Revoke: Mitigating Ransomware Attacks against Ethereum Validators

Alpesh Bhudia, Daniel O'Keeffe, and Darren Hurley-Smith

Royal Holloway University of London, UK
alpesh.bhudia.2018@live.rhul.ac.uk
{daniel.okeeffe,darren.hurley-smith}@rhul.ac.uk

Abstract. Recent research has shown the viability of ransomware attacks on Ethereum Proof-of-Stake (PoS) validators, whereby an attacker that compromises a validator can threaten to perform slashable actions unless a ransom is paid. Given the size of Ethereum validator stakes, validators could become an attractive target for future ransomware. However, there are currently no practical mechanisms to recover from ransomware since even validators that attempt to exit the network are retrospectively slashable during the withdrawal period.

We propose *Revoke*, an extension of Ethereum that mitigates the impact of ransomware attacks on validators. *Revoke* introduces a new decentralised key revocation mechanism that enables validators to change their signing key without withdrawing their stake. A challenge for *Revoke* is balancing the utility of the revocation mechanism for individual validators against potential reductions in overall chain security. *Revoke* exposes a trade-off whereby validators cannot propose or attest to blocks during the revocation process, and hence incur inactivity penalties, but are not susceptible to much larger slashing penalties. Our design extends the Ethereum specification to capture the impact of *Revoke*'s core key-change mechanism on both the beacon-chain state transition function and fork-choice decisions. We also adapt the existing safety and liveness proofs of Ethereum to incorporate the effects of *Revoke*.

1 Introduction

Proof-of-Stake (PoS) blockchains are garnering increased attention among the blockchain community [2, 14, 23, 26, 32, 36]. Most prominently, Ethereum transitioned to a new design (Ethereum 2.0) based on PoS in 2022. In Ethereum, committees of randomly chosen *validator* nodes propose and validate blocks of transactions as part of a new 'beacon-chain' consensus protocol. To participate, validators must deposit a stake of 32 ETH (approx. USD $95k[1]). As of April 2024, there are over 900,000 active validators with a total stake of over 28,800,000 ETH (approx. USD $2.74 Billion).

A major concern for PoS validators is to ensure their stake cannot be stolen if a validator node is compromised by an attacker. In Ethereum, validator security is enhanced by separating the *withdrawal key* that controls access to the

[1] ETH-USD valuation 17/04/2024

staked deposit from the *signing key*, used to sign any messages sent as part of the protocol. The withdrawal key can be kept offline in cold storage, as it is not required for regular operations performed by the validator. The signing key, needed to sign all validator operations, must be kept online and hence is vulnerable if the validator node is compromised. However, access to the signing key does not provide an attacker direct access to the validator's stake.

Although separate withdrawal and signing keys provide a valuable layer of defence, recent work has shown how *ransomware* attacks can still indirectly extort a compromised validator's stake even without direct access to the withdrawal key [6, 7]. The idea behind such attacks is to threaten compromised validators that unless a ransom is paid, the attacker will misbehave using their signing key (e.g. by signing conflicting blocks or attestations). In Ethereum, misbehaving validators are punished heavily through the deduction of *slashing* penalties from their stake. As a result, validators are incentivised to comply with a ransomware attacker's demands. Furthermore, it has been shown that a smart contract can guarantee a validator will be safe from any further slashing on payment of the ransom, minimising the need for a compromised validator to trust the ransomware attacker. Given the size of validator stakes, they could become a lucrative target for future ransomware attacks.

We observe that while *key revocation* would potentially support post-compromise security in the event of a ransomware attack, it is currently unsupported in Ethereum. Unlike in centralised settings, signing key revocation in a decentralised PoS blockchain is non-trivial since validators must agree on each other's signing keys in order to check the correctness of consensus protocol messages (e.g. block proposals and attestations). This requires the key revocation mechanism itself to be decentralised. Such a mechanism must additionally consider that an attacker may monitor messages exchanged over the blockchain peer-to-peer network and cause the validator to be slashed if any attempt to revoke a compromised key is observed. Finally, revocation must not provide an avenue for large-scale attacks to undermine the security of the blockchain by sidestepping slashing penalties.

To overcome these challenges, we propose *Revoke*, a new *decentralised key revocation* protocol for the Ethereum blockchain. *Revoke* enables a validator to recover from an ongoing ransomware attack by submitting a *revocation request* containing a new signing key to the blockchain. Once a block containing the revocation request is finalised, the new signing key becomes operational. In the intervening period, the validator is temporarily disabled, and any block proposals or attestations it signs using either key will be ignored by other validators. Although the validator is subject to inactivity penalties while disabled, any slashable behaviour using the old key that is not finalised before the revocation request will be ignored.

Revoke's design addresses several other subtle issues that arise due to interactions between revocation and the existing consensus protocol. For example, votes by disabled validators using either their old or new key are not considered as part of Ethereum's LMD-GHOST fork-choice rule [11]. This ensures a validator cannot ‘double-vote’ for different branches of the block-tree. In addition, validator revocation requests are rate-limited to avoid facilitating long-range attacks. We adapt the existing safety proofs of Gasper [12], the consensus protocol underpinning the Ethereum beacon chain, to incorporate the effects of Revoke. We also implement Revoke as an extension to the Ethereum executable specification, and evaluate its correctness with respect to an expanded version of the existing Ethereum test suite.

The rest of the paper is organised as follows. We first give background on Ethereum ransomware attacks and introduce our system model (§2). We then outline the high-level design of our revocation protocol (§3), before describing in detail its key algorithms and their integration into the Ethereum consensus protocol (§4). Next, we sketch a proof of correctness (§5), discuss incentives for proposers to include revocation requests (§6) and summarise the changes we made to the Ethereum formal specification (§4.3). We finish with a discussion of related work (§7) and conclusions (§8).

2 Background and Motivation

Ethereum Proof-of-Stake. Ethereum’s core ‘beacon-chain’ consensus protocol [23] is based on Gasper [12], which itself is a combination of ideas from the Casper FFG finality gadget [11] and a hybrid form of the LMD-GHOST fork-choice rule [11, 40]. The Casper ‘gadget’ is a sub-protocol for deciding which blocks in a blockchain should be considered *finalised*, i.e. blocks that everyone will eventually think of as part of the consensus history. More formally, given a view G containing the set of messages observed by a validator, a finality gadget returns the set $F(G)$ of finalised blocks.

The LMD-GHOST fork-choice rule gives a validator a law to follow when deciding what the right block should be in the event of conflicting blocks. More formally, the fork-choice rule is a function $\mathsf{fork}()$ that, when given a view G identifies a single leaf block B that produces a unique chain $\mathsf{fork}(G) = \mathsf{chain}(B)$ from $B_{genesis}$ to B called the *canonical chain*. B is referred to as the *head* of the chain in view G. In Ethereum, new blocks are proposed once per *slot*, where a slot is defined as a constant number of seconds (currently 12s [23]). For efficiency reasons, finalisation considers only a subtree of *checkpoint* blocks. Checkpoints occur at the granularity of *epochs*, where an epoch is some constant number C of slots (currently 32 [23]). Within an epoch, validators are partitioned into committees, with one committee per slot. The assignment of validators to committees is performed at random one epoch in advance.

Within each slot, a single validator from the corresponding committee proposes a block. The other committee members then create an *attestation* containing (i) a vote for the block they consider to be the head of the chain according to the fork-choice rule (ii) a finalisation vote in the form of a *checkpoint edge* that indicates what they consider to be the most recent checkpoint.

Accountability. To incentivise good behaviour on the part of validators, Ethereum employs two accountability mechanisms in the form of *inactivity penalties* and *slashing*. Inactivity penalties are generally small (3x the base reward ≈21,720

Gwei per epoch), targeting validators who fail to participate in consensus activities because they are offline or otherwise unavailable. In contrast, slashing penalises actions that could jeopardise the integrity of the blockchain, such as signing contradictory attestations or block proposals. For a validator to be slashed, evidence of the slashing must be included in a block.

A slashed validator with stake x incurs an initial *base* slashing penalty $b = \frac{x}{32}$ e.g. 1 ETH for a typical stake of 32 ETH [3, 24]. $\frac{b}{16}$ of the base slashing penalty is rewarded to the proposer of the block containing the slashing, amounting to 0.0625 ETH for a stake of 32 ETH. To incentivise nodes to search for and report slashable behaviours [8, 20], a whistleblower reward of $\frac{1}{8}$ of the proposer reward was also suggested initially. However, the current implementation disables this feature since it is trivial for the proposer to claim any slashing evidence for itself.

A slashed validator is also *signed to exit*, i.e. it can no longer participate in consensus activities. Furthermore, it is unable to withdraw its remaining stake for a further 8192 epochs (approx. 36 days). Halfway through the withdrawal period (4096 epochs), an additional correlation penalty β is calculated for the validator. The correlation penalty scales with the number of concurrent slashing events in order to disincentivise large-scale attacks without being overly punitive for isolated slashing incidents. β is based on the total amount of stake slashed across all validators during the 4096 epochs before and after the validator was slashed. Concretely, β is set to $\frac{3SB}{T}$, where T is the total stake of all active validators, B is the current balance of the slashed validator, and S is the sum of all balances of slashed validators in the last 36-days. The correlation penalty can result in a validator losing all of its stake if $\frac{T}{3}$ of stake is slashed in the same period. However, for isolated slashing events the correlation penalty is negligible in comparison to the base penalty. To date, all slashings have been relatively isolated events with respect to the total number of validators (Figure 1). The maximum number of correlated slashings observed was 106 in November 2023, constituting 0.016% of the total validator pool and a maximum observed correlation penalty of around 0.01 ETH. The raw data for these events is taken from [5] and is available in our repository [38].

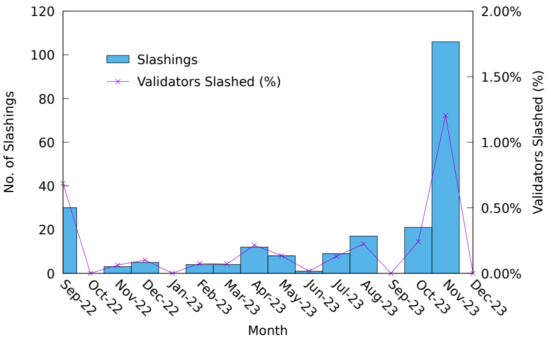

Fig. 1: Monthly Validator Slashings [5]

Blockchain ransomware. While slashing is intended to act as a financial disincentive for validators to misbehave, such incentive schemes can lead to unintended side effects [16, 26].

Blockchain ransomware attacks abuse slashing to extort compromised validators [6, 7, 8]. Such attacks target a validator's online signing key and threaten to perform slashable offences with it unless a ransom is paid. An attacker can even fabricate slashable offences retrospectively. Withdrawing from the validator pool in response to a ransomware attack does not help since validators remain slashable during the minimum withdrawal period for voluntary exits (at least 261 epochs or approximately 1.2 days).

Prior work has explored different strategies a blockchain ransomware attacker can employ from a game-theoretic perspective to maximise their potential rewards [6, 7, 8]. However, there is little support currently in Ethereum for post-compromise recovery from blockchain ransomware. The lack of protocol level mitigations for ransomware increases the need for operational security expertise on the part of individual validators. This potentially undermines the blockchain decentralisation ethos, making revocation mechanisms important for systemic resilience. Standard techniques for recovering from conventional ransomware attacks in other domains (e.g. backups) are inapplicable. Other proof-of-stake blockchains, such as Tezos and Polkadot [1, 2] provide expiry-based key rotation mechanisms, but old keys remain slashable for up to 28-days after rotation.

3 Revoke Design

In this section, we propose *Revoke*, a novel signing key revocation protocol that is robust against Ethereum ransomware attacks. We define several design goals for *Revoke* (§3.1), state our threat model (§3.2) and give an overview of *Revoke*'s design (§3.3).

3.1 Decentralised Key Revocation

Revoke cannot rely on a centralised and trusted key management service [9, 35] to cope with slashing-based ransomware attacks. Instead, *Revoke* introduces a *decentralised* key revocation mechanism that is integrated into the Ethereum consensus protocol. *Revoke*'s decentralised revocation protocol has three key design goals. First, it must enable a compromised validator to revoke its signing key without being slashed (G1). This should be possible even when an attacker has full visibility over the network and can collude with other validators. Second, key revocation should not undermine the security of the consensus protocol (G2). For example, attackers should not be able to leverage it to mount large-scale attacks without incurring slashing penalties. We note that this design goal is in tension with G1. Finally, to ensure correctness and facilitate adoption, the protocol must be compatible with the existing beacon-chain protocol (G3). In particular, it should minimise changes to the existing protocol design (e.g. to facilitate formal verification) and avoid introducing substantial performance or storage overheads.

3.2 Threat Model

We consider the standard Ethereum threat model, where Byzantine validators may misbehave arbitrarily and honest validators will follow the protocol so long as it is economically rational. This threat model differs from the standard Byzantine threat model in that honest validators may deviate from the protocol if there is no economic penalty for doing so (e.g. slashing).

We assume validators have security mechanisms in place to protect their withdrawal key and that it cannot be compromised by a ransomware attack (e.g. it is kept offline in cold storage during normal operation). However, a validator's signing key may be compromised such that an attacker can use it to sign arbitrary messages. We treat the vector by which validators are compromised as an orthogonal issue. An attacker may control or collude with other validators, observe all messages broadcast over the network, and see messages even before they are included in a block (e.g. transactions in the mempool of a proposer). These are conservative but realistic assumptions given the need to capture attackers with many validators distributed throughout the network. Proposers under an attacker's control (either directly or indirectly via collusion) may arbitrarily reorder or delay messages within their own mempool.

3.3 Revocation Overview

Figure 2 gives an overview of *Revoke* revocation processing. Validators wishing to change their signing key first construct a *revocation request* containing the signing key they want to revoke and the new signing key they want to replace it with (e.g. validator `v1` with initial signing key `k1` in Figure 2, step ①). The validator signs the request using its private withdrawal key, which according to our threat model, is unknown to the attacker. The validator broadcasts the revocation request to other validators over the beacon-chain peer-to-peer network.

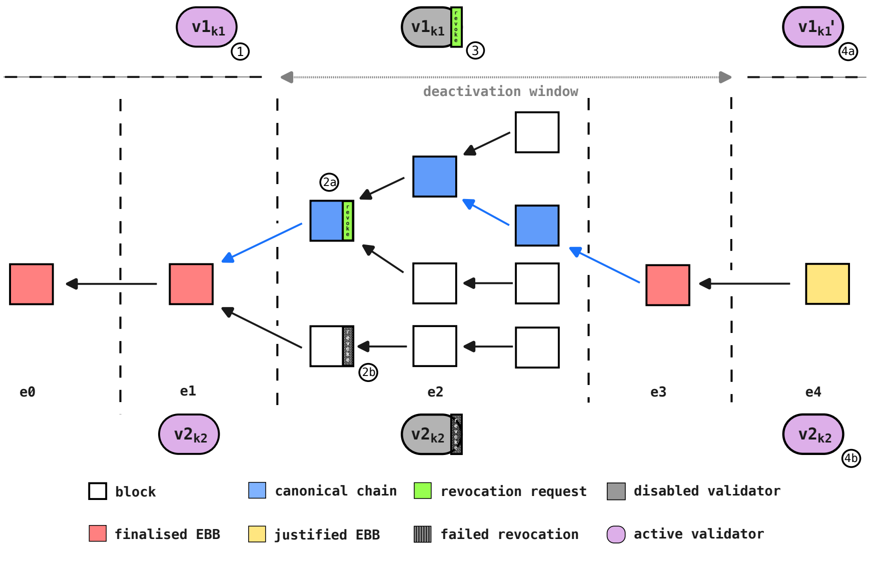

Fig. 2: Overview of the revocation process

Processing of a revocation request starts when a proposer includes it in a beacon block (2a). Before including a request, a proposer performs several validity checks (e.g., to check that the request is signed correctly). If the request is valid, the block containing it *initiates* the revocation process by temporarily *disabling* the validator. While disabled, any block proposals or attestations made by the validator on chains that descend from the initiating block are ignored. The validator may incur inactivity penalties on a chain for which it is disabled, but is *not* subject to any slashing penalties arising from its revoked signing key.

On receiving a block containing a valid revocation request, other validators add it to their local store as with any other block ③. In addition, they mark the validator as temporarily disabled *at the view level*, i.e. they ignore any attestations of the validator on *any* branch of the current unfiltered block tree when making fork-choice decisions. This ensures that validators with ongoing revocations cannot influence fork-choice decisions while slashing penalties are suspended on one or more branches.

A revocation operation completes successfully if a checkpoint containing the revocation is *finalised*. From this point on, the validator is re-enabled and its new signing key is activated (e.g. key `k1′` of `v1` in step ④). `k1′` can now be used to sign block proposals and attestations. Conversely, if a conflicting chain that does not include the revocation request is finalised, the revocation operation fails and must be resubmitted. For example, the revocation request in step 2b for `v2` is not finalised. In this scenario, `v2` is re-enabled in step 4b, but its original signing key `k2` remains active.

Revoke also adapts processing of *slashing evidence* (e.g. conflicting attestations by the same validator) to account for revocations. On a specific chain, *Revoke* enforces that each block may only include slashing evidence for keys that have not already been revoked, since otherwise an attacker can always fabricate slashing evidence retrospectively. However, a validator is still subject to slashing penalties for slashing evidence included on a chain in a block that precedes a revocation request block. At the view level, the situation is more complex, since revocations and slashings may arrive in different orders (or not at all) on different branches. *Revoke* therefore temporarily disables a validator at the view level when slashing evidence is observed on *any* branch of the block tree, as with revocations. If the slashing evidence is eventually finalised, the validator is permanently ignored at the view level. Otherwise, if it can never be finalised it is re-enabled.

4 Revocation Algorithms

In this section, we describe in detail the algorithms underpinning *Revoke*'s decentralised revocation mechanism for Ethereum. We group the algorithms into two categories: *Chain level* algorithms that operate on a specific chain and *View level* algorithms that operate across all chains in a validator's current view.

4.1 Chain level

At the chain level, the revocation process consists of two steps: *initiation* and *completion*, both of which only ever modify signing keys. Revocation initiation happens on a specific chain when a validator's state transition function processes a new beacon block containing a revocation request. Revocation only completes on a chain if a checkpoint block that descends from the initiating block is eventually finalised.

Initiating Revocation. Algorithm 1 describes the revocation initiation algorithm. A revocation (public-key change) request *pkc* received as part of a beacon block header is processed by the ProcessPkChange function (line 1). ProcessPkChange also takes as a parameter a signature *sig* over *pkc* created using the validator's withdrawal key, and the current beacon chain state *bs*. The key tasks when initiating revocation are to (i) ensure each revocation request is valid given the state of the chain (lines 3-5) and (ii) temporarily disable any validator(s) requesting revocation (lines 6-11).

Algorithm 1 Revocation Initiation

```
 1: proc $\textsc{ProcessPkChange}(pkc, sig, bs)$
 2:     with: $v \leftarrow bs.vals[pkc.vIdx]$
 3:     pre: $v.pk = pkc.pk \wedge v.pkEnb \wedge$
 4:         $v.pk \neq pkc.newPk \wedge (\forall\ span \in v.prevPks, span.key \neq pkc.newPk) \wedge$
 5:         $\textsc{VerifySig}(pkc.wKey, pkc, v.wKeyHash, sig)$
 6:     $v.pkEnb \leftarrow \mathsf{False}$
 7:     $v.prevPks \leftarrow v.prevPks \mathbin{\|} [\textsc{PKSpan}(v.pk, v.pkInitSlot, v.pkEnbSlot)]$
 8:     $v.pk \leftarrow pkc.newPk$
 9:     $v.pkInitSlot \leftarrow bs.slot$
10:     $v.pkEnbSlot \leftarrow \top$
11:     $v.pkChgEp \leftarrow \textsc{ComputePkChgEp}(bs, v)$
```

A public key change request $pkc = (vIdx, fromPk, pk, newPk)$, where *vIdx* is the index in the current beacon state *bs* of the validator $v = bs.vals[vIdx]$ requesting the change. To validate *pkc*, ProcessPkChange first confirms that *pkc.pk*, the current public signing key indicated in the request, matches the public signing key of *v*, and that the *v.pkEnb* flag indicates *v*'s signing key is not already disabled due to an ongoing revocation request (line 3).

Next, the uniqueness of the proposed new signing key *pkc.newPk* is verified with respect to the validator's current key *v.pk* and a history *v.prevPks* of the validator's previous keys (line 4). This simplifies slashing handling at the view level (Section 4.2), since attestations or blocks signed with a particular key can be associated with a single contiguous time period. The signing key history is stored as a sequence of *spans*. Each span $span = (ppk, initSlot, enbSlot) \in \mathsf{PKSpan}$ records when the request to change to the previous public key *ppk* was initiated (*initSlot*) and when *ppk* was finally enabled (*enbSlot*). The slot when *ppk* was eventually disabled is inferred from the initiation slot of the key that replaced it. Finally, VerifySig checks the request signature *sig* against *v*'s withdrawal key (line 5). The existing specification only stores a hash of the validator's withdrawal public key in *v.wKeyHash*. To accommodate this, the withdrawal public key is explicitly included in the request as $pkc.wKey$. VerifySig ensures the hash of $pkc.wKey$ matches the one stored in $v.wKeyHash$.

If ProcessPkChange's preconditions hold, it disables v by clearing the $v.pkEnb$ flag (line 6), and appends a span capturing the activity period of the validator's old public signing key $v.pk$ to $v.prevPks$ (line 7). Next, it records the proposed new public key $newPk$ in the validator's state as $v.pk$ (line 8), and the slot in which the request was initiated in $v.pkInitSlot$ (line 9). If revocation is eventually successful, $v.pkEnbSlot$ will record the slot in which $v.pk$ is re-enabled. However, when this occurs is not yet known so it is set temporarily to a dummy value (line 10). To support rate limiting of revocation requests, $v.pkChgEp$ is set to the earliest epoch in which the validator can potentially be re-enabled, as computed by the ComputePkChgEp function (not shown).

Algorithm 2 Revocation Completion

```
1: proc EnableFinalisedPkChg($bs$)
2:     for $v \in bs.vals, \neg v.pkEnb$ do
3:         if $v.pkInitSlot < \textsc{StartSlot}(bs.fcp.ep) \wedge v.pkChgEp < bs.fcp.ep$ then
4:             $v.pkEnb \leftarrow \mathsf{True}$
5:             $v.pkEnbSlot \leftarrow bs.slot$
6:             $v.pkChgEp \leftarrow \bot$
```

Completing Revocation. Revocation completion at the chain level is handled by the EnableFinalisedPkChg procedure, which is triggered whenever a new checkpoint is finalised on the chain (Algorithm 2, line 1). For each currently disabled validator v, as indicated by the flag $v.pkEnb$ (line 2), it checks whether revocation was initiated before the most recently finalised checkpoint block $bs.fcp$. In the absence of rate limiting, it does this by comparing $v.pkInitSlot$ and the starting slot of the finalised checkpoint's epoch $bs.fcp.ep$ (line 3). Since rate limiting may have occurred, it also checks that $v.pkChgEp$ is earlier than the finalised checkpoint epoch (line 3). Finally, the validator is re-enabled (line 4), the current slot is recorded to facilitate subsequent validation of any slashing evidence observed for the new key (line 5), and the $pkChgEp$ field used for rate limiting is reset to a dummy value (line 6).

4.2 View level

We next describe how *Revoke* impacts on fork-choice and finality decisions at the view level. Three main components are affected: block processing, attestation processing, and processing of slashing evidence.

Block Processing. Given a new block b, the OnBlock function processes it and updates a *store* object st representing the validator's current network view (Algorithm 3, line 1). In *Revoke*, OnBlock must be adapted to (i) record in st any revocations and slashings in b and temporarily disable the associated validators (e.g. ignore their attestations) (ii) re-enable or permanently disable validators in *st* as revocations and slashings are finalised as a result of block processing.

Algorithm 3 Block Processing

```
 1: proc $\textsc{OnBlock}(st, b)$
 2:     for $idx \mid pkc \in b.pkcs \land idx = pkc.vIdx \land idx \notin st.equivIdxs$ do
 3:         $st.pendRevIdxs[idx] \leftarrow st.pendRevIdxs[idx] \cup \{\textsc{htr}(b)\}$
 4:     for $idx \mid as \in b.attSlash \land idx \in as.att1.idxs \cap as.att2.idxs \land$
 5:                                                    $idx \notin st.equivIdxs$ do
 6:         $st.pendEquivIdxs[idx] \leftarrow st.pendEquivIdxs[idx] \cup \{\textsc{htr}(b)\}$
 7:     if $\textsc{newFinalization}(st)$ then
 8:         $\textsc{prune}(st)$
 9: proc $\textsc{prune}(st)$
10:     $liveBlks \leftarrow \textsc{getLiveBlks}(st)$
11:     for $(idx, revBlks) \in st.pendRevIdxs$ do
12:         $st.pendRevIdxs[idx] \leftarrow revBlks \cap liveBlks$
13:     for $(idx, equivBlks) \in st.pendEquivIdxs$ do
14:         if $\exists b \in equivBlks \mid \textsc{isFinalized}(b, st)$ then
15:             $st.equivIdxs \leftarrow st.equivIdxs \cup \{idx\}$
16:             $st.pendEquivIdxs[idx] \leftarrow \varnothing$
17:         else
18:             $st.pendEquivIdxs[idx] \leftarrow equivBlks \cap liveBlks$
```

For each revocation request *pkc* in $b.pkcs$, OnBlock extracts the corresponding validator index *idx* unless the validator has already been permanently marked as equivocating in the set $st.equivIdxs$ (line 2). OnBlock then records in $st.pendRevIdxs$ that the resulting validators have a *pending revocation* on the chain for which b is the head block (line 3), where $st.pendRevIdxs$ is a map that records for each validator the Merkle hash tree root HTR(b) of any block b containing a pending revocation for the validator. A validator with index *idx* is thus considered to have a pending revocation if $st.pendRevIdxs[idx]$ is non-empty.

Next, OnBlock checks whether b contains any new slashings. It iterates over the attester slashings $as \in b.attSlash$ and extracts the indices of all potentially slashable validators, ignoring any that are already recorded as permanently equivocating (line 5). For each index *idx*, OnBlock adds the hash HTR(b) to $st.pendEquivIdxs[idx]$, where $st.pendEquivIdxs$ is a map recording the set of *pending equivocations* for each validator (line 6), i.e. the blocks in the current view in which slashing evidence has been observed but not yet included in a finalised block.

The final part of OnBlock is executed when there is a new finalised checkpoint in the current view, as indicated by the NewFinalization function (not shown) (line 7). If so, OnBlock calls the Prune procedure (line 8).

The Prune procedure first calls the getLiveBlks function (not shown) to compute the set of live blocks *liveBlks* in the view, i.e. those blocks that can still potentially be finalised as they do not conflict with the finalised chain (line 10). It then updates $st.pendRevIdxs$ to remove any pending revocations that are not in a live block (lines 11–12). Next, it checks whether there are any pending equivocations in $st.pendEquivIdxs$ such that the block containing the equivocation has been finalised, as indicated by the isFinalized function (not shown) (line 14). If so, it adds the index of the corresponding validator to $st.equivIdxs$, the set of permanently equivocating (slashed) validators (line 15), and clears all pending equivocations for the validator (line 16). Otherwise, it updates $st.pendEquivIdxs$ to remove any pending equivocations (slashings) that are no longer in a live block (line 18).

Attestation Processing. The OnAttestation procedure processes new attestations at the view level (Algorithm 4, line 1). OnAttestation is responsible for (i) validating the well-formedness of each new attestation $att = (blkRoot, slot, idxs, src, tgt)$ (lines 2–3) and (ii) recording the block $att.blkRoot$ as the validators' latest vote for the canonical chain's head if it is more recent than the previous vote (lines 4–8). In addition to the attestation, OnAttestation takes as parameters the current store *st* and an aggregate signature *sigAgg* over *att* that combines the individual signatures of many attesting validators.

Algorithm 4 Attestation Processing

```
1: proc OnAttestation($st, att, sigAgg$)
2:     with: $pubKeys \leftarrow$ GetAttestKeys($st, att.slot, att.idxs$)
3:     pre: IsValidIdxAttest($st, att, sigAgg, pubKeys$)

4:     $validAttIdxs \leftarrow \{idx \in att.idxs \setminus st.equivIdxs \mid$
5:                      $(st.pendEquivIdxs[idx] \cup st.pendRevIdxs[idx]) = \varnothing\}$
6:     for $idx \in validAttIdxs$ do
7:         if $idx \notin st.latestMsgs \lor st.latestMsgs[idx].ep < att.tgt.ep$ then
8:             $st.latestMsgs[idx] \leftarrow$ LatestMsg($att.tgt.ep, att.blkRoot$)
```

The main challenge with validating *att* in Revoke is that Ethereum signature verification assumes validator keys do not change. Revoke must instead verify *sigAgg* using the specific validator keys for the slot the attestation was created ($att.slot$). To achieve this, the GetAttestKeys function (not shown) retrieves from *st* the signing keys for the validators indicated by $att.idxs$ during $att.slot$ (line 2). These keys are passed to the isValidIdxAttest function (not shown), which then verifies *sigAgg* (line 3). In addition, isValidIdxAttest performs a range of sanity checks (e.g. on the *src* and *tgt* checkpoints that form the finality vote of *att*), but these are unchanged from Ethereum.

Once an attestation is validated, OnAttestation records the votes of the attesting validators (lines 4-8). In Ethereum votes from permanently slashed validators, i.e. those in *st.equivIndices*, are ignored. In Revoke, the main change is that votes from validators who are disabled because of a pending revocation or slashing are also ignored (line 5). The set of validators to update are computed and stored in *validAttIdxs*. The votes (*st.latestMsgs*) of these validators are then updated (lines 6-8) if the epoch of the attestation's target checkpoint block (*att.tgt.ep*) is more recent than the current latest vote.

Standalone Attester Slashing Processing. Standalone evidence of attester slashings allows validators to handle equivocation immediately at the view level without waiting for inclusion in a block. This is problematic in *Revoke* as receiving slashings and revocations in different orders at different validators can result in inconsistent decisions on whether a validator should be slashed.

In *Revoke*, standalone slashing evidence is instead handled by initially classifying implicated validators as pending equivocators. These validators' attestations are temporarily disregarded until two conditions are met: (i) the slashing evidence is included in a finalised block, and (ii) no revocation by the validator has been finalised before this block. This ensures more consistent treatment of slashing evidence at different validators. Due to space constraints, we omit the detailed description of the algorithmic changes for handling standalone slashings.

4.3 Ethereum Implementation

We implement *Revoke* based on the Capella version [22] of the Ethereum executable Python specification. Our current prototype supports the core revocation mechanisms outlined in Section 4. In total our implementation adds approximately 325 LoC to the original 2364 LoC of the specification and approximately 5k LoC of tests and testing infrastructure to the existing Ethereum test suite and test framework. Our implementation demonstrates the feasibility of incorporating *Revoke* into a future upgrade (hard fork) of Ethereum. Finally, we have backported *Revoke* to a Dafny based implementation of the Phase 0 version of Ethereum [15] as a stepping stone towards verifying the correctness of our implementation (e.g. to prove the absence of runtime errors). We will release our modified Python specification as open source [38].

5 Correctness

In this section, we sketch a proof for the correctness of *Revoke* with respect to safety and plausible liveness properties of Ethereum. Our proof sketch intentionally follows closely the existing proof strategy for vanilla Gasper [12], but with adjustments to allow for the possibility that some bounded fraction of validators may have changed their key and hence become temporarily immune to slashing. In effect, *Revoke* enables a trade off between the benefits of revocation for individual compromised validators and the stake required by an attacker to violate the blockchain protocol without the risk of slashing.

5.1 Preliminaries

Given a chain, Gasper picks certain blocks to play the role of *checkpoint* blocks. Ideally one block is chosen per epoch, but if no blocks occur in an epoch a block from a previous epoch is used instead. To distinguish checkpoints that use the same block, Gasper introduces the notion of ordered *epoch boundary pairs* (B, j) of a chain, where B is a block and j an epoch. Given a pair (B, j) we define its *attestation epoch* $\mathsf{aep}(B, j) = j$, which is not necessarily the same as the epoch $\mathsf{ep}(B)$ of B.

Given a block B its j-th *epoch boundary block* $B' = \mathsf{EBB}(B, j)$ is defined as the block with the highest slot less than or equal to jC in $\mathsf{chain}(B)$, and $\mathsf{LEBB}(B)$ as the last epoch boundary block of B.

Gasper then defines $\mathsf{J}(G)$ and $\mathsf{F}(G)$ as the sets of justified and finalized *pairs* in view G, respectively. A checkpoint (B, j) is considered justified if it is the genesis pair $(B_{genesis}, 0)$, or a *supermajority link* of attestations with checkpoint edge $(A, j') \rightarrow (B, j)$ exist having total weight over $\frac{2}{3}$ of the total validator stake. A checkpoint (B, j) is considered finalised if (B, j) is justified and there is a supermajority link to some pair $(B', j + 1)$.

Given an attestation α for $\mathsf{block}(\alpha)$ at $\mathsf{slot}(\alpha)$, let $B = \mathsf{LEBB}(\mathsf{block}(\alpha))$. Then for the checkpoint edge $\mathsf{LJ}(\alpha) \rightarrow \mathsf{LE}(\alpha)$, we define $\mathsf{LJ}(\alpha)$ as the *last justified pair* of the view containing all ancestor blocks of B, and $\mathsf{LE}(\alpha)$ as the *last epoch boundary pair* of α, i.e. $(B, \mathsf{ep}(\mathsf{slot}(\alpha)))$.

5.2 Revoke Definitions

To adapt the Gasper proof of correctness for *Revoke*, we begin by redefining Gasper's notion of *p-slashability* [12] to account for the possibility of revocation. We also adjust the slashing conditions of Ethereum to explicitly exclude attestations signed by the same validator with different keys.

(P,R)-Slashability. We define a blockchain running the protocol to be (p, r)-slashable if validators with a total of $\mathsf{max}(p - r, 0)N$ stake can be provably slashed by a validator with the network view, where $r = |R|$ is the weight of the fraction of validators R who have changed their key.

Slashing Conditions. We define the following modified slashing conditions (differences underlined).

- (S1) No validator makes two distinct attestations $\alpha 1$, $\alpha 2$ with epoch $\mathsf{ep}(\alpha 1) = \mathsf{ep}(\alpha 2)$ $\underline{\text{using the same validator key}}$. Note this condition is equivalent to $\mathsf{aep}(\mathsf{LE}(\alpha 1)) = \mathsf{aep}(\mathsf{LE}(\alpha 2))$.
- (S2) No validator makes two distinct attestations $\alpha 1$, $\alpha 2$ with $\mathsf{aep}(\mathsf{LJ}(\alpha 1)) < \mathsf{aep}(\mathsf{LJ}(\alpha 2)) < \mathsf{aep}(\mathsf{LE}(\alpha 2)) < \mathsf{aep}(\mathsf{LE}(\alpha 1))$ $\underline{\text{using the same validator key}}$.

5.3 Safety

Theorem 1 (Safety). In a view G, if we do not have the following properties, then G is $(1/3, r)$-slashable:

1. Any pair in $\mathsf{F}(G)$ stays in $\mathsf{F}(G)$ as the view is updated.
2. If $(B, j) \in \mathsf{F}(G)$, then B is in the canonical chain of G.

Proof. For the proof see Appendix §9.1.

5.4 Liveness

Ethereum's proof of plausible liveness requires $\frac{2N}{3}$ stake worth of honest validators in order to justify and finalise checkpoints. However, in *Revoke*, validators that initiate a revocation are disabled until it is finalised. If these validators are otherwise honest, potentially there will be insufficient honest validators remaining to drive the protocol. We therefore define a parameter $\tau < r$ as the maximum fraction of stake for which the corresponding validators attempt to revoke their key and are hence deactivated at the same time. Hence $\tau > 0$ reduces the security margin of Ethereum, but only includes validators who are currently revoking their keys and only impacts liveness.

Theorem 2 (Plausible Liveness). If at least $\frac{2N}{3} + \tau$ stake worth of the validators are honest, then it is always possible for a new block to be finalised with the honest validators continuing to follow the protocol, no matter what happened previously to the blockchain.

Proof. For the proof see Appendix §9.2.

6 Revocation Incentives

Atomic Revocation Under the threat model of *Revoke* an attacker monitoring the network that observes a revocation can broadcast slashing evidence. This lack of *atomic* revocation results in a race between victim validators and attackers to have their request included in a block.

This problem is related to *order manipulation* in blockchains, whereby adversaries attempt to gain an advantage by manipulating the ordering and inclusion of transactions in blocks [4, 17, 27, 43]. To mitigate this, recent work has explored the design of data-independent ordering protocols [13, 29, 30, 31, 33, 34, 42]. However, these protocols do not take into account the crypto-economic incentives in Ethereum's threat model, assuming instead a classical BFT setting where honest nodes follow the protocol independently of whether it is economically rational. In particular, recent work has demonstrated the impossibility of creating an incentive-compatible order policy enforcement (OPE) framework where colluding parties can be held accountable for violating the ordering protocol [41]. For example, a basic commit-reveal style scheme that allows a validator to first commit to a revocation privately and then reveal its identity in a later block must either forgive any slashable behaviour between the commit and reveal or allow the proposer who receives the reveal transaction to front-run it with slashing evidence.

Revocation reward To work around this impossibility result, we propose an approach whereby proposers are incentivised to include revocations ahead of slashings through the inclusion of a *revocation reward*. Our key insight is that in Ethereum, the incentive for a proposer to include slashing evidence is $b/16$, where b is the base slashing penalty. In the absence of collusion between a proposer and the attacker, a victim validator need only therefore include a small fraction of the base slashing penalty to incentivise a proposer to include a revocation[2].

One concern with this approach is that a dishonest validator caught misbehaving can abuse the revocation mechanism to 'frontrun' slashing evidence, e.g. by submitting a revocation request with a slightly higher reward than the slashing reward (i.e. $b/16 + \epsilon$). To mitigate this, we limit the direct revocation reward to $b/16$ to balance the maximum slashing reward. However, this is only a partial solution, since an attacker can always attempt to pay an additional reward to a proposer out-of-band.

Conversely, a ransomware attacker may choose to collude with a proposer, potentially offering to pay an additional reward to the proposer for front-running a revocation request with slashing evidence. Although in this situation, the ransomware attacker will not receive any payment from the victim and hence lose money, it may be an effective strategy for boosting its credibility for future ransomware attacks. Such an increase in costs for mounting a ransomware attack could be considered a benefit of Revoke.

Finally, we note the increasing importance of MEV and proposer-builder separation (PBS) in the Ethereum ecosystem [10, 17]. Game theoretic modelling of revocation incentives and strategies in the context of emerging PBS platforms is an interesting avenue for future work [37].

7 Related Work

Attacks against PoS. The risks of ransomware attacks against Ethereum PoS systems have been explored in prior work [6, 7, 8] from a game theoretic modelling perspective. No specific mitigations were proposed for individual validators to recover from ransomware attacks. Beyond ransomware attacks, Deirmentzoglou et al. survey [19] long-range attacks against PoS blockchains and potential mitigations. More recently, Schwarz et al. examined [39] vulnerabilities in Ethereum's PoS protocol. However the nature of these vulnerabilities and their associated mitigation techniques are very different to ransomware attacks.

Blockchain Key-Change Mechanisms. Of relevance to Revoke, David et al. [18] employed key evolving cryptography [25, 28] to prevent long-range attacks from reusing the signing keys of compromised validators. Key evolution prevents a ransomware attacker from exploiting deleted signing keys for retrospective slashing. However, since a compromised pri-

Table 1: PoS blockchain key revocation features

Blockchain	Type	Revocation	Slashing	Separation
Algorand	PPoS	✗ / ✓	✗	✓
Cardano	DPoS	✗ / ✓	✗	✓
Polkadot	NPoS	✗ / ✓	✓	✓
Tezos	LPoS	✗ / ✓	✓	✓
Ethereum	PoS	✗	✓	✓
Revoke	PoS	✓	✓	✓

[2] We ignore any increase in Ether's value from burning the remaining slashing penalty

vate signing key implicitly gives access to future keys, a ransomware attacker can simply delay launching an attack to ensure it has a sufficient window of old signing keys for retrospective slashing.

We analysed several prominent blockchains to understand if they support key revocation ([1, 2, 26, 32]). We note that all blockchains other than Ethereum support key *rotation* mechanisms that allow to change the key(s) used for consensus operations (Table 1, column 'Revocation'). However, these mechanisms are not robust against ransomware, since stakers generally remain accountable for actions performed using an old key for a period after key rotation.

Ransomware in other PoS blockchains. Beyond Ethereum, other PoS blockchains could potentially benefit from a revocation mechanism such as *Revoke*.

We consider a blockchain susceptible to ransomware if two conditions hold: (i) the key used for consensus operations must be separate to the key that control's access to stakes, since otherwise an attacker can immediately steal a victim's stake (ii) the blockchain must have some form of direct or indirect accountability mechanism the attacker can subvert to gain leverage over the victim (e.g. stake slashing or reputation loss). As can be seen in Table 1, the blockchains we analysed support key separation. However, only Tezos and Polkadot perform slashing. Algorand and Cardano do not and so are not directly susceptible to ransomware attacks.

8 Conclusions

We propose *Revoke*, a novel decentralised key revocation mechanism for Ethereum to mitigate blockchain ransomware attacks. *Revoke* exposes a trade-off whereby validators cannot propose or attest to blocks during the revocation process, and hence incur inactivity penalties, but are not susceptible to much larger slashing penalties. We implement *Revoke* as part of the Ethereum executable Python specification and adapt the existing safety and liveness proofs for Ethereum's core consensus protocol to model *Revoke*.

Acknowledgements

We thank Aditya Asgaonkar from the Ethereum Foundation for his feedback throughout this research. This project was supported by the Ethereum Foundation Grant #FY22-0720 (REVOKE: Consensus-layer mitigations for validator ransomware attacks) [21]. Bhudia is supported by the UK EPSRC Centre for Doctoral Training in Cyber Security at Royal Holloway University of London (EP/P009301/1). O'Keeffe was partially supported by the CHIST-ERA grant CHIST-ERA-22-SPiDDS-05 (REDONDA project) and EPSRC (EP/Y036417/1).

Bibliography

[1] Abbas, H., et al.: Analysis of polkadot: Architecture, internals, and contradictions. In: Int. Conference on Blockchain. pp. 61–70 (2022)

[2] Allombert, V., et al.: Introduction to the tezos blockchain. In: Int. Conference on HPCS '19. pp. 1–10 (2019)

[3] Aumasson, J.P., Kolegov, D., Stathopoulou, E.: Security review of ethereum beacon clients. arXiv:2109.11677 (2021)

[4] Baum, C., et al.: SoK: Mitigation of front-running in decentralized finance. In: Int. Conference on FC '23. pp. 250–271 (2022)

[5] Beaconcha.in: Ethereum validator slashings (2024), https://beaconcha.in/validators/slashings

[6] Bhudia, A., et al.: Extortion of a staking pool in a proof-of-stake consensus mechanism. In: Int. Conference on COINS '22 (2022)

[7] Bhudia, A., et al.: Identifying incentives for extortion in proof of stake consensus protocols. In: Int. Conference on DBB '22 (2022)

[8] Bhudia, A., et al.: Game theoretic modelling of a ransom and extortion attack on ethereum validators. In: ARES '23. pp. 1–11 (2023)

[9] Boneh, D., Ding, X., et al.: A method for fast revocation of public key certificates and security capabilities. In: 10th USENIX '01 (2001)

[10] Buterin, V.: State of research: Increasing censorship resistence of transactions under proposer/builder separation (pbs) (2021), https://notes.ethereum.org/@vbuterin/pbs_censorship_resistance

[11] Buterin, V., Griffith, V.: Casper the friendly finality gadget. arXiv:1710.09437 (2017)

[12] Buterin, V., Hernandez, D., et al.: Combining ghost and casper. arXiv:2109.11677 (2021)

[13] Cachin, C., et al.: Quick order fairness. In: FC '22 (2022)

[14] Cardano: Cardano home page (2024), https://cardano.org/

[15] Cassez, F., Fuller, J., Asgaonkar, A.: Formal verification of the ethereum 2.0 beacon chain. In: Int. Conference on TACAS '22. pp. 167–182 (2022)

[16] Chen, J., Micali, S.: Algorand. arXiv:1607.01341 (2016)

[17] Daian, P., et al.: Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. arXiv:1904.05234 (2020)

[18] David, B., et al.: Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In: EUROCRYPT '18 (2018)

[19] Deirmentzoglou, E., Papakyriakopoulos, G., Patsakis, C.: A survey on long-range attacks for proof of stake protocols. IEEE access (2019)

[20] Edgington, B.: Upgrading ethereum (2023), https://eth2book.info/capella/part3/helper/predicates/#is_slashable_attestation_data

[21] Ethereum Foundation: Academic grants round grantee announcement (2022), https://blog.ethereum.org/2022/07/29/academic-grants-grantee-announce

[22] Ethereum Foundation: Ethereum proof-of-stake consensus capella specifications (2023), https://github.com/ethereum/consensus-specs/blob/dev/specs/capella/beacon-chain.md

[23] Ethereum Foundation: Ethereum proof-of-stake consensus specifications (2024), https://github.com/ethereum/consensus-specs

[24] Fanti, G., Kogan, L., Viswanath, P.: Economics of proof-of-stake payment systems. In: Working paper (2019)

[25] Franklin, M.: A survey of key evolving cryptosystems. Int. Journal of Security and Networks (2006)

[26] Gilad, Y., Hemo, R., et al.: Algorand: Scaling byzantine agreements for cryptocurrencies. In: 26th SOSP '17 (2017)

[27] Heimbach, L., Wattenhofer, R.: Sok: Preventing transaction reordering manipulations in decentralized finance. In: 4th AFT '23 (2023)

[28] Itkis, G., Reyzin, L.: Forward-secure signatures with optimal signing and verifying. In: Advances in Cryptology—CRYPTO '01 (2001)

[29] Kelkar, M., Deb, S., et al.: Order-fair consensus in the permissionless setting. In: 9th APKC Workshop '22 (2022)

[30] Kelkar, M., Deb, S., et al.: Themis: Fast, strong order-fairness in byzantine consensus. In: CCS '23 (2023)

[31] Kelkar, M., Zhang, F., et al.: Order-fairness for byzantine consensus. In: Advances in Cryptology – CRYPTO '20 (2020)

[32] Kiayias, A., Russell, A., et al.: Ouroboros: A provably secure proof-of-stake blockchain protocol. In: Crypto '17 (2017)

[33] Kursawe, K.: Wendy, the good little fairness widget: Achieving order fairness for blockchains. In: 2nd AFT '20 (2020)

[34] Malkhi, D., Szalachowski, P.: Maximal Extractable Value (MEV) Protection on a DAG. In: 4th Tokenomics '23 (2023)

[35] Myers, S., Shull, A.: Practical revocation and key rotation. In: RSA Conference '18 (2018)

[36] Polkadot: Web3 interoperability (2024), https://polkadot.network/

[37] Research, F.: The future of MEV is SUAVE, https://writings.flashbots.net/the-future-of-mev-is-suave

[38] S3lab-rhul: Revoke: Github repository (2024), https://github.com/s3lab-rhul/revoke

[39] Schwarz-Schilling, C., Neu, J., et al.: Three attacks on proof-of-stake ethereum. In: Int. Conference on FC'22 (2022)

[40] Sompolinsky, Y., Zohar, A.: Secure high-rate transaction processing in bitcoin. In: Int. Conference in FC'15 (2015)

[41] Wadhwa, S., et al.: Breaking the chains of rationality: Understanding the limitations to and obtaining order policy enforcement. IACR '23 (2023)

[42] Zhang, Y., Setty, S., et al.: Byzantine ordered consensus without byzantine oligarchy. In: 14th USENIX Symposium on OSDI '20 (2020)

[43] Zhou, L., Qin, K., et al.: High-frequency trading on decentralized on-chain exchanges. In: 42nd SP '21 (2021)

9 Appendix

9.1 Safety

Lemma 1. In a view G, for every epoch j, there is at most 1 pair (B, j) in $\mathsf{J}(G)$, or the blockchain is $(1/3,\ r)$-slashable. In particular, the latter case means there must exist 2 subsets $\mathcal{V}_1, \mathcal{V}_2$ of $\mathcal{V}$, each with total weight at least $\frac{2N}{3}$, such that at least $(\mathcal{V}_1 \cap \mathcal{V}_2)/R$ violate slashing condition (S1).

Proof. Suppose we have 2 distinct pairs (B, j) and (B', j) in $\mathsf{J}(G)$ (justified blocks in G). This means in epoch j, more than a total stake of $\frac{2N}{3} - r$ attested with a checkpoint edge to (B, j) and more than $\frac{2N}{3} - r$ stake attested with a checkpoint edge to (B', j). These are our desired $\mathcal{V}_1, \mathcal{V}_2$ (validators set).

Lemma 2. In a view G, if $(B_F, f) \in \mathsf{F}(G)$ and $(B_J, j) \in \mathsf{J}(G)$ with $j > f$, then B_F must be an ancestor of B_J, or the blockchain is $(1/3, r)$-slashable. Specifically, there must exist two subsets $\mathcal{V}_1, \mathcal{V}_2$ of $\mathcal{V}$, each with total stake at least $\frac{2N}{3}$, such that at least $\mathcal{V}_1 \cap \mathcal{V}_2/R$ all violate slashing condition (S1) or all violate slashing condition (S2).

Proof. Anticipating contradiction, suppose there is a pair (B_J, j) with $j > f$ and B_J is not a descendant of B_F. By definition of finalisation, in G, we must have $(B_F, f) \to \mathsf{J} \to (B_k, f + k)$, where we have a sequence of adjacent epoch boundary pairs $(B_F, f), (B_1, f + 1), \ldots, (B_k, f + k)$. Since (B_J, j) is justified and B_J is not a descendant of B_F, without loss of generality (by going backwards with supermajority links), we can assume (B_J, j) is the earliest such violation, meaning we can assume $(B_l, l) \to \mathsf{J} \to (B_J, j)$ where $l < f$ but $j > f$.

Here we use Lemma 1, which tells us no two justified blocks have the same $\mathsf{aep}()$, else we are done already with two validator subsets of weight $\frac{2N}{3}$ where their intersection (excluding validators in R) are violating (S1). This is why we do not worry about the equality case.

Since $B_1, \ldots, B_k$ are all justified but are descendants of B_F, we know B_J cannot be any of these blocks, so we must have $j > f + k$. This means the view G sees some subset $\mathcal{V}_1$ of $\mathcal{V}$ with total stake more than $\frac{2N}{3}$ have made attestations justifying a checkpoint edge $(B_l, l) \to (B_J, j)$, so for any such attestation α_1, $\mathsf{aep}(\mathsf{LJ}(\alpha_1)) = l$ and $\mathsf{ep}(\alpha_1) = j$. Similarly, G also sees more than $\frac{2N}{3}$ weight worth of validators $\mathcal{V}_2$ have made attestations justifying $(B_F, f) \to (B_k, f + k)$, so for any such attestation α_2, $\mathsf{aep}(\mathsf{LJ}(\alpha_2)) = f$ and $\mathsf{ep}(\alpha_2) = f + k$. Thus, for anyone in the intersection $\mathcal{V}_1 \cap \mathcal{V}_2$, they have made two distinct attestations α_1 of the former type and α_2 of the latter type. Because $l < f < f + k < j$, we know $\mathsf{aep}(\mathsf{LJ}(\alpha_1)) < \mathsf{aep}(\mathsf{LJ}(\alpha_2)) < \mathsf{ep}(\alpha_2) < \mathsf{ep}(\alpha_1)$, which allows them to be provably slashed by (S2) unless they are in R and have attested with different keys.

Theorem 1 (Safety). In a view G, if we do not have the following properties, then G is $(1/3, r)$-slashable:

1. Any pair in $\mathsf{F}(G)$ stays in $\mathsf{F}(G)$ as the view is updated.
2. If $(B, j) \in \mathsf{F}(G)$, then B is in the canonical chain of G.

Proof. The first property is straightforward from the definitions of $\mathsf{F}(G)$ and $\mathsf{J}(G)$, so we omit the proof.

The definition of the Hybrid LMD GHOST Fork Choice Rule (Algorithm 4.2 in [12]) shows that the canonical chain always goes through the justified pair with the highest attestation epoch j, so by Lemma 2, it must go through the highest finalised pair in $\mathsf{F}(G)$. Thus, it suffices to show that no two finalised blocks conflict because then all the finalised blocks must lie on the same chain, which we just showed must be a subset of the canonical chain.

We now show that if (B_1, f_1), $(B_2, f_2) \in \mathsf{F}(G)$ but B_1 and B_2 conflict, then G is $(1/3, r)$-slashable. Without loss of generality, assume $f_2 > f_1$. Since (B_2, f_2) is finalised, it is justified, and we can apply Lemma 2. This shows that either B_2 must be a descendant of B_1 (assumed to be impossible since they conflict) or G is $(1/3, r)$-slashable, as desired.

9.2 Liveness

Theorem 2 (Plausible Liveness). If at least $\frac{2N}{3}+\tau$ stake worth of the validators are honest, then it is always possible for a new block to be finalised with the honest validators continuing to follow the protocol, no matter what happened previously to the blockchain.

Proof. Suppose we are starting epoch j. Specifically, suppose our current slot is $i = Cj$; then it is plausible (by having good synchrony) for everyone to have the same view. In particular, the proposer, who is plausibly honest, would then propose a new block B with slot i, which is a child of the output block of $\mathsf{HLMD}()$ run on the old view. Call this current view (including B) G and define $\mu = \mathsf{chain}(B)$. Keep in mind that $\mathsf{LEBB}(B) = B$; we are introducing a new epoch boundary block.

Now, we claim that it is plausible that μ extends to a stable chain at the beginning of the next epoch $(j+1)$. To see this, note that since at least $\frac{2N}{3}+\tau$ stake worth of the validators are honest, it is plausible that they all attest for B or a descendant (for example, if they are all synced with the network view and vote immediately after B is created). This creates a supermajority link $\mathsf{LJ}(B) \to \mathsf{J} \to (B, j)$, justifying B. Now, in the next epoch $(j+1)$, it is plausible for the new block B' with slot $i + C = (j+1)C$ to include all of these attestations (which would happen with good synchrony), so $\mathsf{LJ}(B') = (B, j)$ and $\mathsf{chain}(B')$ is indeed stable at epoch $(j+1)$. Thus, by assuming good synchrony and honest validators, it is plausible (possibly having to wait 1 extra epoch) that the chain of the first epoch boundary block of the new epoch is stable, no matter what the state of the blockchain was initially.

Thus, we can reduce our analysis to the case that μ was stable to begin with, meaning that we have a supermajority link $(B', j-1) \to \mathsf{J} \to (B, j)$ in $\mathsf{ffgview}(B)$. Then by the same logic above, it is plausible for the next epoch boundary block (B'') with slot $(j+1)C$ to also be stable, with another supermajority link $(B, j) \to (B'', j+1)$. This finalises the pair (B, j); in particular, it is the special case of 1-finalisation.