V-Guard: An Efficient Client-Centric Blockchain Safeguarding Critical Vehicular Data (Appendix)

A Storage

V-Guard stores committed data batches in a *storage master*. The storage master creates storage master instances (SMIs) for transactions from different vehicles, as a vehicle may operate in different roles in different booths (i.e., $\gamma > 1$). For example, in Figure 1, the storage master has three instance zones containing proposer, validator, and gossiper SMIs (if the gossiping module is applied). A vehicle's storage master has at most one proposer SMI because it is the only proposer of its own V-Guard instance. When the vehicle joins other booths (operating as a validator), its storage master creates validator SMIs. The number of proposer and validator SMIs equals the vehicle's catering factor (γ). When gossiping is enabled, a vehicle may receive gossiping messages, and its storage master creates gossiper SMIs storing consensus results disseminated from other vehicles.

Each storage master instance has two layers: temporary and permanent, with the purpose of maximizing the usage of vehicles' limited storage space. Unlike the blockchain platforms working on servers (e.g., HyperLedger Fabric [1], CCF [5], and Diem [3]), V-Guard operates among vehicles, which usually have only limited storage capability.

The temporary storage temporarily stores transactions based on a predefined policy. V-Guard's implementation uses a timing policy that defines a time period (denoted by τ) that the storage master stores a transaction (e.g., $\tau = 24$ hours). A transaction is registered in the temporary storage by default and is deleted after τ time if the user issues no further command. In addition, temporarily stored transactions can be moved to permanent storage and kept permanently per user request. This option hands over the control of storage to users. For example, when users suspect malfunctions of their vehicles, they may want to keep related transactions as evidence and move them to permanent storage. We introduce four SMI APIs for the layered design.

- `sm.RegisterToTemp(&tx, time.Now())` registers a transaction to the temporary storage layer.

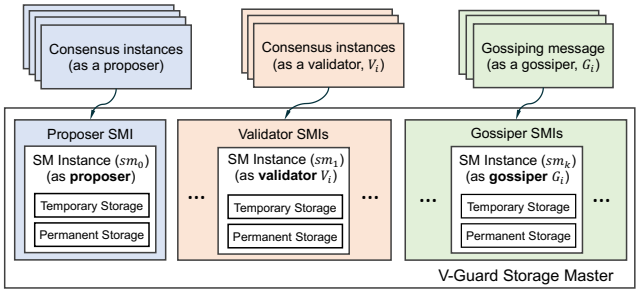

Figure 1: Storage master instances (SMIs). A vehicle operates a proposer SMI when it conducts consensus for itself, validator SMIs when it participates in other vehicles' consensus, and gossiper SMIs when it enables the gossip module.

- `sm.CleanUpTemp(time.Now())` is a daemon process that calculates the remaining time for transactions in the temporary storage and deletes expired ones.

- `sm.MoveToPerm(&tx)` is called by users, moving selected transactions from temporary to permanent storage.

- `sm.DeletePerm(&tx)` is called by users. A user may delete a permanently stored transaction after it has served the user's purpose.

The policy of temporary storage can be implemented differently. For example, a policy that defines a fixed size of storage space may apply. When the temporary storage exceeds the predefined size, it clears up old transactions in a first-in-first-out (FIFO) manner.

The policy-based storage module can significantly reduce the use of storage space as vehicles often have limited storage capability. Note that the pivot validator can permanently store all data as automobile manufacturers often operate on their cloud platforms with scalable storage devices.

B Gossiping

The gossiping module strengthens system robustness by further disseminating committed transactions to the network. This module does not affect the correctness of consensus and can be applied based on preferences. When gossiping is enabled, after the consensus module commits a transaction, the proposer sends a `Gossip` message piggybacking the transaction to other connected vehicles, which will keep propagating this message to their connected vehicles. Therefore, the transmission forms a *propagation tree* where the root node is the *proposer* and other nodes are *propagators*.

Each `Gossip` message has a *lifetime* (denoted by λ) that determines the number of propagators it traverses in a transmission link; i.e., the height of the propagation tree. For example, in Figure 2, V_1 is the proposer and has direct connections with V_2, V_3, and V_4. The gossip message has a lifetime of 2, and its propagation stops when a path has included 2 propagators; e.g., path <V_1, V_2, V_5>. Specifically, we describe the gossiping workflow as follows.

Proposer: The proposer (V_p) starts to disseminate committed transactions by taking the following actions.

1. V_p creates a lifetime (e.g., $\lambda = 3$) for a `Commit` message and a set $\mathcal{G}$ for recording the traverse information of the message's transmission path.
2. V_p hashes the `Commit` message and signs the combination of the hash result (h_c) and λ to obtain a signature σ_{V_p}. Then, it adds a *traverse entry* containing λ, σ_{V_p}, and C_{V_p} to $\mathcal{G}$; i.e., $\mathcal{G}.add([\lambda, \sigma_{V_p}, C_{V_p}])$, where C_{V_p} is V_p's address and public key.
3. V_p sends $\langle$`Gossip`$, \langle$`Commit`$\rangle, h_c, tx, \mathcal{G}\rangle$ to connected vehicles that are not included in the consensus process of the `Commit` message and then creates a list for receiving `ack` messages from propagators.

Propagator: A propagator (V_i) stores a valid `Gossip` message and may further disseminate the message if there is lifetime remaining.

1. V_i verifies a received `Gossip` message by four criteria: ① it has not previously received this message; ② signatures in $\mathcal{G}$ are valid; ③ λ in $\mathcal{G}$ is strictly monotonically decreasing; and ④ the message still has remaining lifetime; i.e., $\lambda_{min} = min\{\mathcal{G}.\lambda\} > 0$. If the verification succeeds, V_i sends an `ack` message to this message's proposer and the pivot validator, registering itself on the propagator list.
2. V_i decrements this gossip message's lifetime; i.e., $\lambda_{new} = \max\{\lambda_{min} - 1, 0\}$. If $\lambda_{new} > 0$, then V_i prepares to further disseminate the message. It signs the combination of h_c and λ_{new} to obtain a signature σ_{V_i} and adds a new traverse entry into $\mathcal{G}$; i.e., $\mathcal{G}.add([\lambda_{new}, \sigma_{V_i}, V_i])$.
3. V_i sends $\langle$`Gossip`$, \langle$`Commit`$\rangle, h_c, tx, \mathcal{G}\rangle$ to other connected vehicles, excluding those it receives this `Gossip` message from (in (1)).

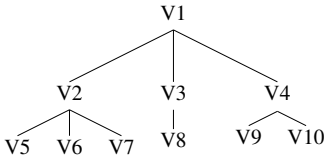

Figure 2: A propagation tree of a gossip message with $\lambda = 2$. V_1 is the proposer and others are propagators.

The gossip module in V-Guard is an additional mechanism that provides an additional layer of redundancy. Since each propagator is registered to the proposer, the proposer can connect to propagators and read its own portion of data. This is especially important when consensus is conducted in small booth sizes, as the failure of vehicles can significantly affect the system's overall resilience. Although gossiping does not guarantee the delivery of all transactions to all participants in the network, it can help to increase the probability of transaction delivery even in the face of network partitions or failures, strengthening the system's robustness without interfering with the core consensus process.

C Correctness argument

Theorem C.1 (Validity)**.** *Every data entry committed by correct members in the consensus phase must have been proposed in the ordering phase.*

Proof. Only a vehicle can be the proposer of the V-Guard instance it initiated. In **O2**, validators sign the same content as the proposer, which includes the hash of a proposed data entry. In **O3** and **O4**, a valid threshold signature is converted from a quorum of signatures that sign the same hash of the proposed data entry. In addition, the proposer signs the hash of the transaction that includes the data entry in **C1**. If a transaction includes non-proposed data entries, the verification process in **C2** fails. Thus, a committed transaction must include data entries proposed in the ordering phase. □

V-Guard's membership management separates the ordering of data batches from consensus and achieves high performance with more granular control of dynamic memberships. The separation sill ensures safety, where no two non-faulty members commit the same data entry with conflicting ordering or consensus IDs.

Lemma C.1. *No two non-faulty members in the same ordering booth agree on conflicting ordering IDs for the same data entry.*

Proof. We prove this lemma by discussing the two following scenarios. First, when the proposer is correct, it will not send conflicting ordering IDs for the same data entry, and f faulty vehicle validators cannot impact the process that the proposer collects sufficient `PO-Reply`s from the remaining $2f$ correct vehicle validators in **O2**.

Second, when the proposer is faulty, it can send conflicting ordering IDs in two cases: ① it assigns the same ordering ID (i) to two data entries, or ② it assigns different ordering IDs to the same data entry. In ①, correct validators will not send a `PO-Reply` to the proposer if i has been used (Criterion 3 in **O2**), so the proposer cannot collect sufficient replies. In ②, although validators will reply to the proposer because they are blindsided, their signatures are signed with different i in the combination of $<i, h_B, h_{\mathcal{V}_o}>$ in **O2**, so the proposer cannot convert collected replies to a valid (t,n)-threshold signature with $t = 2f$. Therefore, no two non-faulty members in the same ordering booth agree on conflicting ordering IDs for the same data entry. □

We reiterate V-Guard's failure assumption and provide its rationale. V-Guard assumes a Byzantine failure model but does not assume the collusion between the proposer and the pivot validator; that is, both the proposer and the pivot validator can be malicious (e.g., stop responding or perform equivocation), but they do not collude to exhibit failures.

This amendment to the failure assumption is specifically tailored for vehicle-centric blockchains. The primary objective of implementing blockchains in this context is to address issues related to data integrity and transparency between the proposer and the pivot validator. The aim is to ensure that neither party can bury evidence (e.g., delete data) or fabricate evidence (e.g., manipulate data). Since the proposer is the only member that produce data entries, allowing collusion between the proposer and the pivot validator would undermine the effectiveness of establishing blockchain solutions, as the data management falls back to be reliant on a single source of truth (i.e., the colluded coalition of the proposer and pivot validator).

Under this failure assumption, V-Guard guarantees that the proposer never causes a fork of ordered data entries through changing memberships.

Lemma C.2. *No two non-faulty members in different consensus booths agree on conflicting consensus IDs for the same entry appended on the totally ordered log.*

Proof. The pivot validator participates all the consensus instances and has a global view of the paired consensus process.

If the proposer performs equivocations by sending different booths with conflicting consensus IDs, the pivot validator will not send a `PC-Reply` message to support a conflicting consensus instance. Thus, the proposer cannot form a valid threshold signature that can be examined by members (**C4**). Since consensus instances are continuous and do not leave uncommitted data entries in between, the formed chain do not permit a data entry ordered and committed with conflicting IDs. □

With Lemma C.1 and C.2, we now prove that V-Guard ensures safety.

Theorem C.2 (Safety)**.** *All non-faulty members agree on a total order for proposed data entries in the presence of no more than f failures in each booth.*

Proof. We prove safety by discussing four cases: the proposer and the pivot validator are both correct (Case ①); the proposer is correct, but the pivot validator is faulty (Case ②); the pivot validator is correct, but the proposer is faulty (Case ③); and the proposer and pivot validator are both faulty (Case ④).

In Case ①, with a correct proposer and pivot validator, f faulty vehicle validators in a booth cannot form a valid quorum certificate of size $2f+1$. We prove safety by contradiction. We claim that two correct members commit the two data entries in the same order. Say if two data entries are committed in consensus instances with the same order, with Lemma C.2, there must exist two quorums constructed in **C3** agreeing on the two entries, which have been appended to the totally ordered log; otherwise, the proposer cannot receive sufficient votes in **C3**. In this case, the two entries must have been ordered with the same ordering ID in an ordering booth, which contradicts Lemma C.1. Therefore, each entry is committed with a unique ordering ID. This is similar to the condition where the leader is correct in traditional BFT algorithms such as PBFT [2] and HotStuff [6].

In Case ②, since the consensus process involves the pivot validator only in consensus instances, a faulty pivot validator may not send a correctly signed `PC-Reply` message to the proposer in **C2**, which will result in the proposer cannot proceed in **C3**. This may lead to a temporary suspension in consensus but has no impact on safety. No data entries with conflicting orders will be committed.

In Case ③, since the pivot validator is involved in all consensus instances, it has a globally consistent view of ordering IDs for date entries. When the proposer is faulty and trying to cause a fork among different booths, the pivot validator will not send a `PC-Reply` message in **C2**; thus, the consensus cannot be achieved.

In Case ④, both the proposer and pivot validator are faulty. Since V-Guard does not assume the collusion of the two parties, the pivot validator will not verify a consensus instance with conflicting ordering IDs of data entries. When both the proposer and pivot validator become quiet nodes, the consensus process temporarily halts, with no data entries committed.

To conclude, the co-witness of the proposer and the pivot validator does not permit data entries with conflicting ordering IDs to be committed. Therefore, all non-faulty members across booths agree on a total order of the data entries. □

V-Guard detects the failure of pivot validators using the failure detector ($\Diamond S$) introduced by Malkhi and Reiter [4]. As discussed in Theorem C.2, a faulty pivot validator can temporarily suspend the consensus process by not responding. $\Diamond S$ guarantees that eventually a faulty pivot validator is permanently suspected by correct nodes.

In addition, since V-Guard yields the replacement of malicious pivot validators to applications. We assume that a non-faulty pivot validator will eventually appear in a correct proposer's V-Guard instance.

Theorem C.3 (Liveness)**.** *After GST, a non-faulty proposer eventually commits a proposed data entry.*

Proof. V-Guard assumes partial synchrony for liveness. After GST, the message delay and processing time are bound. When the pivot validator is correct, since no two adjacent consensus instances leave uncommitted log entries in between, every data entry appended on the totally ordered log will be committed.

When the pivot validator is faulty, it has no impact on the ordering instances; data entries will still be ordered and appended to the totally ordered log. The consensus instances will temporarily suspend, but the failure detector $\Diamond S$ will eventually detect and report a quiet pivot validator. After a correct pivot validator is assigned, the consensus instances resume. Thus, during sufficiently long periods of synchrony, a correct proposer eventually receives replies from validators and completes the consensus for a proposed data entry. □

References

[1] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM, 2018.

[2] Miguel Castro, Barbara Liskov, et al. Practical Byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[3] Diem. The diem blockchain. https://developers.diem.com/main/docs/the-diem-blockchain-paper, 2020.

[4] Dahlia Malkhi and Michael Reiter. Unreliable intrusion detection in distributed computations. In *Proceedings 10th Computer Security Foundations Workshop*, pages 116–124. IEEE, 1997.

[5] Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cédric Fournet, Matthew Kerner, Sid Krishna, Julien Maffre, Thomas Moscibroda, Kartik Nayak, Olya Ohrimenko, Felix Schuster, Roy Schwartz, Alex Shamis, Olga Vrousgou, and Christoph M. Wintersteiger. CCF: A Framework for Building Confidential Verifiable Replicated Services. Technical Report MSR-TR-2019-16, Microsoft, April 2019.

[6] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 347–356. ACM, 2019.