#HashEx
BLOCKCHAIN SECURITY

Lachain consensus

audit report

September 2022

hashex.org

contact@hashex.org

Contents

1. Disclaimer

2. Introduction

HashEx was commissioned by the Lachain team to perform an audit of Honey Badger Byzantine Fault Tolerant (HBBFT) consensus realisation. The audit was conducted between March 14 and April 18, 2022.

HBBFT consensus is a part of Lachain Cross Chain DeFi protocol. The protocol allows seamless access to finance products on major blockchains without gas tokens management. All fees and gas are paid with LA token [1].

HBBFT was introduced in October of 2016 by Andrew Miller et al. It was built to eliminate synchronous protocols' problem of critical dependence on network timing assumptions which make them inappropriate for real deployments scenarios [2].

Using the example of PBFT, the authors have proved [3] that weakly synchronous protocols are not long living in malicious networks and would grind to a halt if the time assumption is violated. HBBFT came out from an improved asynchronous atomic broadcast protocol [4]. Andrew Miller et al. proposed enhancements resulted in bit transmission complexity reduction from O(N^2) to O(N).

The code is available at the @LATOKEN/lachain GitHub repository and was audited after the v0.0.71-main tag.

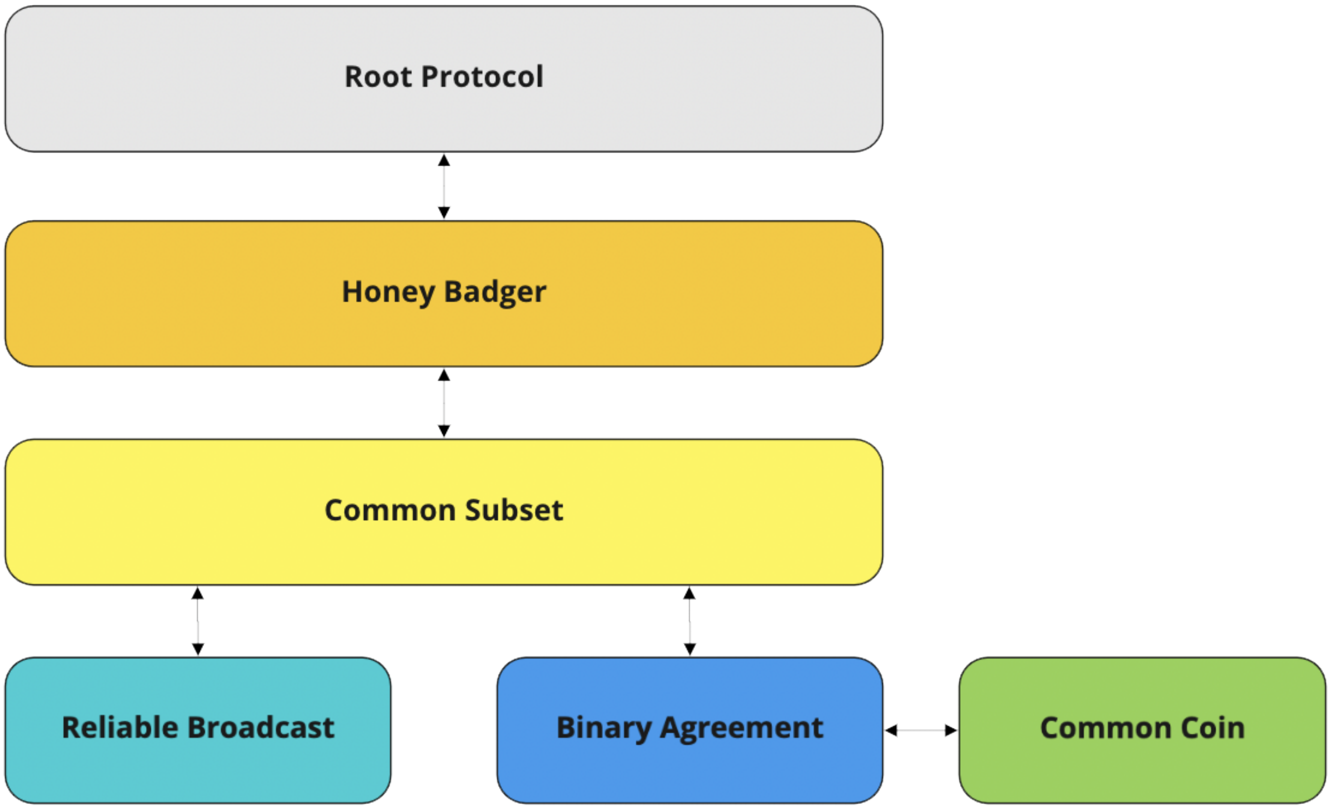
Update: the Lachain team has responded to this report. The updated code is located in the same repository after the v0.0.88 tag.

2.1 Summary

Project name	Lachain consensus
URL	https://lachain.io/
Platform	Lachain
Language	C#

3. Overview

Consensus by itself is not responsible for final block processing and block addition to the blockchain. It serves for achieving an agreement in the list of transactions that will be worked on and will form the final block further. Roughly speaking, consensus ensures that the majority of nodes approves a proposed data for block creation.



The picture demonstrates functional layers interaction during consensus agreement achievement in HBBFT. A detailed layers description and their inner processes explanation can be found in the original Andrew Miller's work [2], article series by POA Network [5, 6], and Appendix B.

The audited folder has a neat and lean structure. Each subfolder name is conventional with whitepaper functional components and modules. The project and the consensus in particular are written in C# language. Observation of public HBBFT realisations revealed that it might be the first protocol implementation in this language. Various implementations are collected in the @initc3 Github [7], some of them have been audited [8].

It's worth to mention that several modifications were introduced into more of the HBBFT implementations compared to the original whitepaper protocol [2]. One can check the Issues section in the @amiller/HoneyBadgerBFT Github repo.

4. Structure of the audited code

Module	Description
BinaryAgreement	Implementation of the Binary Agreement module. This module is used for establishing an agreement on block inclusion. One instance of the module for each validator. When the agreement is established, the result is sent back to the CommonSubset module.
CommonCoin	Implementation of the Threshold sign module. It is used in consensus as a source of randomness (equal results for all validators) to determine the nonce of a new block and a tool for threshold signing that is used in the BinaryAgreement modules.
CommonSubset	Implementation of the CommonSubset module. This module is used as a collector for ReliableBroadcast (one for each validator) and BinaryAgreement (one for each validator) modules' outputs. As one of the ReliableBroadcast modules sends results, the output is sent to the BinaryAgreement module.At the moment, when there are enough outputs from ReliableBroadcast and BinaryAgreement modules, they are combined and sent as the result to the HoneyBadger module.
HoneyBadger	Implementation of the HoneyBadger module. In this module transactions from the node is encrypted (Threshold public-key encryption) and passed to the CommonSubset module.When the CommonSubset module passes back all encrypted blocks from accepted validators, the decryption phase is started. In this phase, all nodes are participating in decryption passing each other their decrypted shares. After collecting enough decrypted shares, all parts are decrypted and after that result is passed back to the RootProtocol module.
Messages	Structures that are needed for communication between internal parts of one node and with other nodes.

ReliableBroadcast	Implementation of the ReliableBroadcast module. For each validator, there is a separated instance of this module.If an instance of this module is assigned to a local node, this module receives an encrypted block from this node, and then using erasure coding splits this input into pieces (coding shards) that are broadcasted to all validators.If an instance of this module is assigned to a remote node, this module receives coding shards. When there is enough valid data collected from other validators, these coding shards are combined and the initial encoded block, that is assigned to this remote node, is interpolated. After, this interpolated data is returned as a result to the CommonSubset module.
RootProtocol	This module proceeds transactions in the blockchain to determine the list of validators, threshold public-key encryption parameters (global public key and private key shares for each validator), and threshold signature parameters (global public key and private key shares for each validator).
AbstractProtocol.cs	Base functionality for all modules. Contains everything the module needs to interact with other internal parts of the node, casting messages to other nodes and public keys information.
PrivateConsensusKeySet.cs	This file contains information about the structure that is used to store information about private keys:- Private key of the validator- Threshold public-key encryption private key- Threshold signature private key share
PublicConsensusKeySet.cs	This file contains information about the structure that is used to store information about public keys:- Threshold public-key encryption public key. The list of transactions of the nodes is encrypted by it.- A list of all threshold signature public keys of all nodes.- A list of all public keys of all nodes. They serve for identifying validators and to sign messages that are broadcasted between nodes.

Dynamic HoneyBadger and Queueing Honey Badger modules were out of the scope of the current audit.

5. Findings

Decrypted share validation problem

Location

Module: Lachain.Consensus/HoneyBadger/HoneyBadger.cs:L169

Description

Once CommonSubset returns the list of shares to HoneyBadger, the latter requests the decryption from other nodes. When the number of decrypted shares from different parties reaches f + 1 (where f is the number of byzantine nodes), the plain text of the initial message can be obtained with TPKE.Dec. The protocol authors assert in the whitepaper [2] that "if C contains invalid shares, then the invalid shares are identified". However, they wriggle out of specifying the exact encryption scheme in the standard that satisfies this requirement, but it's marked out they used threshold encryption proposed by Baek and Zheng [9] in their demo Python realization [10]. It turned out POA Network had stepped aside from using it in their implementation [11], favoring BLS12-381 [12], which was invented after the HBBFT presentation. Since the Lachain team adopted their approach, we imperatively advise to examine this elliptic curve on the ability to validate decrypted blocks that the node receives on the substitution. A similar issue is discussed at @amiller/HoneyBadgerBFT Github repo.

When the decryption phase in HoneyBadger is started, every valid node sends their decryption shares for every encrypted share.

```
private void HandleCommonSubset(ProtocolResult<CommonSubsetId,   ISet<EncryptedShare>>
result)
{
   (...)
   foreach (var share in result.Result)
   {
       var dec = _privateKey.Decrypt(share);
       _taken[share.Id] = true;
```

```
        _receivedShares[share.Id] = share;
        CheckDecryptedShares(share.Id);
        Broadcaster.Broadcast(CreateDecryptedMessage(dec));
    }
    (...)
}
```

At this moment the byzantine node could send a non-valid decryption message (message.Decrypted field in CreateDecryptedMessage(dec) in the snippet above). It is important to send this message fast and to be at the first f+1 values on other nodes. Otherwise, this vulnerability won't work.

When there are enough decrypted shares for some share (not less than f+1) then decryption of this share starts.

This function is called when some decrypted message come:

```
private void HandleDecryptedMessage(TPKEPartiallyDecryptedShareMessage msg)
{
    var share = Wallet.TpkePublicKey.Decode(msg);
    _decryptedShares[share.ShareId].Add(share);
    (...)
}
```

This function is called to start decryption for some share:

```
{
    (...)
    _shares[id] = Wallet.TpkePublicKey.FullDecrypt(_receivedShares[id]!,
_decryptedShares[id].ToList());
    (...)
}
```

This function is called in HoneyBadger.CheckDecryptedShares function:

```
public RawShare FullDecrypt(EncryptedShare share, List<PartiallyDecryptedShare> us)
{
    (...)
    var ys = new List<G1>();
    var xs = new List<Fr>();
    foreach (var part in us)
    {
       xs.Add(Fr.FromInt(part.DecryptorId + 1));
       ys.Add(part.Ui);
    }
    var u = MclBls12381.LagrangeInterpolate(xs.ToArray(), ys.ToArray());
    return new RawShare(Utils.XorWithHash(u, share.V), share.Id);
}
```

This Lagrange interpolation outputs a wrong value because variable ys contains a wrong value for some xs. Because of this, all decryption output will be wrong because this process is tied to this interpolation. Therefore one node can break the whole decryption process.

Update

The fixed realisation contains an extra decrypted share verification stage presented after L143 if-clause in HoneyBadger.cs. The verification has been achieved with the use of threshold and pair-based cryptography principles on the BLS12-381 elliptic curve. Since the approach requires a separate verification public key for each participant, additional modifications to key generation module and node storage scheme were needed. The task was also fraught with difficulties in maintaining compatibility with existing chain history. Intended consensus behaviour for invalid decrypted shares was evidenced in integration tests.

Possible DoS by a byzantine validator

Location

Module: Lachain.Core/Consensus/EraBroadcaster.cs:L143

Description

When the node processes external messages from other nodes, it checks whether a module that corresponds to this message exists. In case there is no corresponding module, the node creates one and starts a new thread for it. Therefore one node can create an unlimited amount of threads on other nodes and this may lead to a Denial of Service (DoS) problem.

There are an unlimited amount of ids for BinaryBroadcast, CommonCoin, ReliableBroadcast, and BinaryAgreement modules. The mechanism of changing the group of validators (i.e. DoS prevention) was out of the scope of the current study.

```
public void Dispatch(ConsensusMessage message, int from)
{
   (...)
   switch (message.PayloadCase)
   {
      case ConsensusMessage.PayloadOneofCase.Bval:
         var idBval = new BinaryBroadcastId(message.Validator.Era, message.Bval.Agreement,
message.Bval.Epoch);
         EnsureProtocol(idBval)?.ReceiveMessage(new MessageEnvelope(message, from));
         break;
      case (...)
      case (...)
      case ConsensusMessage.PayloadOneofCase.SignedHeaderMessage:
         var rootId = new RootProtocolId(message.Validator.Era);
         EnsureProtocol(rootId)?.ReceiveMessage(new MessageEnvelope(message, from));
         break;
      default:
         throw new InvalidOperationException($"Unknown message type {message}");
   }
}
```

Update

The Lachain team agreed with the described concern and modernized the Dispatch function inside the EraBroadcaster class. The updated version includes a message validation for each external request type, whereas inner requests pass without checks as they are produced inside a node and believed not to have malicious objectives. Also, now all messages addressed to unexisting protocol id are postponed and put in a queue for further usage if/when an appropriate protocol is created, in contrast to instant new module initialization.

```
private void HandleExternalMessage(IProtocolIdentifier protocolId, MessageEnvelope
message)
{
    if (ValidateProtocolId(protocolId))
    {
        if (message.External)
        {
            var protocol = GetProtocolById(protocolId);
            if (protocol is null)
            {
                lock (_postponedMessages)
                {
                    _postponedMessages
                        .PutIfAbsent(protocolId, new List<MessageEnvelope>())
                        .Add(message);
                }
            }
            else protocol.ReceiveMessage(message);
        }
        else Logger.LogWarning("Internal message should not be here");
    }
    else
    {
        var from = message.ValidatorIndex;
        Logger.LogWarning($"Invalid protocol id {protocolId} from validator
{GetPublicKeyById(from)!.ToHex()} ({from})");
    }
}

private bool ValidateProtocolId(IProtocolIdentifier id)
{
    switch (id)
    {
        case BinaryBroadcastId bbId:
            return ValidateBinaryBroadcastId(bbId);
        case CoinId coinId:
            return ValidateCoinId(coinId);
        case ReliableBroadcastId rbcId:
        // need to validate the sender id only
            return ValidateSenderId((long) rbcId.SenderId);
        case BinaryAgreementId baId:
        // created only by internal request, external messages never reach
BinaryAgreementId
        // so no need to validate
            return true;
        case CommonSubsetId acsId:
        // created only by internal request, external messages never reach
BinaryAgreementId
        // so no need to validate
            return true;
        case HoneyBadgerId hbId:
        // only has era in the fields
            ValidateId(hbId);
            return true;
        case RootProtocolId rootId:
        // only has era in the fields
            ValidateId(rootId);
            return true;
        default:
            return false;
    }
}
```

Modified common coin algorithm

Location

Module: Lachain.Consensus/BinaryAgreement/BinaryAgreement.cs:L131-137

Description

Coin toss sequence is defined as '*false, true, and the returned value of CommonCoin*'. HBBFT whitepaper describes an algorithm without "forced" common coin values, later implementations mostly use a '*true, false, and returned value of CommonCoin*' pattern. See more at @amiller/HoneyBadgerBFT Github repo.

We expect no significant consequences of this modification.

```
    if ((_currentEpoch / 2) % 3 == 2)
    {
        Broadcaster.InternalRequest(new ProtocolRequest<CoinId, object?>(Id, coinId,
null));
    }
    else
    {
        _coins[_currentEpoch] = ((_currentEpoch / 2) % 3) != 0;
    }
```

6. Conclusion

In this report, we present the results of the audit of the Lachain protocol consensus made upon their request. The Lachain team chose for their network Honey Badger Byzantine Fault Tolerance consensus. BFT consensuses are green (sustainable) consensuses: energy consumption of BFT consensuses is significantly lower than Proof of Work ones because they do not entail a computation puzzle. HoneyBadger consensus is one of the modern implementations of BFT consensuses with their definitive feature, compared to the rest of the BFT consensuses, being asynchronous, i.e. independent from the network timing assumptions. Synchronous and weakly synchronous of consensuses mean that a consensus can be reached only if the network performs as expected. The HB BFT consensus can work independently from the network timing parameters and therefore may work on a larger range of networks (e.g. TOR network with big latencies). The development of the network and consensus by the Lachain team took 2 years. At this time, the 2 most popular realizations of the HB BFT consensus are Andrew Miller's version written in python and POA network's version in Rust. As far as we know, the current version is the first one written in one of the most popular coding languages C#.

We've investigated the consensus algorithm and focused mainly on the consensus sabotage possibilities. Since the Lachain team advertises their consensus as an HBBFT realization, we've referenced A. Miller's paper [2], considering also the already discovered issues in HBBFT repo and other model and production implementations [7].

As a result of our inspection, we've introduced several concerns that may threaten consensus result convergence and a separate node service availability. By substituting a share with an invalid one during the decryption phase a perpetrator could freeze new blocks on adding to the blockchain, halting all transitions for an indefinite period of time until a byzantine node is detected and removed. Also, the lack of external message validation made nodes potentially vulnerable to DoS attack by malicious nodes since addressing to existing module led to its instant initialization.

In the provided update, the Lachain team acknowledged the issues and added fixes for

'Decrypted share validation' and Possible DoS by a byzantine validator' problems. Although the implementation of the coin algorithm preserves a slight discrepancy to the original paper, we have doubts regarding the adverse effects of the modification, as the consensus performance and properties are not compromised. All things considered, we ensure compliance of the audited consensus implementation with the specifications of the HBBFT whitepaper.

7. References

1. https://lachain.io/
2. https://eprint.iacr.org/2016/199.pdf
3. M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In OSDI, volume 99, pages 173–186, 1999.
4. C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. In Advances in Cryptology – Crypto 2001, pages 524–541. Springer, 2001.
5. https://medium.com/poa-network/4b16c0f1ff94
6. https://medium.com/poa-network/c953afa4d926
7. https://github.com/initc3/HoneyBadgerBFT/
8. https://github.com/poanetwork/wiki/tree/master/assets/pdf
9. J. Baek and Y. Zheng. Simple and efficient threshold cryptosystem from the gap diffie-hellman group. In Global Telecommunications Conference, 2003. GLOBECOM'03. IEEE, volume 3, pages 1491–1495. IEEE, 2003
10. https://github.com/amiller/HoneyBadgerBFT
11. https://github.com/poanetwork/hbbft
12. https://electriccoin.co/blog/new-snark-curve/
13. https://docs.rs/hbbft/latest/hbbft/

Appendix A. List of examined vectors

- Randomness sources
- Behavior on invalid input
- Dependencies' safety
- Threshold cryptography
- Documentation compliance
- Replay of messages
- Potentially weak parameters
- General software bugs
- Concurrency issues, deadlocks

Appendix B. HBBFT subprotocols [13]

Honey Badger

The nodes repeatedly input contributions (any user-defined type) and output a sequence of batches. The batches have sequential numbers (epochs) and contain one contribution from at least N - f nodes. The sequence and contents of the batches will be the same in all nodes.

Dynamic Honey Badger

A modified Honey Badger where validators can dynamically add and remove others to/from the network. In addition to the transactions, they can input Add and Remove requests. The output batches contain information about validator changes.

Queueing Honey Badger

A modified Dynamic Honey Badger that has a built-in transaction queue. The nodes input any number of transactions, and output a sequence of batches. Each batch contains a set of transactions that were input by the nodes, and usually multiple transactions from each node.

Common Subset

Each node inputs one item. The output is a set of at least N - f nodes' IDs, together with their items, and will be the same in every correct node.

This is the main building block of Honey Badger: In each epoch, every node proposes a number of transactions. Using the Common Subset protocol, they agree on at least N - f of those proposals. The batch contains the union of these sets of transactions.

Reliable Broadcast

One node, the proposer, inputs an item, and every node receives that item as an output. Even if the proposer is faulty it is guaranteed that either none of the correct nodes output anything, or all of them have the same output.

This is used in Subset to send each node's proposal to the other nodes.

Binary Agreement

Each node inputs a binary value: true or false. As output, either all correct nodes receive true or all correct nodes receive false. The output is guaranteed to be a value that was input by at least one correct node.

This is used in Subset to decide whether each node's proposal should be included in the subset or not.

Common Coin

Each node inputs () to broadcast signature shares. Once f + 1 nodes have input, all nodes receive a valid signature. The outcome cannot be known by the adversary before at least one correct node has provided input, and can be used as a source of pseudorandomness.

Threshold Decrypt

Each node inputs the same ciphertext, encrypted to the public master key. Once f + 1 validators have received input, all nodes output the decrypted data.

Synchronous Key Generation

The participating nodes collaboratively generate a key set for threshold cryptography, such that each node learns its own secret key share, as well as everyone's public key share and the public master key. No single trusted dealer is involved and no node ever learns the secret master key or another node's secret key share.

Unlike the other algorithms, this one is not asynchronous: All nodes must handle the same messages, in the same order.

contact@hashex.org

@hashex_manager

blog.hashex.org

linkedin

github

twitter

HashEx
BLOCKCHAIN SECURITY