View-merge to replace proposer boost

Special thanks to Vitalik Buterin, Michael Sproul, Carl Beekhuizen and Caspar Schwarz-Schilling for editing, feedback and fruitful discussions

Note that the idea of view-merge has previously appeared in theHighway Protocol. This wasn't cited in thepreviousethresear.ch post because this part of their protocol was not known and/or understood by any of us at the time

Note also that view-merge, together with the attestation expiry technique which is later used in this post as well, has appeared in this recent paper. While the setting is not exactly the same, some readers might find it educational to see how view-merge (and reorg resilience, which is derived from it and is another concept which appears in the post) is used as the key tool in constructing a provably secure protocol, in particular one which is very related to LMD-GHOST. Moreover, most of the arguments carry over to LMD-GHOST without committees (i.e. with the whole validator set voting every slot), which informs our understanding of the security of the current protocol, and of the path to improve it.

Background on the problem

Firstly, a (not complete) list of useful references on the attacks whichproposer boost mitigates:

- A balancing attack on Gasper, the current candidate for Eth2's beacon chain
- Attacking Gasper without adversarial network delay
- Three Attacks on Proof-of-Stake Ethereum

The LMD-GHOST component of the fork-choice is susceptible to two kinds of attacks: balancing attacks and ex-ante "withholding and saving" attacks. The former rely on publishing messages near a deadline, in such a way as to split the honest attester, and on using the same technique to keep them split in the future, with a small consumption of attestations. The latter are akin to selfish mining: one withholds blocks and attestations and reveals them later to reorg honest blocks produced in the meantime. The two attack techniques can be combined, because saved attestations can be weaponized to keep the honest attester balanced, which in turn leads to the adversary being able to keep saving their attestations.

High-level mitigation idea

To more comprehensively mitigate both kinds of attack, the general strategy is to empower honest proposers to impose their view of the fork-choice, but without giving them too much power and making committees irrelevant. In the balancing case, the desired outcome is that a 50/50 split of honest views is resolved for whatever the proposer sees as leading. The ex ante case is a bit more complicated, because an adversary which controls many slots in a row can manage to save enough attestations to be able to reorg an honest block even if an entire committee attested to it, but we want to at least give honest proposers the tools to protect themselves in less extreme cases. Namely, from adversaries which don't control a very large percentage of the validator set and which don't control many slots before the honest slot in question. Reorgs from weak adversaries which happen to control many slots in a row are rare, so we are less concerned about them.

Proposer Boost

The current mitigation is proposer boost, which simply gives timely proposals extra fork-choice weight during their slot,

which enables them to defend against all attacks which do not have enough saved attestations to overcome that weight.

If the boost were not temporary, the fork-choice would become a linear combination of a committee-based one and a proposer-based one (think of PoW Ethereum, with the weight of a proposal being the difficulty it adds), giving much fork-choice influence to proposers and greatly reducing the benefit of having committees in the first place. In fact, the boost being temporary means it cannot be saved, i.e. one cannot withhold a proposal to later weaponize its boost, and in particular one cannot stack up boost from multiple proposals, so ex ante reorgs really need the attacker to control a lot of attester, and thus a lot of stake.

Still, proposer boost can be weaponized by malicious proposers to conclude a reorg, lowering the amount of attestations needed for it by the weight of boost.

Other mitigations

Looking to the future a bit, there are some potential changes to the fork-choice and to the consensus protocol in general, which among other things mitigate these issues. Nonetheless, they are not complete solutions, as they fail to mitigate balancing, so we still need either proposer boost or some alternative to it.

- (block, slot) attestations, which are required in current proposals for implementing proposer/builder separation (PBS), make it so that ex ante reorgs must also incorporate balancing, by publishing a block near the deadline and keeping the honest attester split between "the empty block" (i.e. the previous head, pulled up to the current slot) and their block.

Just withholding their block does not constitute a successful attack, because honest validators can vote for the empty block if they don't see any by the deadline. Still, reorgs are made no harder in terms of stake requirement, they just need a bit more sophistication in the execution.

- Ex ante reorgs which rely on simply overpowering a full honest committee by saving attestations with a higher cumulative weight can be entirely solved by removing committees, i.e. by having the whole validator set vote at once, which is one of the paths to single-slot finality. This is simply because LMD would then make it impossible for the adversary to "stack up weight" from multiple slots: all they can do is use one attestation per adversarially controlled validator, which is not enough to overpower all honest attestations, assuming we have an honest majority. On the other hand, balancing attacks are still a threat.

View-merge

High level idea

The general concept of view-merge is the same as in theprevious ethresearch post. Briefly, the idea is the following:

1. attester freeze their fork-choice view \Delta

seconds before the beginning of a slot, caching new messages for later processing

1. the proposer does not freeze their view, or in other words they behave exactly as today, and propose based on their view of the head of the chain at proposal time, i.e. at the beginning of their slot
2. Moreover, the proposer references all attestations and blocks they have used in their fork-choice in some p2p message, which is propagated with the block
3. Attesters include the referenced attestations in their view, and attest based on the "merged view".

Why does it work, and under what conditions?

- If the network delay is < \Delta

, the view of the proposer is a superset of the frozen views of other validators, so the final "merged view" is exactly the view of the proposer, regardless of when an attacker might have sent their attestations (recall, this kind of maliciously timed release of attestations is the key of balancing attacks)

- If the fork-choice output is a pure function of one's view

(or, more precisely, of whatever is being synchronized, i.e. whatever objects the proposer's view-merge message references) then network delay < \Delta

implies agreement on the fork-choice output, and therefore that honest validators attest to honest proposals. This condition is simply a design constraint: we need to ensure that what we do view-merge over, i.e. the objects which are referenced in the view-merge message, fully determine the fork-choice output.

Let's now visualize the slot structure and the two possible cases of maliciously timed attestations, which attempt to split validator views, to convince ourselves that the assumption on the network delay is sufficient to conclude that views are synchronized. At this stage, we ignore attestation aggregation for simplicity.

[

661×522 18.1 KB

](https://ethresear.ch/uploads/default/original/2X/4/470952734032bd048ca3086b364addf321390b3d.png)

Why view-merge

These are the reasons to move from proposer boost to view-merge:

- Not abusable by proposers:

as already mentioned in a previous section, boost can be abused to get extra fork-choice influence and conclude reorgs. View-merge does not lend itself to be weaponized, because it only allows proposers to add real

attestations to validators' views.

Moreover, all honest attestations will under synchronous network conditions already be in their views before the view-freeze deadline, and so will be unaffacted by the view-merge mechanism, maintaining the property that proposers cannot even temporarily exclude honest attestations from the fork-choice of honest validators

- More powerful reorg resilience

:

Let’s first define two useful concepts:

- Reorg resilience

: honest proposals stay canonical

- View-merge property

: Given the condition mentioned in the previous section, that validators which have received the same set of messages also output the same head when running the fork-choice, all honest validators attest to an honest proposal during its slot.

View-merge can almost be viewed as a reorg resilience gadget, able to enhance consensus protocols by aligning honest views during honest slots. In LMD-GHOST without committees, i.e. with the whole validator set voting at once, it is sufficient to have reorg resilience under the assumptions of honest majority and synchronous network.

With committees, i.e. in the current protocol, it is not quite enough for reorg resilience because of the possibility of ex ante reorgs with many adversarial blocks in a row and thus many saved attestations. Nonetheless, the view-merge property is arguably as close to reorg resilience as we can get while having committees.

In the protocol with proposer boost, the adversary can always “simulate playing against view-merge instead of proposer boost”, by just letting honest validators always attest to honest proposals, i.e. never trying to overcome the proposer boost, and only reorging them later if they are able to. In addition, they have access to a second strategy, i.e. to overcome proposer boost before attestations are made. Since it’s not abusable, view-merge then provides strictly superior reorg resilience guarantees than proposer boost.

As an example of the enhanced resilience to reorgs, the current value W_P = 0.4

means that a 20% staker controlling two blocks in a row can reorg the next block, whereas they would need to control 3 slots in a row to do the same reorg with view-merge. Generally, a \beta

adversary needs \frac{1-\beta}{\beta}-1 = \frac{1-2\beta}{\beta}

blocks in a row to be able to do an ex ante reorg (the -1 comes from the fact that the adversarial attestations in the honest slot are also used), instead of min(\frac{1-2\beta}{\beta}, \frac{W_P}{\beta})

.

- Reorg resilience

: honest proposals stay canonical

- View-merge property

: Given the condition mentioned in the previous section, that validators which have received the same set of messages also output the same head when running the fork-choice, all honest validators attest to an honest proposal during its slot.

- Higher balancing resistance

: the two previous points together combine to give us an overall more secure chain when substituting proposer boost with view-merge. The reason is roughly that keeping a balancing attack going needs adversarial attestations to be consumed in order to overcome the influence of honest proposers (be it through proposer boost or view-merge), and view-merge forces a higher consumption of attestations than boost does, while not giving the adversary any further tool to exert influence on the fork-choice without consuming attestations.

- Compatible with dynamic-availability

: proposer boost as it currently exists is not compatible with low participation regimes, because boost does not scale with participation. In particular, the chain becomes completely insecure when the participation drops below 40%, because any proposer can reorg the previous block, and generally security is degraded as participation drops. It is not clear that making boost scale with participation is possible without creating further vectors of adversarial exploitation.

Attestation aggregation

Before diving into view-merge, we recap what attestation aggregation is and how it works, and then try to understand the challenges presented by the implementation of view-merge in the Ethereum Consensus Layer. It can be tempting to want to abstract aggregation away, and stick to the simplified picture we have previously shown. Unfortunately, we cannot, for a simple reason: doing view-merge over thousands of unaggregated attestations is just not practically feasible, whereas doing

it over aggregates restricts the amount of objects over which we are trying to synchronize views to a manageable one. In fact, for view-merge to be practical, it has to be possible for the proposer to make a reasonably sized

message referencing everything which determines their view (or at least everything which they have seen in the last 2\Delta

seconds, i.e. anything which might be in the frozen views of other honest validators). If not, the message won't be able to be delivered to the attester in time, or anyway will strain the p2p network during the key time of block propagation. This requires the amount of references to be relatively small, and not manipulable by the adversary, so we cannot deal with unaggregated attestations, and instead reference exclusively

aggregates.

This is what view-merge looks like with attestation aggregation:

[

821×524 17.5 KB

](https://ethresear.ch/uploads/default/original/2X/b/bc0e919ba967b1dd632c24f49e74f57b815b9414.png)

Aggregation overview

Comprehensive resources on the topic are this note by Hsiao-Wei, the attestation aggregation section of the honest validator spec, the attestation subnets section of the p2p spec, and the annotated phase0 spec by Vitalik.

Why we aggregate

: In each Ethereum epoch, the validator set is evenly distributed over 32 slots, and in each Ethereum slot the validators elected for the slot make an attestation

, including an LMD-GHOST vote for the head of the chain and an FFG vote used for finalizing. Since even 1/32

of the validator set is a very large number of validators (currently over 13k validators), we further subdivide them into small committees, and aggregate the signatures of a single committee, so that only the aggregates have to be gossiped and processed by everyone.

Attestation subnets

: Such a committee is tasked with broadcasting their individual attestations in a corresponding attestation subnet

. This is a temporary subnetwork which is only used for this purpose, i.e. gossiping unaggregated attestations for the committee, without involving the rest of the network. Important parameters are:

- a desired

minimum committee size TARGET_COMMITTEE_SIZE = 128

. For all practical purposes a minimum size, except for degenerate conditions with a tiny validator set.

- a maximum committee size MAX_VALIDATORS_PER_COMMITTEE = 2048

. This can only be reached with ~134 million ETH staked, corresponding to the maximum theoretically supported validator set size 2**22 = 4,194,304

. Currently, there are around 200 validators per committee, and this determines both the gossip load on a single subnet and also the size and processing time of the aggregates.

- There are at most MAX_COMMITTEES_PER_SLOT = 64

subnets, and their number is always maxed out when the validator set size exceeds TARGET_COMMITTEE_SIZE*MAX_COMMITTEES_PER_SLOT*SLOTS_PER_EPOCH=128*64*32=262,144

(which is the case now and is likely to be the case for the foreseeable future), as this ensures that every committee has at least the minimum desired number of validators, which is what we consider to be secure.

Aggregators

: Within each subnet, validators broadcast their individual attestations. Since we don't want all attestations to have to be gossipped to the whole network and processed by everyone, we aggregate

them within a subnet, and only globally gossip the aggregates. Each subnet has some validators with a special role, called aggregators

, which are randomly (self-)elected in a verifiable way. The expected number of aggregators per subnet is TARGET_AGGREGATORS_PER_COMMITTEE = 16

, sufficient to ensure a high probability of there being at least one honest aggregator in every subnet. Aggregators are tasked with collecting individual attestations from the subnet and creating an aggregate attestation, i.e. an attestation whose data is the same as all of the individual attestations and whose signature is the BLS aggregate signature of all of the individual signatures, i.e. their sum, verifiable with the aggregate public key, the sum of all public keys. To be precise, they only aggregate attestations which agree with their own, and this is the only incentive which they currently have to perform their aggregation duty, since it's not rewarded in the Beacon Chain.

Global Aggregate channel

: Aggregators publish the best aggregate they can produce (and which is in agreement with their view), including a proof of their aggregator status, in the global beacon_aggregate_and_proof

gossip channel, which all validators and full nodes participate in in order to update their view of latest messages and be able to follow the chain. The expected number of aggregates in the global topic is MAX_COMMITTEES_PER_SLOT*TARGET_AGGREGATORS_PER_COMMITTEE = 64*16 = 1024

, and is already maxed out since the validator set is large enough to support the maximum number of committees per slot.

Aggregators as designated attestation sources

Today, aggregators are purely a p2p concept, absent both from the Beacon Chain spec and from the fork-choice spec. In practice, this means that blocks can include attestations which do not come from an aggregate, at least not an "official" one with a valid proof, as they look exactly the same to the Beacon Chain. It also means that a Beacon client might include such an attestation in their local view, i.e. use it as an input to update_latest_messages

, which in turns influences the outcome of their local fork-choice. For example, a Beacon node subscribing to an attestation subnet might directly include unaggregated attestations from the subnet in their local view, regardless of whether they are included in some official aggregate that is published in the global beacon_aggregate_and_proof

topic.

This is clearly incompatible with our goal of making view-merge feasible in practice by only needing to reference aggregates, because synchronizing the honest views on the "official" aggregates is not sufficient to guarantee that their fork-choice outputs also agree, as the latter are not a pure function of the former. To prevent this problem, we have to restrict what can influence the fork-choice to only aggregates from designated aggregators

, i.e. aggregators with a valid selection proof, or proposers

. For the latter, the block they propose is itself taken to be the aggregate, aggregating all attestations it includes (in the block.attestations

field). Only attestations from such aggregates are to be used to update the fork-choice Store

, so that agreement on a list of aggregates from designated aggregators is sufficient to agree on the fork-choice output.

Aggregator equivocation

What's necessary for view-merge to work is that validators which have received the same set of messages also output the same head when running the fork-choice, regardless of the order in which they have received the messages. If that's the case, all honest attester will output the same head as the proposer, given a synchronous network with maximum delay \Delta

, which is what the view-merge mechanism needs to ensure consistency of views.

Unfortunately, this is not necessarily the case today. For example, equivocations used to be treated with a first-seen approach, i.e. reject all attestations from a certain validator after the first one. This is liable to splitting view, potentially even permanently. Discounting fork-choice weight from equivocating attester is in this sense a step in the right direction, because evidence of an equivocation can reconcile the views which might have been split by it. A more challenging source of equivocations is the aggregation mechanism.

Aggregator equivocation problem

: An aggregator can produce two valid aggregates for the same slot, differing in the attesting indices, without there being any equivocation in the underlying attestations. The way we deal with this today is first-seen, i.e., only the first aggregate from any given aggregator is accepted and forwarded, to prevent a DoS vector. Again, this can be problematic independently of view-merge, because an aggregator could aggregate attestations that no one else has access to in unaggregated form, e.g. their own, so that anyone not processing such an aggregate necessarily has a different view from all who do. In fact, the reason this is problematic for view-merge is the same exact reason why this is problematic to begin with, i.e. that views

cannot be reconciled.

Solution idea

: In this case, it is not as straightforward to discount equivocations like we do for equivocations in the unaggregated attestations, but with a little bookkeeping it is possible. When we find out about an aggregator having equivocated, reconciling views requires removing all influence they have had on our local fork-choice. The reason this is a bit more involved that with equivocating attestations is that we normally don't track where messages in the fork-choice store "came from", meaning we only know what's the latest message of a validator, and not where we have seen that message. Properly reconciling views requires that we do such tracking, so that we discount only the messages which have only

been seen in aggregates from equivocating aggregators. Moreover, doing that for all aggregates of an equivocating aggregator requires tracking aggregates, rather than just processing them to update the fork-choice store with their attestations and forgetting about them.

Concrete view-merge proposal

We now make a detailed proposal on how view-merge would actually work, largely just making precise all of the ideas developed in the previous sections. We go through the needed changes one by one and explain their reasoning. A WIP attempt at making these changes in the specs is here, though not entirely up to date.

Limiting the amount of references needed

Only attestations from aggregates made by designated aggregators are utilized for the fork-choice. We consider the proposer to be a special designated aggregator, and we accept all attestations included in a block. Unaggregated attestations can be propagated like today, in their relevant attestation subnet but not in the global beacon_aggregate_and_proof

topic, but won't trigger updates to store.latest_messages

. For example, one could add an input is_from_aggregate

to on_attestation

in the fork-choice spec, and only call update_latest_messages

if is_from_aggregate

. This restricts the relevant messages which need to be referenced to only aggregates, which makes it feasible for the proposer to reference all that's needed to reconstruct their "fork-choice view", i.e. what determines the fork-choice output.

Attestation expiry

Attestations expire after 2 epochs as far as the LMD-GHOST component of the fork-choice is concerned, i.e. get_latest_attesting_balance

checks that store.latest_messages[i].epoch in [current_epoch, current-epoch -1]

. The primary reason as far as view-merge is concerned is to bound the scope of what the proposer needs to reference, similarly to the first change.

Extra motivations for attestation expiry

- Fork-choice expiry would be in line with"on-chain expiry", i.e. a valid block only includes attestations from the current or previous epoch. We also already have a concept of "p2p expiry" ([ATTESTATION_PROPAGATION_SLOT_RANGE = 32

](https://github.com/ethereum/consensus-specs/blob/b72afffe264ffcb77cdbfc41c9b59068d77e68f7/specs/phase0/p2p-interface.md#configuration)), which can be modified to match this as well. As a side note, and providing further justification for the introduction of fork-choice expiry of attestations, notice that p2p expiry and on-chain expiry can themselves cause split views, because some validators might see some attestations before their p2p expiry, and other validators might not, plus on-chain expiry would prevent the former validators from including these attestations in blocks, so the latter would never add them to their fork-choice store. Adding a matching fork-choice expiry prevents this.

- Attestation expiry (or something along those lines) is needed to have a secure chain in conditions of low participation. The idea is that LMD-GHOST gives permanent fork-choice influence to the stale votes of offline validators. In certain conditions of low participation, these can be weaponized by an adversary controlling a small fraction of the validator set, to execute an arbitrarily long reorg. Consider for example a validator set of size 2n+1

, and a partition of the validator set in three sets, V_1

, V_2

, V_3

, with $|V_1| = |V_2| = n$

and $|V_3| = 1$

. The validators in V_1

, V_2

are all honest, while the one in V_3

is adversarial. At some point, the latest messages of validators from V_1

and V_3

vote for a block A

, whereas those from V_2

vote for a conflicting block B

, so A

is canonical. The validators in V_2

now go offline. The online honest validators, i.e V_1

, keep voting for descendants of A

for as long as the adversary does not change their vote. After waiting arbitrarily long, the adversarial validator votes for B

, resulting in all blocks produced in the meantime being reorged.

Treatment of aggregator equivocations

1. We add to store

the field store.previous_epoch_aggregates: Dict[ValidatorIndex, Set[Attestation]]

and similarly also store.current_epoch_aggregates

, mapping a validator index to their aggregates from the previous and current epoch. Again, we consider proposers to be aggregators, and add all attestations contained in a block to their set of attestations. We use these maps to make sure that we process every aggregate exactly once, and that we can “undo” changes to reference_count

in the LatestMessage

object (see next bullet point) when we get evidence of an equivocation.

1. We add store.previous_latest_messages: Dict[ValidatorIndex, LatestMessage]

to store

, analogous to store.latest_messages

but tracking the second-to-last messages. When a new latest message is found for a validator with indexi

, we set store.previous_latest_messages[i] = store.latest_messages[i]

before updating store.latest_messages[i]

with the new one. As for the aggregates in the previous bullet point, we only need to track the previous and current one because of attestation expiry.

1. We add a field reference_count: uint64

to the LatestMessage

object, which is increased by 1 anytime “the same latest message” is processed, meaning that a new aggregate or attestation in a block has evidence for the same vote, i.e. a vote from the same validator and with same epoch and root.

reference_count

is updated both for store.latest_messages

and for store.previous_latest_messages

. Since we process every aggregate exactly once, reference_count

will normally (if there's no equivocating aggregators, see 7.) equal the number of unique aggregates which aggregated a certain message if the message did not appear in any block, and some higher number than that otherwise.

1. In get_latest_attesting_balance

we check store.latest_messages[i].reference_count > 0

, and otherwise use store.previous_latest_messages[i]

if possible, i.e. if we also check that it is unexpired and with positive reference count.

1. We accept equivocation evidence for aggregators on the p2p network, though it does not get processed on chain (aggregation is a p2p concept, not known to the Beacon Chain. This proposal keeps things that way, though it does introduce the concept in the fork-choice as well). It consists of two distinct SignedAggregateAndProof

messages from the same aggregator and from the same slot. It is processed by decreasing the reference count of every latest message "referenced" by already processed unexpired aggregates from the equivocating aggregator, and also by adding the aggregator to the (already existing) list of equivocators (henceforth, their aggregates are ignored and their latest messages discounted from the fork-choice).

It is then the case that a store.latest_messages[i].reference_count > 0

(and similarly for store.previous_latest_messages

) if and only if an attestation aggregating this message has been processed either from a block or from the aggregate of a so far non-equivocating aggregator, which justifies checking this condition.

This is how aggregator equivocation evidence is processed (including because of block equivocations), in code:

store.equivocating_indices.add(aggregator_index) unexpired_aggregates_from_equivocator = current_epoch_aggregates[aggregator_index].union(previous_epoch_aggregates[aggregator_index]) for attestation in unexpired_aggregates_from_equivocator: target_state = store.checkpoint_states[attestation.data.target] attesting_indices = get_attesting_indices(target_state, attestation.data, attestation.aggregation_bits) for i in attesting_indices: if (i in store.latest_messages and store.latest_messages[i].root == attestation.data.beacon_block_root and store.latest_messages[i].epoch == compute_epoch_at_slot(attestation.data.slot)): store.latest_messages[i].reference_count -= 1 elif (i in store.previous_latest_messages store.previous_latest_messages[i].root == attestation.data.beacon_block_root and store.previous_latest_messages[i].epoch == compute_epoch_at_slot(attestation.data.slot)): store.previous_latest_messages[i].reference_count -= 1

Changes to proposal and attestation behavior

1. A slot becomes divided in 4 parts of 3 seconds each: attestation deadline at 3s, aggregate broadcasting at 6s, view-freeze deadline at 9s (note that this is just for simplicity. In practice, different parts of the slot do not require the same amount of time, e.g. the first part requires more time because block processing both by the CL and EL has to happen before attesting. Asymmetric breakdowns of the slot are possible, e.g. 4-3-3-2)

2. Attesters freeze their view at the view-freeze deadline, and cache all later messages. They unfreeze their view and add the cached messages to it after attesting in the next slot.

3. The proposer of slot n doesn't freeze their view at slot n-1. During slot n-1, they keep track of newly

received blocks, aggregates and equivocation evidence. Together with the proposal, they propagate a signed message containing a list of references (hashes) to these, except for all of those which are already contained in the ancestry of the proposal, i.e. except all ancestors of the proposed block and all aggregates whose Attestation

is included in one ancestor.

1. On the p2p layer, proposals are propagated together with this message during the proposal slot.

2. Attesters only attest after seeing both the proposal and the message with references and having retrieved all messaged referenced in the latter, or at the attestation deadline, whichever comes first. The view they use when attesting is that obtained by starting with the frozen view, processing all referenced messages which they have retrieved and processing the equivocation evidence.

Propagation costs

Referencing all recent aggregates

The only two components of the view-merge message are references to aggregates and references to aggregator equivocation evidence, i.e. pairs of aggregates. The potential load added by the latter is at worst two references for each adversarial aggregator in the last two epochs, so at most 2*text{slots in two epochs}*\text{aggregators per subnet}*text{# of subnets}*\text{bytes per hash}

= 2*64*16*64*32 = 4096

bytes, or 4 MBs. While this is a large amount, this is a worst case size which requires the whole validator set to equivocate. Generally, the worst case size for this portion of the message is the fraction of 4 MBs corresponding to the fraction of validators which have equivocated. Any meaningful attack of this kind can be made extremely

expensive (as it requires a large fraction of validators to equivocate as aggregators, which we could make slashable), and it also cannot be repeated since a validator who has been found to be equivocating can later be ignored. Moreover, we can also replace equivocation evidence with the validator indices of equivocating aggregators, requiring only 8 bytes per equivocator, and thus a maximum of 512 KBs. Let's now consider the size of the first part of the view-merge message, the references to non-equivocating aggregates, in the honest case and in different attack scenarios.

Honest case

: All aggregators within the attestation expiry period and the current proposer are honest. The only honest aggregates which are referenced by an honest proposer at slot n

are those from slot n-1

, since all others would have been received in previous slots, which means the proposer does not reference them (because it is not necessary for them to do so), as specified in 3., in the previous section. This bounds the honest portion of the list of aggregator indices to about \text{aggregators per subnet}*text{# of subnets}*\text{bytes per hash} = 16*64*32 = 1024*32

bytes, or 32 KBs.

Malicious proposer

: It can reference all aggregates from the last two epochs, which would result in 64*32 = 2048

KBs, or 2 MBs. This can be mitigated with propagation rules, e.g. IGNORE

if the message references aggregates that had already been seen more than two slots before, but not completely prevented, because adversarial nodes can always choose to propagate the message regardless of the rule.

Malicious aggregators

: They can withhold aggregates and then publish them all at once, resulting in a message which contains all adversarial aggregates for up to two epochs, and which the proposer cannot ignore because they have been recently published. It seems inevitable that attempting to defend against this kind of attack is going to compromise the main property of view-merge, since such adversarial behavior is not distinguishable from high latency.

Capping the size of the view-merge message

These malicious behaviors are quite concerning, because introducing cheap ways to strain the p2p network is liable to do more damage than what we gain by going from proposer boost to view-merge. Fortunately, even though being able to reference all of the aggregates would be required to ensure that view-merge always

works, we can ensure that it works most of the time

even with some bound on what can be referenced. For example, let's say we cap the size of the view-merge message at 256 KBs. Honest validators never submit late aggregates, so (when the network is synchronous) there is never any need to reference honest aggregates from slots other than the previous one (the proposer of slot n

only needs to reference the ones from slot n-1

). For the view-merge message to be full, it has to then be the case that at least 224 KBs (256 - 32) are filled by adversarial aggregates. An adversary controlling a fraction \beta

of the validators is only able to generate about 32\beta

KBs worth of references per slot, so they need to have withheld an amount of aggregates corresponding to \frac{224}

{32\beta} = \frac{7}{\beta}

slots. If we had chosen the size cap to be 128 KBs, they'd need to withhold an amount of aggregates corresponding to \frac{96}{32\beta} = \frac{3}{\beta}

slots. This is the key to be able to argue that this weaker (but safer from a networking perspective) form of view-merge still gets us a very similar resilience to reorgs and to balancing attacks.

Resilience to reorgs

: The observation above means that this attack (publishing more withheld aggregates than the view-merge message can reference) requires the adversary to necessarily include aggregates which go at least as far back as \frac{7}{\beta}

slots, with the 256 KBs size cap. In the worst case of a \beta = 0.5

attacker, this is 14 slots. Under normal conditions, i.e. not an ongoing balancing attack, or at least not going as far back as 14 slots, the attack is completely ineffective: the proposer can just order aggregates by slot and reference them until the size cap is hit, with no impact to the view-merge mechanism. This is simply because attestations from 14 slots ago are only relevant if there is a (competitive) fork which goes that far back. Similar considerations apply to a size cap of 128 KBs.

Resilience against balancing attacks

: That same observation also means that the adversary is only able to execute the attack at a rate of once every \frac{7}{\beta}

or \frac{3}{\beta}

slots. This is crucial to extending the kind of rate-based arguments that one would make about balancing resilience. With normal view-merge, the idea is that every honest slot forces the adversary to utilize 1-\beta

attestations (to be precise, that fraction of a committee's weight), because all honest validators attest together due to the view-merge property, and the adversary has to overcome these attestations to maintain the balance. The relevant inequality is therefore (1-\beta)^2 > \beta

: a fraction 1-\beta

of the slots is honest, each of which forces the adversary to consume 1-\beta

attestations, and the adversary can at most accumulate \beta

attestations per slot. When the inequality holds, the adversary eventually fails to maintain the balance, because they run out of attestations to do so. With this weaker view-merge, the "good slots" are now not all honest slots, but only those in which the adversary is not able to fill the view-merge message completely, and we know that a fraction 1-\beta - \frac{\beta}{7}

of slots is "good". Therefore, the relevant equation becomes (1-\beta)(1-\beta - \frac{\beta}{7}) > \beta

. The maximum tolerable \beta

is 0.38 in the first case, and still 0.37 in the second. A size cap of 128 KBs leaves us with (1-\beta)(1-\beta - \frac{\beta}{3}) > \beta

, which corresponds to a maximum tolerable \beta

of 0.35, still only a minor loss of resilience.

Alternative strategies to explore

One can in principle do away with the view-merge message altogether, by asking honest validators to "try to agree with the proposer", meaning that they try to find a subset of messages in their buffer which gives the right fork-choice output when added to their view. Such a subset would always exist, but it's unclear if it can be efficiently computed, and there might well be some hardness result which says otherwise in the worst case.

Miscellaneous questions and concerns

Losing the ability to discourage late blocks

: proposer boost gives us the ability to somewhat mimick the functioning of (block, slot) attestations, without the complexities of it (in particular the difficulty of a safe backoff scheme for it), by prescribing that proposers use their boost to orphan late blocks which have not received many attestations (see here and here). This can be useful to counter the incentive to publish one's block late to get more MEV. Another way to discourage this could be by rewarding block proposers retroactively, based on the attestations which their block receives. Nonetheless, this might not always be sufficient, because MEV might

at times dwarf the potential reward for timeliness.

Actually slashing/exiting aggregators

: We might wish to avoid this “limbo” state in which aggregators which have equivocated are ignored as aggregators, and potentially also as attesters, but they are still in the validator set. Since equivocating as an aggregator cannot lead to double-finality, an option could be to not punish it with slashing, but only with a forced exit. This would of course require the Beacon Chain to be able to process aggregator slashing evidence.

Doing away with the separate view-merge message

: The view-merge mechanism would be a lot cleaner if all of the references could just go in the block, doing away with the current sidecar construction. The issue is that even a 128 KB cap on the view-merge message would result in the maximum block size being nearly doubled.