AIP[1]-007: Provable Blockchain States

platfowner, minsulee2, liayoo 2020-06-17

Goals

- Provide *proof* of blockchain states

Terminology

We define *partial state proof* as follows:

- For a blockchain block and a path in the state tree, *state proof* is defined to be showing that the state root hash in the block header is from the state values at the path
- A state proof is said to be *full* if the given path is the root of the state tree, otherwise it's said to be *partial*

Background

State Proof in Ethereum

For Bitcoin and Ethereum cases, see Merkling in Ethereum and Patricia Tree.

The following shows Ethereum getProof() web3js API:

```
web3.eth.getProof(address, storageKey, blockNumber, [callback])
```

Ethereum States vs. AIN Blockchain States

The following table shows a comparison between Ethereum states and AIN Blockchain states:

	Ethereum States	AIN Blockchain States	Notes
structure	list	tree	
key	account address	arbitrary path	
value	nonce, balance, storageRoot, codeHash	arbitrary tree object (subtree)	

[1] AI Network Improvement Proposal. Visit https://docs.ainetwork.ai for the full list.

state access operations	write(address, value)
	read(address)
		write(path, value)
		read(path)	

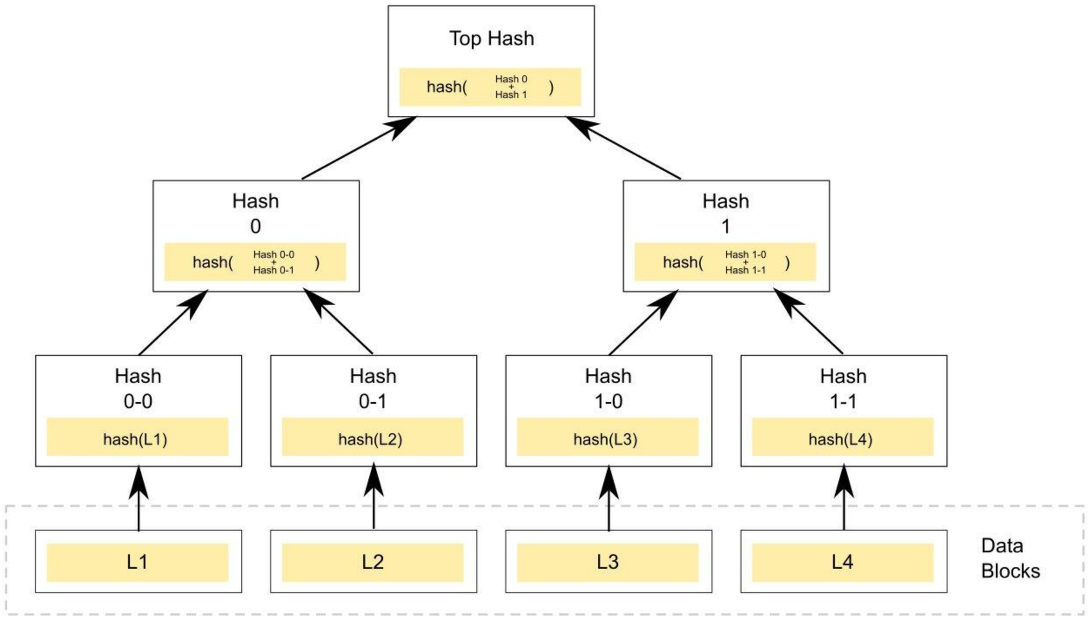
state proof operations	prove(address, storageKey, blockNumber)	prove(path, blockNumber)	

Requirements

- For the latest finalized[2] block, provide partial state proof for any state subtree in O(<path length>) time regardless of the branching factors of the state tree
- For older finalized blocks, provide full state dump and tools for self proof service
- The hash function of the proof hashes should be stable, i.e., independent of state update operation order

Key Ideas

- For partial state proof, build a Hash Tree, i.e., keep proof hash in each internal node of the state tree, reusing the existing state tree



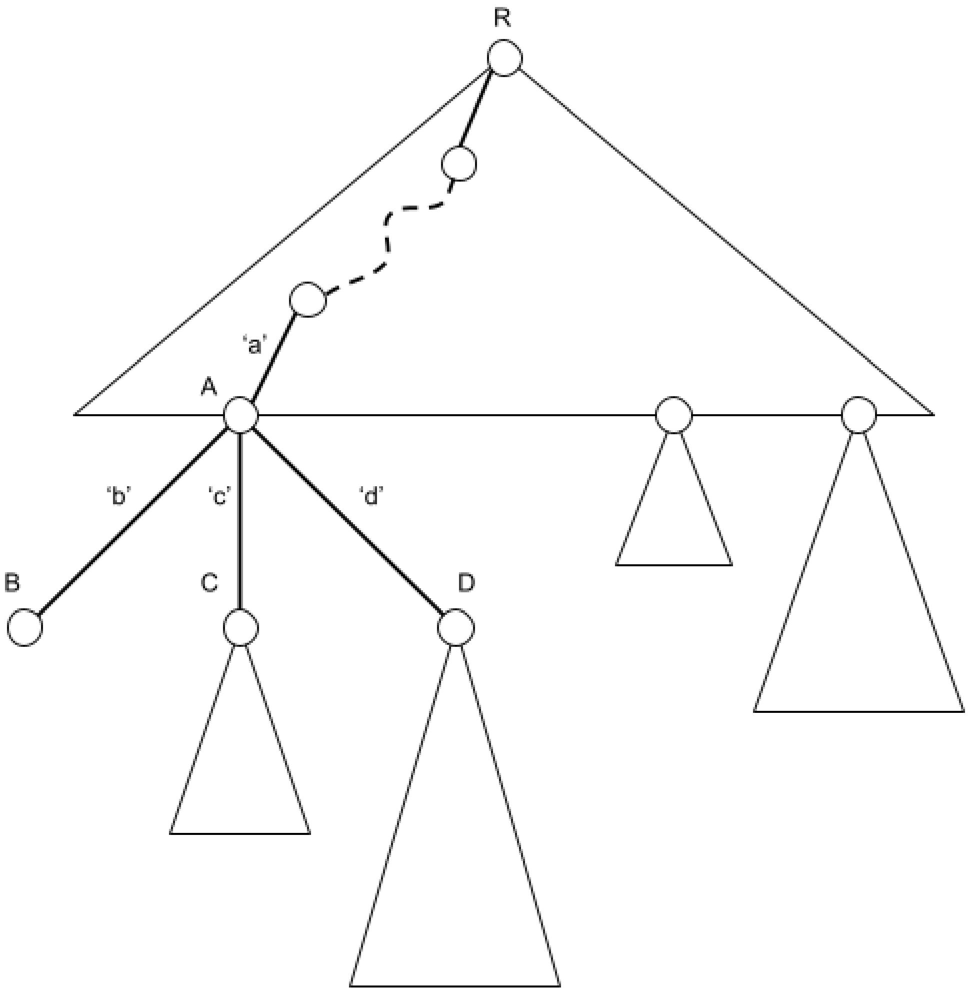
- For handling high branching factors, use Radix Tree
- For full state dump for older states, keep previous state trees in disk

[2] See Streamlet: Textbook Streamlined Blockchains

Proposed Design

Proof Hash

Proof Hash Generation



For a given state node A which has child state nodes B, C, and D, the proof hash for A proofHash(A) is defined as follows:

proofHash(A) = hash(join('b', proofHash(B), 'c', proofHash(C), 'd', proofHash(D)))

where hash() is a one-way hash function and join() is a string join function. For example, we can use a join function like join(‘b’, ‘0xBBB’, ‘c, ‘0xCCC’, ‘d’, ‘0xDDD’) = ‘b.0xBBB.c.0xCCC.d.0xDDD’ .

where dot(.) is a reserved character in AIN Blockchain state path (see Predefined Structure and Built-ins). To remove ambiguity, the child state nodes are sorted by a sorting rule e.g. by the node labels before being joined.

We can also use a cryptographic hash function in order to generate a fingerprint of each corresponding path. The hash function, Sha256 here, is adopted to hash given values. Since hashing of values always generates a unique fingerprint, it can be said that the given path has the unique hash fingerprint.

SHA256(string) = 0x[0...e]; where length = 64

For example, the above combination of child proof hashes ‘b.0xBBB.c.0xCCC.d.0xDDD’ is hashed by SHA256 and then it is shown as the hexadecimal form like ‘1a0c1a0c70ef07fb8b009dc84225eb3e161e06f657ac593d5b24b6b9aa9c72af’.
If any values in the path changes a single bit, the hash fingerprints completely differ from the previous hash value. As given path changes to ‘b.0xBBB.c.0xCCC.d.0xDDDD’,which the last value is add one more ‘D’ at the end of the combination, and then the proof hash will be generated as ‘3cee7b3a7e265aef4a6c82bab2a665c6f3b8cac4537c2c6ade35c2a4d3e0ad85’ for instance.

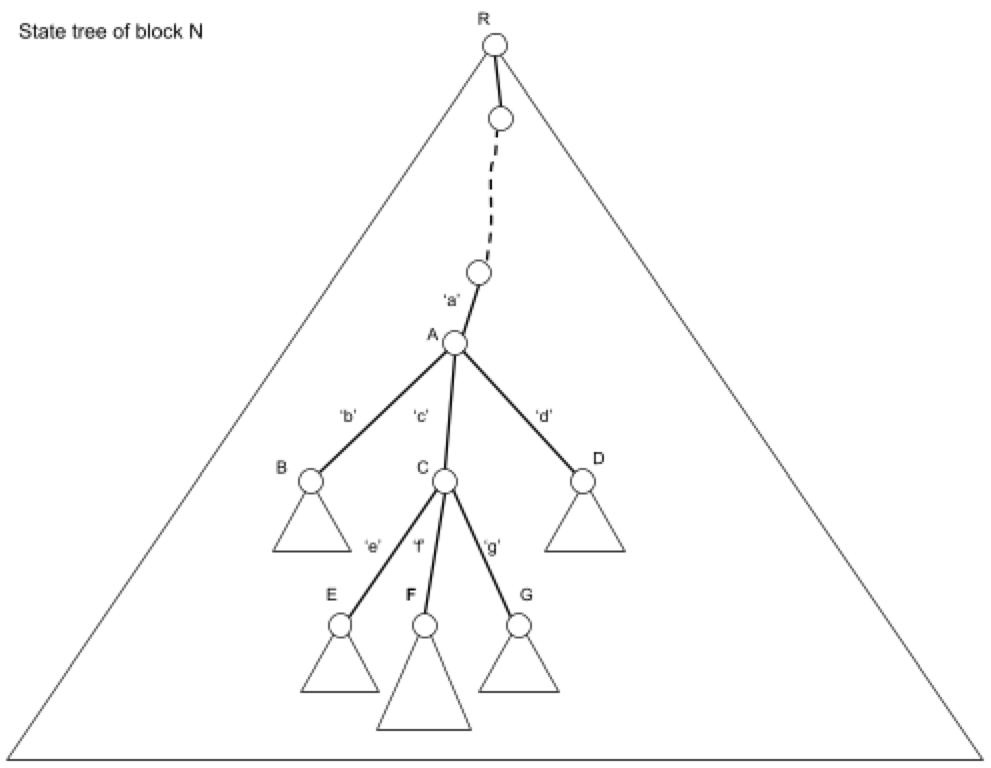
It turns out that with uniqueness of the hash values, it is able to chain the nodes to the root node of the merkle tree through hash chaining. Consequently, the final value ‘0xDDDD’ affects subtree A, and then the changed proof of subtree A also affects the Merkle root R. The R generates a new hash value, which includes the changing history of data from the bottom in the Merkle tree in the end.

Proof Hash As Meta-Data

Proof Hash is stored in the state tree as meta-data. For more information, see AIP-006 State Version Control.

Partial State Proof

We provide partial state proof for the latest finalized block and its state tree. To do that the blockchain needs to have the full states of the latest finalized block. The partial state proof is done via prove(path, blockNumber) function which returns all proof hashes from the given path to the root of the state tree.



Let F be a state node with path /.../a/c/f in the state tree of block N as shown in the above diagram. Then prove('/.../a/c/f', N) function returns the following value:

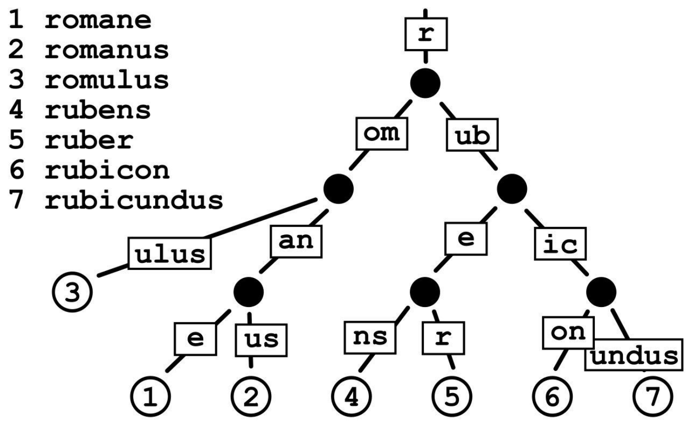
```
{
    ...
        'a': {
            '.proof_hash': proofHash(A),
            'b': {
                '.proof_hash': proofHash(B)
            },
            'c': {
                '.proof_hash': proofHash(C),
                'e': {
                    '.proof_hash': proofHash(E),
                },
                'f': {
                    '.proof_hash': proofHash(F)
                },
                'g': {
                    '.proof_hash': proofHash(G)
                }
            },
            'd': {
                '.proof_hash': proofHash(D),
            },
        },
   ...
}
```

With the returned value, the caller of the prove() function can verify that the proof is valid or not by applying proofHash() function to the nodes in the path recursively.

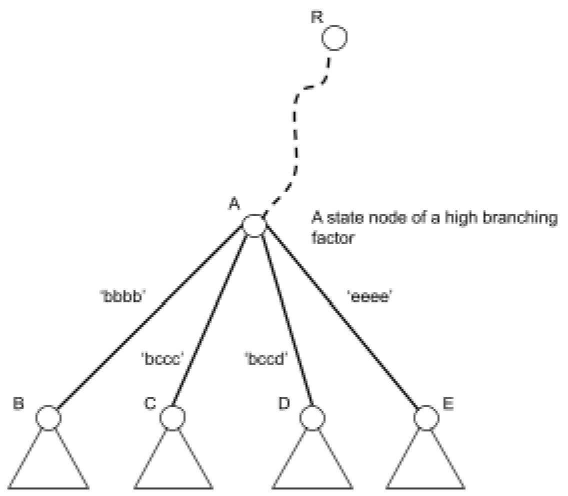
Proof Hash Updates

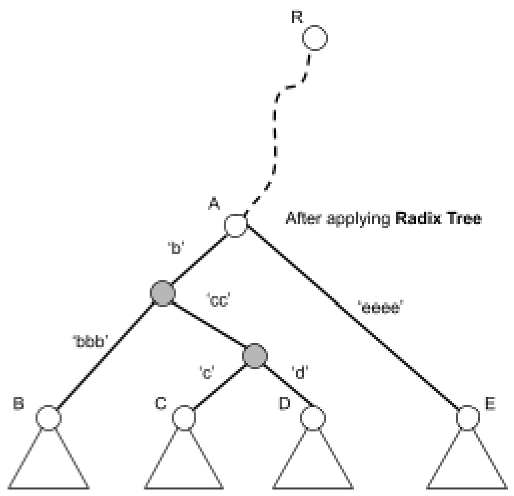
For each write(path, value) operation, the proof hashes in the state tree need to be updated. It's done by traversing the state tree from the state node at the given path to the root. In this case, the time needed for the proof hash update is the summation of the time needed at each state node in the path. If the update time at each state node is O(1), then the whole hash update time is O(<path length>). The thing is that it's not O(1) by the definition of proofHash() if the branching factor of the state node is substantially high. This problem can be solved by using Radix Tree.

Radix Tree



Radix Tree is a space-optimized trie in which each node that is the only child is merged with its parent.





As shown in the diagram above, branching factor can be reduced by using Radix Tree. The reduction effect can be maximized with random labeling, e.g., hash values like addresses. Fortunately, it's often the case as high branching factors come from such use cases like /users/$user_id or /posts/$post_id. In the worst case, however, the branching factor in the Radix Tree is bounded by 256 if we allow only string labels.

Additional methods that can be adopted for keeping the state tree balanced as much as possible is as follows:

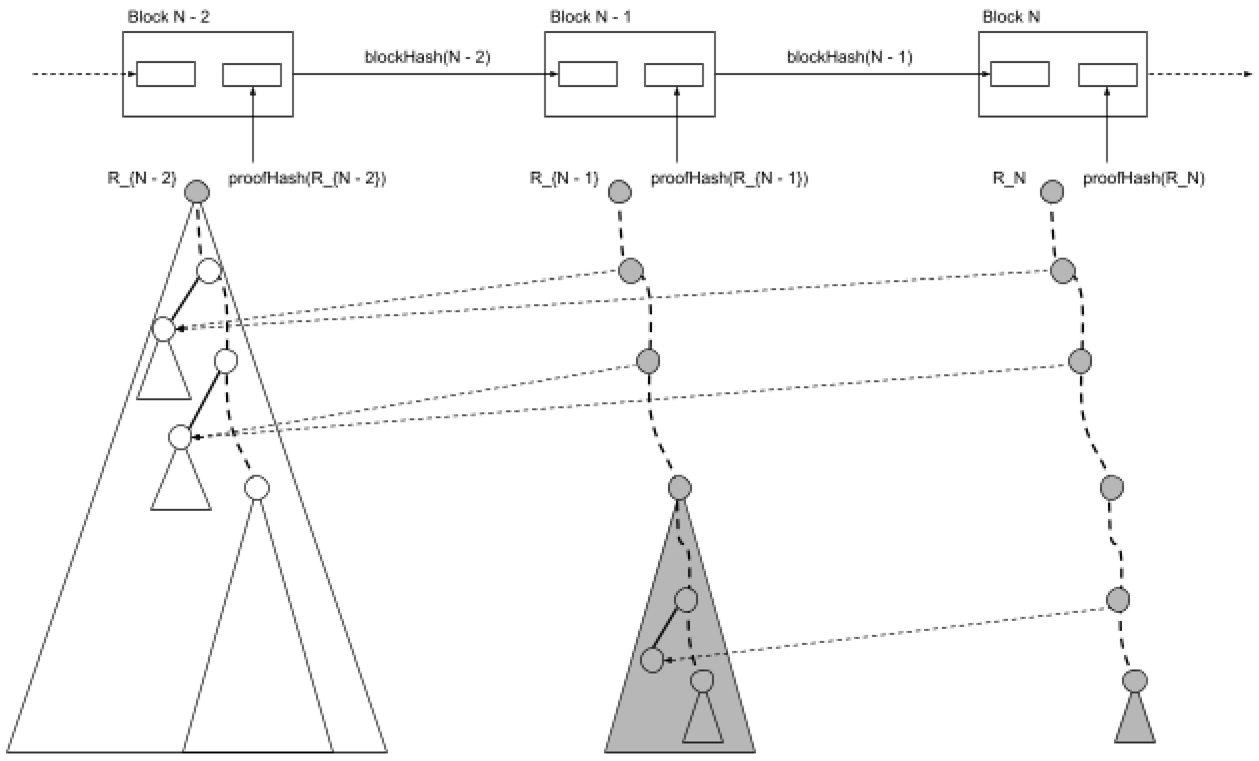
- Charge more for longer paths
- Charge more for high branching factors with non-random labelings

Full State Dump

In order to provide full state dump for older finalized blocks, we need to handle the following issues:

- State trees of non-latest blocks need to be stored efficiently, i.e., with minimized space consumption
- The full state dump can be done in a reasonable time

For the efficient space condition, we can minimize duplicated data in the state trees over generation by using inter-generation references.



The above diagram shows a conceptual visualization of the inter-generation references. In Block N for example, the state tree keeps only the state values altered by the transactions in Block N with the references to the unaltered states in previous blocks.

For the reasonable time condition, we can insert full state trees regularly (e.g. once for K consecutive blocks).

Design Alternatives

Modified Merkle Patricia Tree (MMPT) for Full Paths

Although there are some alternatives to build merkle roots, the modified merkle patricia tree has been adopted and deployed quite well in the Ethereum blockchain network. In the blockchain area, the MMPT can be said to be quite suitable for a blockchain network with transactions.

The MMPT is basically based on a combination of prefix tree and merkle tree. Through the combination of two trees, it takes strong advantages in both efficiency of data store and fast comparison.

Here, blockchain accounts are set as key and a value of each account is set to value.

Before building the MMPT, a key and value are encoded via recursive length prefix encoding and Hex-prefix encoding as prerequisites.

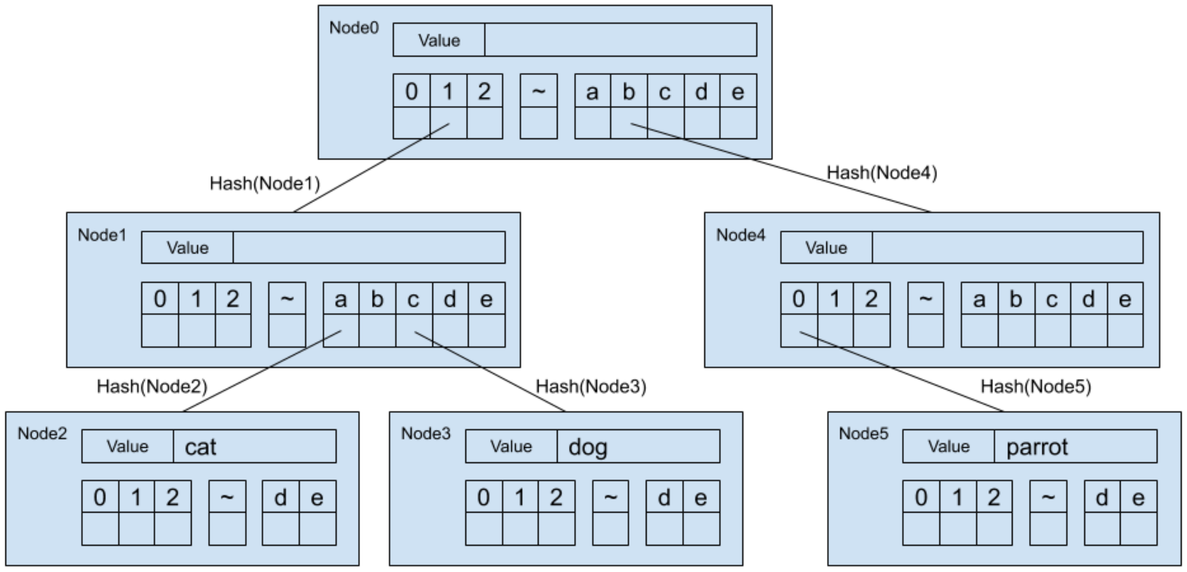
- Through HP encoding, we can distinguish that either a node is a leaf node or an extension node. It also converts the given path as even length.
- The RLP encoding, on the other hand, is used to specify data type as well as compact data and make it simple.

MMPT nodes are simply designed which only include an address table for the next nodes and a value field.

- The leaf node has a value. It generates its own fingerprint through hash function and it affects the merkle proof to the top of the tree.
- The extension node can be said as a “connector”. It literally connects nodes from leaves to the root recording fingerprints of each node.
- An address table stores the next nodes’ hash fingerprint, so that not only is it able to connect children as well as it also achieves enabling a partial comparison of what merkle tree does.

After building the MMPT, merkle roots are, in the end, generated and stored in the block header.

The completed tree can be simply depicted as below:



To adopt the MMPT for the AINetwork blockchain, there are some challenges to deal with.

- The hex-prefix encoding and recursive length prefix encoding are not suitable for the database structure, which is a kind of javascript object form like { foo: { bar: john doe } }.

- Since a key size of the database is no limits, it is difficult to directly use the database keys for merkle path.
- As keys are not predictable, it may have a problem with using extension nodes.
- Forms of the database values are more various than simple number values; such as string, number, array and even object.
- In order to avoid a DOS attack, the key length should be preprocessed.

Roadmap

We have the following roadmap for the proposed features:

1. Introduce an abstract layer for the state operations (read, write, traverse) as described in AIP-006 State Version Control.
2. Add proof hash management for the abstract state operations
3. Add full state proof for new block verification
4. Add partial state proof for user's calls to prove() function
5. Add full state dumping logic for non-latest blocks

Links

- Merkling in Ethereum (link)
- Patricia Tree (link)
- web3js getProof() (link)
- Hash Tree (link)
- Streamlet: Textbook Streamlined Blockchains (pdf)
- Radix Tree (link)
- Predefined Structure and Built-Ins (link)
- Modified Merkle Radix Trees (link)
- AIP-006: State Version Control (link)

Document History

Date	Who	Change	Notes
2020-06-18	platfowner, minsulee2, liayoo	Wrote initial version	
2020-06-25	platfowner, minsulee2	Added details	
2021-05-06	platfowner	Published	
2021-05-12	platfowner	Github IDs Link to full list	

2021-05-12	platfowner	Republished	