HIVE:
A Blockchain Based Group Messaging Service

Christian Hwa
Massachusetts Institute of Technology
chwa@mit.edu

Kevin Yue
Massachusetts Institute of Technology
kevinyue@mit.edu

Abstract

With the advent of a privacy focused digital age, more and more messaging services are providing end to end encrypted applications. However, nearly all popular messaging platforms are run by centralized organizations that not only provides a singular point of vulnerability but also may not have their interest aligned with their users. We introduce HIVE, a blockchain based group messaging service, that aims to mitigate those issues. HIVE is a completely decentralized messaging platform, thus there is no centralized target for malicious users to steal or alter data. The blockchain based nature of the HIVE also ensure the integrity of the data. By having three different points of verification of data integrity including the Ethereum blockchain, HIVE provides the guarantee that a message sent across a network will not be altered. In fact, HIVE also ensures the confidentiality of data by encrypting all information being sent across the network. Ultimately, HIVE provides a messaging platform where data integrity is guaranteed, users cannot equivocate messages, and messages sent across the network cannot be read by malicious users.

1 Introduction

Today, an increasing population is realizing the importance of digital privacy. Most of our communication today occurs online through various different messaging and social media platforms. In many cases however, this communication is not encrypted and stored on some company's servers, meaning that a security compromise could easily lead to the exposure of sensitive message contents. Furthermore, because one individual company/organization controls these centralized servers, individuals that use their communication services must trust them to not use their information and messages, for benign or malicious purposes.

This revelation for some individuals has led to the recent rise of messaging applications that provide end-to-end encryption. With end-to-end encryption, the provider of the communication system is not able to decrypt any messages sent/received, even if they are stored on a centralized server. Nonetheless, even with end-to-end encryption, privacy issues remain. An eavesdropper could attempt to impersonate a message recipient, so that messages are sent to their public key instead of the intended target - a technique known as a man-in-the-middle attack. Furthermore, there is still the question of whether one using a centralized messaging application should or can trust the entity that owns it. Especially if the application code is not open source, many things could be going on behind closed doors that users are oblivious to. Finally, most messengers also require some form of identifying information to sign up, a phone number for example, which can be tied to a multitude of other personal information about the user.

In response to these various concerns regarding centralized messaging applications, we propose HIVE, a decentralized messaging platform built on top of the three pillars that make up the Ethereum network: the blockchain, Whisper [1] (peer-to-peer communication), and Swarm [2] (decentralized file storage). In the following sections, we will show how our system will be able to support confidential and authenticated group messaging.

2 Project Scope

Our goal is to provide a high level system design for a blockchain based group messaging service. As a result, we make several assumptions in the design. We assume that the network is operating in a steady state, meaning that we have sufficient nodes to provide varied routes as well as ensuring that the probability of all nodes on any given path is malicious would be negligible.

3 Prior Work

3.1 Signal

Currently, one of the most used end to end encrypted group messaging services is Signal. Signal provides end to end encrypted communication between any two parties using their application. As a result, the group messaging mechanic that signal provides is central to the product they provide to users. Regardless of the intended use case for group messaging, it is an inherently social construct that requires

Signal provides group messaging functionality through the notion of private groups: Signal does not retain membership lists and group names. On a high level, group messaging is significantly more complicated than direct messaging because of the additional overhead needed to maintain the necessary security properties desired. [3]. Most existing messaging services store the group state and messages as plaintext on their servers. While this simplifies things for users using the service, this presents a singular point of weakness for adversarial users to leverage. Adversarial users can easily break into the servers that store group information and extract all information regarding group message history and state on the entire network. To combat this glaring shortcoming, Signal designed their group message paradigm with the notion that a group of n-individuals consists of n one on one conversations with each individual in the group. That way, they can leverage their existing security guarantees with one on one communications to the group level.

Specifically, Signal approaches group messaging by concentanting messages with a 128-bit secret that represents the Group ID. In essence, clients never reveal which messages are group messages and which messages are private messages. Instead, clients tell each other what they need to know to determine the destination of the message. However, this messaging system does have shortcomings. One of the primary issues is the existence of race conditions as well as key dissemination and verification. If two users in the same group try to update the group state at the same time, for example adding somebody, changing the group description, or kicking somebody, a race condition occurs as the network cannot determinstically order the messages- resulting in an inconsistent group state amongst all users. To address the issue of verification and key dissemination, Signal uses MAC-based key-verification anonymous credentials (KVAC) [4]. Anonymous credentials were created to solve the issue of the service needing to authenticate the group records corresponding to the user making the request. However, the main issue with such anonymous credential systems is that the latency. Signal tackles this issue by improving upon the KVAC system: group members are issued credentials based on their unique identification (UID), and thus can prove to the service they are a member by providing the authentication credential as an encrypted group membership entry. In doing so, the user does not reveal their UID or any other information they wish not to be leaked. However, Signal's KVAC scheme hinges on the centralized nature of the key issuer and verifier.

The effectiveness of Signal group messaging paradigm is obvious in the widespread use of the application. As a result, we designed HIVE using Signal's group messaging architecture as a starting point, improving upon issues and security flaws that we wish to guarantee to the user.

3.2 Blockchain

Blockchain technology is central to our system. Specifically, the peer-to-peer network that manages the related blockchain allows us to decentralize our system, which means our users don't have to place any trust in us like they would have to with a centralized service like Signal. Furthermore, with our decentralized system, we won't be susceptible to downtime as a result of servers going down, and we won't be susceptible to any country looking to censor our application - e.g., China blocking Telegram.

Finally, the blockchain also allows for what might be the most important feature of our system: auditing. With an immutable public ledger of all transactions, it's impossible for someone to claim that they didn't say something, as it'll be on the chain. With our system, we look to place the hash of any message on a block that is going onto the blockchain, so for auditing purposes, all that's necessary is to compare the hash of the purported message with the relevant hash on the blockchain. Additionally, because each block is marked with a timestamp, the timing of messages and when they were sent are also immutable, which greatly benefits our goal of supporting auditing.

3.3 Whisper Protocol

Whisper protocol is a peer to peer (P2P) communication protocol designed for decentralized applications that is built on top of the ethereum network. Whisper provides privacy-preserving routing and messaging when passing whisper messages, envelopes, across the network. Because of the distributed nature of the system, envelopes are gossiped across the network As a result, any node or user on the network can receive messages; thus, the security relies on the ability for only a receiver to decrypt the message. Furthermore, denial of service attack robustness is provided by the proof of work algorithm that supports the message passing between nodes.

Whisper envelopes contain different payloads which determine whether they are chat messages or group state

message updates. The payload of a specific message is defined by a metadata header which contains flags for different envelope configurations, called topics. The main topics used by the whisper protocol are: partitioned topics, contact code topic, negotiated topic, and negotiated topics. Contact code topics are envelopes that begin communication between two parties. For example, if user 1 wanted to chat with user 2, user 1 would first send a contact code message to user 2. Partitioned topics is a construct that ensures information is not being leaked when sent across the network. Because we rely on Signal's group messaging approach of sending a private message to each user in the group, it becomes very easy to detect when two users are in conversation with each other. Partitioned topics ensure that multiple topics can be sent per envelope, thus balancing the efficiency and privacy. Finally, negotiated topics are topics where the receiver must listen to the topic. In general, for user 1 to send a message to user 2, they must adhere to the following flow:

- User 1 must wait for User 2's client code topic
- User 1 then send a message on User 2's partitioned topic
- User 1 can then receive messages from user 2

One thing to note is that Whisper automatically encrypts each message with asymmetric encryption, ensuring that messages that get sent to the transport layer are fully encrypted.

3.4 Swarm: Decentralized Storage

As mentioned in the introduction, our system is supported by the three pillars that make up the Ethereum network. In this section we outline the Ethereum Swarm system and how it allows for decentralized storage.

Th Swarm system is necessary in our system to transport large pieces of data, like videos or images. Currently, Whisper is only able to handle messages that are less than 64k bytes in size, which means that for anything larger, we'll need to use Swarm. We'll see shortly how the Swarm network works to supply decentralized storage, but at a high level, for large messages, instead of sending the actual message across Whisper nodes, we need only send a Swarm hash that will allow us to retrieve that data from the Swarm network later.

At a fundamental level, Ethereum Swarm is a system of peer-to-peer networked nodes that cooperate to create a decentralized service system. The keyword is "cooperate" where it is very important that all nodes in the system are incentivized to work together. First, we'll explain how Swarm stores data across its network, then we'll explain how Swarm's incentive system keeps the network running.

When a file is uploaded to the Swarm network from some node, that file is broken down into much smaller pieces of data called chunks. These chunks have a fixed maximum size of 4KB, and each chunk is individually hashed, where that hash corresponds to the address of some other node in the Swarm network. In this manner, Ethereum Swarm effectively implements a "distributed content-addressed chunk store", where content addressing means that the address of any chunk is deterministically calculated from its content. The addressing hash function takes in a chunk as input and outputs a 32-byte long hash, where clients can use that hash to retrieve the chunk.

Of course, this process of storing chunks at addresses is only possible because nodes in the network dedicate personal resources - diskspace, memory, CPU - to store and serve chunks. When a node wants to retrieve data at some address, it posts a request to the network with an address of the chunk. That request gets forwarded throughout the system until it reaches the desired address. Because the Swarm network is implemented as a Kademlia network, the topology of its graph ensures the existence of a path, as well as a maximum number of forwarding hops required, that is logarithmic in network size.

Additionally, Swarm's incentive system allows it to be self-sustaining, where all nodes in the network want to cooperate. The incentive layer of the network uses deposit-based storage incentives and allows trading resources for payment. The figure below shows a retrieval process in action.

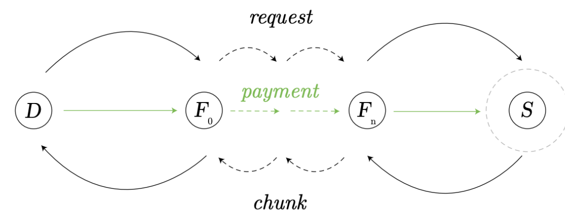

Figure 1: This figure demonstrates the retrieval process. In this example, node D sends a chunk retrieval request that gets forwarded through the network towards the chunk's address at node S. When the request gets to node S, the chunk is sent back through all the forwarding nodes towards node D. Whenever a node receives the chunk, a payment event is triggered, where the receiving node pays the node it received the chunk from. [2]

Uploading a chunk through the network follows a similar forwarding and payment process.

Finally, when a user wants to retrieve the entire contents of a file, a Swarm hash is used. A Swarm hash is the means by which chunks can be combined to represent a larger set, like a complete file. Essentially, chunks are arranged in

a Merkle tree, where leaf nodes correspond to chunks of consecutive input data from the original file, and intermediate nodes correspond to chunks which are themselves composed of chunk references for all their children. At the very top of the Merkle tree is the single root Swarm hash node, which again can be used to retrieve the complete contents of a file.

Ethereum Swarm's implementation of a distributed content-addressed chunk store and incentive system allow it to run as a self-sustaining decentralized storage system that we will utilize in our own system.

4 Privacy Properties

The following section will outline the privacy properties that we wish to guarantee for any user on our network. We aim to provide the following guarantees to our users: lack of equivocation, encrypted communication, and an indisputable history of chat messages.

We ensure that users cannot equivocate messages by signing each message with a signature. Each message sent to the group is signed by a MAC, thus we asssume that if the message can be decrypted and read, the origin of the message is from a user in the group. Next, each message is also signed with the sender's key, this allows us to attribute the sender of each message by virtue of the encryption key. We ensure that all communication passed over the network will be securely encrypted using the KEM-tree data structure. KEM-trees allow us to efficientlydistribute and manage keys in the group.

Finally, we ensure that messages sent on the network will be stored securely and will be essentially immutable as message history can be verified by the three independent components which make our network: the blockchain, a sidechain, and distributed storage. The interconnected nature of the three components allow for HIVE to identify anomalies in the network efficiently.

5 System Design

5.1 System Overview

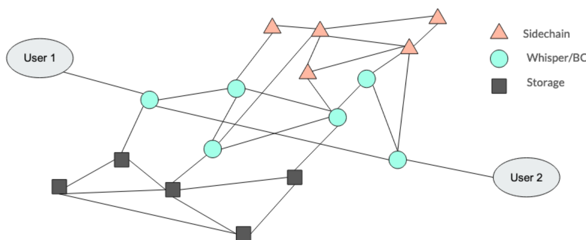

Figure 2: A high level diagram of HIVE. We can clearly see the three components that make up our network: Swarm distributed storage, sidechain nodes, and finally whisper protocol and ethereum blockchain nodes

Our system is built upon three different nodes and four different protocols. The key nodes that make the backbone of our system are the Whisper and Ethereum protocol; nodes in the HIVE network must support both protocols. We do this to ensure that whenever a message is sent over the system, at least one node will update the blockchain. Each Swarm distributed storage node is connected to at least one other HIVE node. That way, the storage system can verify what it stores with the Ethereum blockchain component. Finally, we have sidechain nodes that are also connected to HIVE nodes. Much like how the Swarm nodes can verify their contents with the blockchain, sidechain nodes can also verify their content with the mainchain. The sidechain stores a digest of the message history. The three components allow the HIVE network to quicly identify anomalies between message history. If any one component's data changes, it can be resolved with the other two components. Figure 2 above highlights the components of the diagram as well as any connections that exist between the components of the network.

5.2 Users

5.2.1 Account Creation

When a user creates an account, they are given a public/private key pair on the Ethereum blockchain, where their public key will be their username. However, because these keys can be extremely long, complicated, and hard to remember, we also provide the ability to use the Ethereum Name Service (ENS) to represent these long hashes with much more readable names. Nonetheless, because a user's account isn't associated with any personal information - phone number, IP address, bank account, etc. - we are able to maintain user privacy.

5.2.2 User Actions

1. Adding contacts: A user can add contacts by learning their public address or associated ENS name.

2. Send messages: Once a user learns another user's address/ENS name, they can begin a conversation with them.

3. Create groups: A user can also create groups on HIVE. A much more in-depth analysis into how HIVE accomplishes will be shown in the subsequent subsections.

4. Add/remove member from a group: Along with creating groups, users can add and remove members from any group they are in.

5.3 KEM-Trees

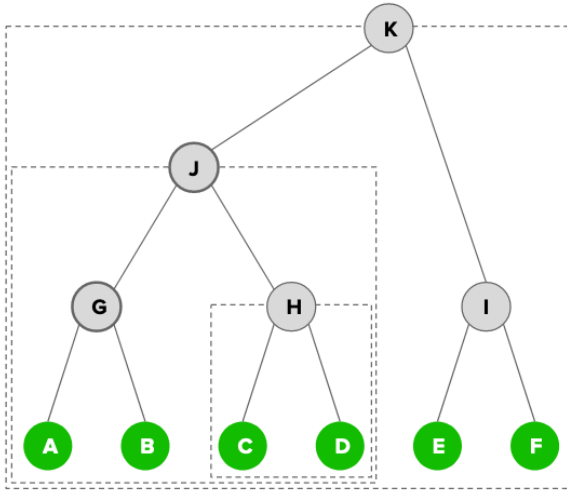

Figure 3: Illustration of a KEM tree. It is evident that leaf nodes only know secret keys for any parent nodes along the path to the root node of the tree

The KEM-tree data structure provides an efficient and quick way of managing group keys regardless of whether users are added, kicked, or when keys need to be refreshed. KEM trees represent a left balanced binary tree of asymmetric keys. From figure 3, we see that members of the group are represented as root nodes and know all secrets of any parent nodes in its path to the root node. We can formalize this as an invariant of the data structure: private keys for any node is only known by nodes in its subtree.

Because of this invariant, it gives us a simple representation for key sharing with group messages. Subtrees of nodes can represent subgroups within the groups, such as admins or regular users. Furthermore, and arguably the most important observation from the invariant is that the root node represents the entire group. That way, we can use the root node as our group key because all leaf nodes have paths to the root node. Thus, the root private key is known to everyone in the group at any given time.

All group level operations a user wishes to execute is reflected in the KEM tree as follows. Let's say $user_1$ is in $group_1$ and wants to add $user_2$ to the group. We first generate a new leaf node with a new key. We then hash up to the root node along the leaf-root path, updating any intermediate nodes if necessary. We then run an update step to confirm the changes on the tree. The update step entails generating a new leaf key with the associated user requesting the update, and then hashes up to the root node along the leaf-root path again. All paths affected by this change then have new key values associated with each node.

If $user_1$ wants to remove $user_2$ from the group, the leaf node associated with $user_2$ gets removed, and any secret keys along the leaf-root path get reset to 0 as well. This way, we ensure that the $user_2$ has no information of the new group key, as well as the inability to receive any new updates since they are not part of the tree. Next, the update steps described before is executed to derive a new group secret and update the KEM tree so that its structure will reflect the removal of $user_2$.

The tree structure of KEM-trees provides us asymptotic behavior that significantly outscales that of Signal. Currently if a user leaves a Signal private group, or if new keys need to be reissued it takes $O(n^2)$ time to execute. This gets prohibitively slow as n increases. However, with KEM-trees operations on the trees are bounded by $O(logn)$. Meaning that if we need to update keys for all n users in our group, it only takes $O(nlogn)$, providing significantly asymptotic behavior when compared to signal.

5.4 MLS

Creating our system on top of the Messaging Layer Security (MLS) protocol allows us to have efficient group messaging [5]. Whereas the Whisper protocol is great for peer-to-peer communication, the goal of the MLS protocol is to allow a group of clients to exchange private and authenticated messages. The MLS protocol takes advantage of the KEM trees - described in section 4.3 - to manage group private keys, as well as an Authenticated-Encryption with Associated-Data (AEAD) encryption scheme to encrypt and decrypt group messages.

The features that are necessary for effective group messaging are: group initialization, adding/removing members from a group, managing the private group key, and a method for members of group to communicate through private and authenticated messages. Let's go through these features one at a time.

5.4.1 Group Private Key

As mentioned in the KEM tree section, the root node of a tree comprised of group members is the group key that every member will have. Each member contributes to the KEM tree, and the group key, through their individual leaf nodes. Every time an operation is performed on the group (we call this an epoch), the group key will be updated as well. An operation is defined as adding a member, removing a member, or a member updating its leaf secret and key pair. Again, in each one of these instances, the group secret will be updated as well to protect from compromises.

5.4.2 Private and Authenticated Messages

Now that all members in a group have access to a shared group secret, messages among them can be encrypted and authenticated through AEAD. It is important to note that authentication in this case doesn't guarantee that a certain member sent a message; it only guarantees that a legitimate member in the group sent the message. In order to authenticate a message from a particular member, signatures are required.

After the initial group secret is generated, every member of the group creates their own sender Application Secret that will be used for its own sending chain. Group members must then only use that Application Secret once before monotonically incrementing the generation of the next secret. This is necessary to ensure Forward Secrecy. This means that if an attacker were to get a member's $n+1$th application secret, they would not be able to derive their nth application secret and associated AEAD key.

5.4.3 Group Initialization

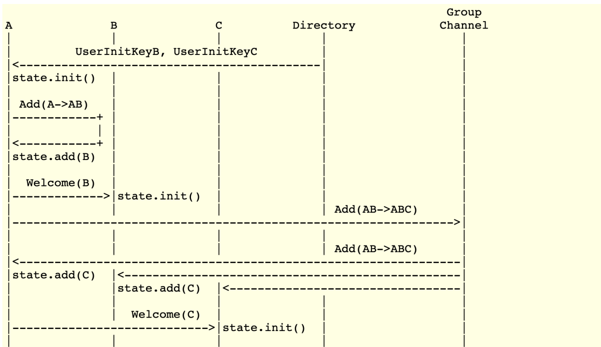

Figure 4: This figure describes the process of group creation in the MLS protocol. [5]

The process of group initialization is as follows: when a user joins HIVE, they are designated a set of initialization keys that are stored in a directory file - corresponding to the directory column in figure 4 - on the Swarm network. If user A wants to create a group with users B and C, he looks up their initialization keys (which are public knowledge) on the directory, then initiates the group initialization process. In the manner laid out in the figure, he sequentially adds B then C to the group. Each member in the group maintains a state of the group, where the state keeps track of the group's KEM tree, the current epoch, and additional information about the group is stored.

5.4.4 Adding/Removing Members to/from Group

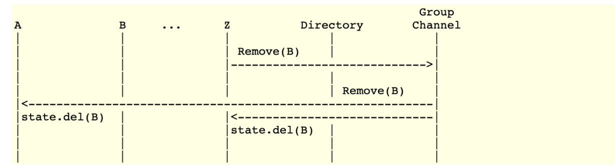

Figure 5: This figure describes the process of removing a member in the MLS protocol. [5]

The process of adding a member is very similar how member A adds user C in figure 4. How removing a member works in the MLS protocol is laid out in figure 5. In figure 5, user Z sends a message to group removing user B. Every new member in the group (past group without user B) then performs a delete operation, incrementing the group's epoch and modifying the group state.

5.5 Group Messaging

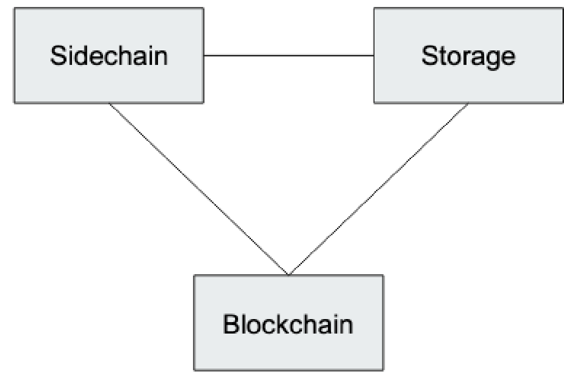

Figure 6: This figure highlights the relationship between the sidechain, distributed storage, and the Ethereum blockchain. This ensures that data is consistent between all three components

When a user sends a message across the HIVE network, it is first signed with the user's secret key, ensuring that the sender cannot equivocate. Users must use the negotiated MLS ciphersuite to AEAD encrypt and decrypt their ciphertext. Each message is signed as follows: group ID, group epoch, group generation, sender, and the ciphertext. Group epoc and generation is needed to determine the associated state the KEM-tree should be in. As shown in the MLS section above, group level operations update the KEM-tree, thus this ensures that any malicious user that was previously removed from the group cannot read any post-removal messages even if they know the group ID. However, a weakness of this encryption scheme still exists: malicious users can see the frequency of messages and payload size of each message being sent from one user to

another. While malicious users cannot decrypt any messages, information can still be gained by the frequency and size of the messages.

Once an envelope enters the HIVE network, it gets passed from one HIVE node to another until the intended recipient receives the message. At any point along the path, any HIVE node has the ability to update the storage with the new message. We use the Swarm storage network to store the chat history. Once the Swarm updates the associated chat history file by using the groupID, the new hash representing the hash of the updated chat history gets sent back to a HIVE node. Since our HIVE nodes also support the Ethereum protocol, the Ethereum main chain gets updated with the new hash. This represents the connection between storage and blockchain as seen in figure 6. At the same time, HIVE nodes are also connected to the sidechain. The sidechain essentially stores a digest of messages sent between groups. The side chain then occasionally checks in with the Ethereum blockchain. This constitutes the connection between the sidechain and the blockchain. If there is disagreement between the blockchain and sidechain, this means that either somebody changed the blockchain or a malicious user changed the contents on the distributed storage system. In either case, the sidechain acts as a canary and can alert users in the group that their privacy has been compromised. As we can see, the sidechain is connected to storage through the blockchain. If at any point the sidechain doesn't agree with the blockchain, Swarm can be used to determine where the compromise occurred. If the distributed storage system receives a message that isn't on the sidechain, users will also be alerted. This could mean one of two cases: either our HIVE nodes didn't post to the sidechain, in which one or more nodes may be compromised, or the group of users are lying about what messages have been sent. Either way, through the sidechain and blockchain, our system is robust enough to identify such discrepancies and notify users of the compromise.

Finally, if a user receives a message that isn't in storage, or the sidechain, then we know all HIVE nodes have been compromised. In this case, the network needs to be shutdown due to the possibility of all nodes being compromised. Furthermore, our group messaging system is also robust against malicious senders; however, the robustness hinges upon a simple assumption: the probability that all nodes along a Whisper envelope path are malicious is incredibly low. With this assumption, we assume that at least one node along the path will be a good node. Thus rendering any attack by a malicious node or sender moot since the message sent will be published on the distributed storage, sidechain, and blockchain as well- preventing users from obfuscating the origin of messages.

6 CIA Security Analysis

6.1 Confidentiality

Confidentiality refers to the protection of user data, ensuring that data can only be read by authorized parties. This security guarantee is the backbone behind HIVE. To do this, all messages across the network are encrypted end to end using asymmetric and symmetric key encryption. Furthermore, all data stored on HIVE, the sidechain, and the blockchain are all also encrypted.

We place a heavy emphasis on ensuring confidentiality is maintained throughout our network as highlight sensitive messages may be sent across our platform, for example messages regarding legal proceedings, or even business to business communication. Consequently, confidentiality is a critical aspect to the HIVE network.

6.2 Integrity

Integrity refers to a system's ability to ensure that the system and information is accurate and correct. The protocols and systems we employ: Whisper, Swarm, and MLS ensure integrity in their individual contributions to the HIVE system. Among the HIVE system itself, section 5.5 on group messaging and the relevant figure 6 that highlights the relationship between the Ethereum blockchain, distributed storage, and our sidechains show that HIVE ensures data consistency among all three components.

6.3 Availability

Finally, availability refers to a system's ability to ensure that services are available for users a vast majority of the time. First, the decentralized property of HIVE means that availability will not be impacted by any server failures or censorship. Additionally, with the Whisper protocol, the latency of messages will be lower than if we used purely the blockchain, where blocks are placed on the Ethereum blockchain every 13 seconds on average.

7 Future Work

While we have outlined a general framework for a distributed blockchain based messaging service, there is still much work to be done. Currently, malicious users can perform a man in the middle attack to get information regarding message frequency and message length. While they cannot decrypt the ciphertext, the length of the message can also reveal a lot of information. This can be addressed by padding messages to a specified length before sending it over the network.

Another focus of future work can also be improving

upon the Whisper protocol. Whisper provides a low-latency solution to peer to peer communication; however, it still relies on gossiping for the Whisper envelope to reach its intended target. As a result, it is inherently slower than direct communication. Optimizations can result in a more instantaneous messaging experience. Additionally, increasing the 64K byte message limit can also be another point of future work.

Finally, actually implementing this model will require a significant amount of work. The specifications for HIVE rely specifically on the steady state assumption, thus any security concerns regarding a young network with few nodes need to be explored. Furthermore, details regarding the encryption algorithm and ensuring that the implementation does not compromise the confidentiality guarantees of the network need to be carefully reviewed.

8 Conclusion

Overall, HIVE offers an anonymous decentralized group messaging solution to users who are looking for complete confidentiality without the need to trust any centralized service. Equally as important, HIVE guarantees data integrity, low latency on messages, and provides the opportunity for the auditing of messages. With the upcoming upgrades to the Ethereum ecosystem - Ethereum 2.0, the improved scalability, speed, and efficiency promise an exciting future for HIVE.

References

[1] Ethereum Foundation. *Whisper*. URL: https://eth.wiki/concepts/whisper/whisper.

[2] Viktor Trón. *The Book of Swarm*. 2020.

[3] Signal. *Group Chats*. URL: https://support.signal.org/hc/en-us/articles/360007319331-Group-chats.

[4] Melissa Chase, Trevor Perrin, and Greg Zaverucha. “The Signal Private Group System and Anonymous Credentials Supporting Efficient Verifiable Encryption”. In: (2020). DOI: https://eprint.iacr.org/2019/1416.pdf.

[5] Network Working Group. *The Messaging Layer Security (MLS) Protocol*. URL: https://tools.ietf.org/id/draft-ietf-mls-protocol-11.html.

[6] *Swarm Guide - 2. Architectural Overview*. URL: https://swarm-guide.readthedocs.io/en/latest/architecture.html#peer-management-hive-kademlia.

[7] Silas Lenz. “Evaluation of the Messaging Layer Security Protocol – A Performance and Usability Study”. In: (2020). DOI: https://liu.diva-portal.org/smash/get/diva2:1388449/FULLTEXT01.pdf.