstream to attack a lending protocol, or if data might have been purposely withheld to pass an invalid claim through a fraud proof, or an oracle transaction censored to manipulate liquidations, etc. In all of these cases, there will be strong social agreement on whether the system was purposely attacked.

1.2.3 Non-attributable Faults

The third category of faults is *not attributable* outside the victim of the fault. For example, consider a secret-sharing system where anyone who wants to store a secret can split the secret into smaller chunks of information and different nodes store different subsets of the data. The secret is revealed only after a certain amount of time. In this case, if the nodes collude together to break the privacy and reveal the secret, it is possible for the secret storer to know that his / her secret is revealed but a third party cannot disambiguate between whether the storer is malicious or the system is majority malicious. Since it is impossible to attribute and punish for these faults, it is only possible to avoid them by making the validators *decentralized* and hence *collusion resistant*.

Subjective "faults" are those where observers make their own judgments, with no guaranteed concordance across the different observers. Examples include taste judgments, reputation scoring of drivers or restaurants, questions about the future price, etc. These squarely fall outside the scope of the present work, because, due to the lack of concordance, we cannot build strong cryptoeconomic guarantees on these type of "faults".

Fig. 1 illustrates the distinction among objectively attributable faults, intersubjective attributable faults and

subjective faults. Fig. 2 illustrates a categorization of different types of tasks and their resolvability with different mechanisms.

1.3 Overview of Core Contributions

In this paper, we introduce EIGEN a new universal method to induce cryptoeconomic penalties for intersubjectively attributable faults committed by stakers, without being vulnerable to the tyranny-of-majority.

Prior Work. The intellectual precursor to this work is the pioneering work by Vitalik Buterin documented in the Ethereum blog (circa 2014-2015): [4], [5], [6]. To the best of our knowledge, Augur built on these posts and introduced the idea of designing a token that forks at the application layer of Ethereum in the context of prediction markets [7]. Kleros and Aragon Court use the system of jurors who are selected probabilistically based on their stake to vote on subjective faults (but no forking) [8, 9]. In UMA, each token holder votes on a price request submitted by the Optimistic Oracle for resolving disputes [10] and relies on Nash equilibria without forking for security.

Contributions of this work. In this paper, we build on the prior work of using social consensus (forking) for resolving intersubjectively attributable faults to design EIGEN. Compared to the prior work, we expand the scope of EIGEN in four significant directions:

1. **Universality.** Building a **universal intersubjective token** rather than a solution for a specific problem.
2. **Isolation.** Solving how to utilize a forking token in fork-unaware applications such as DeFi.
3. **Metering.** Ensuring that the cost to social consensus for resolving any intersubjective fault is metered and the value distributed to those who partake in it.
4. **Compensation.** Designing mechanisms to utilize the intersubjective fork for compensating users against faults of the stakers/operators, rather than just being limited to only punishing them.

EIGEN **unleashes more open innovation.** EigenLayer enables anyone to create new Actively Validated Services (AVS) that have objective slashing conditions to incorporate security from ETH by being able to slash ETH. However, a significant limitation is that if the AVS had a digital task that involves intersubjective fault, it can't resort to the cryptoeconomics of ETH, without creating the tyranny-of-majority problem. In this work, we create the first universally intersubjectively slashable system so that stakers can make a wide range of commitments about the digital work that they perform. This enables a wide range of new AVSs to utilize the cryptoeconomics of EIGEN without being capped to their own economic value. We envision a renaissance in the design of new services structured as AVSs utilizing EIGEN cryptoeconomic security, in addition to ETH security. Any Actively Validated Service (AVS) on EigenLayer that has intersubjectively attributable digital faults can now obtain the cryptoeconomic safety guarantee of EIGEN. To make the faults intersubjectively attributable, the AVS may need to focus on developing robust monitoring infrastructure including light clients. This can lower the *cost-of-monitoring*, ensuring that there will be a wide net of community members from EIGEN's social consensus who will be operating the AVS's light node software for monitoring the EigenLayer operators that have opted into the AVS.

2 Core Idea

The main question we wish to address is whether we can extend cryptoeconomic security to digital tasks on EigenLayer in the context of intersubjective faults. We answer this question in the affirmative by creating EIGEN: the universal intersubjective work token.

2.1 Chain forking in Ethereum

Before we get to the description of the protocol on how to get cryptoeconomic security for intersubjectively attributable tasks, we glean some critical insights on how we escape the problem of the tyranny-of-majority in existing protocols like Ethereum. Ethereum has been a leader in recognizing the importance of the fact that the security of any blockchain is comprehensively inherited from two factors:

- **Cryptoeconomic security** that is obtained from the stake deposited by the validator set as part of participating in the proof-of-stake (PoS) protocol, and
- **Layers of social consensus** which surround and moderate the objective consensus mechanisms of the blockchain. Each layer of social consensus reveals a form of *watching the watcher*.

We can easily distinguish three layers of consensus that are integral to the operation of Ethereum:

1. Chain Validity
   (a) which rule for determining the validity of a given chain of blocks is to be used.
   (b) whether a given chain of blocks is valid, according to a given validity rule (see [11] for details).
2. Chain Liveness
   (a) what rule to follow if new valid blocks are not being proposed or attested.
3. Chain canonicity
   (a) which chain of blocks, valid under a given validity rule, constitutes the canonical chain at a given point in time.

While it might seem that a majority of Ethereum validators would have carte-blanche power to determine the content of the canonical chain completely, this is not true. Suppose that the majority of Ethereum validators were to turn malicious and collude to either double-sign off on a block to do a reorg[2] or engage in liveness attack by refusing to sign on any new blocks for at least an observable period[3]. This would set in place the following sequence of events:

1. Any validator not a part of the colluding set would reject any invalid blocks under the current validity rule while continuing to propose and attest to valid blocks.
2. Depending on the type of attack being executed, the Ethereum's social consensus can intervene to reject the block:
   - Incorrect state transition is an objectively attributable fault that is proactively observable. Any honest validator or any user can verify the correctness of state execution themselves by re-executing the block.
   - Building blocks that are in conflict with any weak subjectivity checkpoint is a proactively observable objective fault. Any honest validator or user can outright reject such blocks.
   - Double-signing on finalized blocks is an objectively attributable fault but only retroactively observable. Any honest validator or a user in the social consensus, when given evidence of a validator having attested to block headers for conflicting blocks, would be able to objectively attribute the fault.
   - Liveness attack against chain growth is an intersubjective attributable fault with concurrent observability. Therefore, those users in social consensus who are active at the moment when the attack is being executed and are able to tap into the P2P network of Ethereum by themselves or via someone they trust will be able to detect the attack[4].
3. In order to provide a cryptoeconomic guarantee against the above attacks, there are two distinct provisions for slashing in Ethereum [12]:
   - in the event of chain forking due to double-signing on finalized blocks, Ethereum offers the provision to slash the offending validators by including transactions for slashing the adversarial validators.
   - in the event of chain forking contingencies due to a liveness attack, Ethereum has a special provision of "inactivity leak," wherein the colluding validators that are not proposing or attesting to new blocks can be slashed over the course of time [13].

   This fund to slash the adversarial validators comes from the stake that was deposited as part of staking with Ethereum, thus lending to cryptoeconomic security.
4. In either case, if the social consensus of Ethereum agrees with a fork of the chain where the slashing is happening, then that fork would end up being perceived as the canonical fork, which lends to cryptoeconomic security against the adversary.
5. The digital assets of the other fork would be excluded from any interface between onchain and offchain[5] and their market value would trend to zero.

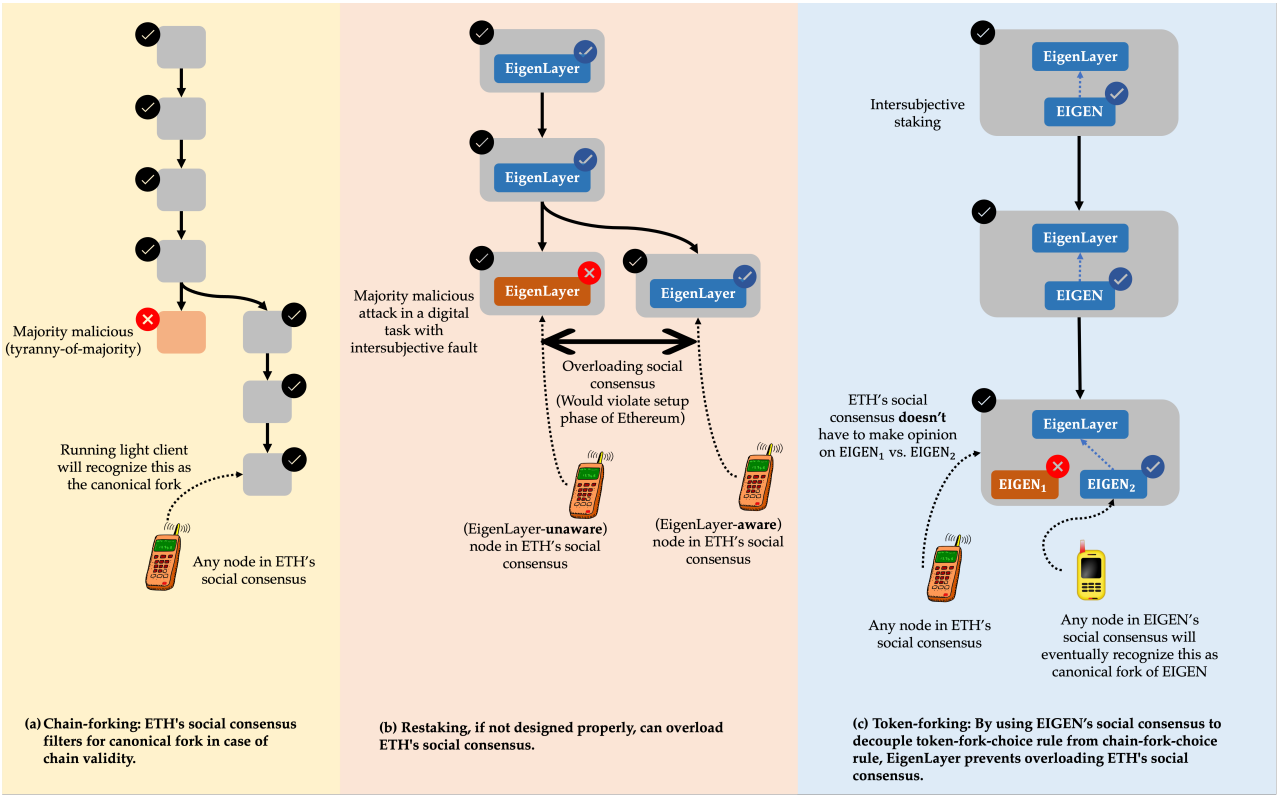


Figure 3: A high-level overview on how intersubjective staking in EigenLayer resolves intersubjective faults without overloading ETH's social consensus. Fig (a) illustrates how Ethereum's social consensus resolves any violation of chain validity despite majority collusion in the validator set. Fig (b) illustrates how extending restaking to resolve anything beyond objectively attributable faults would overload Ethereum's social consensus, thus violating Ethereum's setup phase. Fig (c) illustrates that intersubjective staking with `EIGEN` utilizes token forking to resolve any intersubjective attributable fault.

Thus, on inspection, we find that the influence of Ethereum validators ultimately extends *only* to determining which of any possible set of valid blocks should be added to the canonical chain at each moment in time. In particular, we see that there exists an easily overlooked but powerful social consensus that decides whether to uphold the fork where slashing is happening as canonical or not. See fig. 3(a) for an illustration. This is an illustration of how social consensus is protecting the Ethereum ecosystem from a *tyranny-of-majority* of Ethereum's validator set. Unlike the set of validators staked on the chain, a key observation is that the coverage of social consensus is the set of all possible observers, and is an unsized and potentially infinite set. Even if all the nodes that are running the software agree on a malicious fork, new nodes that come in might detect this problem and choose only to build on the fork perceived as canonical by the social consensus. Thus, truth becomes a self-enforcing equilibrium and becomes nearly impossible to take over by an adversary.

2.2 The Two Phases of Intersubjective Agreement

Systems that utilize intersubjective agreement usually proceed in two phases. There is a *setup phase*, in which the rules of intersubjective agreement are codified, and then there is a second phase, called the *execution phase*, in which the pre-agreed rules are executed. As a first example, consider the establishment of a modern nation-state. The setup phase comprises codifying a constitution, which all later laws must then follow. The execution phase then requires the creation of laws, which must comply with the Constitution. The clearer the Constitution is, the easier it is to agree on whether a given law complies with the Constitution.

Let's take blockchains as a second example. The setup phase for blockchains involves specifying the fork-choice rule of the consensus protocol (for example, the longest chain rule specifies that the current block

[2]Incorrect state execution is not a problem as any honest validator running full node can inspect the correctness immediately.

[3]Note that one doesn't need collusion among majority of Ethereum validators for executing either of the attacks. As long as more than 33% of stake is colluding, then either of the attacks can be executed. We are considering "majority corruption" here to illustrate the power of social consensus.

[4]Here we are assuming that there is no partitioning of the underlying P2P network.

[5]Examples are exchanges, real-world shops where crypto is accepted as a medium of exchange for physical goods, etc.

Concept	US Gov.	PoW consensus	Weak subjectivity	Chain validity/DA	Censorship slashing	Rollup	Sovereign Rollup
Setup phase: Pre-agree on a rule	US Constitution	Longest-chain rule (LCR)	Weak subjectivity checkpoint rule	Accept only valid, available blocks	Slash validators for censorship using UASF	Follow bridge contract	Example: Follow social consensus to revert hacks
Execution phase: Execute the rule	Laws passed compliant with constitution	Decide on latest block (per LCR)	Computation of weak subjectivity checkpoint	Nodes reject invalid/unavailable blocks even when header signed by majority	When observing censorship, activate UASF to slash validators	Decide on current rollup block (using bridge state)	Decide on current rollup block (using social consensus)

Figure 4: An illustration of the setup and execution phase for for different scenarios in government settings and blockchain settings.

is the tip of the longest chain of blocks). In the execution phase, any node can now identify, based on its observed blockchain, the current block. Thus, the setup phase requires consensus, and later, the execution phase requires purely local information. This is an example of a powerful intersubjective system, as there is little ambiguity left in the execution phase. As a more complex example, in a proof-of-stake blockchain, there are two distinct consensus rules possible: (i) light-node-rule: accept any block header signed by 67% of the nodes, or (ii) full-node-rule: accept only blocks that you can download **and** that are valid **and** that are signed by 67% of the nodes. It is pre-agreed in the setup phase of the blockchain that the full-node-rule is the real rule, and the light-node-rule is a proxy, so when there is a disagreement between the two rules, nodes already know that they should go with the full-node rule during the execution phase.

As another complex example, consider two distinct kinds of rollups: (a) standard rollup, where the setup phase defines the current block in the chain as the latest valid block written to the bridge contract, and a (b) sovereign rollup, where the setup phase defines that the current block in the chain is the latest valid block written to the bridge contract, *except if it is reverted by social consensus*. In the standard rollup case, the execution phase is totally self-evident: each node can find the current block directly without requiring to talk to each other. In the sovereign rollup case, the nodes are required to talk to each other in order to know if social consensus has reverted that block or not. The more an intersubjective system makes the execution phase *self-verifiable*, the stronger the intersubjective cohesion is, i.e., participants will agree on the right version of the world more easily. If too much coordination is required during the execution phase, it is possible for the system to experience intersubjective fracture, i.e., participants disagree on the right version of the world.

2.3 Extending the power of social consensus beyond chain validity via Forking Token

Ethereum's social consensus. As described in sec. 2.1, Ethereum's security comes from a combination of cryptoeconomics of slashing of stake and social consensus. Given the benefit of using Ethereum's social consensus to resolve any take-over of Ethereum's validator set to attack chain validity, an immediate question would be whether the usage of Ethereum's social consensus can be extended beyond chain validity and liveness. One example of services that can benefit from using Ethereum's social consensus to resolve tyranny-of-majority is any generic tasks where faults are intersubjectively attributable in nature (see sec. 1.2 for details). Examples of such tasks are oracles or data availability (see sec. 1.2). In such services, fraud can't be attributed objectively by submitting any fraud proof or ZK proof as part of the execution of the chain state. However, fraud is subjectively visible to any outside observer. As a concrete example, if an oracle reported that 1BTC equals 1USD, which is far from the real truth, this fraud is obvious from the outside. A naive solution extension would be to require Ethereum validators to participate in such services (for example, the validators vote on price feeds as part of the block proposal) and assume the (super-)majority of Ethereum validators are honest. In case of an adversarial take-over of the majority of Ethereum validators to break this oracle, the social consensus of Ethereum would be expected to subjectively step in to resolve the tyranny-of-majority of the Ethereum validator set. However, the setup phase for Ethereum stipulates the usage of its social consensus only for violation in the context of chain validity and chain liveness. This ensures that Ethereum's social consensus is used only sparingly. Any effort to extend the usage of Ethereum's social consensus for resolving tyranny-of-majority in additional digital tasks will be a violation of the thesis of not overloading Ethereum's social consensus that was agreed upon in Ethereum's setup phase [14]. See fig. 3(b) for an illustration.

Extending the power of social consensus. Our proposal to extend the benefit of social consensus in resolving tyranny-of-majority without overloading Ethereum is as follows:

- **Restaking: Tap into Ethereum's security only for resolving objectively attributable faults.** Recall that Ethereum's security comes from the combination of staking, slashing, and Ethereum's social consensus. Any digital task where fault can be objectively attributed and malicious operators can be penalized programmatically within EVM should use `ETH` restaking in EigenLayer to tap into the deep pool of security offered by Ethereum. In such tasks, the mechanism for dispute resolution can be programmed within the chain state as a smart contract[6]. Correct execution of the dispute resolution is subsumed into chain validity. Therefore, cryptoeconomic security for digital tasks with an objectively attributable fault can uniquely tap into Ethereum's cryptoeconomic guarantees without overloading the social consensus of Ethereum.
- **Intersubjective staking: Tap into `EIGEN`'s social consensus for resolving intersubjectively attributable faults.** The core idea of `EIGEN` is to recognize that the principle of intersubjectivity can apply to digital tasks other than that of the chain itself. In particular, `EIGEN` expands the idea of subjective choice from choosing a fork of chain to choosing a fork of token for tasks that have faults intersubjectively attributable in nature. Resolving any potential faults in such tasks due to tyranny-of-majority requires the use of the technology of social consensus. In such tasks, `EIGEN` offers the tooling for unconditional slashing of the `EIGEN` staked by malicious operators on EigenLayer by tapping into the power of `EIGEN`'s social consensus to canonize the fork of `EIGEN` token that penalizes the malicious operators.

2.3.1 The structure of Forking Tokens

The core problem with intersubjective faults is that the right answer is unknown inside the reference frame of blockchain, whereas from the vantage point of observers outside the system, the right answer is known. The system needs to have a way to utilize this exogeneous information to impose a cryptoeconomic penalty on the violators.

First, we highlight an example of a forking token based on prior work [6, 4] and its implementation, [7]. Augur has an ERC20 token called the REP token (short for reputation), a forking token. The REP token is staked to report what happened in the real world to resolve the results of prediction markets (where the answer is binary, i.e., true or false). The REP token holders are slashed if they disagree with a majority of token holders, but if a certain-sized minority of REP holders dissent, they can initiate a forking event. The forking event now creates two ERC20 tokens, each of which represents a new forked version of the token corresponding to true/false (let's call it REP1 and REP2, say). Every REP holder now has to decide if they want to choose REP1 and REP2 tokens. Furthermore, future Oracle users (prediction markets) need to specify whether they want to utilize REP1 or REP2 as the token staked. Similarly, exchanges and other financial services will need to choose between REP1 and REP2. The set of players involved in valuing whether REP1 or REP2 is the right fork is not fixed, and a priori known, set. All of these together ensure that if REP1 clearly represents the truthful answer, then everyone will choose it, and all of the values of REP will filter into REP1. This will make the value of REP2 go to zero. If this were not the case, the whole machinery of the market can then be leveraged (for example, short the wrong fork) and can be engaged permissionlessly to get the market to converge on the right answer. One way of thinking about this token model: it utilizes market forces to ensure that everyone converges to the intersubjective truth.

We note that while Augur made major progress in how to induce cryptoeconomic penalty for intersubjective faults, many open problems remain:

1. Specialization: Because Augur was application-specific (focused on prediction markets), it had a simple mechanism to measure the total value at risk (profit-from-corruption). Generalization beyond this special example requires a mechanism to solicit the value-at-risk for arbitrary computaion.
2. Fork-aware: Every holder of the REP token needs to be aware of forking and claim the right version of REP1 or REP2 tokens, even if they were not participants in the oracle.
3. Parasitic (free-riding) behavior: It was possible for other protocols to build parasitic prediction markets on top of the prediction market. More severe than the Augur protocol losing fees, the protocol loses security properties in this setting.

In this paper, we design `EIGEN` to circumvent all these problems. The token achieves the properties of (a) universality in being able to use the token for any intersubjective fault, (b) isolation of defi from forking, (c) metering the cost of social consensus, and (d) compensating users for their loss due to faults.

[6] Any evidence of objective fault can be submitted as fraud proof to the chain. Here, the assumption is that at least one honest participant in Ethereum's social consensus will submit this evidence and will be accepted within the censorship-resistance period of Ethereum.

2.4 Universality of EIGEN token

In this section, we describe how EIGEN token achieves universality. There are two distinct components to this: (i) conceptual universality and (ii) technical universality.

The Problem: Imagine someone stakes an existing ERC20 token and proclaims this token should be forked if something bad happens in my newly designed oracle. This will not work because the token does not have the setup phase to permit it. The setup phase of the ERC20 token comprises of what the people believe it will do (the concept of the token), and what functionalities the token allows. Both of these parameters are fixed when the token is created. We call this the **setup phase** of the token. If someone claims the token should be forked, neither has this conceptual authority nor the technical ability (as forking the token will have downstream consequences on other applications of the token, which cannot be containerized). We note that, even when the token has some amount of functional universality, it may not have conceptual universality: for example, even though the REP token is forkable, it will not fork for other applications. The social consensus around the REP token has been created for utilization in the prediction market when **it was setup**.

The core setup of the EIGEN token will be that it will be the universal intersubjective work token. This means, that when this token is staked in EigenLayer and stakers opt into covenants from AVSs, the token has conceptual universality to fork for this purpose if the stakers are viewed as malicious. We note that the setup phase of the token should also specify where each AVS specifies its slashing conditions, as these can be thought of as amending the setup.

Exclusions: The EIGEN token is designed to adjudicate faults where there will be near-universal agreement among observers, i.e., where there is intersubjective consensus. It is not designed for problems which are highly subjective in nature. Examples of excluded types of interactions include "what is the predicted price of this NFT in 20 days?", or "Is Paris the most beautiful city?".

A grounding principle in the EIGEN token is to "slash stakers" only when their fault is beyond a reasonable doubt. Under these conditions, slashing requires large collusion as well as a large potential loss of funds, and this suffices to ensure honest behavior. Thus we expect EIGEN forking to be a *last-resort nuclear option*, but nevertheless one that is needed to ensure that the stakers behave correctly.

As EIGEN features its own social consensus for resolving intersubjective faults of intersubjective tasks, this responsibility is now no longer being imposed on Ethereum's social consensus, thus, not overloading it. Actively Validated Services (AVSs) on EigenLayer tap into the ETH quorum for objective faults but into EIGEN quorum for intersubjective faults. See fig. 3(c) for an illustration. Now there could be forking in EIGEN for resolving intersubjective faults without requiring forking the chain state of Ethereum itself.

2.5 Isolation between Staking and Fork-Unaware Applications

2.5.1 Existing PoS protocols

PoS protocols typically feature only a limited amount of their native tokens being staked at any given time, while the remaining tokens are utilized for other fork-unaware purposes, such as DeFi or users holding the staking token without deploying it. A similar scenario is expected to happen with EIGEN. In this section, we will explain how the intersubjective (forking) nature of EIGEN makes the interaction complex. To understand this interaction, we will understand the interaction in two existing protocols.

We note that in chains like Ethereum, the native ETH token forks along with the chain. So as long as the ETH is deployed into DeFi protocols that are deployed in the same fork of the chain, the protocols always include the right version of the ETH. We coin the term *intersubjective frame-of-reference*, in order to understand from which frame forking is happening. Protocols and (standard) tokens that share the chain will fork together when the underlying chain forks, so they can be said to inherit the intersubjective frame-of-reference from the chain. With EIGEN, the token has its own intersubjective frame-of-reference, so if the token forks, DeFi protocols may not know that it has forked, and will need to handle it with care. This consideration is shown in Fig. 6a.

2.5.2 Solid representation: Isolation barrier between intersubjective staking and DeFi

Interaction between intersubjective staking of EIGEN and using EIGEN of DeFi is complicated. There can be forking of EIGEN but there is no forking of the chain state of Ethereum and, by extension, the rest of the

application layer on Ethereum. This leads to a complicated scenario where EIGEN's social consensus has agreed upon a new canonical fork, but the EIGEN which were locked in non-staking applications such as DeFi would still represent the stale version of EIGEN token from before the fork. In order to solve this problem, we propose creating an isolation chamber between the DeFi use cases of EIGEN and the staking use cases of EIGEN. We consider EIGEN to have a *solid representation* if the tokens belonging to any fork of EIGEN can be redeemed for the same number of tokens belonging to any of its descendent fork of the EIGEN token that the holder desires. With solid representation, an EIGEN holder can afford to be inactive for an indefinite time or be locked in long-term DeFi positions but still be able to redeem tokens from any of the descendent forks that they agree with. This property is akin to saying that non-staking use of EIGEN can be unaware of any fork in EIGEN. See fig. 6a for an illustration.

In order to achieve solid representation for EIGEN, we proposed an intersubjective forking protocol with two tokens: EIGEN and bEIGEN (i.e., backing EIGEN). bEIGEN will be used solely for intersubjective staking and EIGEN will be used for non-staking applications such as DeFi. The forking protocol is designed such that EIGEN is a wrapper over bEIGEN with special clauses on when and how wrapping and unwrapping can be executed which imparts the behavior corresponding to a solid representation. Even if bEIGEN is subjected to intersubjective forking, any EIGEN holder who is using it for non-staking applications doesn't have to worry as it can redeem the forks of bEIGEN at any time later. For any developer of a non-staking application, as long as their surface area of interaction with the intersubjective forking is limited to only using EIGEN, the application can be agnostic of any intersubjective forking of bEIGEN.

2.6 Compensating for the Cost of Intersubjective Consensus

We observe that intersubjective (social) consensus incurs a cost, and if it is not metered properly gives rise to two kinds of problems: (i) if it is free, then it might lead to griefing attacks, where someone tries to abuse social consensus without incurring a cost, and (ii) if the users who participate in social consensus are not sufficiently compensated, then it might lead to users not participating in it. These two problems lead to core principles: (i) anyone who tries to fork the token should incur a cost, and (ii) some portion of the cost should be distributed to users who participate in social consensus.

We formalize this by requiring two types of economic costs in the original token $\texttt{bEIGEN}_1$ and the forked token $\texttt{bEIGEN}_2$:

- **Deflation-per-fork** (DPF). DPF represents the minimum fraction of total $\texttt{bEIGEN}_1$ that should be tagged as malicious (see sec. 2.7.2 for a description on this tagging) and burnt while creating the fork $\texttt{bEIGEN}_2$. If $\texttt{bEIGEN}_2$ ends up being considered as the canonical fork, then each unit of $\texttt{bEIGEN}_2$ would appreciate by $\frac{1}{1-\texttt{DPF}}$ multiplication factor. If the challenger intersubjective challenge isn't slashing at least DPF fraction of total $\texttt{bEIGEN}_1$, then that challenge will not be considered legitimate.
- **Commitment-per-fork** (CPF). CPF represents the minimum fraction of total $\texttt{bEIGEN}_1$ that needs to be committed by the challenger to trigger the intersubjective challenge. Here, commitment refers to the fact that the challenger will burn CPF fraction of $\texttt{bEIGEN}_1$ so that they cannot access this portion of tokens if the original token $\texttt{bEIGEN}_1$ continues to be considered the canonical fork. The challenger will, in return, be able to redeem these tokens if $\texttt{bEIGEN}_2$ is the correct version of the fork. An important point to note is that the challenger has to be convinced of the intersubjective fault that has happened as it is burning its $\texttt{bEIGEN}_1$ holdings and is betting on the value extracted from the $\texttt{bEIGEN}_2$ that it is receiving as part of the challenger reward. So, anyone harmed by the intersubjective fault in the AVS has the highest incentive to raise the challenge.

Both the parameters DPF and CPF are enshrined in the setup phase of the EIGEN token. Setting the values of DPF and CPF properly is critical to ensuring the system's healthy functioning.

Here, DPF pays off the cost for the social consensus to switch from $\texttt{bEIGEN}_1$ to $\texttt{bEIGEN}_2$. This switching cost comes in the form of the need for the social consensus to validate the claim of intersubjective fault, gas cost of redeeming $\texttt{bEIGEN}_2$, gas cost of against staking $\texttt{bEIGEN}_2$, etc. On the other hand, CPF acts as a deterrence against any adversary raising vacuous intersubjective challenge. Even in the case of vacuous intersubjective challenges, anyone participating in the social consensus must verify the fault's authenticity before rejecting the challenge, which incurs a cost. CPF serves as a mechanism to pay off that social cost. See fig. 5 for an illustration and sec. 3.3 for a detailed analysis on setting up the value for DPF and CPF.

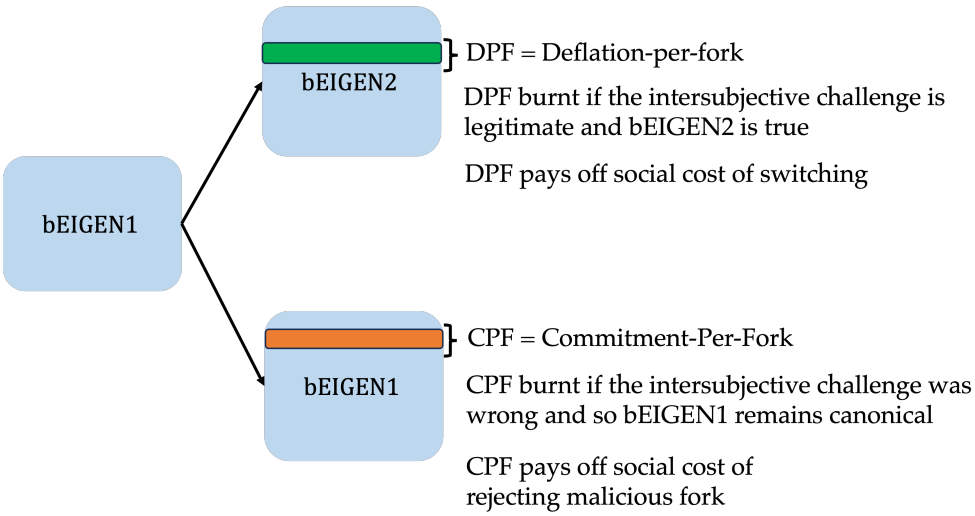


Figure 5: Simple illustration of CPF and DPF. Here $\texttt{bEIGEN}_2$ is a fork of $\texttt{bEIGEN}_1$ as a consequence of an intersubjective challenge.

2.6.1 Setting values for CPF and DPF

CPF pays off for the social cost of a malicious fork, acting as a deterrent against an adversary raising vacuous challenges and requiring everyone in the social consensus to verify the authenticity of the challenge and reject it. If CPF is set too high, large off-chain coordination is needed to pool enough $\texttt{bEIGEN}_1$ for raising the intersubjective challenge. This sets an upper bound on the value of CPF.

On the other hand, DPF pays off the cost for the social consensus to switch from $\texttt{bEIGEN}_1$ to $\texttt{bEIGEN}_2$. This switching cost comes in the form of the need for the social consensus to validate the claim of intersubjective fault, gas cost of redeeming $\texttt{bEIGEN}_2$, gas cost of against staking $\texttt{bEIGEN}_2$, etc. Deflation is created because some stakes are slashed in the forked token. Therefore, if DPF is set too high, it imposes a requirement on the AVS that at least DPF fraction of stake is attributable and slashable when a fault happens on the AVS. This sets an upper bound for DPF.

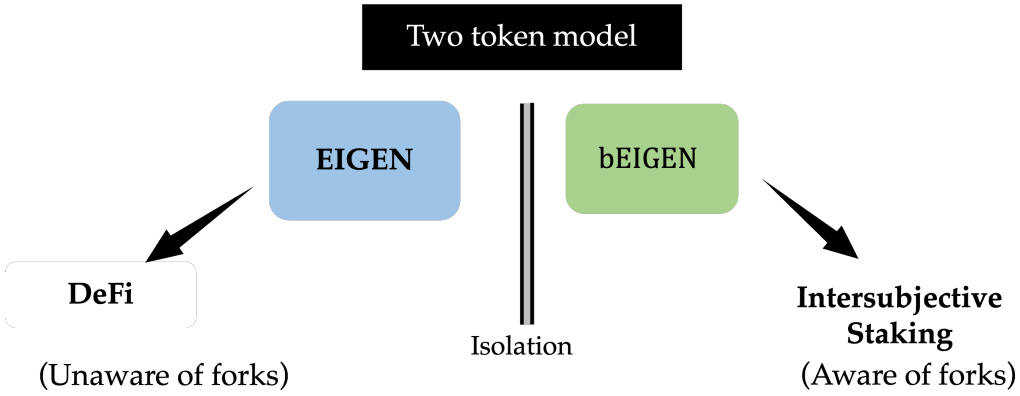
2.7 Overview of Protocol for Intersubjective Staking on EigenLayer

Next, we will provide a very high-level explanation of intersubjective staking on EigenLayer. Note that we present the baseline design in sec. 2.7.1 for the purpose of only motivating the two-token design. We won't be ever implementing the baseline design.

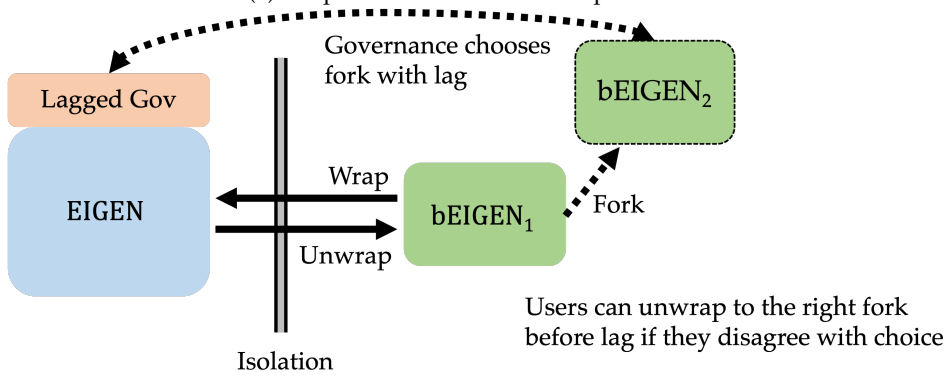
2.7.1 Baseline Design

We start with a baseline design for the intersubjective protocol that features a single token model. Note that this baseline design doesn't offer solid representation, but we describe to motivate the necessity of a two-token design. This is described as follows:

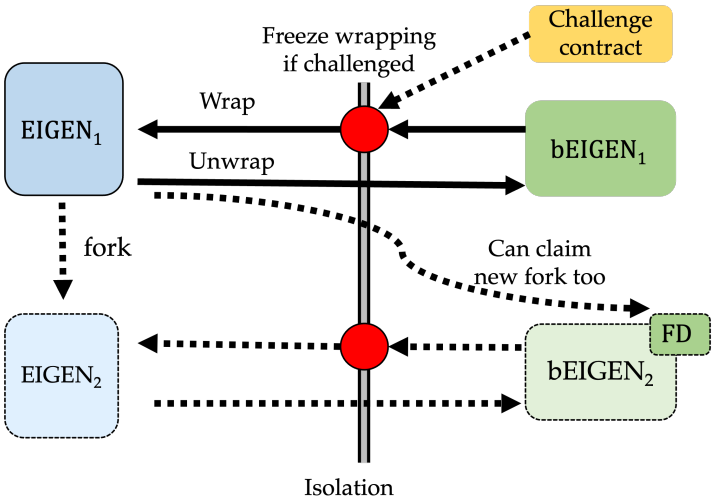
1. **Intersubjective staking with EIGEN.** Stakers stake the EIGEN for staking into EigenLayer. By staking EIGEN, the staker is opting to respond to digital tasks and subjecting its stake to intersubjective slashing. Here, EIGEN is an ERC20 token. The staking contract ensures that withdrawal from staking is subject to a withdrawal lag - this lag is important as it allows a time period for the token to fork before affecting other holders.
2. **Intersubjective forking and intersubjective slashing.** In order to raise a challenge against an intersubjectively attributable fault, the challenger must burn some amount of EIGEN and intersubjectively fork EIGEN. *Intersubjectively forking* the EIGEN requires the challenger to deploy: (a) a new ERC20 contract that represents a new fork of EIGEN (recall that the existing EIGEN was also an ERC20 contract) and, (b) a fork distributor (FD) smart contract that specifies how stakers and other holders of existing EIGEN can redeem tokens from this new ERC20 contract. We call this new ERC20 contract a fork of EIGEN. This FD should allow stakers delegated to honest operators and the challenger to claim tokens from this new fork of EIGEN, whereas stakers delegated to malicious operators should not be able to redeem tokens from the new fork. For simplicity of explanation, we call the tokens from pre-forked EIGEN as $\texttt{EIGEN}_1$ and post-forked EIGEN as $\texttt{EIGEN}_2$.



(a) Simple illustration of solid representation.



(b) A simplified illustration of V1. Here, $\texttt{bEIGEN}_2$ is fork of $\texttt{bEIGEN}_1$ after the intersubjective challenge is raised.



(c) A simplified illustration of V2. Here, $\texttt{bEIGEN}_2$ and $\texttt{EIGEN}_2$ are corresponding forks of $\texttt{bEIGEN}_1$ and $\texttt{EIGEN}_1$ after the intersubjective challenge is raised. The red dot here represents the control for freezing wrapping.

Figure 6: Overview of the intersubjective forking protocol that is being enabled by `EIGEN`.

3. **Redemption mechanism.** The FD also specifies how a non-staked holder can redeem the fork. Any holder of $\texttt{EIGEN}_1$ is able to redeem $\texttt{EIGEN}_2$ by locking their $\texttt{EIGEN}_1$ in the FD of $\texttt{EIGEN}_2$ for a period of T_{lock} slots. This redemption is permitted for a period of T_{redeem} slots after the attack has happened. Note that here it is necessary to have $T_{lock} > T_{redeem}$ in order to avoid the issue of double redemption[7].

By including the details on who is not able to redeem $\texttt{EIGEN}_2$ tokens, the challenger is effectively slashing certain existing stakers who had staked $\texttt{EIGEN}_1$. This slashing is *intersubjective* in nature as others might not agree with the challenger and can ignore the $\texttt{EIGEN}_2$. In such a case, $\texttt{EIGEN}_1$ will continue to be held as the canonical fork of EIGEN by the social consensus. Therefore, we call such a slashing as *intersubjective slashing*. Ideally, the challenger should deploy the $\texttt{EIGEN}_2$ and its FD after deliberation with the rest of the social consensus.

Baseline design lacks "solid representation." The baseline design lacks "solid representation." If an $\texttt{EIGEN}_1$ holder is not active for the period of T_{redeem} slots after the malicious attack was executed by operators, then the holder will never be able to redeem $\texttt{EIGEN}_2$. Therefore, the baseline design doesn't offer a solid representation. In the next few sections, we will give a high-level outline of an intersubjective forking protocol that will progressively achieve this property. We also want the intersubjective protocol to be resilient against malicious behavior of the security council[8] of the staking contract. Towards that end, we will start by describing a protocol V1 that guarantees neither solid representation nor is safe against corruption of the Security Council but lays the foundation of the eventual protocol that guarantees both these properties. We build V2 on top of V1 and it guarantees solid representation. Finally, in V3, we achieve resilience against the malicious behavior of the Security Council.

2.7.2 V1: Two-token model for the intersubjective staking protocol

Following is the high-level overview of the V1 version of the intersubjective forking protocol on EigenLayer. For a full description of V1, please refer to Appendix. A.1.

1. There are two separate tokens: bEIGEN (backing EIGEN) is used for staking while EIGEN is used for non-staking purposes, such as DeFi. The bEIGEN can be subjected to forking. Both bEIGEN and EIGEN are ERC20 contracts. The latter features governance that decides which fork of bEIGEN is configured to back the EIGEN.

2. **Wrapping and unwrapping.** Anyone can instantly wrap 1 bEIGEN to 1 EIGEN and vice versa. Only the current fork of bEIGEN backing the EIGEN will be considered for wrapping and unwrapping (see sec. A.1.3 for details).

3. **Triggering the intersubjective forking.**

   (a) **Social-deliberation.** During their respective setup phase, each AVS must specify and implement a social-deliberation mechanism that anyone can use (a) for raising the alarm among the EIGEN's social consensus about any intersubjective fault during the execution phase and/or (b) for subsequent deliberation to resolve the intersubjective fault. In the event of a "potential" intersubjective fault during the execution phase, anyone would be able to alert the social consensus of EIGEN about the fault. At the end of the mechanism, all the honest members of the social consensus who actively participated should be able to form their own subjective opinion on whether the fault is genuine, converge on who the culprits are, and agree on who will be responsible for raising the challenge[9]

   (b) **Challenge.** Raising a challenge requires launching a new ERC20 contract representing a fork of bEIGEN, a new challenge contract for the new fork, a new fork-distributor (FD) contract specifying which operators were malicious and how much the challenger will be rewarded. For explanation purposes, we refer to the pre-forked bEIGEN as $\texttt{bEIGEN}_1$ and post-forked bEIGEN as $\texttt{bEIGEN}_2$. The FD of $\texttt{bEIGEN}_2$ must prevent the $\texttt{bEIGEN}_1$ stakers delegated to operators identified as malicious/culprits during the fork-choice mechanism from being able to redeem $\texttt{bEIGEN}_2$. Furthermore, the challenge

[7] By "double redemption," we mean that for the same 1 unit of $\texttt{EIGEN}_1$ token, the holder or staker shouldn't be able to redeem more than 1 unit of $\texttt{EIGEN}_2$ token.

[8] We partition the governance of the staking contract into two parts: (1) governance for regular contract upgrades for adding new features and will have lags in place for these upgrades, (2) security council for triggering emergency upgrades or pausing. Clearly, the security council has a lot of power.

[9] The challenge against an attack needs to be raised within a limited window of length T_{chal} after the attack was socially agreed to have happened. There is potentially a need for the feature of a buffer period in the event that the selected challenger doesn't raise the challenge. In this buffer period, anyone should be able to raise the challenge.

must tag at least DPF fraction of total $\texttt{bEIGEN}_1$ as malicious. The challenger is also required to submit a bond that amounts to at least CPF fraction of total $\texttt{bEIGEN}_1$ tokens. See sec. A.1.2 for details[10] on the forking and see sec. 2.6 for details on CPF and DPF.

(c) **Configuring the EIGEN contract.** If the challenge is correct from the perspective of the governance of EIGEN, the governance then must queue an upgrade transaction to reconfigure EIGEN to be backed by $\texttt{bEIGEN}_2$. This upgrade gets executed after a lag. See sec. A.1.3 for details.

4. **Redemption clauses.**

   (a) **For bEIGEN active stakers/holders.** After the challenge is raised, there is a redemption period during which any $\texttt{bEIGEN}_1$ staker or holder and the challenger could redeem $\texttt{bEIGEN}_2$. See sec. A.1.4 for details.

   (b) **For EIGEN holders.** Any EIGEN holder locked in a long-term position in DeFi or inactive can trust EIGEN's governance for selecting the canonical fork of bEIGEN. If the EIGEN holder disagrees with the EIGEN governance's decision to reconfigure, the holder can unwrap to $\texttt{bEIGEN}_1$ within the lag. If the EIGEN holder disagrees with the EIGEN governance's decision not to reconfigure, the holder can unwrap to $\texttt{bEIGEN}_1$ within the redemption period and redeem $\texttt{bEIGEN}_2$. See sec. A.1.5 for details.

5. **Staking and Withdrawal.** Anyone can either use the UI to stake EIGEN or directly deposit bEIGEN with the staking contract. Under the hood, using the UI first unwraps EIGEN to the fork of bEIGEN that the EIGEN contract is currently configured to and then deposits the bEIGEN with the staking contract. An opposite flow is executed in the case of withdrawal. Staking is not subjected to any lag while withdrawal is subjected to a lag. See sec. A.1.1 for details.

See fig. 6b for a high-level illustration.

Assumptions needed for security. V1 makes the following assumptions:

1. Either Governance of EIGEN wrapping contract is honest, *or,* EIGEN holders are engaged in non-staking activities and have not locked their token for over a certain period.
2. Security council governing EIGEN staking contract is honest.

V1 doesn't offer solid representation. Elaborating more on the first assumption, in the event of a reconfiguration decision by EIGEN wrapper's governance with which an EIGEN holder disagrees, the holder has a limited period to unwrap its EIGEN tokens to the bEIGEN fork of its choice. This leads to a lack of solid representation in V1. For example, after there was a fork from $\texttt{bEIGEN}_1$ to $\texttt{bEIGEN}_2$, consider that EIGEN wrapper's governance decides to reconfigure to $\texttt{bEIGEN}_2$ and queues an upgrade that will be executed after the governance lag. If the EIGEN holder disagrees with the governance decision but is locked in a DeFi position that will be unlocked only after the governance upgrade is executed or is inactive during this period, the EIGEN holder won't be able to unwrap to $\texttt{bEIGEN}_1$ before its EIGEN gets backed by $\texttt{bEIGEN}_2$. Now, consider an opposite scenario where the EIGEN wrapper's governance makes the decision not to reconfigure to the new fork $\texttt{bEIGEN}_2$, but the EIGEN holder disagrees with that decision. If the EIGEN holder is inactive or locked in a long-term DeFi position that will be unlocked only after the redemption period of $\texttt{bEIGEN}_2$ is over, the EIGEN holder will not be able to redeem $\texttt{bEIGEN}_2$ ever.

Trust assumption on Security Council. Furthermore, there is a trust assumption on the Security Council of the staking contracts. The Security Council can arbitrarily change the unstaking period in staking contracts and steal value from the staked bEIGEN tokens. We will remove these shortcomings in the next two versions of the protocol.

With some additional upgrades in V2, as described in the next section, solid representation can be achieved.

2.7.3 V2: Getting solid representation

In addition to the features described for V1, we propose certain new features to achieve solid representation in V2. For a full description of V2, please refer to Appendix. A.2.

1. **Wrapper-fork.** The wrapper contract EIGEN is now an immutable contract; that is, it has no governance. This immutability requires that the wrapper EIGEN contract can now also be forked socially,

[10] See sec. 3.3.3 for details on cryptoeconomic guarantees around multiple AVSs using intersubjective staking

just like bEIGEN. A legitimate intersubjective challenge now also requires forking EIGEN by deploying a new ERC20 contract, which is in addition to the requirement from V1. We call this forking of EIGEN wrapper contract as *wrapper-forking*. For explanation purposes, we refer to pre-forked EIGEN and bEIGEN as $\texttt{EIGEN}_1$ and $\texttt{bEIGEN}_1$, respectively, and post-forked EIGEN and bEIGEN as $\texttt{EIGEN}_2$ and $\texttt{bEIGEN}_2$, respectively. See sec. A.2.2 for details.

2. **Freeze wrapping.** Raising an intersubjective challenge now also requires freezing any further wrapping of $\texttt{bEIGEN}_1$ to $\texttt{EIGEN}_1$. See sec. A.2.1 for details.

3. **New redemption clause for passive users.** Suppose an $\texttt{EIGEN}_1$ holder unwraps to $\texttt{bEIGEN}_1$ anytime after the redemption period of $\texttt{bEIGEN}_2$ is over. This unwrapping action is recorded in the storage of the $\texttt{EIGEN}_1$ contract. Now, this holder can just submit the proof of this unwrapping to the FD of $\texttt{bEIGEN}_2$ for redeeming $\texttt{EIGEN}_2$. See sec. A.2.4 for details.

See fig. 6c for a high-level illustration.

Achieving solid representation. With the additional redemption clause in V2, an $\texttt{EIGEN}_1$ holder can now choose to be inactive for a long time or be locked in long-term Defi positions. In the event of a fork, when the $\texttt{EIGEN}_1$ holder comes back online or gets back its $\texttt{EIGEN}_1$, the holder can still redeem $\texttt{EIGEN}_2$ if it agrees with the fork. The holder can just unwrap its $\texttt{EIGEN}_1$ to $\texttt{bEIGEN}_1$ and use the pointer to that unwrapping action to redeem $\texttt{EIGEN}_2$. See sec. C for details.

In Appendix. A.2.6, we provide examples of how DeFi protocols like lending protocols and DEXs can interact with the EIGEN's solid representation that is being enabled by V2.

2.7.4 V3: Making intersubjective staking protocol resilient to corruption of security council

With respect to V2, V3 features a modification in staking and withdrawal to make the protocol resilient to corruption in the security council.

Staking and Withdrawal. Depositing bEIGEN to the staking contract requires depositing the bEIGEN tokens to an immutable gateway contract with a fixed unstaking period. The gateway contract holds the bEIGEN tokens and notifies the staking contract about the deposit which only does accounting. Any withdrawal from staking would be subjected to an immutable unstaking period by the gateway contract. In the event of corruption of the Security Council, anyone can submit a bond to raise a challenge in the gateway contract that freezes all withdrawals. The bond should amount to at least the social cost of disruption due to the freezing of all withdrawals.

For further details on the full description of V3, please refer to Appendix. A.3.

2.7.5 Summary

The following table summarizes the differences between different protocols:

	Solid representation	Resilient against a malicious security council?
V1	No	No
V2	Yes	No
V3	Yes	Yes

3 Intersubjective Staking for Actively Validated Services

EigenLayer enables services beyond consensus to inherit security from a shared security system, which comprises a pool of stake as well as operators, who are equipped to run various software. These services are called Actively Validated Services (AVSs) and there is a wide variety of AVSs, including new chains, data availability (DA) layers, oracles, verifier-as-a-service, fast-finality-layer, and coprocessors.

In this section, we talk about how EigenLayer can be used to share security across AVSs using intersubjective staking. When an intersubjective token such as bEIGEN is staked, in the event of a fork in bEIGEN, the AVS has to decide whether it wants to accept the new fork of bEIGEN or not. This leads to a new complexity not

present when staking objective tokens like ETH and other ERC20 tokens in EigenLayer.

In the next section, we consider the case when there is only one AVS using bEIGEN for intersubjective staking and describe how this AVS handles forking of bEIGEN. For the purpose of explanation, we represent the canonical bEIGEN before the intersubjective challenge as $\texttt{bEIGEN}_1$ and the new fork of bEIGEN arising out of the challenge as $\texttt{bEIGEN}_2$.

3.1 Single AVS with Intersubjective Staking

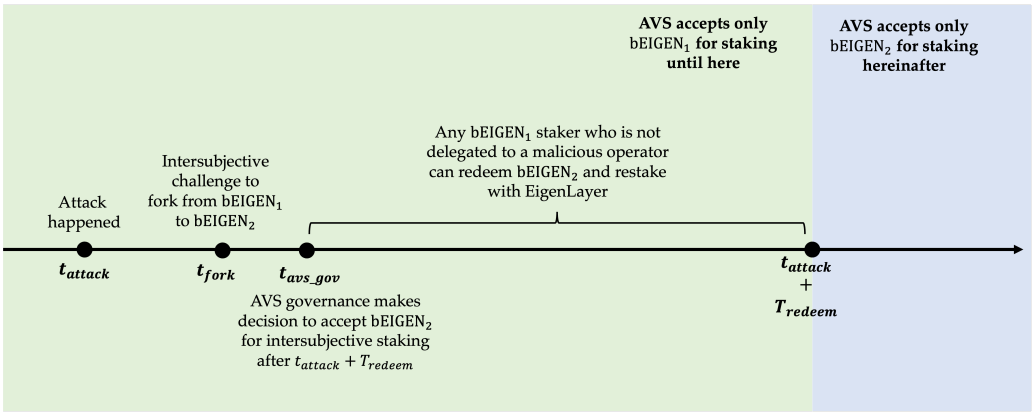


Figure 7: An illustration of how AVSs would handle intersubjective forking,

Consider a single AVS built on the intersubjective staking of bEIGEN. We note that in the two-token system explained earlier, bEIGEN is the staking token, which needs to be fork-aware, and EIGEN is the fork-unaware token, which can be used in non-staking applications. Let us consider the initial case where $\texttt{bEIGEN}_1$ is staked into a single AVS and no other AVSs are present.

Suppose that some of the stakers are attributed to having participated in an intersubjective fault in the AVS. This results in $\texttt{bEIGEN}_1$ being subjected to an intersubjective challenge, and a new fork $\texttt{bEIGEN}_2$. The governance of the AVS now needs to make a decision on whether to accept $\texttt{bEIGEN}_2$ for intersubjective staking or not and to enact that policy at the smart contract level. At any point in time, it is natural for the AVS governance to accept only one fork of the bEIGEN depending on whether the forked token is correct or the original one is correct.

Assume that the governance of the AVS agrees with the intersubjective challenge and the forking. See fig. 7 for a timing diagram. In order to ensure that the stakers have enough time to redeem $\texttt{bEIGEN}_2$ and opt into the AVS with the new $\texttt{bEIGEN}_2$ quorum, there is a need for governance lag before the AVS starts accepting $\texttt{bEIGEN}_2$. Furthermore, the lag before opting into a fork ensures that AVS users who do not agree with the fork can gracefully move on to a different AVS that actually accepts the correct fork. Therefore, the AVS governance must make an announcement that after T_{redeem} slots since the attack was executed:

> After T_{redeem} slots from the attack, the AVS will accept only $\texttt{bEIGEN}_2$ stake (instead of $\texttt{bEIGEN}_1$).

The governance will have to queue a governance upgrade for this decision soon after the intersubjective challenge of bEIGEN is finalized on Ethereum and this upgrade will be executed at T_{redeem} slots from when the attack was executed. During the interval of T_{redeem} slots since the attack happened, any $\texttt{bEIGEN}_1$ staker who agrees with the intersubjective challenge will be able to redeem $\texttt{bEIGEN}_2$ from its associated FD and then again stake with EigenLayer and opt into the AVS. On the other hand, those $\texttt{bEIGEN}_1$ stakers who don't agree with the AVS governance's decision would be able to simply opt out of the AVS.

In the intervening period, $\texttt{bEIGEN}_1$ stakers are still responsible for running the AVS. This is important, because, in case the challenge is malicious and the AVS governance is malicious, the AVS users are still protected by $\texttt{bEIGEN}_1$ staking. On the other hand, if the challenge and the AVS governance are correct, then the $\texttt{bEIGEN}_1$ stakers may engage in continued malicious behavior. Therefore, when performing a cryptoeconomic analysis, it is important to ensure that the penalty incurred by the $\texttt{bEIGEN}_1$ stakers is greater than the harm that they can inflict throughout this T_{redeem} period.

3.2 Defintions of Cryptoeconomic Safety

We first begin with the definition of cryptoeconomic safety which is inspired from [7, 10] and expanded in [15].

Cryptoeconomic Security

For any attacker, the maximal profit extractable from attacking the safety (profit-from-corruption) is smaller than the minimum cost enforced by the system on the attacker (cost-of-corruption).

However, it suffers from a fundamental problem: the profit-from-corruption is non-measurable. The adversary may have perverse incentives outside the system's locus of measurement. Furthermore, this notion does not guarantee that the user actually gets compensated for the value that the user lost if the attack happened.

In [15], a new notion of strong cryptoeconomic safety is defined:

Strong Cryptoeconomic Security

Any user should be compensated a pre-specified amount in the event that the safety guarantee to the user is violated.

Since this definition models the user's guarantee in case the system breaks its guarantee, it does not depend on extraneous assumptions about the adversary's goals and exogenous interests. In [15], it is shown how to achieve this property for an onchain observable fault. In the following sections, we will create mechanisms that achieve these two properties in succession.

3.3 Cryptoeconomic Analysis of Intersubjective Staking

In this section, we will analyze the cryptoeconomic guarantees of the intersubjective staking continuing on the single AVS case.

3.3.1 Pooled security for Single AVS

In the case of when there is only a single AVS using the `EIGEN` quorum, profit-from-corruption is computed by evaluating the total cost that all the consumers on top of the AVS will incur in a period of length T_{redeem} slots if the malicious operators attack. We note that this is because, the AVS is allowed to switch to a different quorum only after the T_{redeem} slots, which is important to ensure AVS users are protected in case of a malicious challenge (see sec.3.1 for details). In case the challenge is correct, though, the malicious operator with $\texttt{bEIGEN}_1$ stake can continue to attack the AVS during this period.

Therefore, for the AVS to be considered to be *cryptoeconomically secure*, it is essential that:

$$\textit{Cost-of-corrupting the AVS} \geq \textit{Profit-from-corrupting the AVS in any interval of length } T_{redeem} \textit{ slots} \tag{1}$$

The cost of corrupting the AVS is measured by the amount of stake that is slashable. For any AVS, it is possible to calculate the fraction of stake that is attributable as malicious if there was any safety violation in the AVS. We call this parameter *Fraction-slashable-for-AVS*, emphasizing that this parameter depends on the structure of the AVS. For example, for a standard BFT consensus protocol with quorum threshold $\frac{2}{3}$, the *Fraction-slashable-for-AVS* is $\frac{1}{3}$ [16]. Therefore, we have:

$$\begin{aligned} \textit{Cost-of-corrupting the AVS} &= \textit{Stake slashable for AVS safety failure} \\ &= \textit{Fraction-slashable-for-AVS} \times \textit{Amount-staked} \end{aligned} \tag{2}$$

Recall that cost-of-corruption is the cost incurred by any adversary to attack a system. From the definition of `DPF`, it is evident that the adversarial stake can be slashed in an AVS only if at least `DPF` fraction of total $\texttt{bEIGEN}_1$ has behaved maliciously in the AVS. Therefore, we must have

$$\textit{Fraction-slashable-for-AVS} \times \textit{Amount-staked} \geq \texttt{DPF} \times \textit{Value}(\texttt{bEIGEN}_1), \tag{3}$$

$$\iff \textit{Fraction-slashable-for-AVS} \times \textit{Fraction-staked} \geq \texttt{DPF}. \tag{4}$$

Satisfying this condition will ensure that a "socially" legitimate intersubjective challenge can be raised to slash adversarial $\texttt{bEIGEN}_1$ stake via intersubjective forking.

3.3.2 Attributable Security for Single AVS

An important caveat for pooled security is that it is not possible for the EigenLayer protocol to know the profit-from-corruption for the AVS. So, there is no in-protocol guarantee whether the security invariant in eq. 1 is satisfied. Moreover, assuming eq. 1 is satisfied, even if the malicious operators's stake is slashed in the event of an attack, there is no redistribution to the harmed parties: AVS and consumers of AVS. Therefore, we would like to improve the mechanism to achieve *strong cryptoeconomic security*, i.e., honest AVSs should be compensated a pre-specified amount if the safety of the AVS is compromised.

Consider that there is a legitimate intersubjective forking from $\texttt{bEIGEN}_1$ to $\texttt{bEIGEN}_2$. See fig. 8 for an illustration. In the pooled security model, all of the adversarial stake is slashed and burnt. Instead, in the attributable security mechanism, only a fraction of the adversarial stake is burnt by the FD of $\texttt{bEIGEN}_2$ (that is, it remains non-redeemable under $\texttt{bEIGEN}_2$'s redemption mechanism). Instead, the remaining adversarial stake is *redistributed to the harmed AVS*. In the general case of multiple AVSs, a mechanism is needed to apportion the redistribution across the various AVSs. However, in the case of a single AVS, all of the redistributed funds will obviously go to that AVS.

Even though redistribution is the key property we seek here, the mechanism ensures that a portion of adversarial stake is burnt during the fork. This is necessary for mitigating any griefing attack, where an adversary represents all the malicious stake but also holds redistribution claims for the AVS, thus making sure it does not suffer any loss. This strategy becomes unprofitable with the burning mechanism as a portion of the stake is burnt and not returned to the redistribution. Furthermore, a portion of the stake *has to be burnt* to satisfy the deflation condition for the fork and pay off the social cost of forking.

$$\texttt{DPF} \times \mathit{value}(\texttt{bEIGEN}_1) \leq \gamma_{burn} \times \mathit{Stake\text{-}slashable}. \tag{5}$$

$$\iff \texttt{DPF} \leq \gamma_{burn} \times \mathit{Fraction\text{-}slashable\text{-}for\text{-}AVS} \times \mathit{Fraction\text{-}staked} \tag{6}$$

When slashing happens, out of the amount slashed, γ_{burn} is burnt and $\gamma = 1 - \gamma_{burn}$ is redistributed to the AVS. Therefore

$$\mathit{Attributable\text{-}security} = \gamma \times \mathit{Stake\ slashable\ for\ an\ AVS\ safety\ fault} \tag{7}$$

$$= \gamma \times \mathit{Fraction\text{-}slashable\text{-}for\text{-}AVS} \times \mathit{Amount\text{-}staked} \tag{8}$$

For the system to be strongly cryptoeconomically secure, we need the attributable security to be greater than the harm that any user of the AVS can suffer during the entire T_{redeem} slots. This is because switching from one fork of bEIGEN to the other incurs a delay of T_{redeem} slots, and the adversarial stake can cause harm throughout this T_{redeem} period. Therefore, we require the following:

Formal Definition of Strong Cryptoeconomic Security

If an AVS acquires more attributable security than the harm it can suffer from an attack within T_{redeem} slots, then it achieves strong cryptoeconomic security.

$$\mathit{Attributable\text{-}security} \geq \mathit{Harm\text{-}from\text{-}corruption\ for\ AVS\ in}\ T_{redeem}\ \mathit{slots}. \tag{9}$$

Example 1: As a simple example, consider a fast bridge AVS for a rollup, which has a (X, T)-rate-limit, i.e., that it does not allow more than X total transaction value in any T period. Now if the bridge had (X, T_{redeem}) as the rate limit, then we know that the total value transacted by the bridge is less than X during any attack period, and therefore the harm from corruption for the T_{redeem} period is less than or equal to X. If the attributable security is greater than X then this AVS works correctly. For this example, we have the following conditions for strong cryptoeconomic safety.

$$\gamma \times \mathit{Fraction\text{-}slashable\text{-}for\text{-}AVS} \times \mathit{Amount\text{-}staked} \geq X. \tag{10}$$

Example 2: Consider an oracle for price feed that requires any operator participating in the oracle to respond with the price feed at every fixed time. These operators are required to stake $\texttt{bEIGEN}_1$ in order to participate in the oracle (we assumed that the pre-forked bEIGEN is called $\texttt{bEIGEN}_1$ and post-forked bEIGEN is called $\texttt{bEIGEN}_2$). In the event that operators behave maliciously, which impacts the price-feed from the oracle, then $\texttt{bEIGEN}_1$ will be intersubjectively forked to $\texttt{bEIGEN}_2$. Under attributable security, the oracle must beforehand evaluate the profit that any malicious operators can extract by corrupting the price feed over the next period of length T_{redeem} slots (here the oracle is designed not to pause when a challenge is raised but instead continues to use $\texttt{bEIGEN}_1$ as the stake until the redemption period of $\texttt{bEIGEN}_2$ is over). The adversary

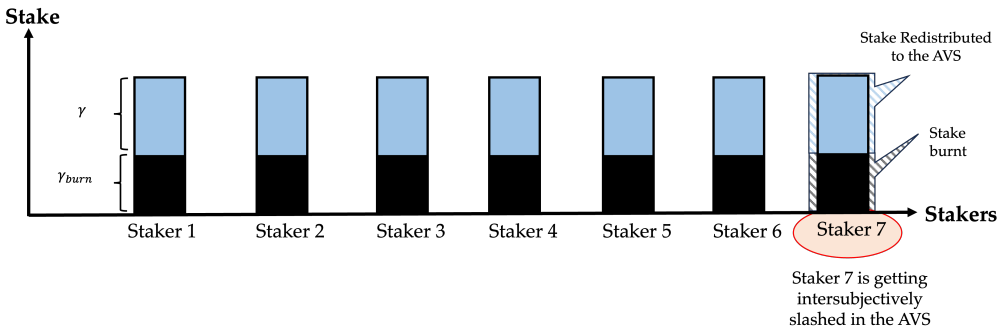


Figure 8: Assume that there are 7 stakers with an equal stake of $\texttt{bEIGEN}_1$ who are opted into the AVS and self-delegated to themselves as an operator. Consider that staker 7 participated in an attack that successfully corrupted the AVS. Here, γ_{burn} represents the fraction of the total stake delegated to an operator that is earmarked for burning if the operator gets slashed. γ represents the fraction of the total stake delegated to the operator that is earmarked to be claimable by the AVS in the event that the operator gets slashed in any intersubjective fault of the AVS.

will extract this profit from the Dapps or co-processors who might utilize this corrupted price feed. The oracle has to purchase attributable security equal to this profit every T_{redeem} slots (assuming this is less than $\gamma \times$ Value of total number of $\texttt{bEIGEN}_1$ staked). The oracle must utilize some form of *circuit breakers* to ensure that, over the course of a period of T_{redeem} slots, the Dapps and co-processors who are sourcing their on-chain price feed from this oracle can be harmed at max the attributable security that it purchased. This ensures all users of the oracle are compensated for the harm that has been caused to them due to the attack.

3.3.3 Multiple AVS Case: Synchronized Quorum Forking

One of the core features of EigenLayer is using the same stake to secure multiple AVSs. This enables capital efficiency while ensuring that the cost-of-corruption in EigenLayer remains greater than the profit-from-corruption. We want to have a similar feature for intersubjective staking.

Consider that there are multiple AVSs that are using $\texttt{bEIGEN}_1$ as their quorum for resolving intersubjective faults in their AVS. Suppose that $\texttt{bEIGEN}_1$ is subjected to an intersubjective challenge for a fault in one of the AVS, say $\texttt{AVS}_k$, which results in a new fork $\texttt{bEIGEN}_2$. Consider any other AVS $\texttt{AVS}_j$. If the governance of $\texttt{AVS}_j$ agrees with both the intersubjective challenge in the context of a fault in $\texttt{AVS}_k$ and with the fork, then the $\texttt{AVS}_j$ governance must make an announcement that:

> After T_{redeem} slots from the attack, the AVS will accept only $\texttt{bEIGEN}_2$ stake (instead of $\texttt{bEIGEN}_1$).

This governance upgrade for this should be queued as soon as the transaction for the intersubjective challenge is finalized. The interesting thing with this policy is that all of the AVS switches to the new quorum at the same time, which is T_{redeem} time slots after the attack.

There are many complex timing games, where an attacker may attack an AVS and then, while that AVS is transitioning to a forked quorum (within the T_{redeem} period), attack a different AVS. We need to make sure that this effect is accounted for in both cases - pooled security and attributable security.

In the next section, we will analyze the pooled security of intersubjective staking under multiple AVSs. There is a general case of where some stakers opt into some AVSs. But, for the simplicity of explanation, here we only consider *Full Staking*: every operator in the $\texttt{bEIGEN}_1$ quorum is opted into every AVS. The set of AVSs who are considered to be part of full staking are assumed to be whitelisted. We leave the analysis for the general setting for future research.

3.3.4 Pooled Security for multiple AVSs under Full Staking

Let us assume that if an EigenLayer operator in the $\texttt{bEIGEN}_1$ quorum behaves maliciously to participate in an intersubjective fault in one AVS, then that operator's $\texttt{bEIGEN}_1$ stake is completely slashed and burnt.

$$\textit{Cost-of-corruption-for-AVS}_i = \textit{Fraction-slashable-for-AVS}_i \times \textit{Amount-staked} \tag{11}$$

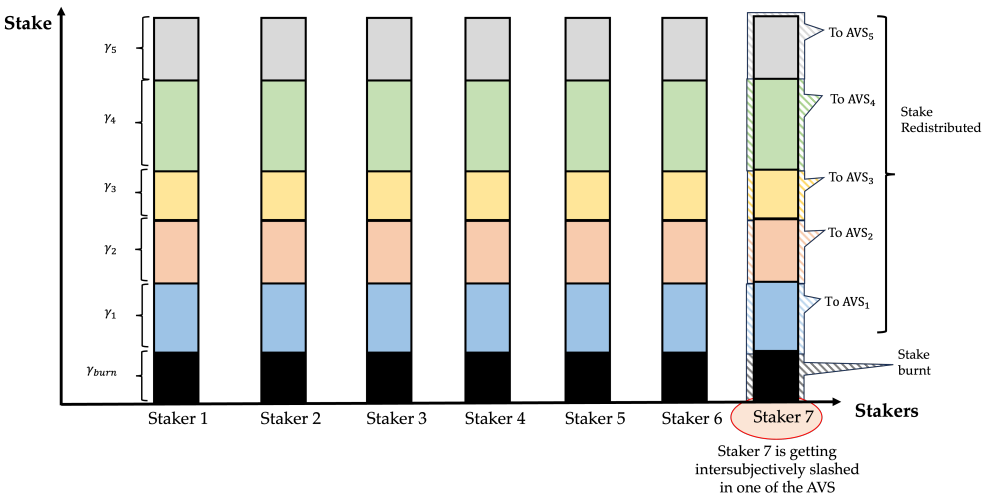


Figure 9: Assume that there are 5 AVSs in full staking and there are 7 stakers with an equal stake of $\texttt{bEIGEN}_1$ who have self-delegated to themselves as an operator. With full staking, all the 7 stakers have opted into all the 5 AVSs who are using the EIGEN quorum for resolving intersubjective faults. If a staker behaves maliciously as an operator in even one of the AVS, its full bEIGEN stake is subjected to slashing. Here, γ_{burn} represents the fraction of the total stake delegated to an operator that is earmarked for burning if the operator gets slashed. γ_i represents the fraction of the total stake delegated to the operator that is earmarked to be redistributed to the AVS_i in the event of the operator being slashed in any intersubjective fault across any of the AVSs enlisted in full staking. Here, $\gamma = \sum_i \gamma_i$ and $\theta_i = \frac{\gamma_i}{\gamma}$.

To attack the system, the attacker must attack at least one AVS. Thus a simple bound says,

$$\textit{Cost-of-system-corruption} = \{\min_i \textit{Fraction-slashable-for-AVS}_i\} \times \textit{Amount-staked} \tag{12}$$

Hence the system is cryptoeconomically safe as long as the following security invariant is guaranteed:

$$\begin{aligned} &\textit{Cost-of-system-corruption} \\ &\geq \sum_{i \in \mathcal{A}} \textit{Profit-from-corrupting } \texttt{AVS}_i \textit{ in an interval of length } T_{redeem} \textit{ slots.} \end{aligned} \tag{13}$$

We need to ensure that the amount slashed and burnt satisfies the DPF constraints for forking, i.e., the amount burnt should be greater than a DPF fraction.

$$\texttt{DPF} \times \textit{value}(\texttt{bEIGEN}_1) \leq \gamma_{burn} \times \min_i \textit{Stake-slashable-for-}\texttt{AVS}_i. \tag{14}$$

$$\iff \texttt{DPF} \leq \gamma_{burn} \times \left(\min_i \{\textit{Fraction-slashable-for-}\texttt{AVS}_i\} \right) \times \textit{Fraction-staked} \tag{15}$$

3.3.5 Attributable Security for Multiple AVSs under Full Restaking

Because there are multiple AVSs, when slashing happens, we need a mechanism to know how to redistribute/allocate the slashed fund *across the different AVS*. One example mechanism to provide attributable security to an AVS is to make it proportional to the security fees paid by a given AVS over a past epoch period. The security fees paid by AVSs to stakers and operators over an epoch period are observable on-chain and so the attributable security held by the AVS is on-chain observable and computable. This is just one example of implementing attributable security for intersubjective staking, but there can be other examples of allocating attributable security.

We will now argue that it is insufficient to compensate only for the AVS that got attacked. If the protocol does that, it is possible for an attacker to attack $\texttt{AVS}_1$, and then wait for a fork of the token to be created with redistribution only to $\texttt{AVS}_1$, and then switch to attacking $\texttt{AVS}_2$, etc. This leads to complex timing races between the attacker and the challenger. We will now propose a simple mechanism that does not have this timing race.

Redistribution mechanism. As was done in single AVS scenario, attributable security under multiple AVS requires that whenever any operator is slashed, γ_{burn} fraction of the stake held by the operator will always

be burnt. The remaining γ fraction of slashed stake is allocated across *all the AVS*. The fraction of this redistribution allocated to an AVS can be determined based on different mechanisms, but for concreteness, we consider simply allocating funds proportional to the security fee paid by the AVS.

Now, we can proceed to calculate how to make the system strongly cryptoeconomically secure.

First we note the amount of stake slashed must satisfy the following relationship:

$$\textit{Amount-slashed} \geq \left(\min_i \{\textit{Fraction-slashable-for-}\texttt{AVS}_i\} \right) \times \textit{Amount-staked}. \tag{16}$$

Out of the amount slashed, $\gamma = 1 - \gamma_{burn}$ is redistributed and out of that, $\texttt{AVS}_i$ gets a fraction θ_i of the redistribution (where θ_i is the proportional fraction of fees paid by the AVS in the particular mechanism).

$$\textit{Attributable-security-for-AVS}_i = \gamma \times \theta_i \times \left(\min_i \{\textit{Fraction-slashable-for-}\texttt{AVS}_i\} \right) \times \textit{Amount-staked} \tag{17}$$

See fig. 9 for an illustration.

Strong Cryptoeconomic Security under Multiple AVS

If AVS_i ensures that the total harm from corruption during the T_{redeem} period is smaller than the attributable security, then AVS_i is strongly cryptoeconomically secure.

$$\textit{Attributale-security-for-AVS}_i = \gamma \times \theta_i \times \left(\min_i \{\textit{Fraction-slashable-for-}\texttt{AVS}_i\} \right) \times \textit{Amount-staked}. \tag{18}$$

3.4 Risks & Mitigations

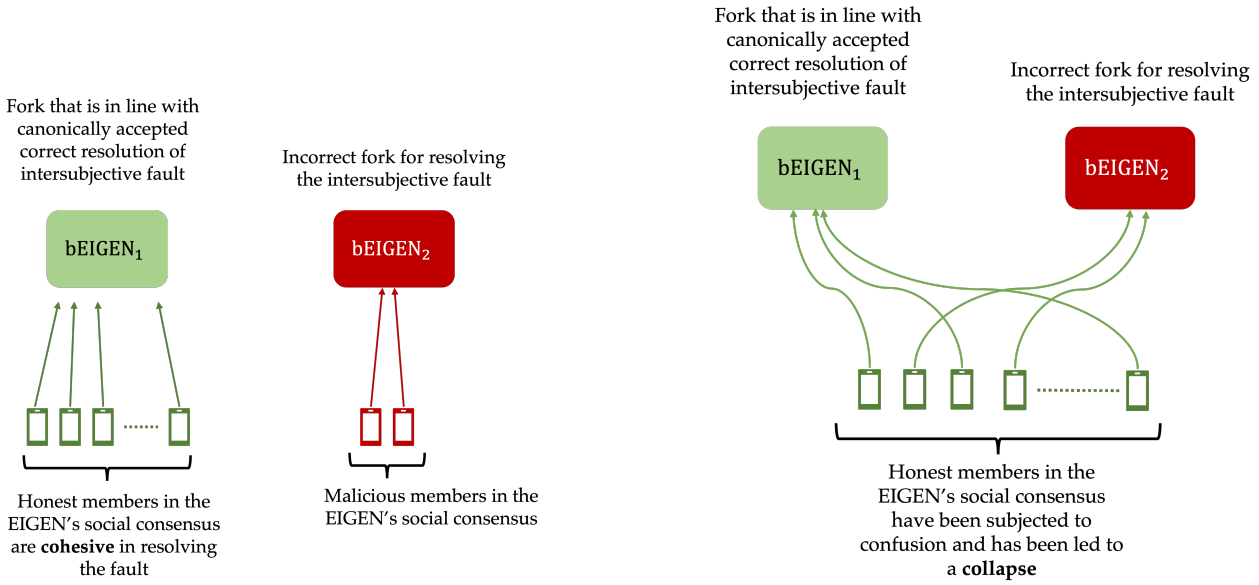
3.4.1 Risk Landscape

As we have seen in sec. 4, intersubjective slashing opens the door toward building cryptoeconomically secured AVSs with intersubjectively attributable faults, enforcing conditions whose verification may depend on information that is external to the blockchain history or temporal in nature. In this section, we will consider some of the risks related to ways in which such a system might break down and the mitigations of these risks.

Intersubjective cohesion. An important requirement for social consensus to be able to resolve intersubjective faults for potentially verifiable digital tasks is that all honest members of social consensus should be in cohesion about what the correct fork of `bEIGEN` after an intersubjective challenge is triggered. This is referred to as *intersubjective cohesion*. See fig. 10a for an illustration. AVSs that rely on intersubjective slashing must focus on making sure that a large body of users in the social consensus of `EIGEN` has direct (first-hand) access to evidence about whether the validators of an AVS have behaved correctly or not. Users who do not have direct evidence can rely on indirect (second-hand) evidence and their trust relationships to determine their own evaluation of the status of any slashing event. For instance, in a Data Availability AVS, users running light nodes will directly observe the AVS validators to ensure that they are properly serving data; for an oracle AVS, users running the light nodes will make observations of the source and observations of price attestations made on the chain and check whether the two are consistent.

Intersubjective fracture. If an AVS doesn't carefully design its light node architecture for users to utilize when resolving intersubjective faults, it presents the risk of a fracture in the social consensus. Here, fracture comes in the form of disagreement among the honest users of social consensus on whether the fault has happened or not and whether it can be properly attributed. We term such a scenario as *intersubjective fracture*. See fig. 10b for an illustration. Such a fracture can happen either by accident or by artifice:

- **Edge cases**: Consider an oracle AVS that specifies that slashing happens when the price feed is incorrect. The intersubjective fracture can arise if it is difficult to decide if the price feed is slightly incorrect or approximately correct.
- **Corruption of Information Sources.** By making use of security vulnerabilities of world information sources, operators, or intermediate internet infrastructure, an adversary may be able to feed disparate observations to sow confusion in the social consensus.
- **Light node monitoring attacks.** If an AVS relies on light node monitoring to detect concurrently attributable faults, it is possible for intersubjective fracture when different nodes have different observations.



(a) Intersubjective cohesion.

(b) Intersubjective fracture.

Figure 10: Cohesion vs. Fracture.

3.4.2 Mitigations

There is no one-size-fits-all approach for mitigating the risk of intersubjective fracture intrinsic to an intersubjective staking ecosystem. Rather, protocol developers need to be disciplined about considering how to insulate their protocols from common sources of breakdown in intersubjectivity. A key general pattern is that the Setup phase of the AVS intersubjective staking should be made comprehensive so that the execution phase can be made as self-serve as possible. A comprehensive setup phase specifies what happens during all imaginable observations accounting for imperfections in the intersubjective process.

We give some examples here.

- **Edge Cases.** As an example, an oracle AVS prespecifies that stakers will only be slashed if the price deviates more than 1% but the AVS is guaranteed that redistribution will happen only if the deviation is more than 5%. Thus if the price deviates between 1% and 5%, there is a buffer zone where social consensus can make decisions gracefully.
- **Information Source Contamination.** By pre-agreeing on which information sources are used, and having routine monitoring of these information sources by light nodes, it is possible to minimize confusion due to equivocation by these sources, as such equivocation is more widely observed.
- **Circumstantial Evidence.** One way to boost the detectability of malicious behavior is to observe the downstream consequences of the said behavior. For example, if an oracle manipulation is used to steal funds from a DeFi protocol, this may leave additional traces that can help arrive at a common view of whether this is an attack.
- **No harm done.** The core principle in the design of the intersubjective system should be that no honest participant should lose funds. Thus, similar to the principle in jurisprudence, a staker should be slashed only if their action is malicious beyond a reasonable doubt. In cases where the AVS software is misbehaving rather than the stakers are misbehaving, the risk is shifted to the AVS (users) and stakers will not be slashed for honestly running bad software.

3.4.3 Graceful Degradation

We note that, intersubjective token forking has a graceful degradation with respect to fracture. This is because, even if two distinct forks are created in the event of intersubjective fracture, AVSs can choose either fork, or even choose to allow both forked tokens for staking with an appropriate weighting across the two tokens. This is fundamentally different from the situation of a chain fork, where all applications need to decide which fork they want to reside on. As an example, if Ethereum forks, a stable coin issuer needs to decide which fork to reside on, and this may play the effect of king-maker, in deciding which fork can survive. This is, because,

when a chain forks, all the applications and tokens built on top of the chain fork in a bundled manner. In the case of token forking, however, it is only the value of the token that is forking according to the perceived correctness. Thus we can view token forking as an unbundling of chain forking to only the value of the token. This makes token forking naturally more graceful with respect to intersubjective fracture. Furthermore, since each forking event requires the burning of certain amount of EIGEN, unlike in chain forks, the forking events are designed to be even rarer.

4 A World With Intersubjective Staking

4.1 Relationship between Intersubjective Staking and Restaking

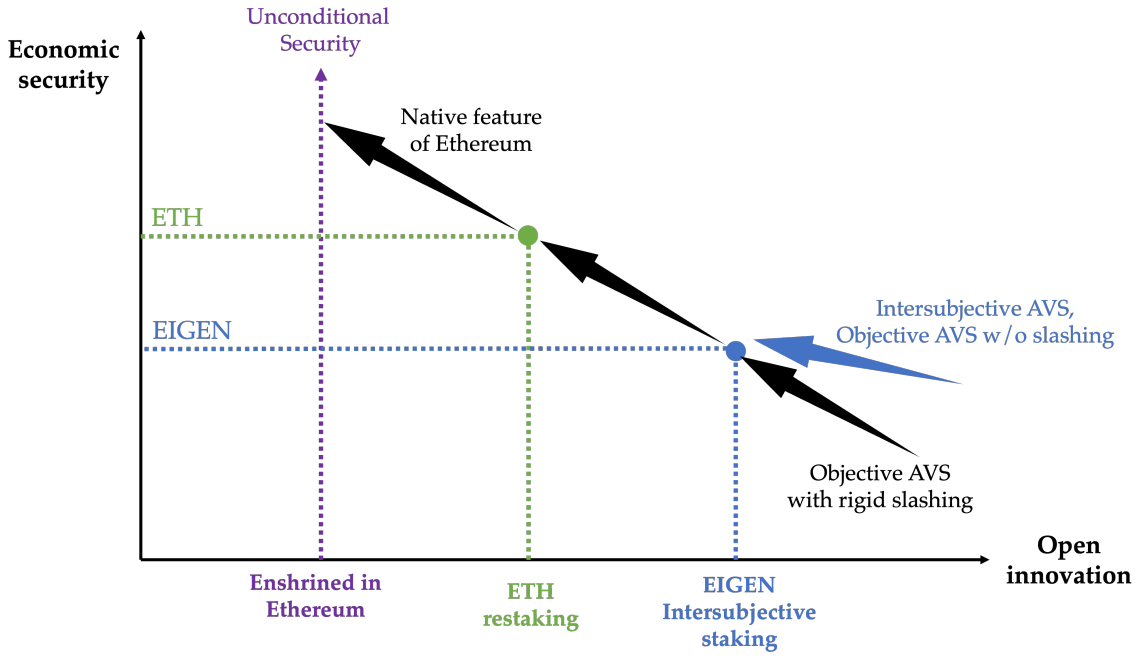


Figure 11: An illustration of innovation-vs-security tradeoff and how a service can progress towards "potential enshrinement".

4.1.1 Complementary roles of ETH and EIGEN

Intersubjective staking and restaking can each fill complementary roles within a staking system such as EigenLayer. Many mature AVS protocols consist of both safety properties that can be secured via objective slashing and liveness or censorship-resistance properties that would previously depend on majority-assumptions which rest on stake decentralization. As discussed above, for such properties, there are often strong advantages in moving to an intersubjective, cryptoeconomic security model for those types of faults. For a service that uses restaking for safety and intersubjective staking for liveness, fees can be split between the two quorums. Furthermore, for core services provided to the Ethereum ecosystem, we envision many services that will use dual staking between ETH and EIGEN, where the native restaking absorbs the decentralization / collusion resistane and operator alignment that comes with ETH restaking, and the EIGEN staking can actually support cryptoeconomic slashing. In this model, the former serves as a mechanism to obtain majority trust from the Ethereum participants, and the latter serves as a mechanism to obtain economic security. The system together absorbs the better of the two safety models, even if liveness requires both quorums separately. For example, an oracle built in this model can be thought of as a halfway house between the enshrined oracle model proposed by Justin Drake [17] and the cryptoeconomic oracle model proposed in [7] and [18].

4.1.2 EIGEN enables higher rate of innovation in objective AVSs

Intersubjective staking can **also** be used to support digital tasks which could in principle be secured via objective fraud proofs, but where doing so would involve a large amount of technical complexity and associated risk. In particular, in envisioning the lifecycle of a new objective AVS, we can identify a progression that

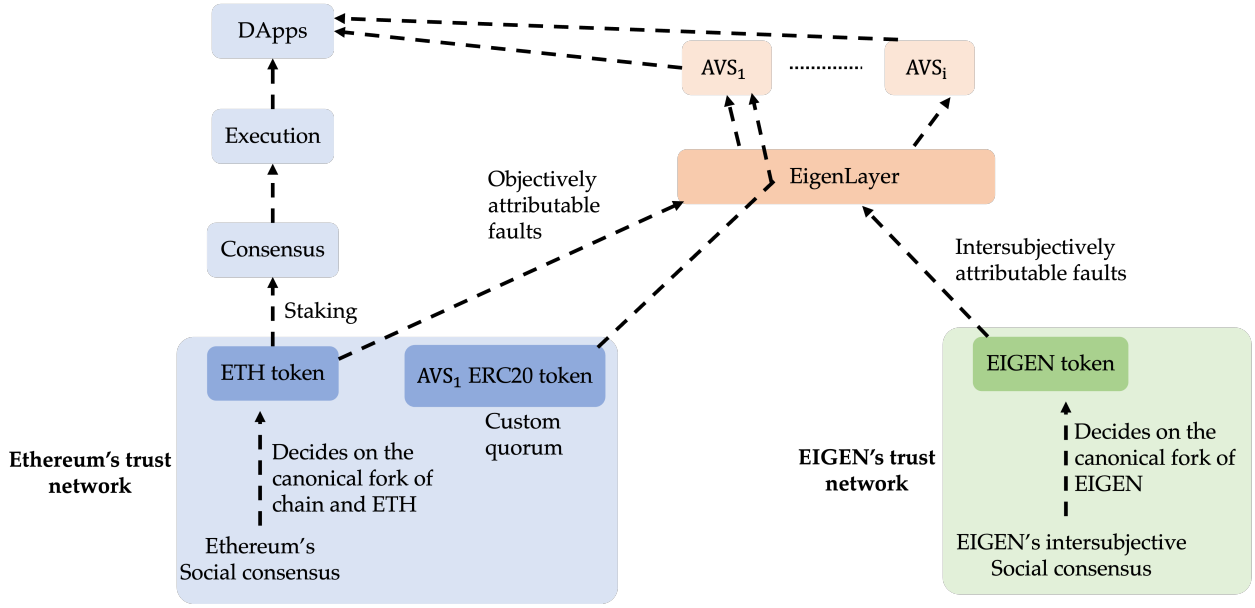


Figure 12: Quorum composition.

leverages intersubjective staking for security early in the bootstrapping phase of the protocol and transitions to restaking or even native protocol adoption as the protocol matures and ossifies.

1. **Intersubjective staking for fast innovation.** Early in the lifecycle of an AVS, intersubjective staking can be used in order to launch the AVS protocol and prove product-market fit without assuming the cost/risk associated with developing and deploying fraud proofs on an accelerated schedule. Most existing rollup projects have illustrated the need for such a phase. Intersubjective staking could allow projects in such stages to proceed with bonafide cryptoeconomic security.

2. **Restaking for higher economic security.** The economic security affordable by intersubjective staking tops out at the staked `EIGEN` market cap. AVSs which require even greater amounts, or less volatile sources, of cryptoeconomic security will need to provision their protocols with fraud proofs in order to support restaking and access the higher amounts of cryptoeconomic security available from Ethereum.

3. **Protocol integration for unconditional security.** AVSs which support credibly neutral capabilities and whose implementations have matured to a highly ossified and stable state may be eligible for an even higher level of security via adoption by the Ethereum protocol itself. This extends unconditionality to the correctness of the operation of the protocol. Obviously this requires that the Ethereum community should want such a feature enshrined into the core protocol.

4.1.3 Quorum Composition

In keeping with the spirit of open innovation, EigenLayer allows AVSs to mix and match these two modalities of objective staking from `ETH` and intersubjective staking from `EIGEN` in addition to utilizing the native AVS tokens for providing additional validation from an aligned community of AVS token stakers. Fig.12 illustrates the scenario when `ETH` restaking should be utilized and when `EIGEN` staking should be utilized.

4.2 Examples

4.2.1 Grounding Example: Data Availability

In this section, we will provide a high-level overview of the architecture of how an AVS such as `EigenDA` can take advantage of intersubjective staking in EigenLayer.

For simplicity, we consider a blockchain for data availability, that is built as an AVS on EigenLayer. Consider that the EigenLayer operators participating in the consensus of this chain collude to withhold data from all full nodes. Observe that this action of withholding data is observable from outside the Ethereum as anyone requesting the underlying blobs will not have access to it. Therefore, this data withholding attack falls under the scope of the intersubjectively attributable fault, and `EIGEN`'s intersubjective staking can be used to resolve

this fault.

Lightweight monitoring. Utilizing EIGEN's intersubjective staking requires that the honest members of social consensus must be able to come to a uniform agreement on whether a data withholding attack has been perpetuated or not. Adding mechanisms for light clients to monitor the data availablity can provide broader agreement on this fault. Methods such as Data Availability Sampling (DAS) can be used by light nodes to monitor the availability of enough chunks to reconstruct the blob [19, 20].

Risks and mitigation. One difficult edge case in data withholding happens, when data is withheld for some period of time, and then data is subsequently released. Depending on the time period during which this happens, there can be disagreement on whether this fault occured or not. To mitigate such a scenario, the setup phase for such a system should specify the maximum period for which data witholding is tolerable. Lets say this is specified to be 1-day. Now, if data was withheld for something very close to 1-day, then there maybe disagreement on whether data was withheld or not in this edge-case.
In order to solve this edge-case problem, the system can specify that the stakers may get slashed if data was withheld for more than 1-day but a challenger raises a fork only if the data was withheld for 2-days. The creation of this buffer zone, between 1 and 2 days, ensures that when the challenger raises the fault, the others will agree if the data was withheld more than 1-day.

4.2.2 Examples: Foundational Modules

To showcase example applications that can benefit from intersubjective staking, we first analyze some of the foundational building modules for any AVS and understand what type of faults can be attributed to each of these modules (an AVS might not require all of the foundational modules but only some):

- **Reorg resistance.** Any reorg of finalized blocks in a ledger is caused by either double-signing or not building on the latest finalized blocks. This is an objectively attributable fault, as anyone with access to reorged blocks can unilaterally identify the faults.
- **Validity.** Validity of any deterministic execution, such as EVM execution, is typically done on-chain via optimistic fraud proofs/zk proofs or simply zk proofs. This is possible because of the objective attributable nature to any fault in deterministic execution. However, writing fraud proofs and zk proofs is a time-consuming exercise. An alternative mechanism is to make the fault intersubjectively attributable by having social consensus replicate the execution off-chain whenever there is a challenge and perform intersubjective forking of the token to penalize the nodes who did the wrong execution. This feature is especially useful for running ML models, gaming VMs, etc., where writing fraud proofs and zk proofs is difficult.
- **Censorship-resistance.** An AVS might censor requests/tasks it receives but this censoring is not possible to establish algorithmically in a smart contract but is observable. If a transaction has been propagated in the peer-to-peer network but languishes in the mempool for a long enough time and has not been included in the ledger, then the validators who participated in censoring can be intersubjectively slashed.
- **Data availability.** Any fraud in serving of data chunks can't be established objectively on a smart contract. However, this fraud is observable from outside the chain by employing data-availability sampling [19] and so the data providers not serving data can be intersubjectively slashed.
- **Ledger growth.** The AVS can stop executing any new requests/tasks, thus stopping the growth of the ledger of executions. It is impossible to establish this fault on a smart contract, but it is observable off-chain, thus making it an intersubjectively attributable fault.
- **Oracle.** Typically, oracles assume that the majority of the nodes participating in the network are honest and are honestly feeding the correct data on-chain. However, such designs are vulnerable to tyranny-of-majority. Establishing the violation of the underlying honest majority assumption on smart contracts is impossible. However, suppose the input data by this adversarial majority of oracle nodes is different from the actual data by a significant margin. In that case, it is observable from outside the chain and malicious oracle nodes can be intersubjectively slashed.
- **Threshold cryptography.** Current state-of-the-art threshold cryptography protocols like MPC, threshold signature, and secret sharing are vulnerable to collusion by the majority of the participating nodes in private without being attributable intersubjectively or objectively on-chain. This excludes them from

benefiting from intersubjective staking. These services can still leverage a decentralized operator set to get some assurance of collusion resistance.

4.2.3 Examples: Application Use Cases

These foundational modules can then be used as a mix-and-match to more complex modules that can be employed in different AVSs. For example:

- **New Chains.** Any new chain is simply a combination of some of the core primitives laid out above. Thus, intersubjective staking enables developers to build new chains that can include components of intersubjective staking.
- **Composable Modules.** A blockchain can be constructed by composing several modules, one providing one property each: (i) validity checker using ZK or cryptoeconomic means for validation, (ii) (multilateral) transaction ordering engine for censorship resistance, (iii) DA system: for assuring data availability, (iv) proposer auction and replacement protocol: for auctioning off rights to become block proposer - proposer replacement, and (v) a finalization service - for guaranteeing reorg resistance. With intersubjective staking providing assurances against all of these types of failures, it is possible for open innovation at the level of each of these modules so that end users can plug and play these modules to arrive at all the properties of a chain.
- **Intents, Order Matching, and MEV Engines.** Intent and order matching engines have complex offchain software for compiling from intents to transactions. An intent engine may require validation checking for which present techniques for validation may prove difficult to write onchain slashing contracts for. Intersubjective staking and slashing provide a great opportunity for open innovation here.
- **Databases.** There are many execution environments for which writing fraud proofs or validity proofs may be complex in full generality. Therefore, intersubjective slashing can provide an intermediate step before slashing contracts for these AVS can be rigidly built.
- **Gaming VM.** Gaming virtual machines where the game state transition function is hard to fully determine due to the extensive use of GPUs and floating point arithmetic in their underlying logic. However, significant violations in the game state can be intersubjectively attributed.
- **AI Training, Benchmarking, Inference.** AI systems do not yet have full determinism due to their reliance on floating point arithmetic and non-determinism. This compounds the difficulty to write slashing contracts for incorrect execution, however, significant violations in these systems are intersubjectively attributable if the code snippets and hardware architecture are precommitted to.
- **Prediction markets.** Prediction markets require an oracle to resolve at the end of the market. Therefore using intersubjective staking for these oracles makes them more permissionless and rigid.
- **Storage services.** One can create new storage services with a proof-of-replication or a proof-of-custody protocol, where, if the nodes do not custody distinct units of data, the nodes can be slashed. The slashing protocol can be intersubjective rather than fully onchain.
- **Migrate cloud microservices to crypto.** Cloud microservices are usually composition of compute, network and storage. One can start building verifiable versions of these microservices with intersubjective slashing. For example, consider a platform like Apache Kafka, a distributed event store and stream-processing platform. One can now start building a blockchain version of such a service.

5 Roadmap

The launch of EIGEN introduces intersubjective staking. This has broad ramifications for the crypto ecosystem as a whole. With its design being completely novel, the concept needs to be absorbed and discussed widely by the ecosystem participants. The initial implementation of intersubjective staking along with EIGEN and bEIGEN token at launch mirrors the V1 protocol to some extent. However, there are still several parameters, such as CPF, DPF, γ_{burn}, timing parameters like T_{redeem}, etc., that need to be determined for full actuation of V1 protocol. To accomplish that, we are beginning in a meta-setup phase for the EIGEN's intersubjective protocol itself. This period lasts between the initial launch of the token and before transitioning to the full deployment of the V1 protocol. This meta-setup phase will serve as a call-to-action where researchers, experts, and the broader community engage in public discourse to discuss the necessary parameters to make the protocol and its interaction with the rest of the Ethereum ecosystem safe and most effective. One way

to view the setup phase for EIGEN's intersubjective staking is that it is run with training wheels. During this phase, furthermore, there is a whitelisting process for intersubjective AVSs and all whitelisted AVSs are run by all the EIGEN stakers. Furthermore, this ensures a high degree of token toxicity (loss of ecosystem value) if a large fraction of stakers behaves maliciously on a universal intersubjective token. The token toxicity is an initial mechanism for securing the AVSs before the system can take off the training wheels.

We also plan to run a setup phase for EigenDA to discuss the optimal values for parameters for intersubjective slashing: for example, the right amount of time required for data withholding before stakers are slashed, etc. Furthermore, this phase will offer an opportunity to establish a concrete mechanism for computing the parameter $\theta_{\texttt{EigenDA}}$ required for allocating attributable security during its execution phase. The setup phase for EigenDA will serve as a template on how to conduct the setup phase for any AVS that wants to get benefits of intersubjective staking from EIGEN.

In summary, we will be engaging with the community transparently on two major topics:

- **Research community.** The design and parameters of the intersubjective token, how the conditions of forking are specified for new AVSs to inherit security from the token, how to avoid edge cases and risks, etc.
- **Rollup community.** The protocol for determining the precise parameters in EigenDA that would offer sufficient cryptoeconomic security for all rollups using EigenDA.
- **AVS community.** We want this to be useful for AVSs outside of EigenDA, so we want to get feedback from the "customers" of this product.

References

[1] Vitalik Buterin. *Proof of stake: The making of Ethereum and the philosophy of blockchains*. Seven Stories Press, 2022.

[2] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[3] Elinor Ostrom. *Governing the commons: The evolution of institutions for collective action*. Cambridge university press, 1990.

[4] The p + epsilon attack. https://blog.ethereum.org/2015/01/28/p-epsilon-attack.

[5] Schellingcoin: A minimal-trust universal data feed. https://blog.ethereum.org/2014/03/28/schellingcoin-a-minimal-trust-universal-data-feed.

[6] The subjectivity / exploitability tradeoff. https://blog.ethereum.org/2015/02/14/subjectivity-exploitability-tradeoff.

[7] Augur: a decentralized oracle and prediction market platform (v2.0). https://www.allcryptowhitepapers.com/wp-content/uploads/2018/05/Augur-white-paper.pdf.

[8] Kleros. https://kleros.io/whitepaper.pdf.

[9] Aragon court. https://docs-staging.aragon.org/court/.

[10] Hart Lambur, Allison Lu, and Regina Cai. Uma data verification mechanism: Adding economic guarantees to blockchain oracles. *Risk Labs, Inc., Tech. Rep., Jul*, 2019.

[11] Slashing in ethereum. https://eth2book.info/capella/part2/incentives/slashing/.

[12] A technical handbook on ethereum's move to proof of stake and beyond. https://eth2book.info/capella/.

[13] Inactivity leak. https://eth2book.info/capella/part2/incentives/inactivity/.

[14] Don't overload ethereum's consensus. https://vitalik.eth.limo/general/2023/05/21/dont_overload.html.

[15] Soubhik Deb, Robert Raynor, and Sreeram Kannan. Stakesure: Proof of stake mechanisms with strong cryptoeconomic safety. *arXiv preprint arXiv:2401.05797*, 2024.

[16] Peiyao Sheng, Gerui Wang, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath. Bft protocol forensics. In *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, pages 1722–1743, 2021.

[17] Enshrined eth2 price feeds. `https://ethresear.ch/t/enshrined-eth2-price-feeds/7391`.

[18] A not-quite-cryptoeconomic decentralized oracle. `https://ethresear.ch/t/a-not-quite-cryptoeconomic-decentralized-oracle/6453`.

[19] Mustafa Al-Bassam, Alberto Sonnino, and Vitalik Buterin. Fraud proofs: Maximising light client security and scaling blockchains with dishonest majorities. *arXiv preprint arXiv:1809.09044*, 160, 2018.

[20] Mingchao Yu, Saeid Sahraei, Songze Li, Salman Avestimehr, Sreeram Kannan, and Pramod Viswanath. Coded merkle tree: Solving data availability attacks in blockchains, 2019.

[21] Aave. `https://aave.com/`.

[22] Uniswap. `https://uniswap.org/`.

A Protocol

In this section, we will explain the full details of the protocol around intersubjective staking and provide a gradual roadmap toward guaranteeing solid representation and resilience against the Security Council turning malicious.

A.1 V1: Two-token model

We propose a two-token model for intersubjective staking protocol on EigenLayer: (a) backing `EIGEN` (represented as `bEIGEN`) and (b) wrapped representation around `bEIGEN` (represented as simply `EIGEN`). Here `EIGEN` would be a wrapper around `bEIGEN` (more details on this wrapper later). Only `bEIGEN` would be employed for actual staking and subjected to intersubjective forking. On the other hand, the wrapped token `EIGEN` should be used for all non-staking applications.

A.1.1 Staking and Withdrawal

Under normal mode, where no adversarial attack occurs, `bEIGEN` will be solely used for staking purposes, and `EIGEN` will be used for non-staking purposes. No one is expected to hold `bEIGEN` under normal circumstances.

Staking. Anyone can either stake `bEIGEN` by directly interacting with the staking contract or can use the UI to deposit `EIGEN`. If the UI is being used for staking, then the UI first unwraps the deposited `EIGEN` and uses the resulting `bEIGEN`, that the `EIGEN` wrapper is currently configured to, for staking in the staking contract.

Withdrawal. As for withdrawal, when the staker triggers a withdrawal request directly in the staking contract, the contract returns the fork of `bEIGEN` that the staker deposited while staking. Once a withdrawal request is queued, completion of the queued withdrawal request is subjected to a delay of $T_{withdraw}$ slots. On the other hand, if the user is using the UI for withdrawing, then the `bEIGEN` that is returned by the staking contract is wrapped to `EIGEN` which is then returned back to the staker[11]. Note that this wrapping will work only if `EIGEN`'s wrapper contract is still configured to the fork of `bEIGEN` that the staker has deposited in while staking (more on this in sec. A.1.3).

A.1.2 Intersubjective forking of `bEIGEN`

We will explain the working details of intersubjective forking of `bEIGEN`. If there is a need for raising a challenge, the challenger needs to complete the following steps within T_{chal} slots since the malicious actions of the stakers:

1. **Social-deliberation.** In the event of a "potential" intersubjective fault in an AVS, anyone would be able to alert the social consensus of `EIGEN` about the fault. This should trigger a deliberation mechanism among the social consensus to be able to form their own subjective opinion on whether the fault is genuine and socially agree on who the culprits are. Furthermore, someone from the social consensus is selected as a challenger to trigger the intersubjective challenge.

[11] UI is much more opinionated than the staking contract. Any staker who disagrees with the UI can still interact with the staking contracts directly.

2. **Forking** bEIGEN**.** The challenger has to deploy a new bEIGEN ERC20 contract that comes with all the new bEIGEN tokens minted. We call this event "forking the bEIGEN." For explanation purposes, we subsequently refer to $\texttt{bEIGEN}_1$ as the canonical bEIGEN before the forking event and the new fork of bEIGEN as $\texttt{bEIGEN}_2$.

3. **Specifying distribution.** They also deploy a ForkDistributor (FD) contract which contains the following parameters that are socially agreed upon beforehand:

   - address of the $\texttt{bEIGEN}_1$ token contract,
   - addresses of the operators who behaved maliciously,
   - amount of $\texttt{bEIGEN}_2$ to distribute to the challenger,
   - **For attributable security**. Each AVS can redeem an amount of $\texttt{bEIGEN}_2$ based on two factors: the total amount of $\texttt{bEIGEN}_1$ that was staked with malicious operators as attributable security and a fraction of the total attributable security that the AVS is owed to.

   The total $\texttt{bEIGEN}_1$ stake of the operators whose address has been tagged malicious must amount to at least DPF fraction of total $\texttt{bEIGEN}_1$. Furthermore, the FD must meet the following requirements:

   - any $\texttt{bEIGEN}_1$ holder who wants to redeem $\texttt{bEIGEN}_2$ from it should lock their $\texttt{bEIGEN}_1$ tokens for a period of T_{lock} slots. After the end of the lockup period, the $\texttt{bEIGEN}_1$ tokens are returned. See sec. A.1.4 for details.

4. **Specifying enshrined challenge contract.** The challenger needs to deploy an enshrined challenge contract that only accepts $\texttt{bEIGEN}_2$ as the challenger bond. This challenge contract is for future forking of $\texttt{bEIGEN}_2$ as part of the intersubjective challenge. Every fork of bEIGEN will have its own enshrined challenge contract.

5. **Triggering intersubjective challenge onchain.** A challenger then must burn CPF worth of total $\texttt{bEIGEN}_1$ for triggering a challenge in the enshrined challenge contract of $\texttt{bEIGEN}_1$. While raising the challenge, the challenger must supply the addresses of the $\texttt{bEIGEN}_2$ contract, its FD contract, and the enshrined challenge contract. One clause for a valid challenge is that the challenger must trigger this challenge within T_{chal} slots since the attack has happened, otherwise, the challenge wouldn't be considered valid socially (this is not programmatically enforced).

6. **Transfer.** All of the $\texttt{bEIGEN}_2$ tokens are transferred to the FD contract of $\texttt{bEIGEN}_2$ which manages the redemption based on the clauses specified in it.

Social-resolution-of-deadlocks. Once a challenge has been raised, no new challenge should be raised within a period of T_{redeem} slots since the previous challenge was raised. This rule will be enforced socially (not algorithmically in smart contracts!) by not accepting any challenge that contravenes this rule. To understand the cryptoeconomics around this restriction, please see sec. 3.3.

A.1.3 Wrapping interface EIGEN

The wrapping interface EIGEN is an ERC20 smart contract. The core thesis behind the interface is that a user who holds bEIGEN tokens can delegate their own decision-making power of deciding which fork of bEIGEN is the canonical fork to the governance of this EIGEN wrapper contract. In return, the user will now hold EIGEN tokens.

Wrapping and unwrapping. At any time, anyone can instantly wrap the fork of the bEIGEN token that EIGEN wrapper is currently configured to and receive an equivalent amount of EIGEN tokens. As for unwrapping, at any point in time, anyone can instantly unwrap EIGEN into an equivalent amount of the fork of bEIGEN token that the EIGEN is currently configured to.

Backing. Wrapping involves transferring bEIGEN tokens to the EIGEN wrapper's contract[12]. These bEIGEN tokens will be held by the wrapper contract, and the user will be issued an equivalent number of EIGEN tokens. Therefore, 1 unit of EIGEN is backed by 1 unit of the fork of bEIGEN that EIGEN wrapper is currently configured to and is held by the wrapper contract.

Lagged governance: mitigation in case of disagreement with EIGEN **wrapper's governance.** The EIGEN wrapper contract must require that the contract is upgradeable by its governance to reconfigure to a different

[12] As mentioned above, these bEIGEN tokens should be from the fork of bEIGEN that the wrapper contract is configured to.

fork of bEIGEN with a delay of $T_{wGovDelay}$ slots. Suppose an existing EIGEN holder agrees with the decision of the governance of the EIGEN wrapper. In that case, the user doesn't have to do anything and can continue to use EIGEN for non-staking purposes as usual. However, if the user disagrees, the user will have $T_{wGovDelay}$ slots to unwrap to get the tokens belonging to the existing fork of bEIGEN that the EIGEN wrapper contract is configured to before the governance decision gets executed.

Governance steps for upgrading configuration to a new fork of bEIGEN**.** Suppose there is a fork of bEIGEN from $\texttt{bEIGEN}_1$ to $\texttt{bEIGEN}_2$ and the governance of the EIGEN wrapper wants to configure the wrapper to $\texttt{bEIGEN}_2$ (from $\texttt{bEIGEN}_1$). To do that, the governance of EIGEN has to queue an upgrade transaction to configure EIGEN to $\texttt{bEIGEN}_2$ and a transaction for locking $\texttt{bEIGEN}_1$ with the FD of $\texttt{bEIGEN}_2$ to mint $\texttt{bEIGEN}_2$. Once these two transactions are executed after $T_{wGovDelay}$ slots, the $\texttt{bEIGEN}_1$ tokens of the EIGEN wrapper are locked with the FD of $\texttt{bEIGEN}_2$ for a period of T_{lock} slots and are minted back equivalent amount of $\texttt{bEIGEN}_2$ tokens. After the lock-up period is over, the locked $\texttt{bEIGEN}_1$ tokens are given back to the wrapper EIGEN contract, but they won't be returned as part of the unwrapping of EIGEN.

A.1.4 Redemption clause for $\texttt{bEIGEN}_1$ holders, stakers, and challengers if in agreement with the intersubjective fork

Once the $\texttt{bEIGEN}_1$ tokens are transferred to the FD associated with $\texttt{bEIGEN}_2$, a correct FD must respect the following set of rules for $\texttt{bEIGEN}_1$ holders, stakers and challengers to be able to redeem:

1. **HODLER path: those holding $\texttt{bEIGEN}_1$ before the end of redemption period.** The redemption period for $\texttt{bEIGEN}_1$ holders ends T_{redeem} slots after the attack was executed. Before the redemption period ends, anyone holding $\texttt{bEIGEN}_1$ has to lock their $\texttt{bEIGEN}_1$ tokens with the FD for a lockup period of T_{lock} slots. The FD will instantly issue the equivalent amount of new $\texttt{bEIGEN}_2$ tokens to the depositor. After the lockup period is over, the FD will release the $\texttt{bEIGEN}_1$ tokens back to the depositor.
2. **STAKER path: those staked when the challenge was raised.** Before T_{redeem} slots have passed since the attack was executed, the staker can prove to the FD that they were staked to a non-malicious operator for a certain amount of $\texttt{bEIGEN}_1$ tokens at the time when the challenge was raised. If the check passes, the FD will distribute the equivalent amount of $\texttt{bEIGEN}_2$ tokens[13].
3. **PENDING-WITHDRAWAL path: those with a pending withdrawal.** Suppose a staker queued a withdrawal before the attack happened but the withdrawal didn't complete by the end of T_{redeem} slots after the attack was executed. During the period from the end of T_{redeem} slots to the $T_{redeem} + T_{pending-redeem}$ slots since the attack happened, the staker can then redeem an equivalent amount of $\texttt{bEIGEN}_2$ tokens by proving that it had a pending withdrawal which satisfies the above criteria.
4. **CHALLENGER path.** The challenger can redeem their share of the $\texttt{bEIGEN}_2$ tokens from FD at any time. However, observe that, unlike others, the challenger has to sacrifice its holding of $\texttt{bEIGEN}_1$ tokens (while raising the challenge).

See fig. 13 for an illustration.

A.1.5 Redemption clause for EIGEN holders

EIGEN holder is in agreement with the governance of EIGEN. If an EIGEN holder is in agreement with the decision of the governance regarding the challenge that was raised, then the holder doesn't have to do anything.

Wrapper-holder-in-time redemption: EIGEN holder disagrees with the governance of EIGEN. On the other hand, suppose the EIGEN holder has a disagreement with the decision of the governance of the interface. This disagreement appears in one of the following scenarios:

- EIGEN holder agrees with the challenge and considers $\texttt{bEIGEN}_2$ as the canonical fork but the governance of EIGEN didn't queue any upgrade request to configure from $\texttt{bEIGEN}_1$ to $\texttt{bEIGEN}_2$ within T_{wait} slots since the challenge was raised.
- EIGEN holder disagrees with the challenge but the governance has queued an upgrade request to configure EIGEN from $\texttt{bEIGEN}_1$ to $\texttt{bEIGEN}_2$. The upgrade would be executed $T_{wGovDelay}$ slots after the upgrade is queued[14].

[13] Note that the redeemed $\texttt{bEIGEN}_2$ tokens are not automatically staked with EigenLayer. The staker has to stake them separately.

[14] In the event of a censorship attack, the upgrade transaction might be executed more than $T_{wGovDelay}$ slots after the upgrade was queued. In our analysis in sec. B, we consider that Ethereum can be censored for at max a certain period of length $T_{censorship}$ slots.

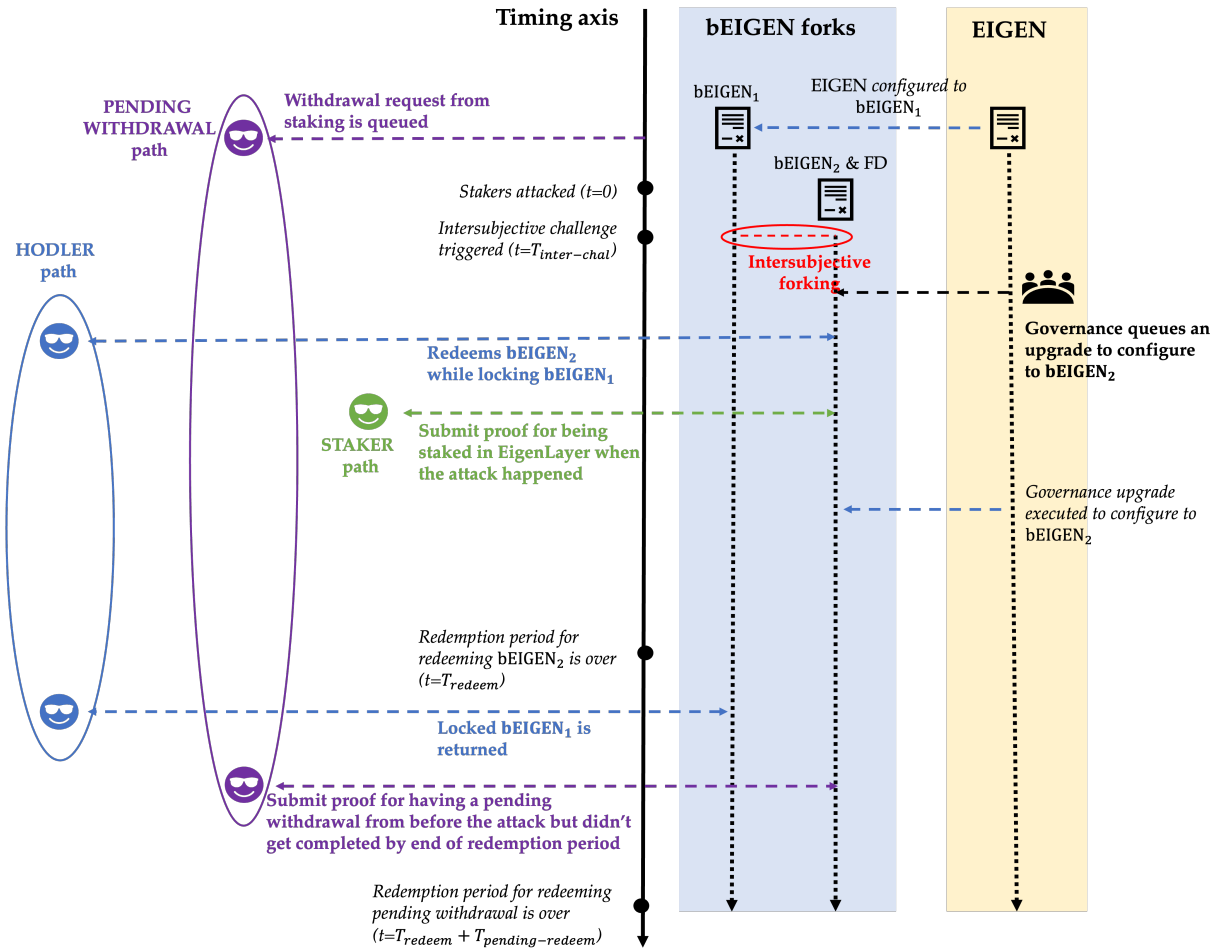


Figure 13: An illustration of redemption mechanism under HODLER, STAKER and PENDING WITHDRAWAL path. Here, for brevity, we assumed that the stakers attacked at time 0.

In either of the scenarios, the EIGEN holder needs to perform the following steps in sequence:

1. get access to its EIGEN on Ethereum
2. unwrap for the $\texttt{bEIGEN}_1$ at the EIGEN contract.

Also, in the scenario where the EIGEN holder is in agreement with the challenge but the governance of EIGEN wrapper contract doesn't, the $\texttt{bEIGEN}_1$ holder afterward has to redeem for $\texttt{bEIGEN}_2$ by locking $\texttt{bEIGEN}_1$. This redemption must be done within T_{redeem} slots after the attack was executed.

In summary, the redemption mechanism for an EIGEN holder is as follows:

- EIGEN holders can be passive if they agree with the wrapping interface's governance.
- EIGEN holders must actively perform a certain set of actions if they disagree with the governance.

A.1.6 Licensing

For intersubjective forking to work, it is necessary that a challenger be able to list the new forked token $\texttt{bEIGEN}_2$ as a strategy in the StrategyManager contract. To ensure that any legitimate fork gets listed as a strategy permissionlessly, we will have the following licensing clause[15]:

> *Anyone can use the license of EigenLayer's strategy contract to list strategy (gated by governance of staking contract) for the fork $\texttt{bEIGEN}_2$ in the staking contract as long as the intersubjective challenge is tagging at least DPF fraction of $\texttt{bEIGEN}_1$ as malicious for an intersubjectively attributable fault and burning them while creating the fork $\texttt{bEIGEN}_2$.*

See sec. B for security analysis of V1 protocol.

[15]It's specifically a clause added to a license to allow other uses than those in the license body. As an example, check out the Additional Use Grant in Arbitrum Nitro's license here: https://github.com/OffchainLabs/nitro/blob/master/LICENSE.md

A.2 V2: Getting solid representation

Consider a legitimate fork that results in $\texttt{bEIGEN}_2$ tokens. One of the shortcomings for V1 is that if the governance of EIGEN wrapper is malicious and the EIGEN holder is either not active for more than $T_{wGovDelay}$ or is not able to access it within a $T_{wGovDelay}$, then the EIGEN holder won't be able to redeem for $\texttt{bEIGEN}_2$ in the worst case. In V2, we will remove this shortcoming.

In the following protocol description, we will only state the changes with respect to V1.

A.2.1 Updated intersubjective fork

Deploy new wrapper $\texttt{EIGEN}_2$ contract. Recall that from V1, as part of the challenge mechanism, the challenger had to burn CPF fraction of total $\texttt{bEIGEN}_1$ and deploy $\texttt{bEIGEN}_2$ ERC20 contract, an FD contract and an enshrined challenge contract. In V2, the challenger must deploy a new wrapper interface $\texttt{EIGEN}_2$ with additional functionality that enables anyone to stop wrapping in return for posting a sufficient amount of challenge bond.

Supply address of all wrapper-forks of $\texttt{bEIGEN}_1$ to the FD. In addition to specifying the address of $\texttt{bEIGEN}_1$, the address of malicious stakers, amount to be redeemed to the challenger in the new FD contract for $\texttt{bEIGEN}_2$, the challenger must also specify the addresses of all wrapper-forks of $\texttt{bEIGEN}_1$ that it considers to be legitimate (more on wrapper-forks below).

Wrap and Transfer. The minted $\texttt{bEIGEN}_2$ supply should be wrapped in the $\texttt{EIGEN}_2$ contract and transferred to the FD.

Stop further wrapping in the canonical $\texttt{EIGEN}_1$ wrapper. Assume that $\texttt{EIGEN}_1$ is the current canonical wrapper-fork on $\texttt{bEIGEN}_1$. While triggering the challenge in the enshrined challenge contract of $\texttt{bEIGEN}_1$, the challenger must also call the stop function in the $\texttt{EIGEN}_1$ wrapper contract to stop any further wrapping of $\texttt{bEIGEN}_1$ to $\texttt{EIGEN}_1$ as part of the challenge.

A.2.2 Updated wrapper interface EIGEN

In contrast to one wrapping contract in V1 where its governance decides upon the fork of bEIGEN it can configure to, the wrapping interface EIGEN in V2 doesn't feature any governance at all. Instead, each intersubjective fork would feature its own wrapping contract, and each wrapping contract is immutable.

Backing. For each intersubjective fork, 1 unit of the associated fork of EIGEN will be backed by 1 unit of the corresponding fork of bEIGEN. That is, if there is fork from $\texttt{bEIGEN}_1$ to $\texttt{bEIGEN}_2$, then 1 unit of $\texttt{EIGEN}_1$ is backed by 1 unit of $\texttt{bEIGEN}_1$ and 1 unit of $\texttt{EIGEN}_2$ is backed by 1 unit of $\texttt{bEIGEN}_2$.

Wrapping and unwrapping when challenged. When a challenge is triggered by depositing sufficient amount of challenge bond, further wrapping in the $\texttt{EIGEN}_1$ wrapper contract will be stopped. Essentially, further production of new $\texttt{EIGEN}_1$ is stopped. On the other hand, $\texttt{EIGEN}_1$ holders can continue to unwrap to $\texttt{bEIGEN}_1$ for all time in perpetuity. Once the wrapping has been paused after the intersubjective challenge was raised, any further unwrapping from $\texttt{EIGEN}_1$ to $\texttt{bEIGEN}_1$ is recorded in the $\texttt{EIGEN}_1$ smart contract's storage (precisely the address that unwrapped, the unwrapped amount, and the time of unwrapping). Note that under normal operation where no challenge is raised, the wrapping and unwrapping remains the same as in V1.

Wrapper-fork: mitigation against potential grieving by raising a fraudulent challenge. A malicious challenger can burn $\texttt{bEIGEN}_1$ to trigger a malicious challenge that will stop any further deposit of $\texttt{bEIGEN}_1$ in the $\texttt{EIGEN}_1$ wrapper for the purpose of wrapping. Unfortunately, despite $\texttt{bEIGEN}_1$ being still considered the legitimate fork by most of social consensus, no one can wrap their $\texttt{bEIGEN}_1$ to $\texttt{EIGEN}_1$ anymore. Under such scenario, social consensus will agree to have one of them deploy a new wrapper interface $\texttt{EIGEN}_1^*$ that points to the $\texttt{bEIGEN}_1$. $\texttt{EIGEN}_1$ holders will eventually migrate to $\texttt{EIGEN}_1^*$ by first unwrapping for $\texttt{bEIGEN}_1$ and then wrapping to get $\texttt{EIGEN}_1^*$. We call such a scenario where there is only fork from $\texttt{EIGEN}_1$ to $\texttt{EIGEN}_1^*$ without the underlying $\texttt{bEIGEN}_1$ getting forked as a wrapper-fork. Note that the deployment of wrapper-forks doesn't require transferring the tokens to the corresponding FD.

A.2.3 Redemption clause for $\texttt{bEIGEN}_1$ holders, stakers, and challengers if in agreement with the intersubjective fork

All the redemption clauses for $\texttt{bEIGEN}_1$ holders, stakers, and challengers from V1 still hold true except redemption pays out $\texttt{EIGEN}_2$ tokens, instead of $\texttt{bEIGEN}_2$.

A.2.4 Redemption clause for $\texttt{EIGEN}_1$ holders

In addition to the redemption rules specified for V1, there is one more redemption rule that needs to be respected by the FD of the $\texttt{bEIGEN}_2$ fork:

> **Wrapper-holder-late redemption.** The FD of $\texttt{bEIGEN}_2$ must allow holder of any legitimate wrapper-fork of only the previous fork $\texttt{bEIGEN}_1$ (there could be many wrapper-forks for the same $\texttt{bEIGEN}_1$) be able to redeem $\texttt{EIGEN}_2$ via the following process:
>
> 1. Assuming $\texttt{EIGEN}_1$ to be the wrapper-fork of $\texttt{bEIGEN}_1$, a $\texttt{EIGEN}_1$ holder first unwraps to $\texttt{bEIGEN}_1$ after T_{redeem} slots since attack (as claimed by the challenger) was executed. As the unwrapping is happening after the stopping of wrapping in the $\texttt{EIGEN}_1$ wrapper contract, this unwrapping action is recorded in the storage of the $\texttt{EIGEN}_1$ contract.
> 2. $\texttt{EIGEN}_1$ holder submits to the FD of $\texttt{bEIGEN}_2$ a pointer to the unwrapping action recorded in the storage of $\texttt{EIGEN}_1$ contract . This will release the corresponding amount of $\texttt{EIGEN}_2$ to the $\texttt{EIGEN}_1$ holder.

Note that wrapper-holder-in-time redemption mechanism from v1 is also part of the redemption mechanism for $\texttt{EIGEN}_1$ holders. See fig. 14 for an illustration.

A.2.5 Updated withdrawal

Withdrawal. If the staker has deposited $\texttt{EIGEN}_1$ token for staking, the staker will receive $\texttt{EIGEN}_1$ while withdrawing from staking if it chooses to receive the wrapper EIGEN token. On the other hand, if the staker has deposited $\texttt{EIGEN}_2$ for staking, the staker will receive $\texttt{EIGEN}_2$ while withdrawing if it chooses to receive the wrapper token. The withdrawal will continue to be subjected to a delay of $T_{withdraw}$ slots by the staking contract before the withdrawal is completed.

A.2.6 Illustration of interaction between DeFi and V2

We will consider two examples to elucidate interaction between DeFi and EIGEN that demonstrates the power of solid representation:

1. **Lending.** Suppose that an $\texttt{EIGEN}_1$ holder puts its $\texttt{EIGEN}_1$ tokens as collateral to borrow USDC via a lending protocol, say Aave [21]. Now consider that later a fork happens in the light of some intersubjectively attributable fault. This forks $\texttt{bEIGEN}_1$ and $\texttt{EIGEN}_1$ to $\texttt{bEIGEN}_2$ and $\texttt{EIGEN}_2$, respectively. Due to the solid representation offered by V2, the $\texttt{EIGEN}_1$ tokens held by the lender now actually give it the right to be able to redeem $\texttt{EIGEN}_2$ tokens. Therefore, the true value of $\texttt{EIGEN}_1$ tokens held by the lender is actually a summation of $\texttt{EIGEN}_1$ tokens held by the lender and $\texttt{EIGEN}_2$ tokens that one can redeem using those $\texttt{EIGEN}_1$ tokens. Any liquidation must take into consideration this aggregated value from the tokens belonging to the descendent forks of $\texttt{EIGEN}_1$.
2. **LPing.** Suppose that an $\texttt{EIGEN}_1$ holder deposits an equal value of $\texttt{EIGEN}_1$ and ETH into a liquidity pool in Uniswap and, in return, receives a liquidity token [22]. Assume that later a fork happens in the light of some intersubjectively attributable fault. This forks $\texttt{bEIGEN}_1$ and $\texttt{EIGEN}_1$ to $\texttt{bEIGEN}_2$ and $\texttt{EIGEN}_2$, respectively. Owing to solid representation in V2, the liquidity token would now represent the ETH and $\texttt{EIGEN}_1$ contribution of the LP to the pool along with the value of $\texttt{EIGEN}_2$ tokens that the LP can redeem with those $\texttt{EIGEN}_1$ tokens in the event if the LP withdraws from the pool.

In both these examples, the DeFi protocol itself is unaware of the forks, but the users using the protocol are able to adjust and embed the relative value of the fork.

See sec. C for an FAQ.

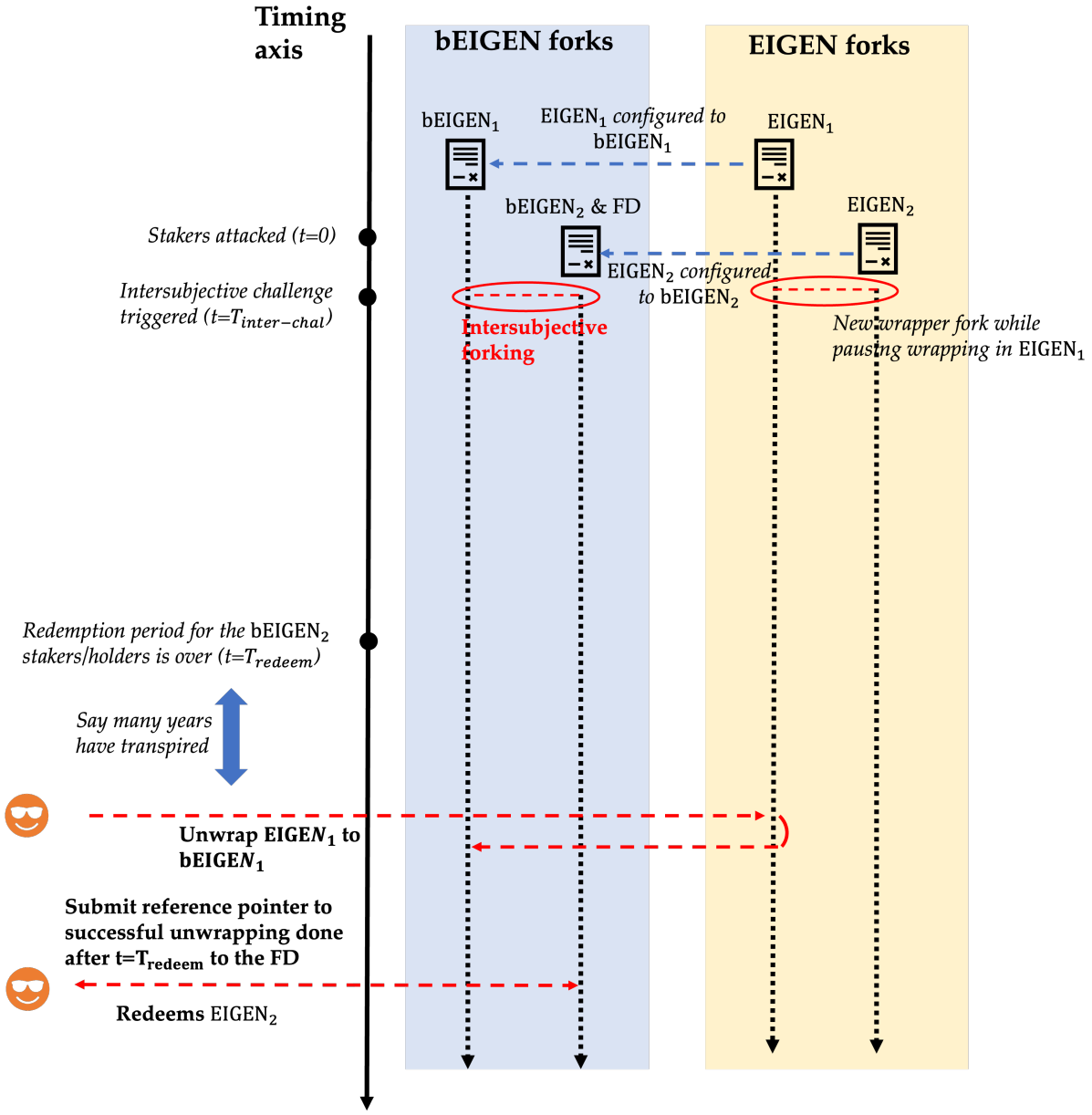


Figure 14: An illustration of wrapper-holder-late redemption. Here, for brevity, we assumed that the stakers attacked at time 0.

A.3 V3: Making intersubjective protocol resilient to corruption of security council

With respect to V2, the only change in the intersubjective forking protocol will be the removal of enforcement of withdrawal lag from the staking contract and instead enforce it via a separate immutable contract that doesn't have any governance.

A.3.1 Adding gateway for withdrawal lag with a challenge mechanism

Gateway contract. An immutable gateway contract is deployed at the genesis of introducing intersubjective forking in EigenLayer. This contract will have no governance. Every time someone wants to stake any `EIGEN`, they have to first call this contract to deposit their stake to this contract while also supplying the metadata on the amount of `EIGEN`, contract address of `EIGEN` wrapper contract, contract address of staking contract, and the address of the associated strategy contract of corresponding fork of `bEIGEN` that is registered in the staking contract (note that there would be strategy contracts for each intersubjective fork of `bEIGEN`). After the gateway contract does unwrapping, in the `bEIGEN`'s ERC20 contract, the deposited stake would be owned by the gateway contract while the staking contract would just be doing accounting and delegation. To withdraw from staking, the staker must make the withdrawal call via the gateway contract. The gateway contract would forward the call immediately to the staking contract, where the appropriate accounting for deductions

will be done immediately, but the exact transfer of bEIGEN from the gateway contract will be subjected to the withdrawal lag of $T_{withdrawal}$ slots. See fig. 15 for an illustration.

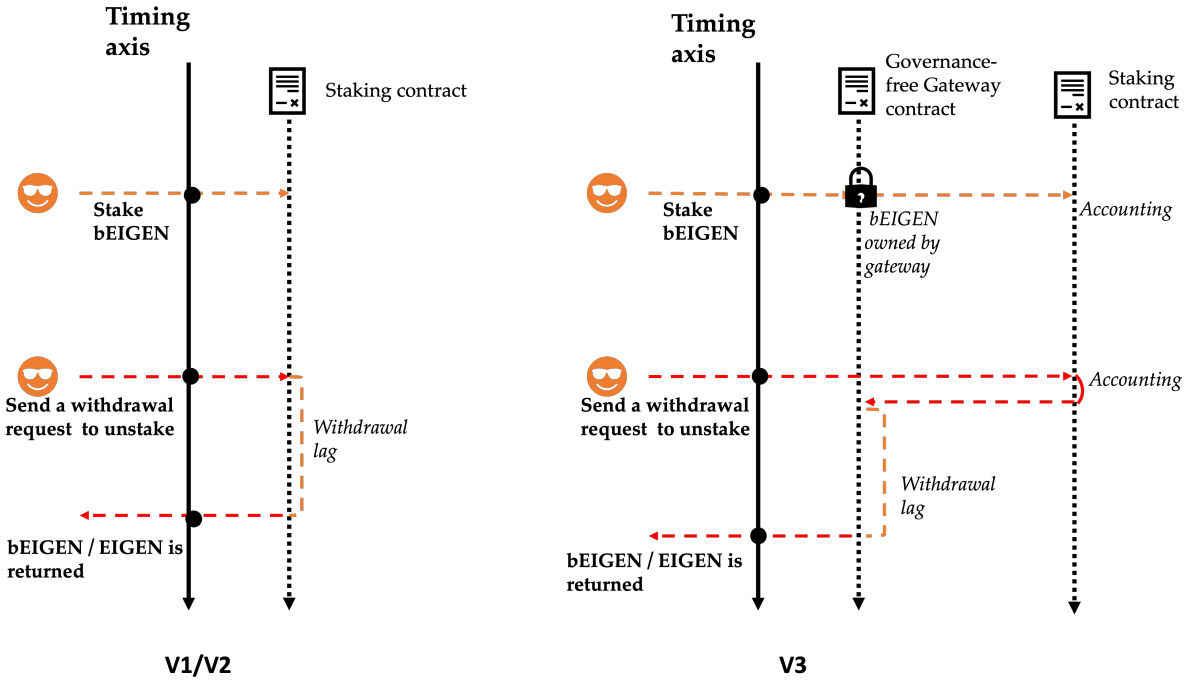


Figure 15: An illustration of the interaction between gateway and staking contracts.

Challenge. By putting a challenge bond in $\texttt{bEIGEN}_1$, anyone can trigger the freezing of all withdrawals from the staking contract to the gateway contract.

See sec. D for an FAQ.

B Security analysis for V1

B.1 Assumptions

In our V1 design, we make the following assumptions:

- A strong adversary model wherein the adversary's transaction for attacking AVSs on EigenLayer is not subjected to censorship attack on Ethereum.
- The 33% adversarial threshold for Ethereum holds true.
- Any transaction on Ethereum can get censored for a maximum of $T_{censorship}$ slots before getting finalized in the ledger.
- The maximum time it takes for detecting any malicious action by a majority/supermajority of stakers and for the challenger's transaction to be finalized in the ledger under potential Ethereum censorship is T_{chal} slots.
- Each staker is always actively monitoring the challenges being triggered.
- [removed in V2] Any holder of EIGEN, if in non-staking activity, can get access to its EIGEN tokens in a period less than $T_{wGovDelay}$ slots.
- [removed in V3] The security council of the staking contract is honest.

B.2 Timing Specification

- $T_{withdraw}$: Any withdrawal from staking is subjected to a lag (see relevant sec. A.1.1). We call the lag as "unstaking period."

- T_{redeem}: When a challenger forks $\texttt{bEIGEN}_1$ to $\texttt{bEIGEN}_2$, the challenger is required to specify a slot $t_{end-of-redeem}$ in the FD until which anyone holding $\texttt{bEIGEN}_1$ can redeem $\texttt{bEIGEN}_2$ via HODLER path or STAKER path (see relevant sec. A.1.4). Now, the time of the attack for which the challenge has been raised is considered to have happened at the slot $t_{attack} = t_{end-of-redeem} - T_{redeem}$.
- T_{lock}: In order for any holder of $\texttt{bEIGEN}_1$ tokens (**not** stakers) to redeem $\texttt{bEIGEN}_2$ via HODLER path, $\texttt{bEIGEN}_1$ tokens would be locked at the time of redeeming $\texttt{bEIGEN}_2$ tokens (see relevant sec. A.1.4).
- T_{chal}: There is a limited window of length T_{chal} after the attack during which anyone can raise a challenge against the attack. Once the challenger specifies slot $t_{end-of-redeem}$ in the FD of $\texttt{bEIGEN}_2$ (see above), that automatically sets the challenge window to be $(t_{attack}, t_{attack} + T_{chal}]$. The enforcement of the restriction that the challenge should be raised within this window is done socially: any challenge not respecting this restriction will be ignored by default.
- $T_{deliberationSlack}$: Before the challenge is raised onchain, the social consensus of EIGEN needs to engage in deliberation to recognize whether the challenge is correct or not, who are the culprits, and who should raise the challenge (see relevant sec. A.1.2). We assume that the length of this deliberation period is $T_{deliberationSlack}$ slots.
- $T_{wGovDelay}$: Anytime the EIGEN governance agrees with the new fork $\texttt{bEIGEN}_2$, the governance needs to queue an upgrade transaction to reconfigure from $\texttt{bEIGEN}_1$ to $\texttt{bEIGEN}_2$. Once queued, the upgrade is subjected to a lag of $T_{wGovDelay}$ slots before execution (see relevant sec. A.1.3).
- T_{wait}: Immediately after a challenge is raised, any EIGEN holder who agrees with the fork will wait for further T_{wait} slots (since the challenge was raised) for the governance of EIGEN to queue an upgrade to reconfigure to the new fork.
- $T_{wGovSlack}$: In the event of a forking, we assume that a small slack of length $T_{wGovSlack}$ slots after the challenge is raised is necessary for the governance of EIGEN to decide whether to reconfigure to the new fork or not. This slack period should be considered within the waiting period of length T_{wait} slots described above.

B.3 Safety Definitions

Safety for honest stakers and EIGEN holders. The protocol is considered to be safe for honest stakers and EIGEN holders if all the following conditions hold true:

1. with a rule-conforming FD for $\texttt{bEIGEN}_2$ deployed due to a challenge, the stakers described as malicious in the FD shouldn't be able to redeem the $\texttt{bEIGEN}_2$ tokens,
2. with a rule-conforming FD for $\texttt{bEIGEN}_2$ deployed due to a challenge, any deserving staker as specified by the FD can redeem the $\texttt{bEIGEN}_2$ tokens only once,
3. any decision by the governance of EIGEN doesn't prohibit active EIGEN holders who disagree with the governance from redeeming their preferred choice of $\texttt{bEIGEN}_1$ or $\texttt{bEIGEN}_2$ tokens.

B.4 Results

Lemma 1. Under the above assumptions, the proposed two-token protocol is safe for honest stakers and EIGEN holders.

Proof. We will proceed with the proof by explaining all possible permutations of events (exempting code bugs) that can happen with the fork and governance of the EIGEN interface.

Scenario 1: $\texttt{bEIGEN}_1$ staker agrees with the challenge

At slot t_{attack}, the malicious operators attack an AVS and immediately queue their withdrawal from staking. Now, consider the following sequence of events:

1. $(t_{attack}, t_{attack} + T_{chal}]$: Within this period, social consensus finishes deliberation on the details of the $\texttt{bEIGEN}_2$ and FD contracts, deploys them, triggers a challenge in the enshrined challenge contract. We represent the slot where the relevant transactions get included in the ledger by t_{chal}.

2. $(t_{chal}, t_{attack} + T_{redeem}]$: Any staker of $\texttt{bEIGEN}_1$ who agrees with the challenge should prove to the FD of $\texttt{bEIGEN}_2$ that it was staked with a non-malicious operator. Beyond this period, no redemptions are allowed for honest stakers. Observe that here, no locking of $\texttt{bEIGEN}_1$ is happening (as it is happening under the STAKER path).

3. $t_{withdraw} > t_{attack} + T_{redeem}$: Completion of withdrawal by any malicious operator after participating in attacking in the slot t_{attack}.

Observe that for the malicious stakers who queue withdrawal after the attack at t_{attack} to be not able to redeem for new $\texttt{bEIGEN}_2$ tokens (safety clause 1), it is required that (assuming $t_{attack} = 0$ hereinafter):

$$T_{redeem} < T_{withdraw}. \tag{19}$$

This condition guarantees that any malicious staker as specified by the FD of $\texttt{bEIGEN}_2$ cannot redeem $\texttt{bEIGEN}_2$.

For the challenge to be raised, we ideally want to give some slack for deliberation among social consensus, say $T_{deliberationSlack}$ slots. However, this challenge transaction then could be subjected to censorship. Therefore, we must have:

$$T_{chal} > T_{deliberationSlack} + T_{censorship} \tag{20}$$

To guarantee safety clause 2, we need the following conditions:

$$T_{redeem} - T_{censorship} > max\{t_{chal}\} = T_{chal} \tag{21}$$

$T_{censorship}$ appears in the LHS because the redemption transaction could be subjected to censorship. This condition basically guarantees that after the end of the challenge period of maximum length, there is enough time for any deserving $\texttt{bEIGEN}_1$ staker to be able to redeem $\texttt{bEIGEN}_2$[16].

Scenario 2: `EIGEN` holder agrees with the challenge and `EIGEN`'s governance also supports the fork

At slot t_{attack}, the malicious operators attack an AVS and immediately queue their withdrawal from staking. Now, the following sequence of events must happen:

1. $(t_{attack}, t_{attack} + T_{chal}]$: By some slot t_{chal} within this period, social consensus finishes deliberation on the details of the $\texttt{bEIGEN}_2$ and FD contracts, deploys them, triggers a challenge in the enshrined challenge contract.

2. $(t_{chal}, t_{chal} + T_{wait}]$: At some slot t_{queue} in this period, honest governance of `EIGEN` queues an upgrade that would configure `EIGEN` to point to $\texttt{bEIGEN}_2$ and a transfer transaction that would be used for redeeming $\texttt{bEIGEN}_2$.

3. $[t_{queue} + T_{wGovDelay}, t_{queue} + T_{wGovDelay} + T_{censorship}]$: The upgrade in the `EIGEN` wrapper and the transaction to lock old $\texttt{bEIGEN}_1$ and redeem the $\texttt{bEIGEN}_2$ is executed at some point in time within this window ($T_{censorship}$ is appearing here as the upgrade execution transaction could be subjected to censorship).

An honest governance ideally should send the scheduling transaction for queuing upgrade and redemption as soon as after slot t_{chal}. We offer a small slack of $T_{wGovSlack}$ slots for governance to send the scheduling transaction. But this might be subjected to Ethereum censorship and so we have the following condition:

$$T_{wait} > T_{censorship} + T_{wGovSlack} \tag{22}$$

In order for `EIGEN`'s governance to be able to redeem $\texttt{bEIGEN}_2$, we require

$$T_{redeem} > \max(t_{queue} + T_{wGovDelay} + T_{censorship}) = T_{chal} + T_{wait} + T_{wGovDelay} + T_{censorship} \tag{23}$$

Note that this requirement is part of safety clause 2.

[16]Here, it is being assumed that any honest $\texttt{bEIGEN}_1$ staker sends its transaction for redemption of $\texttt{bEIGEN}_2$ immediately after it observes the challenge transaction getting finalized on Ethereum. In practice, Ethereum finalization delay and some observation slack should be included too.

Scenario 3: bEIGEN forks and EIGEN holder agrees with the challenge but EIGEN governance doesn't agree with the challenge

The EIGEN's governance decides not to upgrade to $\texttt{bEIGEN}_2$. Note that this governance decision of EIGEN doesn't impact the flow from scenario 1 which $\texttt{bEIGEN}_1$ holders and stakers have to follow. Only the flow for EIGEN holders involved in non-staking operations is impacted. With an active honest EIGEN holder, we will have the following sequence:

1. $(t_{attack}, t_{attack} + T_{chal}]$: At a slot t_{chal} within this period, a challenge is triggered in the challenge contract.
2. $(t_{chal}, t_{chal} + T_{wait}]$: The EIGEN holder waits until the end of this period but the governance of EIGEN doesn't queue any upgrade that would configure EIGEN to point to $\texttt{bEIGEN}_2$ (since governance doesn't agree with the challenge).
3. $(t_{chal} + T_{wait}, t_{attack} + T_{redeem}]$: Any honest EIGEN holder must unwrap to get $\texttt{bEIGEN}_1$ during this period and then redeem for $\texttt{bEIGEN}_2$ from the FD.

This scenario illustrates that despite malicious governance of the EIGEN interface not pushing for an upgrade, an EIGEN holder can redeem the new $\texttt{bEIGEN}_2$ tokens. This is the safety clause 3. The desirable condition is:

$$T_{redeem} > \max(t_{chal} + T_{wait}) + T_{censorship} + T_{censorship} = T_{chal} + T_{wait} + 2T_{censorship} \tag{24}$$

The first term $T_{censorship}$ is for the unwrap transaction from EIGEN to $\texttt{bEIGEN}_1$ which might be subjected to Ethereum censorship. The second term $T_{censorship}$ is for the redemption transaction that might be subjected to Ethereum censorship.

Scenario 4: bEIGEN is forked and EIGEN governance supports the fork but the EIGEN holder doesn't agree with the fork.

Given our assumption of honest governance for staking contracts, stakers are not impacted. Also, $\texttt{bEIGEN}_1$ holders are not impacted. The only impacted party are EIGEN holders.

1. $(0, T_{chal}]$: at a slot t_{chal} within this period, the challenge is triggered in the challenge contract. The challenger also deploys $\texttt{bEIGEN}_2$ and the FD. The governance of EIGEN queues an upgrade that would point to this $\texttt{bEIGEN}_2$ and a redemption transaction.
2. $(t_{chal}, t_{chal} + T_{wGovDelay}]$: Any time the governance of EIGEN queues the upgrade to have EIGEN configured to point to $\texttt{bEIGEN}_2$, the disagreeing EIGEN holder must unwrap to get $\texttt{bEIGEN}_1$.

This scenario illustrates despite EIGEN governance pushing for an upgrade to reconfigure EIGEN wrapper which the active EIGEN holder disagrees with, the EIGEN holder can retrieve $\texttt{bEIGEN}_1$ tokens[17]. Again this is part of safety clause 3. The necessary condition is

$$T_{wGovDelay} > T_{censorship}. \tag{25}$$

Scenario 5: Any $\texttt{bEIGEN}_1$ holder at the time of challenge shouldn't be able to redeem $\texttt{bEIGEN}_2$ more than once

Recall that a $\texttt{bEIGEN}_1$ holder has to lock its $\texttt{bEIGEN}_1$ with the FD of $\texttt{bEIGEN}_2$ as part of the redemption process. Unless the following condition is satisfied,

$$T_{lock} > T_{redeem} \tag{26}$$

what can happen is that a $\texttt{bEIGEN}_1$ holder might be able to lock its $\texttt{bEIGEN}_1$ for T_{lock} slots and get $\texttt{bEIGEN}_2$ and then after $\texttt{bEIGEN}_1$ are unlocked, the holder can again lock $\texttt{bEIGEN}_1$ to get additional $\texttt{bEIGEN}_2$ tokens. We call such a loophole as *double redemption*.

Mitigation of race conditions due to concurrent challenges. Recall that any legitimate challenge must be triggered at least T_{redeem} slots after the previous legitimate challenge was triggered (from social-resolution-of-deadlocks in Appendix A.1.2). Hence, at any point in time, there can't be two legitimate challenges going on simultaneously. This observation nullifies any attack scenario involving race conditions between two or more concurrent challenges.

[17]Here, we are assuming that active EIGEN holders can observe queued governance transaction as soon as it is queued into mempool. In practice, Ethereum finalization delay and some observation slack should be included.

Therefore, the system is safe under the following requirements:

$$
\begin{aligned}
&T_{lock} > T_{redeem} \\
&T_{chal} > T_{censorship} + T_{deliberationSlack} \\
&T_{redeem} > T_{censorship} + T_{chal} \\
&T_{redeem} < T_{withdraw} \\
&T_{redeem} > T_{censorship} + T_{chal} + T_{wait} + T_{wGovDelay} \\
&T_{redeem} > 2T_{censorship} + T_{chal} + T_{wait} \\
&T_{wGovDelay} > T_{censorship} \\
&T_{wait} > T_{censorship} + T_{wGovSlack} \\
&T_{redeem} < T_{withdraw}
\end{aligned} \tag{27}
$$

Additionally, we also need $T_{pending-redeem} > T_{censorship}$ for any staker who has a pending withdrawal from before the attack but has not completed yet by the end of the redemption period to be able to redeem by the PENDING-WITHDRAWAL path.

Hence, the proof. □

B.5 FAQ

Question 1. Consider a staker who staked EIGEN. When staking, EIGEN was backed by $\texttt{bEIGEN}_1$. Suppose much later there is a fork from $\texttt{bEIGEN}_1$ to $\texttt{bEIGEN}_2$ and the governance of EIGEN decides to configure it to $\texttt{bEIGEN}_2$. Assume that the staker was not active within the redemption period. What happens when the staker tries to withdraw EIGEN after the redemption period?
Answer. Note that this question is breaking the assumption of stakers being inactive, but we want to state the invariant that guarantees the assumption. If the staker is using the UI for withdrawing, it is impossible to wrap $\texttt{bEIGEN}_1$ to EIGEN as EIGEN now only accepts $\texttt{bEIGEN}_2$ for wrapping. So, this withdrawal transaction will fail. Instead, the staker can withdraw $\texttt{bEIGEN}_1$ by directly interacting with the contract, instead of using the UI (see Appendix A.1.1).

Question 2. What is a possible solution for the set of equations 27 stated in the security analysis?
Answer. The reduced set of equations are:

$$
\begin{aligned}
&T_{lock} > T_{redeem} \\
&T_{chal} > T_{censorship} + T_{deliberationSlack} \\
&T_{redeem} < T_{withdraw} \\
&T_{redeem} > T_{censorship} + T_{chal} + T_{wait} + T_{wGovDelay} \\
&T_{wGovDelay} > T_{censorship} \\
&T_{wait} > T_{censorship} + T_{wGovSlack} \\
&T_{pending-redeem} > T_{censorship}
\end{aligned}
$$

Assume $T_{censorship} = 3.5$ days, $T_{deliberationSlack} = 3.5$ days, $T_{wGovSlack} = 1$ day. Substituting $T_{chal} = 8$ days, $T_{wait} = 5$ days, $T_{wGovDelay} = 10$ days, $T_{redeem} = 28$ days, $T_{pending-redeem} = 10$ days, $T_{lock} = 30$ days and $T_{withdraw} = 30$ days solve the equation.

Question 3. How are Ethereum reorgs for non-finalizing blocks affecting the intersubjective forking protocol?
Answer. We will illustrate the impact of reorgs by explaining different scenarios. Note that we are operating under the assumption that the 33% adversarial assumption for Ethereum holds true, and so reorgs can impact only non-finalized blocks within an epoch due to the lack of single slot consensus in Ethereum.

Scenario 1: The challenge transaction for intersubjective forking is subjected to Ethereum reorg

Consider the following sequence of events:

1. At slot t_{attack}, an AVS is attacked by operators of $\texttt{bEIGEN}_1$ stakers who are opted into it.
2. At $t_{queueChal}$ slot, the challenge transaction is sent but gets queued in mem-pool or reorged out.

3. At $t_{queueWithdrawl}$ slot, the adversary withdraws $\texttt{bEIGEN}_1$ from staking.
4. At t_{chal} slot, the challenge transaction to get a new intersubjective fork $\texttt{bEIGEN}_2$ is executed.

Here, $t_{attack} < t_{queueChal} \leq t_{queueWithdrawl} < t_{chal}$. Assume $t_{attack} = 0$.

Basically, the challenge transaction is being subjected to censorship in Ethereum, with maximum censoring that can happen for $T_{censorship}$ slots. Here the worry is that the adversary might be able to withdraw $\texttt{bEIGEN}_1$ before the redemption period for $\texttt{bEIGEN}_2$ is over. However, observe that we already have the condition that

$$T_{chal} > T_{censorship} + T_{deliberationSlack},$$

which means that $t_{chal} < T_{chal}$. Given the redemption period T_{redeem} already takes the worst-case censorship into account as in the following condition

$$T_{redeem} > T_{censorship} + T_{chal} + T_{wait} + T_{wGovDelay}$$

and

$$T_{withdraw} > T_{redeem}$$

the above attack of the adversary slipping out with its stake is impossible.

Scenario 2: Challenge transaction got reorged and reordered with the redemption transaction of $\texttt{bEIGEN}_1$ staker

Consider the following sequence of events:

1. At slot t_{attack}, an AVS is attacked by operators of $\texttt{bEIGEN}_1$ stakers who are opted into it.
2. At slot $t_1 \in (t_{attack}, T_{chal}]$, the challenge transaction is sent but reorged out within the finalization period.
3. At slot t_2, $\texttt{bEIGEN}_1$ staker who agrees with the challenge and is not delegated to any malicious operator doesn't wait for the challenge transaction to be finalized and sends its redemption transaction to the FD of $\texttt{bEIGEN}_2$ which gets finalized.
4. At slot t_3, the challenge transaction is finalized.

Here $t_1 < t_2 < t_3$. Observe that in step 3, the execution of the redemption transaction would be reverted as there is no FD for $\texttt{bEIGEN}_2$ yet due to reorg in step 2. However, the loss to the $\texttt{bEIGEN}_1$ staker is only in gas fees. Ideally, this staker should wait for the finalization of the block containing the challenge transaction.

Scenario 3: Challenge transaction got reorged and reordered with the redemption transaction of $\texttt{EIGEN}_1$ holder

Recall that an $\texttt{EIGEN}_1$ holder would always be waiting for a period of T_{wait} slots

$$T_{wait} > T_{censorship} + T_{wGovSlack}$$

before sending any redemption transaction, if needed. Under the assumption that 33% adversarial assumption for Ethereum holds true, by the time $\texttt{EIGEN}_1$ holder sends any redemption transaction, the challenge transaction will already be finalized in the ledger. Hence, the above reordering scenario is never going to happen.

Question 4. Why does the challenger have the incentive to raise a challenge?
Answer. The challenger has to burn $\texttt{CPF}$ fraction of $\texttt{bEIGEN}_1$ while being able to receive some amount of $\texttt{bEIGEN}_2$ from the FD of $\texttt{bEIGEN}_2$. In case of a legitimate intersubjective challenge for a potentially verifiable digital task, $\texttt{bEIGEN}_2$ will be accepted as the canonical fork and ideally, inherit all worth of $\texttt{bEIGEN}_1$ that was there before the attack. Note that, unlike other honest players, the challenger has to burn $\texttt{bEIGEN}_1$ and has access to only $\texttt{bEIGEN}_2$. That infers the harmed parties from the attack executed by malicious $\texttt{bEIGEN}_1$ stakers have the highest incentive to incur the cost to raise the intersubjective challenge.

Question 5. What is the incentive to prevent malicious challenge from being raised?
*Answer.*It is possible that a malicious entity might raise a vacuous challenge against a digital task for an AVS. If the social-deliberation mechanism for the AVS is properly designed during the setup phase of the AVS, then no honest member of the social consensus of $\texttt{EIGEN}$ will respect the new fork $\texttt{bEIGEN}_2$ and $\texttt{bEIGEN}_1$ will continue to be perceived as the canonical fork of $\texttt{bEIGEN}$ by the social consensus. Instead, the malicious challenger will end up losing $\texttt{CPF}$ fraction of $\texttt{bEIGEN}_1$ that it submitted as a challenge bond.

C Security analysis for V2

C.1 FAQ

Question 1. Why is pausing of wrapping necessary?
Answer. Without pausing in the wrapping of $\texttt{bEIGEN}_1$ to $\texttt{EIGEN}_1$ when a challenge is raised but still permitting for the perpetual redemption for $\texttt{EIGEN}_1$ holders would enable any $\texttt{EIGEN}_1$ holder to engage in double redemptions. More specifically, an $\texttt{EIGEN}_1$ holder can unwrap to get $\texttt{bEIGEN}_1$ which is then recorded in the log maintained by the wrapper $\texttt{EIGEN}_1$. Now, this holder can show a pointer to the record in the storage of $\texttt{EIGEN}_1$ as proof to the FD of $\texttt{bEIGEN}_2$ to redeem $\texttt{EIGEN}_2$. Without the feature of stopping any further wrapping from $\texttt{bEIGEN}_1$ to $\texttt{EIGEN}_1$ when the challenge is raised, the holder can then wrap its $\texttt{bEIGEN}_1$ to get $\texttt{EIGEN}_1$, send it to a different sybil account it controls and then unwrap again to $\texttt{bEIGEN}_1$ and use that to redeem more $\texttt{EIGEN}_2$, which would have to lead to double redemption.

Question 2. How is daisy chaining getting resolved?
Answer. Consider the following daisy-chaining scenario:

> Suppose an $\texttt{EIGEN}_1$ holder is locked in a 1-year position in some non-staking application. During this 1 year period, there have been 10 forks which resulted in transitioning from $\texttt{bEIGEN}_1$ to $\texttt{bEIGEN}_{10}$ as the canonical fork of bEIGEN. Now, when the $\texttt{EIGEN}_1$ holder unlocks from its non-staking position after 1 year, suppose the holder considers all the 10 forks legitimate. How does the $\texttt{EIGEN}_1$ holder transition to holding $\texttt{EIGEN}_{10}$?

Recall that under the redemption mechanism for v2, a correct FD for the N-th fork $\texttt{bEIGEN}_N$ token will permit the holders of $\texttt{EIGEN}_{N-1}$ token to be able to redeem $\texttt{EIGEN}_N$ token anytime after the redemption period for bEIGENvN fork has passed over. Depending on the exact value for T_{redeem}, it is possible that after 1 year, the redemption period for many forks $\texttt{bEIGEN}_N$ is over. For our example, we assume the redemption period until $\texttt{bEIGEN}_8$ is over when $\texttt{EIGEN}_1$ holder's positions become unlocked. Therefore, what this holder can do is first unwrap $\texttt{EIGEN}_1$ to $\texttt{bEIGEN}_1$ that records the unwrapping action in $\texttt{EIGEN}_1$ wrapper contract which enables redeeming $\texttt{EIGEN}_2$ with the FD of $\texttt{bEIGEN}_2$. Then, the holder can do unwrapping from $\texttt{EIGEN}_2$ to get $\texttt{bEIGEN}_2$. This unwrapping action is recorded in the wrapper $\texttt{EIGEN}_2$, which can then be referred to as proof to the FD of $\texttt{bEIGEN}_3$ to redeem $\texttt{EIGEN}_3$. This inductive redemption continues until the holder receives $\texttt{EIGEN}_8$. Assuming that the redemption period for $\texttt{bEIGEN}_9$ is not over yet, the holder can unwrap $\texttt{EIGEN}_8$ to get $\texttt{bEIGEN}_8$ and then lock it with the FD of $\texttt{bEIGEN}_9$ to redeem $\texttt{bEIGEN}_9$ (note that its $\texttt{bEIGEN}_8$ gets locked here). Now, the holder can then use the $\texttt{bEIGEN}_9$ to lock it up with the FD of $\texttt{bEIGEN}_{10}$ to redeem $\texttt{bEIGEN}_{10}$ (note that its $\texttt{bEIGEN}_9$ gets locked here).

Question 3. How to disincentivize liquidity fracturing?
Answer. Under malicious challenge to pause $\texttt{EIGEN}_1$ wrapper contract, the migration to $\texttt{EIGEN}_1^*$ might happen gradually, leading to liquidity fracturing for a while. This gives us another lower bound on the challenge bond:

$$\texttt{CPF} \times \text{Total } \texttt{bEIGEN}_1 > \text{Profit from causing the fracturing of liquidity}$$

Question 4. In the redemption mechanism for $\texttt{EIGEN}_1$ holders, why does the FD of $\texttt{bEIGEN}_2$ need to permit redemption for all wrapper-forks of $\texttt{bEIGEN}_1$?
Answer. Consider that at the beginning, the canonical fork is $\texttt{bEIGEN}_1$ and the current wrapper on it to be $\texttt{EIGEN}_1$. Now let us have the following sequence of events:

1. At t_1 slot, a malicious challenge is raised that stops further wrapping in $\texttt{EIGEN}_1$.
2. At t_2 slot, a new wrapper $\texttt{EIGEN}_1^*$ is released over $\texttt{bEIGEN}_1$.
3. At t_3 slot, a legitimate intersubjective fork $\texttt{bEIGEN}_2$ is done and the FD of $\texttt{bEIGEN}_2$ considers both $\texttt{EIGEN}_1$ and $\texttt{EIGEN}_1^*$ as the legitimate wrapper-forks on $\texttt{bEIGEN}_1$ (under the wrapper-holder-late redemption mechanism described in sec. A.2.4). Note that this challenge stops further wrapping in $\texttt{EIGEN}_1^*$.

Consider a different version of redemption protocol where the FD of $\texttt{bEIGEN}_2$ had considered only the latest wrapper-fork of $\texttt{bEIGEN}_1$, that is $\texttt{EIGEN}_1^*$ in the above case, to be the one that is eligible for redeeming $\texttt{EIGEN}_2$. Now if an $\texttt{EIGEN}_1$ holder wakes up say 1 year later, they can't redeem $\texttt{EIGEN}_2$ from the FD of $\texttt{bEIGEN}_2$ directly. They would have to unwrap to get $\texttt{bEIGEN}_1$ and then wrap to get $\texttt{EIGEN}_1^*$. However, this wrapping to $\texttt{EIGEN}_1^*$

is not possible as wrapping was already stopped in step 3 above. That's why by socially enforcing the FD of $\texttt{bEIGEN}_2$ include all legitimate wrapper-forks of $\texttt{bEIGEN}_2$ for being eligible to redeem, the $\texttt{EIGEN}_1$ holder can redeem $\texttt{EIGEN}_2$.

Question 5. In the wrapper-holder-late redemption mechanism for $\texttt{EIGEN}_1$ holders, why does the FD of $\texttt{bEIGEN}_2$ require holders of wrapper-forks of $\texttt{bEIGEN}_1$ to unwrap only after T_{redeem} slots since the challenge to fork $\texttt{bEIGEN}_1$?
Answer. We will show the necessity of this requirement by considering an alternative definition of wrapper-holder-late redemption where holders of wrapper-forks of $\texttt{bEIGEN}_1$ can redeem $\texttt{EIGEN}_2$ if they have unwrapped any time after the intersubjective challenge was triggered. For simplicity of explanation, let us consider that the wrapper fork is represented by $\texttt{EIGEN}_1$. However, we can have the scenario now where this holder of $\texttt{EIGEN}_1$ is also receiving $\texttt{bEIGEN}_1$ from the unwrapping within the redemption period and can:

- lock that $\texttt{bEIGEN}_1$ to redeem $\texttt{bEIGEN}_2$ directly within the redemption period via the lock-and-mint policy.
- use the unwrapping from $\texttt{EIGEN}_1$ to $\texttt{bEIGEN}_1$ as a proof to redeem $\texttt{EIGEN}_2$.

Clearly, this is a case of double redemption.

This double redemption can be avoided if the $\texttt{EIGEN}_1$ holder cannot redeem $\texttt{bEIGEN}_2$ by locking $\texttt{bEIGEN}_1$. That is possible by putting the requirement of T_{redeem} slots in the wrapper-holder-late redemption mechanism.

D Security Analysis for V3

D.1 FAQ

Question 1. How does putting a lag in a separate gateway contract protect against adversarial action from the Security Council?
Answer. Suppose the security council maliciously updates the staking contract with the wrong accounting. Note that this update will be done immediately. However, if now the security council wants to withdraw the $\texttt{bEIGEN}_1$ stake, it will be subjected to the withdrawal lag in the gateway contract. During this withdrawal lag, the harmed parties will have every incentive to raise a challenge to stop withdrawals from the staking contract to the gateway contract. After pausing happens, someone in social consensus (after deliberation) will launch a new staking contract with a new security council and a new fork $\texttt{bEIGEN}_2$ with its FD which compensates back the stake to the stakers harmed by the security council.