TRAIL: Cross-Shard Validation for Byzantine Shard Protection *

Joseph Oglio, Mikhail Nesterenko, and Gokarna Sharma

Department of Computer Science, Kent State University, Kent, OH 44242, USA
{joglio@, mikhail@cs., sharma@cs.}@kent.edu

Abstract. We present *TRAIL*: an algorithm that uses a novel consensus procedure to tolerate failed or malicious shards within a blockchain. Our algorithm takes a new approach of selecting validator shards for each transaction from those that previously held the asset being transferred. This approach ensures the algorithm's robustness and efficiency. *TRAIL* is presented using *PBFT* for internal shard transaction processing and a modified version of *PBFT* for external cross-shard validation. We describe *TRAIL*, prove it correct, analyze its message complexity, and evaluate its performance. We propose various *TRAIL* optimizations: we describe how it can be adapted to other Byzantine-tolerant consensus algorithms, how a complete system may be built on the basis of it, and how *TRAIL* can be applied to existing and future sharded blockchains.

1 Introduction

In this paper, we present *TRAIL* – an algorithm for robust blockchain design. A blockchain is a shared, immutable, append-only distributed ledger, typically maintained by a peer-to-peer network [2], [3]. This design eliminates centralized control over transaction processing and makes the system potentially more scalable, flexible, and efficient.

Blockchains are usually designed to tolerate Byzantine faults [4]. A Byzantine peer may deviate from the algorithm and behave arbitrarily. Therefore, such faults encompass a variety of failures and security threats. Despite the faults, correct peers need to be able to arrive at consensus on proposed transactions.

Popular blockchains use proof-of-work based consensus algorithms [2] in which peers compete for the right to publish records on the blockchain by searching for solutions to cryptographic challenges. Such algorithms tend to be conceptually simple and robust. However, they are resource intensive and environmentally harmful [5]. Therefore, modern blockchain designs often focus on cooperative consensus algorithms.

In these cooperative consensus algorithms, rather than compete, peers exchange messages to arrive at a joint decision. Such algorithms may tolerate some number f of faulty processes. This number is called tolerance threshold. It is usually a fraction of the network size n. One of the most widely used algorithms in this category is *PBFT* [6].

* A technical report [1] contains a more extensive version of the paper.

Scaling up a Byzantine-robust algorithm is challenging as it usually involves system-wide broadcasts. Such broadcasts are expensive in large systems. A prominent approach of improving scalability in blockchains is sharding. In sharding, the network peers are divided into committees or shards. Each shard is made responsible for a subset of the processing done or the data stored by the network. Every shard internally runs a consensus algorithm, such as *PBFT*, and coordinates with other shards to achieve global consistency. Thus, the overall workload is distributed and the processing of records is potentially accelerated.

However, such sharding is at cross-purposes with fault tolerance: the network is only as reliable as any of its shards. For example, given a fixed number of peers, decreasing the shard size increases the number of available shards. This results in greater parallelism in transaction processing. Yet, a small shard is more vulnerable to failure since it has lower tolerance threshold f of its internal consensus algorithm. The sharded blockchains presented in the literature usually assume that no shard tolerance threshold is breached. This places a limit on the efficiency of the sharding approach to performance improvement since shards need to be made large enough to ensure that they never fail.

In this paper, we address the handling of complete shard failures which potentially allows aggressively small shards and removes the shard size scalability obstacle. A naive approach would be to group shards into static meta-shards. Such a meta-shard would treat individual shards as peers and run a meta-consensus algorithm among them to validate transactions across shards to withstand individual shard failures. However, concurrent transactions that are assigned to different shards would be verified by the same static meta-shard, regardless of the transactions' nature or history. This may create a performance bottleneck.

Paper contribution. We propose *TRAIL*: a novel approach to cross-shard validation. With this technique, a trail of shards dynamically tracks each coin according to its transaction history. The source shard runs an internal shard consensus algorithm to validate and linearize transactions. The trail of shards runs a cross-shard consensus algorithm to confirm the transaction and fortify it against shard failure. We present *TRAIL* using *PBFT* for both internal shard transaction processing and external cross-shard validation. We utilize *PBFT* since it is well-known and widely used. Our solution may use various *PBFT* efficiency enhancements such as parallel transaction processing and transaction pipelining. Moreover, *TRAIL* is independent of the specifics of sharding operation and may be adapted to enhance the robustness of consensus algorithms other than *PBFT*.

We evaluate the performance of *TRAIL* using an abstract simulator and study its transaction confirmation rate, scalability and robustness against peer and shard failure. Our experiments indicate that *TRAIL* adds shard failure protection with relatively modest resource expenditure.

2 Network Model, Problem Statement, *PBFT*

System model. We assume a peer-to-peer network. Each peer has a unique identifier. A peer may send a message to any other peer so long as it has the receiver's identifier. Peers communicate through authenticated channels: the receiver of the message may always identify the sender. The communication channels are FIFO and reliable.

Network peers are grouped into *shards*. Every shard has a unique identifier. For simplicity, we assume that all shards are the same size s. Each shard maintains a portion of the blockchain's data. Each shard peer stores a copy of its shard's data. Any peer may determine the shard identifier of any other peer in the network. Peers are either correct or faulty. Faults are *Byzantine* [4]: a faulty peer may behave arbitrarily. A *peer tolerance threshold* f is the maximum number of faulty peers that a shard can tolerate. A shard is correct if it has at most f faulty peers. The shard is faulty otherwise.

Data model and the problem. A *coin* is a unit of ownership whose movements are recorded by the network. Each coin has a unique identifier, which can track fungible assets like currency (e.g., UTXO) or non-fungible assets like NFTs or smart-contracts, without affecting the fungibility of the currency. A *wallet* is a collection of coins. Each shard is responsible for storing and updating a disjoint subset of the network's wallets. A *client* is an entity that owns a wallet. We assume a client is external to the peer-to-peer network but may communicate with any peer. Clients may submit transactions to the network requesting a coin to be moved from a *source wallet* to a *target wallet*. The peers are able to authenticate the wallet owner; the peers accept transaction requests for the wallet owned by the client. The approval of the target wallet owner is not required.

A blockchain algorithm constructs a sequential ledger of transactions reflecting coin movements. Two transactions $t1$ and $t2$ are *consequent* in this ledger if they operate on the same coin and there is no transaction $t3$ also operating on this coin such that $t3$ comes after $t1$ and before $t2$.

An algorithm *state* is an assignment of values to variables in all processes. Algorithm code contains a sequence of actions guarded by boolean guard predicates. An action whose guard evaluates to **true** is *enabled*. An algorithm *computation* is a sequence of steps such that for each state s_i, the next state s_{i+1} is obtained by executing an action enabled in s_i.

To make the *TRAIL* correctness argument more rigorous, we formally state the problem that it solves.

Definition 1. *An algorithm solves* the Coin Transmission Problem *if it constructs a transaction ledger satisfying the following two properties:*
ownership continuity – *for any pair of consequent transactions* $t1$ *and* $t2$*, the target of* $t1$ *is the source of* $t2$*;*
request satisfaction – *if the owner requests a coin movement from its wallet, this request is eventually satisfied.*

Ownership continuity is a safety property that requires that a coin can only be moved out of a wallet once for each time it is moved into it. This is crucial because it prevents the same coin from being spent more than once—thereby precluding double-spend attacks and disallowing spending money that the client does not have. The request satisfaction property guarantees liveness: the client request is eventually fulfilled.

PBFT. *PBFT* is a Byzantine-robust consensus algorithm. Its tolerance threshold is $f = \lfloor (n-1)/3 \rfloor$, where n is the total number of peers in the system.

Peers communicate directly with each other via message broadcast. One of the peers is a *leader*. The leader linearizes client requests. A period of single leader continuous operation is a *view*. *PBFT* is in *normal operation* if the leader is correct. Normal operation has three phases: pre-prepare, prepare, and commit.

Once the leader receives a client transaction request, it assigns it a unique sequence number and starts the *pre-prepare* phase by broadcasting a pre-prepare message containing the transaction and the sequence number to all peers. In the *prepare* phase, each peer receives the pre-prepare message and broadcasts a prepare message containing the information that it received from the leader. If a peer receives $n - f - 1$ prepare messages (excluding itself) that match the initial pre-prepare, this peer is certain that correct peers agree on the same transaction. In this case, the peer starts the *commit* phase by broadcasting a commit message. Once a peer receives $n - f$ commit messages, normal *PBFT* operation concludes and the peer informs the client that the transaction is committed. The transaction is confirmed once the client receives $f + 1$ commits.

If the leader is faulty, the client requests may not be carried out. In this case, the client or the peers initiate a *view change* to replace the leader. The view change process is designed to maintain transaction consistency through the transition: if a transaction is committed by a correct peer in the old view, the new leader submits it with the same sequence number so that the rest of the peers commit it in the new view.

If the number of faulty peers does not exceed the peer tolerance threshold f, *PBFT* guarantees the following three properties: *agreement* – if a correct peer confirms a transaction, then every correct peer confirms this transaction; *total order* – if a correct peer confirms transaction $t1$ before transaction $t2$, then every correct peer confirms $t1$ before $t2$; and *liveness* – if a transaction is submitted to a sufficient number of correct peers, then it is confirmed by a correct peer. We assume these properties also apply to correct clients.

The first two properties are satisfied regardless of the network synchrony. The liveness property is guaranteed only if the network is partially synchronous, meaning that the message transmission delay does not indefinitely grow without a bound.

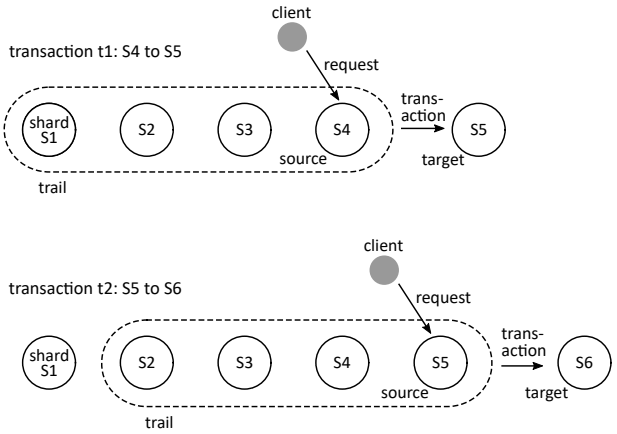

Fig. 1: Trail membership modification under consequent transactions for $t = 4$. The first transaction moves a coin from a wallet in shard $S4$ to a wallet in shard $S5$. The second moves the same coin from $S5$ to $S6$.

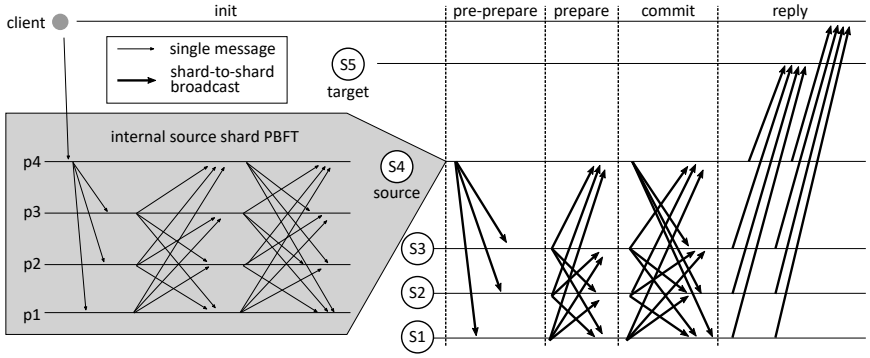

Fig. 2: Message transmission in *TRAIL*'s normal operation. The coin trail contains shards: $S1 - S4$. The coin is located in a wallet stored by shard $S4$. A client sends a transaction moving the coin from $S4$ to a wallet of shard $S5$. First, the $S4$ runs internal $PBFT$; then, the phases of external shard $PBFT$. After committing, the shards notify the client and the target shard.

3 *TRAIL* Description

Algorithm outline. The objective of the algorithm is to ensure the validity of coin transitions between wallets despite faulty peers and shards. To counter malicious behavior of faulty shards, *TRAIL* requires a collection of shards to agree on coin movement. This collection is called a *trail*. A trail is composed of the t unique shards whose wallets the coin visited most recently. Refer to Figure 1 for an illustration. Notice that this trail is specific to a coin and changes as the

coin moves from wallet to wallet. At any point in the computation, each coin may have its own separate trail of shards. We assume that each client knows the identities of the coins in its wallet as well as their trails. *Shard tolerance threshold* F is the maximum number of faulty shards that *TRAIL* may tolerate. The length of the trail, $t \geq 3F+1$. Note that F and the peer tolerance threshold f are not related.

TRAIL consists of two parts: (1) *internal* source shard *PBFT* and (2) *external* trail *PBFT*. To initiate the movement of a coin, the client that owns the source wallet sends a transaction request to the shard that holds it. To linearize received transaction requests and ensure that each individual transaction has an agreed-upon sequence number, the source shard peers execute internal *PBFT*, see Figure 2 for illustration, in which shard $S4$ is the source shard.

Once the source shard peers agree on this transaction, they initiate a modified external trail *PBFT*. For that, each source shard peer broadcasts a *pre-prepare* message to every peer of every trail shard. Once a trail peer receives $s - f$ such *pre-prepare*s from the source shard, it initiates the next *PBFT* phase by sending *prepare* messages to every peer in every trail shard. In this way, each shard-to-shard broadcast emulates an individual message transmission in classic *PBFT*. This continues until the external *PBFT* instance commits. After that, each trail shard peer records the transaction in its ledger and notifies the target shard and the client.

Once the target shard and the client are notified by $t - F$ trail shards, they record the transaction in their ledgers. The target shard, which is shard $S5$ in Figure 1, becomes the source shard for the next transaction. If the leader of the source shard, $S4$, is faulty, the other peers of the source shard execute a view change, switch to a new leader, and continue with internal *PBFT*.

If the source shard as whole is not faulty, i.e. the number of faulty peers in the source shard is below the tolerance threshold f, then the faulty peers may not influence the trail shard. Indeed, for each external *PBFT* message, each peer of the trail shard expects at least $s - f$ individual messages.

If the number of peers in the source shard exceeds the tolerance threshold f then the whole shard is faulty. In this case, the individual messages of the faulty source shard peers are equivalent to the faulty messages of the source shard. The external *PBFT* guarantees that, despite the faulty shard, no spurious transactions will be recorded by the trail shards and that eventually the faulty leader shard is replaced.

Specifically, the trail shards execute a view change, switch to a new shard as a leader, and continue with the consensus process, including a new internal *PBFT* instance being performed within the new leader shard. Note that in the latter case, the record of the transaction may be placed in the trail shards but not in the faulty source shard that is nominally responsible for maintaining the source wallet record. This is an essential feature of our algorithm: the faulty source shard that stores the client wallet may be bypassed.

Let us now describe the algorithm in detail.

***TRAIL* constants, variables, and functions.** These constructs are shown in Algorithm 1. Each peer with id p knows the following constants: f – peer tolerance threshold (the maximum number of faulty peers in a correct shard); F – shard tolerance threshold (the maximum number of faulty shards); s – shard size; and t – trail size.

Several variables are common across transactions. We list them in a single place for convenience. Each transaction uses a coin identifier *coin*; a source wallet id $sWallet$; a target wallet is $tWallet$; a transaction sequence number *seq* assigned by the source shard; and the sequence of shard ids *trail* that indicates the trail shards for this coin at its present location. Each peer maintains a *ledger*, which is a sequence of transaction records that the peer confirmed in a trail or received as a target.

TRAIL functions are shown in Algorithm 2. They are grouped by their purpose. Ledger maintenance functions are in Lines 16–27. *TRAIL* has two such functions.

Function Record appends the transaction record to the ledger. Function IsPresent($coin, wallet$) returns **true** if the ledger's most recent transaction record about *coin* moved it to *wallet*, i.e. there is a transaction where *wallet* is the target wallet and this record is not followed by a transaction moving *coin* from *wallet* to a different target wallet.

TRAIL uses several functions for wallet lookup and communications. They are shown in Lines 28–38. Function GetShard($wallet$) returns the id of the shard that stores *wallet*. We assume that every peer is able to identify which shard maintains each *wallet*.

Functions Send and Receive are single-message transmissions to the specified sender and receiver with straightforward functionality. In function SendToShard($shard, message$), the sender peer broadcasts a *message* to all peers in *shard*. Function ReceiveFromShard($shard, message$) returns **true** once the peer receives *message* from $s - f$ unique peers of *shard*.

The internal source shard *PBFT* is represented by two functions in *TRAIL*. They are shown in Lines 39–46. Function StartShardPBFT initiates the *PBFT* operation. The last function CompleteShardPBFT signifies that the internal *PBFT* is completed and the peers assigned sequence number *seq* to the transaction.

***TRAIL* phases.** The actions for the algorithm are presented in Algorithms 2 and 3. We only show normal operation code for *TRAIL*. View change code is added accordingly. Client and target code is not shown. See Figure 2 for the illustration of algorithm operation.

TRAIL phases execute the internal source *PBFT* and the external trail *PBFT*. **Phase 0: Init** (see Lines 48–53) starts when a peer receives a transaction request from a client. If the peer contains the source wallet, i.e. it is the source shard for the transaction, the peer initiates internal shard *PBFT*. After the source shard runs classic *PBFT*, if the shard is not faulty, all the source

Algorithm 1: *TRAIL*: Normal Operation, Variables

```
1  Constants
2  p                                                    ▷ process id
3  f                                                    ▷ peer tolerance threshold
4  F                                                    ▷ shard tolerance threshold
5  s                                                    ▷ shard size; s ≥ 3f + 1
6  t                                                    ▷ trail size; t ≥ 3F + 1

7  Transaction variables
8  coin                                                 ▷ coin to be transmitted
9  sWallet                                              ▷ coin owner wallet
10 tWallet                                              ▷ coin recipient wallet
11 seq                                                  ▷ transaction sequence number
12 trail                                                ▷ sequence of confirming shard

13 Process variables
14 ledger                                               ▷ sequence of records of committed
15                                                        transactions
   ▷ record format: ⟨coin, sWallet, tWallet, seq, trail⟩
```

shard peers agree on the transaction and its sequence number. The completion of internal *PBFT* starts **Phase 1: Pre-prepare** (Lines 54 through 58). Each source shard peer sends a *pre-prepare* message to a peer of every trail shard.

The receipt of $s-f$ messages from the source shard starts **Phase 2: Prepare** (Lines 59–64) in all the trail shards. Once a peer of the trail shard ascertains that the coin is present in the source wallet, i.e. the transaction is valid, the peer sends a *prepare* message to all of the trail shards.

In **Phase 3: Commit** (Lines 65–74), each trail peer assembles the *prepare* messages. Variable *prepShards.coin.seq* collects the identifiers of the shards from which this peer has received $s - f$ *prepare* messages. If the number of these identifiers is $t - F$, the peer sends *commit* message to all trail shards, signifying that it is ready to commit.

Phase 4: Reply (Lines 75–90) is similar to the **Commit** phase. Once enough *commit* messages from trail shards arrive, the peer records the committed transaction to its ledger and notifies the peers of the target shard and the client.

4 *TRAIL* Correctness and Efficiency

Theorem 1. *Algorithm* TRAIL *solves the Coin Transmission Problem with at most F Byzantine shards and at most f individual Byzantine faults in each correct shard.*

See [1] for the proof of the theorem.

4.1 *TRAIL* Algorithmic Extensions and Implementation Considerations

Parallelizing transactions, splitting, merging and mining coins. The same source shard may run multiple external or internal transactions so long as they concern different coins.

Algorithm 2: *TRAIL*: Normal Operation, Functions

```
16 Ledger functions
17 function
     Record(coin, sWallet, tWallet, seq, trail)
18       ▷ add transaction record to ledger
19 function IsPresent(coin, wallet):
20       if ∃r1 ≡ ⟨coin, x, wallet, ···⟩ ∈ ledger
           and
21          ∀r2 ≡ ⟨coin, wallet, y, ···⟩ ∈
           ledger ⇒
22          r2 precedes r1 in ledger then
23          return true
24       else
25          return false
26 function GetTrail(coin):
27       r ≡ ⟨coin, sWallet, tWallet, seq, trail⟩
           such that r is the last record for coin
           in ledger return trail

28 Communication and wallet functions
29 function GetShard(wallet)
30       ▷ returns the id of the shard that stores
           wallet
31 function Send(peer, message)
32       ▷ send message to peer
33 function Receive(peer, message)
34       ▷ receive message from specific peer
35 function SendToShard(shard, message)
36       ▷ send message to all peers of shard
37 function
     ReceiveFromShard(shard, message)
38       ▷ receive message from s − f unique
           peers of shard

39 Internal PBFT functions
40 function
     StartShardPBFT(coin, sWallet, tWallet)
41       ▷ initiate internal shard PBFT
42       ▷ by sending request to shard leader
43 function CompleteShardPBFT()
44       ▷ finish internal shard PBFT
45       ▷ return ⟨coin, sWallet, tWallet, seq⟩
46       ▷ with unique sequence number seq

47 Basic PBFT operation
48 Phase 0: Init ▷ done by source shard leader
49 upon Receive (client,
50          request⟨coin, sWallet, tWallet⟩):
51       if p ∈ GetShard(sWallet) and
52          IsPresent(coin, sWallet) then
53          StartShardPBFT(coin, sWallet, tWallet)
```

Algorithm 3: *TRAIL*: Normal Operation, Cross-Shard Actions

```
54 Phase 1: Pre-prepare ▷ done by source
     shard
55 upon ⟨coin, sWallet, tWallet, seq⟩ ←
56       CompleteShardPBFT() :
57       forall shard ∈ GetTrail(coin) do
58          SendToShard(shard,
              prePrepare⟨coin, sWallet, tWallet, seq⟩)

59 Phase 2: Prepare ▷ done by non-source
     shards
60 upon ReceiveFromShard(senderID,
61          prePrepare⟨coin, sWallet, tWallet, seq⟩):
62       if IsPresent(coin, sWallet) then
63          forall shard ∈ GetTrail(coin) do
64             SendToShard(shard,
                 prepare⟨coin, sWallet, tWallet, seq⟩)

65 Phase 3: Commit ▷ done by all trail shards
66 prepShards.coin.seq ← ∅
67 upon ReceivedFromShard (
68          senderID ∈ GetTrail(coin),
69          prepare⟨coin, sWallet, tWallet, seq⟩):
70       prepShards.coin.seq ←
71          prepShards.coin.seq ∪ {senderID}
72       if |prepShards.coin.seq| = t − F − 1
           then
73          forall shard ∈ GetTrail(coin) do
74             SendToShard(shard,
                 commit⟨coin, sWallet, tWallet, seq⟩)

75 Phase 4: Reply ▷ done by all trail shards
76 cmtdShards.coin.seq ← ∅
77 upon ReceivedFromShard (
78          senderID ∈ GetTrail(coin),
79          commit⟨coin, sWallet, tWallet, seq⟩):
80       cmtdShards.coin.seq ←
81          cmtdShards.coin.seq ∪ {senderID}
82       if |cmtdShards.coin.seq| = t − F then
83          if GetShard(tWallet) ∉
              GetTrail(coin) then
84             newTrail ←
                 GetShard(tWallet) ∥
85                 all of GetTrail(coin) except
                   last
86          else
87             newTrail ← GetTrail(coin)
88          Record(coin, sWallet, tWallet, seq,
              newTrail)
89          SendToShard(GetShard(tWallet),
              reply(coin, sWallet, tWallet, newTrail))
90          Send(client, reply(coin, sWallet, tWallet,
              newTrail))
```

Multiple coins may be merged and a coin may be split to accommodate more complex transactions: this is analogous to creating 1 dollar out of 100 cents or vice versa. Both operations may be convenient to simplify transactions or make them more efficient. For example, a client wants to send 100 people each 1 cent of their 1 dollar, with coin splitting this can be done in a single transaction. If a coin is split, all its portions inherit the old coins' trail. Coin merging is a bit more involved since the merging coins, even if they are located in the same wallet, may have different trails. To merge, the two coins are marked as merging and their movement transactions are executed jointly. The coins are finally merged once they travel together for the length of the trail. At this point, they share all shards of the trail. This allows clients to combine coins and transfer them to another wallet in a single transaction.

To create, or mine, a new coin, it needs to acquire a trail of length t. This may be accomplished by forming a committee of arbitrary t shards and running a *PBFT* on this committee to agree on the new coin's trail. These processes guarantee that all coins have a trail of the required length regardless of the origin of the coin.

Optimizing internal transaction validation. To decrease message overhead, transactions are divided into internal and external. In an *internal transaction* the source and target wallet are maintained by the same shard. To confirm this transaction, the source shard does not consult the trail shards; it runs internal *PBFT*, and thus relies on the internal shard fault tolerance to maintain wallet integrity. External transactions are processed as usual: with external *PBFT*.

The trade-off for this optimization is decreased shard fault tolerance: the trail shards are not aware of the source shard internal transactions. However, shard failure may be determined by a failure detector [7]–[12]. Such a detector establishes a shard failure and notifies other shards. In the event of a detected shard failure, the trail shards perform a *failed shard recovery procedure* to restore the integrity of the system: the wallets maintained by the failed shard are moved to other shards and their contents are restored to the last known external transaction. The clients have to re-submit internal transactions.

Wallet location, client data recovery, shard maintenance. While coins move between shards, the wallets are assumed to be stationary. For quick shard lookup by the client, the wallet id might contain the shard number. Alternatively, the wallet-to-shard mapping may be recorded in the same or in a separate ledger. For efficiency, wallets frequently participating in joint transactions may be moved to the same shard.

If a client loses its local information about the coin contents of its wallet, it may be able to recover it by conducting a network-wide query. Note that asking the shard that keeps the wallet information alone is not sufficient: the shard may be faulty. Instead, the complete network broadcast is required. The trail shards that confirmed moving the coin should answer to the recovering client.

Again, since some shards may be faulty, the client considers the coin present in its wallet if the trail confirms its location.

In *TRAIL* description, we assumed that the shard sizes are uniform. However, this does not have to be the case. Instead, the shards may grow and shrink as peers join or leave system. Shard sizes may also be adjusted in response to transaction load requirements. Shard membership may be maintained in the shard ledger or, alternatively, in a separate membership ledger.

Algorithm parameter selection. *TRAIL* operates correctly regardless of the concrete values of shard size s and trail size t. These parameters, however, affect the algorithm performance. Larger s makes it less likely that the complete shard fails. Yet, larger s makes the internal consensus algorithm less efficient. The smaller s necessitates larger trail size t to protect against shard failure.

5 Performance Evaluation

Simulation setup. We evaluate the performance of *TRAIL* in an abstract algorithm simulator QUANTAS [13]. QUANTAS simulates multi-process computation, message transmission and has extensive experimental setup capabilities. The simulator is implemented in C++. It is optimized for multi-threaded large scale simulations [14]. The code for our *TRAIL* implementation in QUANTAS as well as our performance evaluation data is available online [15], [16].

The simulated network consists of individual peers. Each pair of peers is connected by a message-passing channel. Channels are FIFO and reliable. A computation is modeled as a sequence of rounds. In each round, a peer receives messages that were sent to it, performs local computation, and sends messages to other peers.

Peers are divided into shards. Shard leaders propose transactions; clients are not explicitly simulated. A transaction has a 25% probability of having source and target wallets in separate shards. Internal transactions are not externally verified by the trail of shards. If a shard is Byzantine, it generates invalid cross-shard transactions only. Specifically, the shard creates transactions moving coins that it has already spent.

Experiment description. Figures 3, 4, 5, and 6 show the dynamics of transaction processing during a computation. In these simulations, a computation runs for 500 rounds. The internal faulty peer tolerance threshold f is 7. The shard size is $s = 3 \cdot f + 1 = 22$. The faulty shard tolerance threshold F is 2. This makes the trail size $3 \cdot F + 1 = 7$. In the experiments, the number of actual faulty shards is equal to the shard fault tolerance threshold F; that is, we run the experiments with maximum tolerance. The faulty shards behave correctly at the start of the simulation and fail at round 100. The total number of shards in the system is $S = 50$. Therefore, the network size is $S \cdot s = 1100$. We run 15 experiments per data point and show the average of the results.

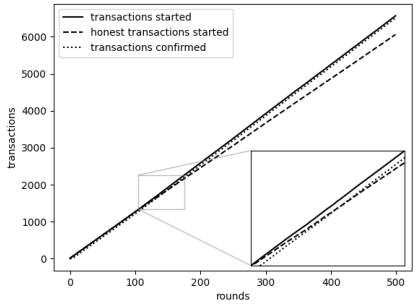

Fig. 3: Transactions approved over time without *TRAIL* shard validation. The network approves both honest and malicious transactions.

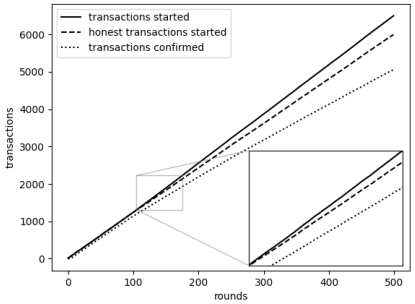

Fig. 4: Transactions approved over time with *TRAIL* shard validation. The network approves honest transactions only.

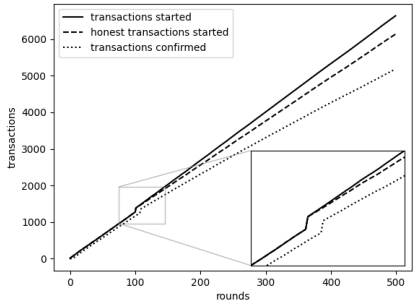

Fig. 5: Transactions approved over time in *TRAIL* with shard validation and wallet recovery from the failed shards. Correct shards detect the failure and submit additional transactions moving coins from the failed shards.

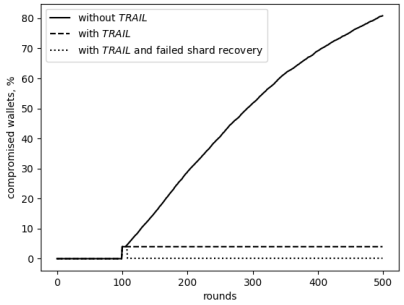

Fig. 6: Percentage of wallets compromised by malicious transactions.

Figures 3, 4, and 5 show the accumulated counts of started and confirmed transactions. We distinguish between the *honest* transactions generated and the total number of transactions, which includes *malicious* transactions generated by faulty shards. The number of honest confirmed transaction is lower.

In Figure 3, no cross-shard validation is performed. In this figure, the number of confirmed transactions matches the total number of transactions; that is, transactions are confirmed whether they are malicious or not. The graph indicates a certain delay before transaction starting and confirmation due to the operation of external and internal *PBFT*.

In Figure 4, *TRAIL* validates the external transactions. Malicious transactions are not confirmed, and the total number of confirmed transactions only accounts for the honest transactions.

In graph shown in Figure 5, *TRAIL* uses the failed shard recovery procedure. Specifically, at round 100, when F shards fail, *TRAIL* detects the faults and generates transactions to move the coins from the faulty shard wallets to the correct ones. This explains the increase in the transaction generation and confirmation rates near round 100 in the figure.

Figure 6 shows the effect of malicious transactions on the overall system integrity. A wallet is *compromised* if it is in a faulty shard or if it receives a coin from a compromised wallet that is not possessed by that compromised sender wallet. A wallet is *safe* otherwise. The safety of a compromised wallet can be restored by the failed shard recovery procedure. The solid line in Figure 6 shows the wallet compromise trend if no cross-shard validation is used. In this case, the failed shards continuously generate malicious transactions, compromising progressively larger number of wallets in the correct shards of the network. In the case in which cross-shard validation is used, the dashed line in Figure 6, the number of compromised wallets does not exceed the number of wallets in the failed shard. In the case with the failed shared recovery procedure, all wallets eventually become safe again: the correct shards generate coin wallet recovery transactions. The delay in wallet recovery shown in the graph is due to the validation of these transactions.

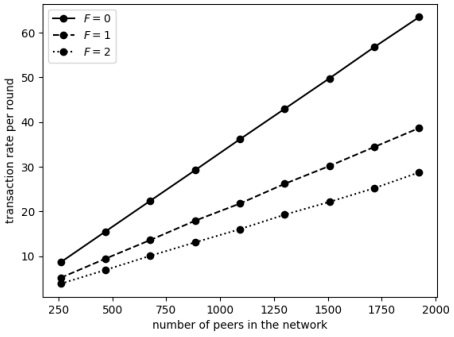

Fig. 7: Throughput with respect to the number of peers in the network for different fault tolerance levels.

Figure 7 shows the performance of *TRAIL* at scale. We run these experiments with a maximum of 148 shards made up of 13 peers per shard. Each data point represents the average throughput from 5 simulations of 200 rounds each. We plot *TRAIL*'s performance with three shard tolerance thresholds F: 0, 1 and 2. There is no cross-shard validation in case of $F = 0$. The figure indicates that the performance of *TRAIL* scales well with network size increase. Larger fault tolerance thresholds incur more overhead. Therefore, the transaction rate is lower for higher values of F.

6 Related Work and Its Application to *TRAIL*

Sharding blockchains. A number of sharding blockchains are presented in the literature. See Le *et al.* [17] for an extensive recent survey. We, however, have not seen an approach where sharding is done on the basis of the coin trail. We are not aware of any blockchain that is robust to shard failure. We believe that most of the published blockchains, even if they do not use *PBFT*, can employ *TRAIL* to fortify themselves against shard failures. For this, consensus on transactions has to be deferred until the transaction's trail confirms it.

***PBFT* optimizations and replacements.** There are numerous proposals to optimize *PBFT* performance. See Wang *et al.* for a survey [18]. Several propose using multiple leaders concurrently [19]–[21]. *Mir-BFT* [19] and *RCC* [21] suggest accelerating *PBFT* by processing non-conflicting requests concurrently. The algorithms have multiple leaders that process these requests simultaneously. *Big-BFT* [20] further enhances parallelism by pipelining subsequent requests. The above *PBFT* optimizations can be applied in *TRAIL* to the internal shard consensus protocol in a straightforward manner. Most of these optimizations can also be applied to the external *TRAIL* algorithms as well.

7 Future Work

The *TRAIL* algorithm presented in this paper is the first to systematically address Byzantine shard failure protection in blockchains for cryptocurrencies. We foresee that it might be developed into a fully-fledged system. Alternatively, *TRAIL* may be used as an add-on component to fortify existing blockchains against shard failure. As a third alternative, *TRAIL* may be enhanced to handle more challenging conditions, such as network partitioning [22] or dynamic networks [23]. Any and all of these alternative directions will increase the robustness of future blockchains.

Acknowledgements. We would like to thank Mitch Jacovetty of Kent State University for his contributions to the paper.

References

[1] M. Jacovetty, J. Oglio, *et al.*, *Trail: Cross-shard validation for cryptocurrency byzantine shard protection*, 2024. arXiv: `2405.07146 [cs.DC]`. [Online]. Available: `https://arxiv.org/abs/2405.07146`.
[2] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system,* 2008.
[3] G. Wood, "Ethereum: A secure decentralized generalized transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.
[4] L. Lamport, R. SHOSTAK, *et al.*, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[5] M. Wendl, M. H. Doan, *et al.*, "The environmental impact of cryptocurrencies using proof of work and proof of stake consensus algorithms: A systematic review," *J. of Env. Management*, vol. 326, p. 116 530, 2023.
[6] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, Nov. 2002.
[7] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *JACM*, vol. 43, no. 2, pp. 225–267, 1996.
[8] T. D. Chandra, V. Hadzilacos, *et al.*, "The weakest failure detector for solving consensus," *JACM*, vol. 43, no. 4, pp. 685–722, 1996.
[9] Q. Bramas, D. Foreback, *et al.*, "Packet efficient implementation of the omega failure detector," *Theory of CS*, vol. 63, pp. 237–260, 2019.
[10] A. Doudou and A. Schiper, "Muteness detectors for consensus with byzantine processes," in *PODC*, 1998, p. 315.
[11] K. P. Kihlstrom, L. E. Moser, *et al.*, "Byzantine fault detectors for solving consensus," *The Computer Journal*, vol. 46, no. 1, pp. 16–35, 2003.
[12] R. Baldoni, J.-M. Hélary, *et al.*, "Consensus in byzantine asynchronous systems," *Journal of Discrete Algorithms*, vol. 1, no. 2, pp. 185–210, 2003.
[13] J. Oglio, K. Hood, *et al.*, "Quantas: Quantitative user-friendly adaptable networked things abstract simulator," in *AppLIED*, 2022, pp. 40–46.
[14] B. Shoshany, "A C++17 Thread Pool for High-Performance Scientific Computing," *arXiv e-prints*, arXiv:2105.00613, May 2021. DOI: 10.5281/zenodo.4742687. arXiv: 2105.00613 [cs.DC].
[15] *Trail implementation in quantas*, https://github.com/QuantasSupport/Quantas / tree / 54ca5de9d556338d1281f85317fb555afd2171fb / quantas/TrailPeer, Aug. 2023.
[16] *Trail perfromance evaluation data*, http://www.cs.kent.edu/~mikhail/Research/trail.output.tar.gz, Aug. 2023.
[17] Y. Li, J. Wang, *et al.*, "A survey of state-of-the-art sharding blockchains: Models, components, and attack surfaces," *Journal of Network and Computer Applications*, p. 103 686, 2023.
[18] X. Wang, S. Duan, *et al.*, "Bft in blockchains: From protocols to use cases," *ACM Computing Surveys (CSUR)*, vol. 54, no. 10s, pp. 1–37, 2022.
[19] C. Stathakopoulou, D. Tudor, *et al.*, "Mir-bft: Scalable and robust bft for decentralized networks," *Journal of Systems Research*, vol. 2, no. 1, 2022.
[20] S. Alqahtani and M. Demirbas, "Bigbft: A multileader byzantine fault tolerance protocol for high throughput," in *IPCCC*, IEEE, 2021, pp. 1–10.
[21] S. Gupta, J. Hellings, *et al.*, "Rcc: Resilient concurrent consensus for high-throughput secure transaction processing," in *ICDE*, IEEE, 2021, pp. 1392–1403.
[22] K. Hood, J. Oglio, *et al.*, "Partitionable asynchronous cryptocurrency blockchain," in *ICBC*, IEEE, 2021, pp. 1–9.
[23] R. Bricker, M. Nesterenko, *et al.*, "Blockchain in dynamic networks," in *SSS*, 2022, pp. 114–129.