Attachment for #515

OUTLINE

This attachment mainly covers the following points:

- For *Reviewer#3*:
  - More comprehensive description of BCR construction algorithm in §1.1.1. A new novel theoretical finding on storage cost is presented in §1.1.2.
  - Literature review for BCR in §1.2
  - ToR protocol specification in §1.3
  - Gateway latency cost evaluation in §1.4
- For *Reviewer#4*:
  - Proof of BCR's uniqueness and completeness in §2.1

1. For Reviewer#3

1.1. BCR construction algorithm

For a more comprehensive description of constructing BCR, we update and present it in a more detailed and clearer style in §1.1.1. During the rebuttal response period, we also demonstrate in §1.1.2 that the algorithm could achieve only 1 hash persistent storage at the best-case, $\lceil log_2 k \rceil$ at the worst-case space, and $\frac{\lceil log_2 k \rceil}{2}$ at the average.

1.1.1. Algorithm1 Description. This Algorithm1 needs to achieve two objectives: 1) Update the BCR by treating the cross-chain message as a new leaf node and using the MerkleUpdatePath. The MerkleUpdatePath is a list that stores several intermediate nodes of the Merkle tree, allowing the cross-chain message to compute the new BCR with those nodes. 2) Generate the MerkleUpdatePath for the next leaf node. As an online algorithm, it generates and stores a new MerkleUpdatePath for the next leaf node.

The algorithm obtains several key global variables stored from the cross-chain service contract on the parachain: 1) $MerkleUpdatePath[h]$ stores the latest MerkleUpdatePath for the block at height h, used for updating the BCR. 2) $Cnt[h]$ stores the number of cross-chain messages in the block at height h and serves as the index of the current cross-chain message in the leaf nodes (starting from 1). 3) $Bcr[h]$ stores the BCR for the block at height h. The algorithm takes the cross-chain message cm and the block height h as inputs. By height h, we can get $MerkleUpdatePath[h]$ and $Cnt[h]$

As the algorithm iterates from the leaf node to the root (line:6 and line:25) layer by layer, it progressively achieves the above two objectives.

To achieve the first objective, during the upward iteration:

- If the current node's index in the current layer is odd (line:11), then it needs to calculate the parent node's hash with the right sibling node. Since the right node is empty, its hash value is assigned to the hash of the current node. And then the hash of the parent node is calculated (line:16);
- If the current node's index in the current layer is even (line:17), then it needs to calculate the parent node's hash with the current node's left sibling node. Since the left sibling node is a historical node that has been already saved in $MerkleUpdatePath[h]$, it can be obtained from that (line:18). Then calculate the parent node's hash value (line:23). The nodes in $MerkleUpdatePath[h]$ are saved when executing the algorithm with the cross-chain message at the index $Cnt[h] - 1$;

To achieve the second objective, during the upward iteration, it is necessary to add the key nodes which are used to update BCR for the next cross-chain message at the index $Cnt[h] + 1$. Importantly, we introduce a variable $flag$ to record the distribution of odd and even nodes in the upward iteration process. The $flag$ has two possible values: ALLEVEN and ANYODD. ALLEVEN means that all nodes in the upward iteration are even, and ANYODD means that at least one node in the upward iteration is odd. Specifically:

- If the current node's index in the current layer is odd (line:11), check $flag$ state. If $flag$ is ALLEVEN (line:12), it means all nodes in the iteration are even, indicating that the subtree rooted at the current node is a full binary tree. Any new cross-chain messages will not be appended to that subtree, but the one rooted at the right sibling node. Therefore, the current node needs to be added to mp (line:13, mp is a local variable list which will be assigned to $MerkleUpdatePath[h]$ at the end). If $flag$ is ANYODD, meaning at least one node in the iteration is odd, indicating that the subtree is not full. And new cross-chain messages will added to this subtree, so the current node does not need to be added to mp. Finally, set $flag$ to ANYODD (line:15) because the current node is indexed as odd.
- If the current node's index in the current layer is even (line:17), calculate the parent node's hash with the left sibling node, whose hash is obtained from $MerkleUpdatePath[h]$ (line:18). If $flag$ is ANYODD (line:20), it means the subtree is not full, and the left

sibling node needs to be added to mp (line:21) for the subsequent cross-chain message. If $flag$ is ALLEVEN, the subtree is full, and no new leaf nodes will be added, so the left sibling node does not need to be added to mp.

Algorithm 1 Constructing BCR: A^u_{BCR}.

Global States:
$Mp[h]$ ▷ Merkle path for updating BCR;
$Cnt[h]$ ▷ number of the cross-chain messages;
$Bcr[h]$ ▷ BCR at block h;
Input:
h ▷ block height;
cm ▷ cross-chain message;
Output:
r ▷ the root of the Merkle tree that updated;
1: $tempNode \leftarrow \mathcal{H}(cm)$ ▷ hash of cm
2: $index \leftarrow Cnt[h] + 1$ ▷ index of cm
3: $pathPoint \leftarrow 0$ ▷ point to the position in $Mp[h]$
4: $flag \leftarrow$ ALLEVEN ▷ initialize flag
5: $mp \leftarrow EmptyList$ ▷ store the new Merkle path
6: **while** $index \geq 1$ **do**
7:   **if** $index = 1$ **then**
8:     mp.append($tempNode$)
9:     **break**
10:   **end if**
11:   **if** $index \bmod 2 = 1$ **then**
12:     **if** $flag$ is ALLEVEN **then**
13:       mp.append($tempNode$)
14:     **end if**
15:     $flag \leftarrow$ ANYODD
16:     $tempNode \leftarrow \mathcal{H}(tempNode \parallel tempNode)$
17:   **else**
18:     $node \leftarrow Mp[h][pathPoint]$
19:     $pathPoint \leftarrow pathPoint + 1$
20:     **if** $flag$ is ANYODD **then**
21:       mp.append($node$)
22:     **end if**
23:     $tempNode \leftarrow \mathcal{H}(node \parallel tempNode)$
24:   **end if**
25:   $index \leftarrow \lceil index/2 \rceil$
26: **end while**
27: $Mp[h] \leftarrow mp$
28: $Bcr[h] \leftarrow tempNode$
29: $Cnt[h] \leftarrow Cnt[h] + 1$
30: **return** $tempNode$

1.1.2. Storage Analysis. Assume that the block at height h contains k cross-chain messages.

In the best-case with $k = 2^p$ and $p \in \mathbb{N}$, only 1 hash(i.e. 32 bytes) is stored persistently in a block. In Algorithm1, we can see that during the upward iteration, if the index is always even(line:17), then only node in the toppest layer(line:8) needs to be appended to mp. Because ALLEVEN indicates that the whole tree is full, the new incoming cross-chain message should be added to the adjacent tree, thus, only the current tree root is needed to calculate the next BCR for the new cross-chain message. Therefore, the number of hash persistent storage is 1, with $k = 2^p$ and $p \in \mathbb{N}$. It also means that the cross-chain message at the index $k = 2^p + 1$ only needs one old node hash to calculate BCR.

In the worst-case with $k = 2^p - 1$ and $p \in \mathbb{N}$, $\lceil log_2 k \rceil$ hash persistent storage is needed in a block. In Algorithm1, we can see that during the upward iteration, if the index of the leaf node is odd(line:11) but all the upward iterated nodes' indexes are even(line:17), then the leaf node, and upward iterated nodes' left sibling nodes should be appended to the mp. The size of mp is $\lceil log_2 k \rceil$. Therefore, the number of hash persistent storage is $\lceil log_2 k \rceil$, with $k = 2^p - 1$ and $p \in \mathbb{N}$. It also means that the cross-chain message at the index $k = 2^p$ needs $\lceil log_2 k \rceil$ old hashes to calculate BCR.

For the average storage, $\frac{log_2 k}{2}$ hash persistent storage is needed in a block. In Algorithm1, we can find that if a node's index at a layer is even, then its left sibling node is needed to calculate the parent node's hash. If the current node's index is odd, then it needs to calculate the parent node's hash with its right sibling node which is a copy of the current node. Therefore, for a cross-chain message at the index k, during the upward iteration, the number of even nodes is equal to the size of mp which generated by the cross-chain message at the index $k - 1$. For each iteration, the probability that the node's index is even is $\frac{1}{2}$. The algorithm always executes iteration $\lceil log_2 k \rceil$ times, leading to the expected mp size $\frac{\lceil log_2 k \rceil}{2}$.

1.2. BCR Construction Literature Review

There are some Merkle tree constructing methods to calculate the root:

- **UpdateByLeaves.** Construct a new Merkle tree using the current and previous messages with $O(k)$ time complexity and $O(1)$ a space complexity (k, number of messages). For example, Bitcoin [1] uses a simple Merkle tree to construct the root committing to all transactions in a block, without any intermediate hashes storage. It is more suitable where updating is not frequent.
- **UpdateByTree.** Store the whole tree and directly update its intermediate nodes when a new leaf is appended. Existing blockchain platforms such as Ethereum [2], Solana [3], and Cosmos [4], Zilliqa [5], Near [6], and Quorum [7] use this method to construct the root committing to all states like account balances and constract storage values. This method has $O(\log(k))$ time complexity but $O(k)$ space complexity, making it more suitable where nodes update frequently and storage is cheap;

Some other methods are also proposed to improve the efficiency of Merkle tree construction, while not practical for constructing the BCR within blockchain contracts. [8] implementes an append-only Merkle tree structure with a time complexity of $O(1)$ and space complexity of $O(k)$. However, it does not store the root on-chain. BIP-0098

[9] proposes a more efficient Merkle tree structure, but it modifies the SHA-256 hash function, not compatible with existing mainstream blockchains. [10] accelerates Merkle Patricia tree construction using GPU, while not suitable for contracts. Above algorithms may not fully meet the requirements for updating BCR efficiently, economically and practically.

1.3. ToR Protocol Specification

We present the ToR protocol description. It mainly includes three parts: contract functionalities on parachains (Fig. 3), contract functionalities on the relay chain (Fig. 2), and gateway transmission protocol (Fig. 4).

In the parachain's cross-chain service contract, there are four core functionalities, including `Initialize`, `IssueCM`, `SyncHeaderFromRC`, and `ReceiveCM`. The `Initialize` is executed to initialize several key global variables when deploying the contract on-chain. The `IssueCM` takes $< dstId, payload >$ as input, in which the $dstId$ is target chain's identifier and $payload$ is for the cross-chain application. The `SyncHeaderFromRC` receives, verifies and stores the relay chain's header from the gateway by the light client LC. `ReceiveCM` happens on the target chain for receiving the source chain's cross-chain messages and related DLV proofs.

In the relay chain's cross-chain service contract, only two functionalities are necessary. `Initialize` is executed upon deploying the contract. `VerifyTrustRoot` accepts the parachain header and its corresponding BCR with the state proof (a Merkle path from BCR to the root in the header). It first verifies header validity by the parachain's (identified by pid) light client LC_{pid}. Then the BCR is verified based on the parachain's Merkle tree algorithm MTA_{pid}.

The gateway runs three core processes, including `TrustRoot_S2R` synchronizing trust roots from parachains to the relay chain's `VerifyTrustRoot`, `Header_R2D` synchronizing the relay chain's headers to parachain's `SyncHeaderFromRC`, and `Message_S2D` transmitting cross-chain messages from the source chain to the target chain. In the `Message_S2D` process, upon receiving a new cross-chain message cm finalized in the source chain's $block_h$ (at the height h), the gateway needs to wait $block_h$'s BCR until it is finalized on the relay chain. Then the gateway queries BCR's state proof (as $vp1$) on the relay chain and the finalized block height rh. Combined with cm's proof (a Merkle path from cm to BCR, as $vp2$), the gateway sends $< cm, vp1, vp2, rh >$ to the target chain's `ReceiveCM`

1.4. Gateway Latency Cost

We present the latency of the gateway transmitting messages from the time listening to the a message on the source chain to the time sending the message to the target chain in Fig. 1. We issue 100,000 messages in the cross-chain system with pn=[10, 60, 120, 180] fully connected parachains, and control the cross-chain message ratio to $ratio$=[0.2, 0.4, 0.6, 0.8, 1.0]. We set the y-axis to $log_2(latency)\cdot sec^{-1}$ for better visualization. Fig. 1 shows that the latency of ToR remains consistently low across different ratios, while the latency of AoR increases sharply with $ratio$. The NoR latency also remains relatively stable, but there is a significant increase in latency as pn increases. In a small-scale system (pn=10), the gateway latency of ToR is a bit higher than NoR, because the header synchronization overhead is relatively lower in NoR. But with pn increasing, the latency of ToR is much lower than NoR, because the header synchronization overhead grows exponentially which makes the cross-chain message queuing delay more serious.

2. For Reviewer#4

2.1. Proof of BCR's Uniqueness and Completeness

2.1.1. Uniqueness Proof. Assume by contradiction that there exist two different valid BCRs, BCR_1 and BCR_2, for the same block B. By the BCR definition, both BCR_1 and BCR_2 must be constructed using the same algorithm MTA_0 and the same ordered list of cross-chain messages CMs.

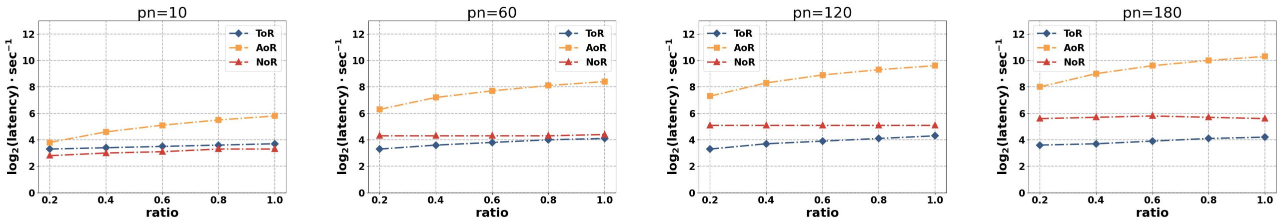
First, the ordered list of CMs is deterministic in each block. Each cm is identified by $< block.height, msgId >$. Both $block.height$ and $msgId$ are increasing monotonically. Second, transaction execution order in each block is deterministic among different blockchain nodes. Therefore, each block generates a deterministic unique CMs. Finally,since MTA_0 is deterministic, MTA_0 must produce the same root for the same input CMs. Therefore, $BCR_1 = MTA_0(CM) = BCR_2$, which contradicts our assumption that $BCR_1 \neq BCR_2$.

2.1.2. Completeness Proof. For the BCR in $block_i$, we have:

Part1: All cross-chain messages in $block_i$ are included in the BCR. Each cross-chain message must be as input of A^u_{BCR} and update the old BCR to a new one. And each one is identified by $< block.height, msgId >$ which is unique in $block_i$, so the cross-chain message can not be overwritten by other messages. If any cross-chain message is not included in the BCR, then the contract execution is broken which contradicts with the blockchain's correctness.

Part2: Any non-cross-chain data are not included in the BCR. Only the cross-chain service contract can update the BCR by calling A^u_{BCR}. The contract's logic is deterministic and each honest node executes the same logic. Malicious nodes can run a fake contract to add non-cross-chain data to the BCR, but it won't be finalized on the chain.

Part3: Any cross-chain messages in $block_j (i \neq j)$ are not included in the BCR. If a cross-chain message is in $block_j$, then it serves as the input of A^u_{BCR} with argument $< block_j.height >$ and finally $blockMerkle[j]$ is updated. Because the message is identified by $< block.height, msgId >$ and $i \neq j$, the intersection of messages in $block_j$ and in $blockMerkle[i]$ is empty. That means



(a) (b) (c) (d)

Figure 1: The the gateway transmitting messages' latency from listening to a new message on the source chain to sending the message to the target chain ($ratio$ and pn).

Cross-chain Service Contract on Relay Chain

Initialize()
Initialize the following global states:

- Other chains' headers $H = \{$chainId $\leftarrow \{$height $\leftarrow$ header$\}\}$
- Other chains' BCR $B = \{$chainId $\leftarrow \{$pcHeight $\leftarrow$ bcr, rcHeight$\}\}$. $pcHeight$ is the height of the BCR finalized on the parachain, $rcHeight$ is the height of BCR finalized on the relay chain.

VerifyTrustRoot(pid, header, bcr, proof)
pid is the parachain's id, $header$ is the parachain's header, bcr is the BCR in the parachain's block at $header.height$, $proof$ is bcr's state proof on the parachain.

- $h \leftarrow header.height$
- if $H[pid][h]$ exists, then revert(EXIST)
- if $LC_{pid}.verify(header)$ is false, then revert(INVALID-Header)
- if $MTA_{pid}.calc(bcr, proof) \neq header.root$, then revert(INVALID-BCR)
- $H[pid][h] \leftarrow header$
- $B[pid][h] \leftarrow bcr, block.height$
- emit event $EntVerifyTrustRoot(pid, h)$

Figure 2: Protocol description of Cross-chain Service Contract on Relay Chain

any cross-chain messages in $block_j$ can not be included in the BCR in $block_i$.

From part1, part2 and part3, we can conclude that the BCR satisfies the completeness.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," https://bitcoin.org/bitcoin.pdf, 2008.

[2] W. Gavin, "Ethereum: A next-generation smart contract and decentralized application platform," https://ethereum.github.io/yellowpaper/paper.pdf, 2024.

[3] "Solana," https://solana.com/zh/developers/guides/advanced/state-compression#what-is-state-compression, 2025.

[4] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on bft consensus," 2019. [Online]. Available: https://arxiv.org/abs/1807.04938

[5] "Zilliqa," https://github.com/Zilliqa/Zilliqa, 2024.

[6] "Near," https://near.org/, 2025.

[7] "Quorum," https://github.com/ConsenSys/quorum, 2024.

[8] A. Kang, "Merkle tree with o(1) append," https://ethresear.ch/t/merkle-tree-with-o-1-append/14748, 2023.

[9] B. Mark Friedenbach, Kalle Alm, "Bip-0098," https://github.com/bitcoin/bips/blob/master/bip-0098.mediawiki, 2024.

[10] Y. Deng, M. Yan, and B. Tang, "Accelerating merkle patricia trie with gpu," Proc. VLDB Endow., vol. 17, no. 8, p. 1856–1869, Apr. 2024. [Online]. Available: https://doi.org/10.14778/3659437.3659443

Cross-chain Service Contract on Parachains

Initialize(_chainId)
Deploy contract on the parachain with chainId and initialize the following global states:

- Current parachain id $srcId$ = _chainId
- Message id $msgId$ = 0
- Relay chain block headers H_{rc} = {height: header}
- Other chains' BCR B = {chainId: {height: bcr}}
- Current chain's BCR Merkle path $blockMerkle$ = {height: {root, path, index}}
- Other chains' messages ids M = {chainId: {msgId: {0,1}}}

IssueCM(dstId, payload)

- $msgId \leftarrow msgId + 1$
- $height \leftarrow$ block.height
- make message $CM \leftarrow \{srcId, dstId, msgId, height, payload\}$
- update BCR in $blockMerkle$ with $A^u_{BCR}(height, CM)$
- emit event $EntIssueCM(CM)$

SyncHeaderFromRC(rch)
rch is a relay chain header.

- if rch.height in H_{rc}, then revert(EXIST)
- if not $LC.verify(rch)$, then revert(INVALID)
- $H_{rc}[rch.height] \leftarrow rch$
- emit event $EntSyncHeader(rch)$

ReceiveCM(CM, vp1, vp2, rh) $vp1$ and $vp2$ are the DLV proofs of CM. rh is height of CM's BCR finalized on the relay chain.

- $sId \leftarrow CM.srcId$;
- $mId \leftarrow CM.msgId$;
- $h \leftarrow CM.height$;
- if $M[sId][mId] = 1$, then revert(EXIST)
- $BCR' \leftarrow MTA_0.calc(CM, vp1)$
- if $B[sId][h] = \perp$, then
  - $root' \leftarrow MTA_0.calc(BCR', vp2)$
  - if $root' \neq H_{rc}[rh].root$, then revert(INVALID)
  - else $B[sId][h] \leftarrow BCR'$
- else
  - if $BCR' \neq B[sId][h]$, then revert(INVALID)
- $M[sId][mId] \leftarrow 1$
- dApp.execute(CM)
- emit event $EntReceiveCM(CM)$

Figure 3: Protocol description of Cross-chain Service Contract on Parachains

Protocol description of Gateway

Initialize(EP_s, EP_d, EP_r)
EP_s is the source chain's endpoint, EP_d is the destination chain's endpoint, EP_r is the relay chain's endpoint. The endpoint could call the corresponding contract and chain's basic functions.
Initialize the following global states:

- Other chains' headers H = {chainId $\leftarrow$ {height $\leftarrow$ header}}
- Other chains' BCR B = {chainId $\leftarrow$ {height $\leftarrow$ bcr}}

TrustRoot_S2R()
Transmit the trust root from the source chain to the relay chain.
Keep listening to the source chain's new block Blk:

- $h \leftarrow Blk.height$
- $header \leftarrow Blk.header$
- $bcr \leftarrow EP_s.getBCR(h)$
- $proof \leftarrow EP_s.getProof(h, bcr)$
- $pid \leftarrow EP_s.getChainId()$
- call $EP_r.VerifyTrustRoot(pid, header, bcr, proof)$

Header_R2D()
Transmit the header from the relay chain to the destination chain.
Keep listening to the relay chain's new block Blk:

- $header \leftarrow Blk.header$
- call $EP_d.SyncHeaderFromRC(header)$

Message_S2D()
Transmit the message from the source chain to the destination chain.
Keep listening to the source chain's event $EntIssueCM(cm)$:

- $sh \leftarrow cm.height$
- $sId \leftarrow cm.srcId$
- $vp1 \leftarrow EP_s.getProof(sh, cm)$
- wait until BCR of sh on the relay chain is finalized
- $(bcr, rcH) \leftarrow EP_r.getBCR(sId, sh)$
- if $EP_d.B[sId][sh]$ not exists, then $vp2 \leftarrow EP_d.getProof(sh, bcr)$
- else $vp2 \leftarrow \perp$
- call $EP_d.ReceiveCM(cm, vp1, vp2, rcH)$

Figure 4: Protocol description of Gateway