SE-DHT: A Securely Encrypted Distributed Hash Table

Roy Shadmon
rshadmon@ucsc.edu
UC Santa Cruz

Karthik Balakrishna
kabalakr@ucsc.edu
UC Santa Cruz

Alex Williamson
alswilli@ucsc.edu
UC Santa Cruz

March 21, 2019

Abstract

Distributed Hash Tables (DHTs) provide a scalable, fault-tolerant infrastructure for users to efficiently store and retrieve files or data. Since the information being stored in the DHT is distributed among a network of other nodes who may or may not try to learn or alter the data, DHTs also need to be secure. Moreover, Chord is a well-recognized DHT implementation because it supports an efficient log-based lookup algorithm and O(log N) and it mitigates the amount of memory each node needs to provide to lookup the data. Additionally, DHTs are able to autonomously adapt and update what node is storing which data among a massive amount of nodes. Unfortunately, the vanilla version of Chord assumes that its nodes are completely honest and pose no threat(s), and therefore does not need any security features to protect users' data from malicious nodes within the network. In order to improve this aspect of Chord, we propose SE-DHT, a secure version of Chord that maintains all of its reputable features while also ensuring security principles. Our main contribution is the implementation of the Information Dispersal Algorithm (IDA) for encrypted and distributed data storage, as well as, an improved data insert and lookup technique that helps enforce our security guarantees. Our results show that the use of this technique allows for confidentiality, authentication, integrity, availability, and obliviousness at the negligible cost of a 10x lookup speed. In addition, we provide good load balance among nodes and that the data is highly available.

1 Introduction

Previously, we implemented a Distributed Hash Table that uses the Chord protocol to provide a scalable, peer-to-peer (P2P) lookup and storage service for any user-specified resource (data). Our project allowed for lookup that needed only $O(\log N)$ messages between nodes to successfully locate a key, where N is the total number of nodes in the network. In addition, each node only needed to store the information of $O(\log N)$ other nodes for efficient routing, with each node only needing to be responsible for relocating O(1/N) keys upon a node leaving the network. This additional feature allowed our DHT to operate efficiently under heavy churn.

One problem that we have with our initial implementation, however, is that there is no protection from an adversarial attack. For example, an arbitrary node A wishing to store their data on the network has no confidentiality, because the data stored at another node, say node B, is unencrypted. Node B is then able to know every detail about an inserted resource, including which node inserted the respective resource. In addition, a node storing the data can learn which resource is queried the most by monitoring which of its stored resources is most often queried; the more a resource is queried, the more valuable it is[1]. Moreover, our network has no authentication mechanism, allowing any node to potentially query for any resource (given the node knew the resource key)[2], and there is no guarantee on the integrity of another node's resource. Without proper mechanisms to handle these faults, the DHT is useless because no node would ever want to store real, valuable data in the network (which is its purpose). Lastly, only one node stores another node's resource. This limits the availability of a node's resource

[1]Data that is queried more is inherently more valuable than data that is not used.

[2]This would be easy because the nodes know the possible key ranges and could request each key in the range to see if there is any valuable resource to them being stored in the network.

when the node storing the resource goes offline, forcing the node requesting its resource to wait an unspecified amount of time before being able to retrieve their data.

In order to improve on these limitations of our previous work, we propose SE-DHT (Securely Encrypted Distributed Hash Table), which is a secure Chord-based DHT implementation. SE-DHT aims to guarantee confidentiality, integrity, authenticity, high-availability, and obliviousness of the resources being inserted and queried in the DHT, and achieves these guarantees through encryption, decryption, and ORAM techniques [1]. More specifically, we implemented an Information Dispersal Algorithm (IDA) in SE-DHT that encrypts a specified resource and partitions the resource into K partitions, where K is a predetermined number. Each partition is then assigned a key using a collision-resistant hash function and inserted at the node in charge of storing the respective partition, based on what the Chord protocol decides. When any given node later tries to find its resource, only that specific node is able to calculate all K keys of its resource since only the node inserting the resource knows the resource hash to calculate each of the K keys. If a node(s) were to fail, the node querying for its K partitions only needs to receive in return a threshold number of partitions. For example, if the number of partitions is 5 and the threshold number is 3, then any 3 out of the 5 total partitions are needed in order to recover the entire resource. This property ensures that the DHT is fault-tolerant given that a node is able to retrieve its resource if it receives at least the threshold number of partitions, where the threshold number is less than K total partitions. Lastly, the transactions in the DHT are oblivious because after every query for the resource, all partitions of that resource are deleted from the DHT, rehashed, and reinserted to the network. If an adversary were to monitor the access patterns it would therefore only able to learn two facts about the resource: it was inserted to the network, and it was queried once. After a resource is queried, the next inserted resource could either be the same or different resource.

Using the following techniques, we present our novel approach to a secure DHT that scales, is secure and oblivious, and has a legitimate use. The rest of the report will be structured as follows. In section 2, we discuss the related work in improving Chord, identify the imitations of the work, and describe how our implementation provides even greater security benefits with marginal performance loss. In section 3, we outline the potential threats to a DHT and define the threat model for our implementation. In section 4, we talk about the use case for SE-DHT. In section 5, we introduce SE-DHT. In section 6, we discuss SE-DHT's guarantees. In section 7, we discuss our encryption and hashing algorithms in detail. In section 8, we evaluate the lookup and load balancing performance of our protocol through two experiments. In section 9, we propose future work to our model and describe some of its limitations. Finally, in section 10 we conclude by summarizing our contributions and results.

2 Previous Work

The original DHT that used the Chord protocol (also known as vanilla Chord [2]) provided a single guarantee for routing requests: an efficient lookup time to find a resource of $O(\log N)$, where N is the total number of nodes in the network. However, vanilla Chord does not detail any security guarantees such as confidentiality, integrity, authenticity, availability, and obliviousness, placing the onus on the person using the DHT. Specifically in Vanilla Chord, a node inserts unencrypted data to the network, a node storing another node's data is able to modify that data, any node is able to query for any data in the network, and a node is not guaranteed to find its data if the node storing the data becomes offline (no fault-tolerance protection), and nodes are able to analyze the access patterns of each individual piece of data.

Thomas Wölfl from the University of Regensburg proposed a public-key based infrastructure to secure a P2P network that uses the Chord protocol. Their solution offers the following guarantees: load distribution, scalability, availability, authentication, and fault resistance. However, they come short compared to our implementation. For example, a DHT is inherently fault-resistant, but Wölfl's implementation is not fault-tolerant regarding being able to access a specific piece of data. If a specific node, that is storing another node's data, becomes offline, Wölfl's implementation will not be able to retrieve that data until the node becomes online again. Moreover, Wölfl's method for authenticating a node is stronger than our implementation, however, their method is significantly more computationally expensive than ours, and it requires a trusted-third party (such as a certificate authority) to verify a node's public key[3]. Lastly, our implementation matches Wölfl's regarding load distribution and scalability[4].

3 Threat Model

A distributed hash table (DHT) is a large P2P data structure that processes data lookups in an efficient manner. A DHT works by peers trusting other peers to store their data for them (possibly to backup their data). However, there is no way to ensure in Vanilla Chord that the entrusted peer would not read or modify the data if it were unencrypted. Moreover, even if the data were encrypted, it is still possible for the peer to figure out the decryption key, possibly through a man-in-the-middle attack, a known plaintext attack (if it's a symmetric key), or simply a brute force or dictionary attack. Additionally, a node storing the data can keep track of how many times a specific resource is queried. Every time a resource is queried some leakage occurs–specifically leaking the overall value of the data based on the total number of times it is queried. Lastly, given a known key range, any node can query for any specific resource–encrypted or not.

These threats allow for the adversary to be a node within or outside the network. Moreover, these attackers can either be classified as honest but curious or active attackers. For example, an honest but curious attacker could attempt to read another node's data or count the total number of times a resource is queried, but they can be trusted to always respond correctly and timely to another node querying for a resource that the honest but curious node is storing (in our implementation, this resource is the correct partition[5]). An active attacker, on the other hand, could attempt to read another node's data, possibly even be brazen enough to modify it, and they aren't guaranteed to return the correct resource or even a resource in general.

Overall, our threat model assumes that a majority of the nodes are honest but curious (they always respond to queries but could read a resource that doesn't belong to them) and that the minority are completely untrusted (they cannot be trusted to return the correct resource or respond to any query). To solve these threats, we use an information dispersal algorithm (discussed in Section 7.1), where we also must assume that each node is capable of decrypting, encrypting, and hashing their own data appropriately and securely.

4 DHT Use Case

There are many reasons to use a DHT, though some of the main reasons involve being able to distributively back up data or to have a distributed dictionary for a fast lookup. For example, a user who who doesn't trust their personal computer to store some data[6] could have another node in the network keep a copy of their data[7]. Moreover, on the organizational level, something like a hospital wanting to store their patient's data in a fault-tolerant data structure could use a DHT to efficiently query for their patient's medical records. Most use-cases that require efficient lookups are for important or sensitive data. Therefore, with SE-DHT, we allow the user to store any type of object into the network, while preserving the confidentiality, authenticity, integrity, availability, and obliviousness of their data.

5 SE-DHT

SE-DHT[8] is our novel secure DHT implementation that uses hashing and encryption to secure user's data. Moreover, SE-DHT uses serialization to convert any object into a byte stream, as well as, AES encryption and SHA-256 hashing techniques. Lastly, we made significant data-related modifications to our insert and lookup functions (particularly recovering and inserting partitions of a resource).

[3]A trusted-third party inherently centralizes the solution and, as a result, weakens the decentralization of the network.

[4]Unfortunately, we were not able to find Wölfl's code implementation to compare lookup run-times, however, we suspect that Wölfl's run-time is much greater because of the long verification time of public-key cryptography.

[5]See Section for more information.

[6]Possibly because their computer doesn't have enough memory or they want to backup their data in case their hard-drive becomes corrupt.

[7]If the data is sensitive, they could also encrypt the data before transmission.

[8]Our implementation is located https://github.com/royshadmon/Secure-Distributed-Hash-Table

5.1 Serialization

Serialization is a crucial first step in the implementation of SE-DHT. Not only is it required to enable the transmission of the object (more correctly its state) over the network, Java's serialization process also provides for a small degree of integrity verification, through the addition of a serialization header. In essence, Serialization is a simple method to convert any complex object into a simple Byte Stream. Our implementation leverages Serialization - Deserialization as a method to introduce integrity checks, and abstract away certain complicated transformation steps, required to truly make the DHT as generic as possible. Treating objects as byte streams in Java, gives us a quick (if not the best) way to do this. Integrity is automatically verified by the JVM, which checks the serialization header. If it is considered to be invalid (a sign of possible tampering), the object is simply not de-serialized. At its core, a serialized object can be stored in any way necessary, and that is simply an implementation detail. Our implementation of SE-DHT does not make use of any complex databases, or even file storage, though it would be easy to extend the DHT to achieve that.

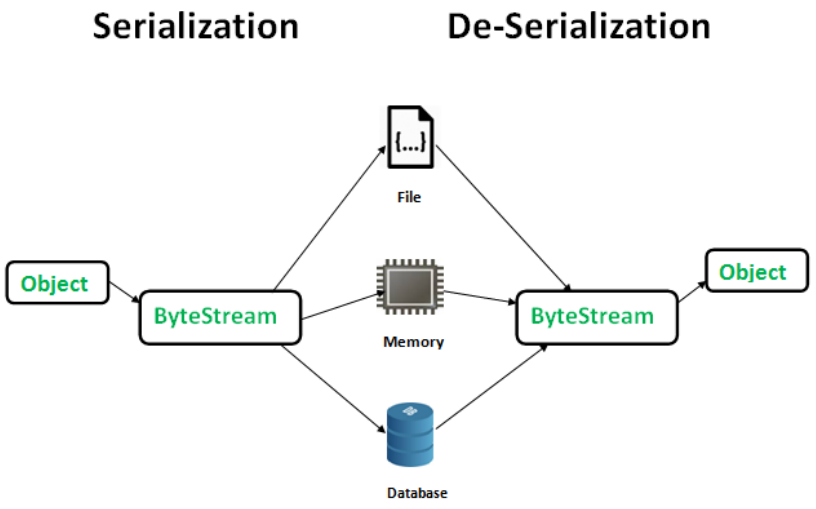


Figure 1: This figure shows the Serialization - Deserialization of a Java Object

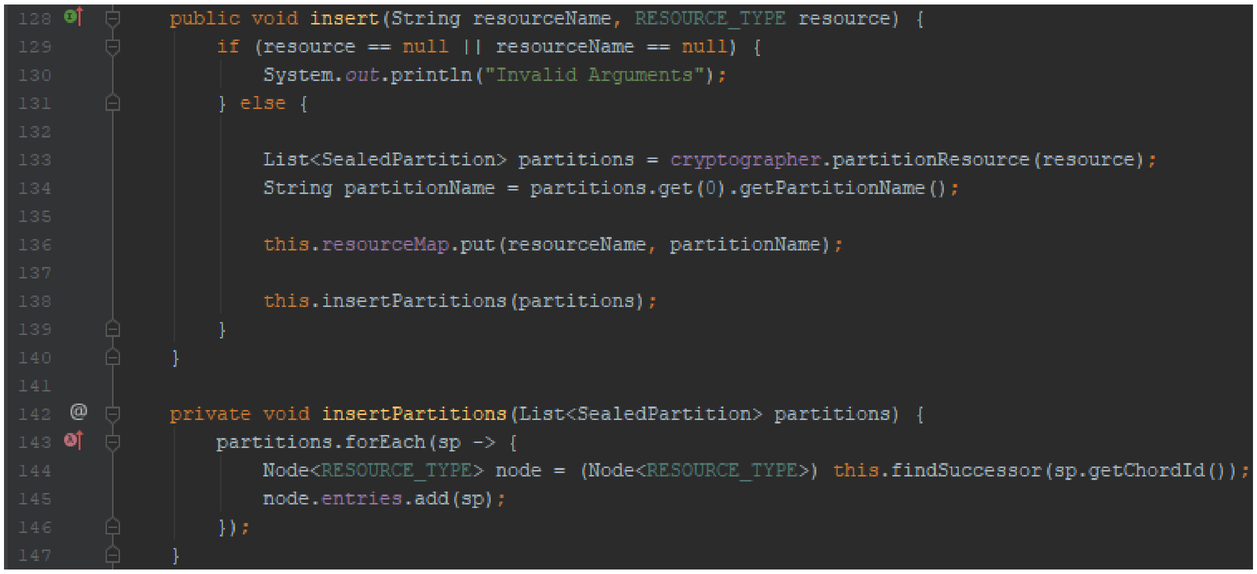
5.2 Insert Function

The insert() function, as seen in Figure 2 below, has been modified from its initial Chord implementation to support the insertion of all partitions for the resource. First, we use the cryptographer function partitionResource() on line 133 to return the properly partitioned resource via serialization, IDA partition encoding, and encryption. Next, we add the name[9] of the first partition to the resource map[10] of the inserting node. We only add the name of the first partition to the map because that is all that is needed to recover the all partitions later using chain-block hashing[11]. Finally, the partitions are inserted into the network via the same process as vanilla Chord, which involves finding and inserting the partitions at the successor of their respective chord ids[12].

5.3 Lookup Function

The find() function, as seen in Figure 3 below, has also been heavily modified from its original Chord implementation. Starting on line 32, we first look up the name of the first partition from the requesting node's resource

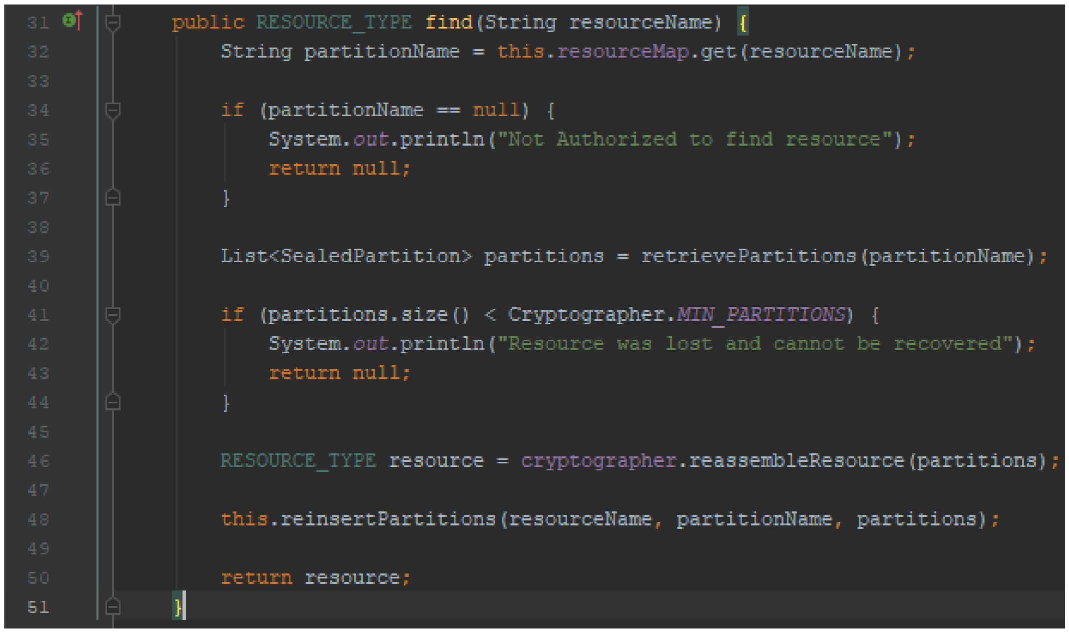
[9]The name of a partition is more specifically the hashed serialization of their resource or the previous partition name
[10]The resource map is a per node dictionary that maps the name of their resource to the partitions of that specific resource
[11]Chain block hashing is further discussed in Section 7.3
[12]Chord Ids are assigned via hashing the partition name and mapping to the id space of the chord ring



```java
public void insert(String resourceName, RESOURCE_TYPE resource) {
    if (resource == null || resourceName == null) {
        System.out.println("Invalid Arguments");
    } else {

        List<SealedPartition> partitions = cryptographer.partitionResource(resource);
        String partitionName = partitions.get(0).getPartitionName();

        this.resourceMap.put(resourceName, partitionName);

        this.insertPartitions(partitions);
    }
}

private void insertPartitions(List<SealedPartition> partitions) {
    partitions.forEach(sp -> {
        Node<RESOURCE_TYPE> node = (Node<RESOURCE_TYPE>) this.findSuccessor(sp.getChordId());
        node.entries.add(sp);
    });
}
```

Figure 2: This figure shows the code sequence for the insert function

map (which was inserted using the insert() function). Next, we retrieve the partitions from their respective nodes using the name of the first partition, with the details of this process described later in section 5.3.1. Once the partitions are gathered, we call the cryptographer function reassemblePartitions() to properly recover the final resource via decryption, IDA partition decoding, and deserialization. Finally, in order to preserve obliviousness, we reinsert the partitions to the DHT with different hashes that map to different nodes. The details of this process are described in section 5.3.2.

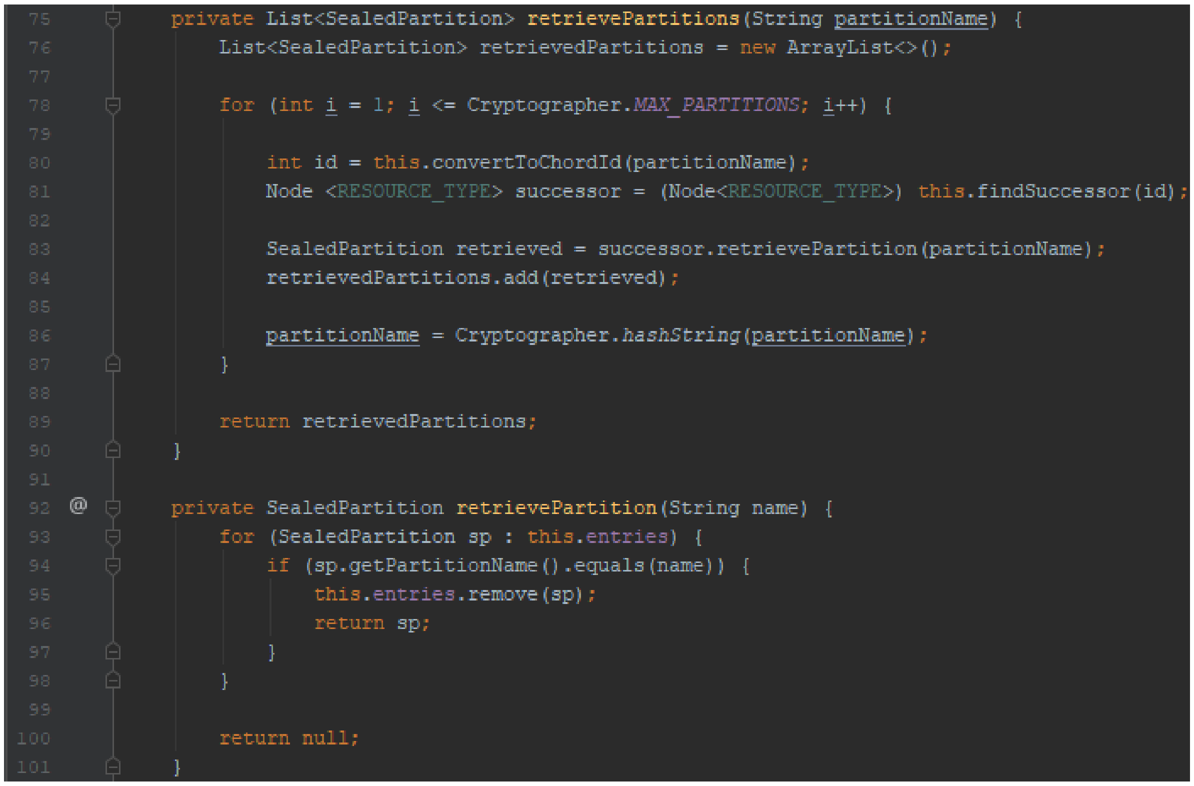


```java
public RESOURCE_TYPE find(String resourceName) {
    String partitionName = this.resourceMap.get(resourceName);

    if (partitionName == null) {
        System.out.println("Not Authorized to find resource");
        return null;
    }

    List<SealedPartition> partitions = retrievePartitions(partitionName);

    if (partitions.size() < Cryptographer.MIN_PARTITIONS) {
        System.out.println("Resource was lost and cannot be recovered");
        return null;
    }

    RESOURCE_TYPE resource = cryptographer.reassembleResource(partitions);

    this.reinsertPartitions(resourceName, partitionName, partitions);

    return resource;
}
```

Figure 3: This figure shows the code sequence for the lookup function

5.3.1 Recovering the Partitions

The retrievePartitions() function that is used during lookup can be seen in Figure 4. For every partition, we calculate its chord id using its partition name and then use Vanilla Chord's findSuccessor() function to locate the node which is currently storing the partition. We then retrieve the partition from the successor node by searching through all the entries at the node for the name of the partition and return the object associated with it[13]. During this process, we also remove the partition from the node to prepare for reinsertion of the partition. Once the partition is gathered, we use chain block hashing again to find the name of the next partition and repeat the loop.



```
75      private List<SealedPartition> retrievePartitions(String partitionName) {
76          List<SealedPartition> retrievedPartitions = new ArrayList<>();
77
78          for (int i = 1; i <= Cryptographer.MAX_PARTITIONS; i++) {
79
80              int id = this.convertToChordId(partitionName);
81              Node <RESOURCE_TYPE> successor = (Node<RESOURCE_TYPE>) this.findSuccessor(id);
82
83              SealedPartition retrieved = successor.retrievePartition(partitionName);
84              retrievedPartitions.add(retrieved);
85
86              partitionName = Cryptographer.hashString(partitionName);
87          }
88
89          return retrievedPartitions;
90      }
91
92  @   private SealedPartition retrievePartition(String name) {
93          for (SealedPartition sp : this.entries) {
94              if (sp.getPartitionName().equals(name)) {
95                  this.entries.remove(sp);
96                  return sp;
97              }
98          }
99
100         return null;
101     }
```

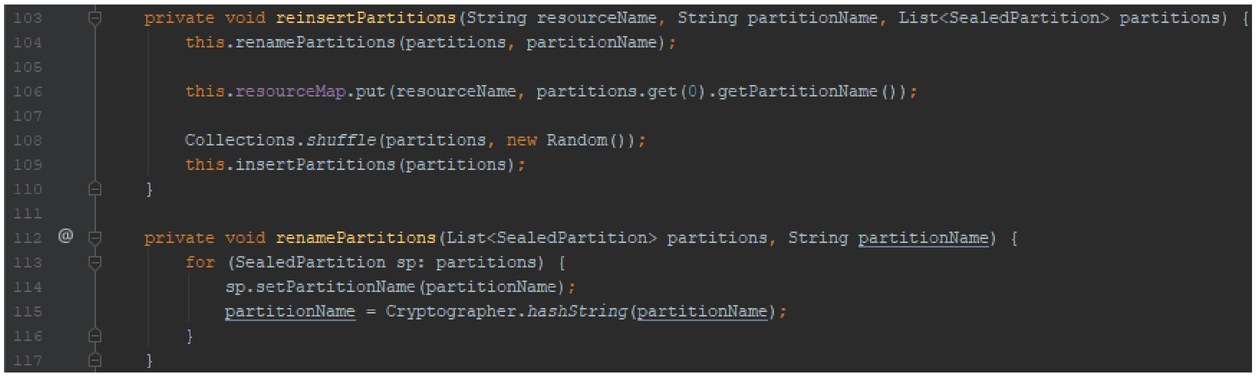
Figure 4: This figure shows the code sequence for retrieving the partitions during lookup using the name of the first partition via chain bock hashing

5.3.2 Reinsertion of the Partitions

The reinsertPartitions() function that is used during lookup can be seen in Figure 5. The first step of the function involves renaming the partitions so that they can be reinserted to different nodes in the DHT. We perform this step so that any frequency information about resource acquisition becomes limited[14]. To rename each partition, we simply continue the process of chain block hashing! Using the hash of the final partition name from the previous set of partition names, we set the name of the first partition to this final hash digest name, and the continue to hash this name to reset the names of all other partitions. The resource map also gets updated to reflect the updated name of the first partition, and lastly the ordering of the partitions are shuffled upon reinsertion to the DHT so that the ordering of the reinsertions does not spoil the identity of the partitions as well.

[13]Note that if the node is adversarial, it may hide the partition name from its entries. In this case, we return null, as shown on line 100

[14]We discuss this in more detail in section 6.5



```java
    private void reinsertPartitions(String resourceName, String partitionName, List<SealedPartition> partitions) {
        this.renamePartitions(partitions, partitionName);

        this.resourceMap.put(resourceName, partitions.get(0).getPartitionName());

        Collections.shuffle(partitions, new Random());
        this.insertPartitions(partitions);
    }

@   private void renamePartitions(List<SealedPartition> partitions, String partitionName) {
        for (SealedPartition sp: partitions) {
            sp.setPartitionName(partitionName);
            partitionName = Cryptographer.hashString(partitionName);
        }
    }
```

Figure 5: This figure shows the code sequence for reinserting the partitions with different partition names to preserve obliviousness after lookup

6 SE-DHT Guarantees

In this section, we will discuss in more detail how our DHT implementation provides the following guarantees: confidentiality, authenticity, integrity, availability, and obliviousness.

6.1 Confidentiality

We provide confidentiality by encrypting the data using the Advanced Encryption Standard (AES) technique. Specifically, for each partition created using the Information Dispersal Algorithm (IDA)[15], we seal each partition with a layer of AES encryption before they are assigned an ID and stored throughout the DHT network. Our encryption and IDA techniques provides confidentiality for two reasons. First, only the owner of the resource is able query for all the partitions of its respective resource because only the owner is able to reproduce each partition key for a specific resource (which acts as the "password"). This is because the partition keys can only be reproduced using chain-block hashing[16] using SHA-256[17], which can only be done by the owner of the resource. More importantly, since SHA-256 is considered a secure hashing algorithm in practice, we believe it is currently implausible to efficiently break our encryption with the current available computer power.

Our confidentiality is illustrated in the following way: only the owner of the resource is able to reproduce other partition keys by hashing the first partition using SHA-256 to produce the second partition key, hashing the second partition using SHA-256 to produce the third partition key, and so on. Secondly, since only the owner of the resource can produce the keys to query all the required partitions of a respective resource, only the owner is able retrieve all of the partitions. Therefore, no other nodes in the network storing the partitions are able to produce the keys to decrypt any of the partitions, nor do they know which partitions are needed to successfully recover a resource.

6.2 Authenticity

We provide authenticity by using the SHA-256 hash function. Specifically, a user only needs to store the hash of its first partition to figure out which node is responsible for storing its partition's key. Since SHA-256 has no proven collisions, only the user who owns the resource would be able to query for its resources' partitions. In addition, for any other user U' to query for a resource's x^{th} partition, the user U' would need to know the $x-1$, $x-2$, ..., first partition. This technique is called chain-block hashing (see Section 7.3 for details why we use that

[15]Discussed more in Section 7.1
[16]We describe chain-block hashing in Section 7.3
[17]See Section 7.2 for more information about SHA-256

technique). Therefore, since it is (currently) computationally NP-hard[18] to figure out the first hash, only the owner of the resource would be able to query for that respective resource. This property is especially sound since each resource could have X total partitions, and would require at least Y partitions to reassemble the partitions, where $Y \leq X$ and is at least equal to or greater than the required threshold ($Y \geq$ threshold. Therefore, since only the owner would be able to generate each resources' partitions key, only the owner will be able to retrieve its resource and know the AES decryption key to unseal each partition.

6.3 Integrity

We mainly provide integrity of the resource through serialization, but also via hashing, encrypting, and partitioning of the resource. The serialization process, which is implemented by Java, trivially prepends a serialization header that is used to check if there have been any manipulation in the resource. Since each node only stores a fraction of an encrypted resource, the node won't be able to modify the resource. Moreover, if a node were to modify the bits of the encrypted partition, the owner of the resource would be able to detect that the bits were modified because it wouldn't be able to use that partition to reassemble the resource via serialization as the header would change, as well[19]. Even though a malicious node modified the bits of a single partition of a resource, the user owning the resource would still be able to retrieve its resource because not all partitions (fewer than the total number of partitions for a given resource) are needed to reassemble the entirety of the resource.

6.4 Availability

Our network is fault-tolerant, and thus provides availability, because even if one or more nodes fail or act maliciously, the user is still able to reassemble its resource given that at least the threshold number of partitions are stored at honest nodes. Therefore, as long as this requirement holds, we are able to guarantee a user that they will always be able to recover their resource from the network. For example, if a node were to insert a resource and it becomes partitioned into 5 pieces, as long as 3 of the partitions are returned when a lookup request is made (with the other two being lost from misbehaving or absent nodes) the node is still able to recover its resource. The three partitions returned in this case can be any of the partitions, due to IDA, and we assume malicious nodes to resemble only a small fraction of the whole network. Therefore, as the network grows in size, to resemble a real-world situation, the likelihood of a node not being able to retrieve the resource it requests is extremely low. Additionally, SE-DHT provides excellent load balancing properties (see section 8.2 for more details about load balance), so it is ensured that the partitions are evenly spread across the network, thus limiting the chance that an adversarial node affects the query of a resource.

6.5 Obliviousness

We provide obliviousness through the rehashing of resource partition names after every access to a stored resource. With every call to the find() function, the DHT first reassembles the resource for the requesting node and then begins the process of inserting the partitions to different, random nodes. Reinsertion is carried about by first renaming the partitions by rehashing their names sequentially using SHA-256. The hash of one resource name is used as the input for the hash of the other, just like when the partitions were first inserted into the network. Next, the resource map stores a new head partition for the resource, allowing for a corrected lookup upon reinsertion. Finally, the partitions are inserted into the network to nodes corresponding to their newly hashed names. This process ensures obliviousness because an attacker can no longer monitor the frequency of resource access in the DHT. Once a resource is requested, the attacker cannot logically create an association to that resource since it will be stored randomly in the network (possibly to the same place or not and under different hash values). In addition, the attacker is not able to infer any information about each partition because they are all of the same size (thus indifferential). Therefore, the attacker is not able to learn anything about the contents of the stored resource nor its access pattern, because it can only know at most that a resource was queried at most once.

[18]NP-hard problems whose computational complexity can only be solved in non-deterministic polynomial time. For example, to determine the input of a SHA-256 hash function given just its output is a NP-hard problem.

[19]Future work here would be to add an incentive layer discuss in section 9.1

7 Encryption & Hash Functions

7.1 Information Dispersal Algorithm

One key building block of SE-DHT is the use of the Information Dispersal Algorithm [3]. IDA works by creating a set of n chunks that reflect a specific object, file, or item, where any m (where $m \leq n$) chunks randomly selected from the respective set are required to reproduce the object, file, or item. In our case, we first ensure data integrity and the support of any type of resource by serializing the resource before being split into these chunks. Next, the chunks are encoded into their recoverable partitions using the IDA algorithm one at a time, with the inverse operation of the algorithm being performed when the chunks become decoded.

The main quality IDA ensures when being used in our DHT scenario is that we don't need to put all of the trust on a single node to return the object in a timely manner. By implementing IDA into the network, we are able to recover from faulty and malicious nodes not responding to queries or returning incorrect or modified partitions when queried. As long as at least m of n total partitions are recoverable, then we can guarantee that a user will be able to retrieve their resource. Moreover, our insert algorithm tries to load balance the partitions so that no one node has to do more work than the other nodes and if a specific node fails, it only affects a small minority of the resources.

7.2 SHA-256

SHA-256 is a secure, cryptographic and deterministic hashing algorithm used to map any sized input to a 256 bit hashed output. The definition of a deterministic, cryptographic hashing algorithm is to map one unique input to one unique output. Moreover, given the same input, the hashing algorithm will output the same n-bit hash. In the case of SHA-256 then, we get a 256-bit hash which, given its output, is cryptographically NP-hard to produce the input. In addition, SHA-256 is considered collision resistant in industry, however, in theory it is not. Although it is not considered collision resistant in theory, there have been no documented cases of a collision occurring when using SHA-256 in practice. This fact is in contrast with other common hashing functions, such as SHA-1 and MD5, which both have been proven to produce collisions with relative ease. Therefore, we use SHA-256 as a crucial building-block to authenticate nodes querying for their resource through knowing the 256 bit hashed resource.

7.3 Chain-Block Hashing

A key aspect of this implementation is the fact that the resource is split into partitions. Crucially, it is only possible to retrieve a partition using a key that is associated with this specified partition. Since every resource is reconstructed by re-assembling its associated partitions, two possible ways to manage these keys arise. In the first method, we can simply store every partition key associated with a given resource at the node that is responsible for it. Then we simply need to lookup all partitions based on their key. The drawback associated with this method, stems from the fact that the Node is unable to leverage the redundancy, and ends up storing keys that it may never use. Alternatively, we chose an approach where the node only associates the resource key with the first partition's key, and all other partition keys are derived from this key via hashing. This significantly improves storage requirements while simplifying the storage process. Instead of relying on a complicated data structure that maps one resource key to multiple partition keys, we now simply generate all other partition keys from the first one by successively hashing them until we reach the threshold specified by the system.

Consider the case when all resources as split into N partitions, of which only M are required. In the first design, each node would have to store all N partition keys, even if only M of them are required. In the second design, a Node only stores 1 partition key, and at most M partition keys are generated via chain-block hashing.

8 Evaluation

For our evaluation of SE-DHT, we ran a few tests to measure the lookup performance of SE-DHT compared to vanilla Chord for a single resource, and the results are promising[20]. We also measured the load balance of SE-DHT on each node when inserting 2,751 different resources. All tests were run on a MacBook Pro with an Intel i7 CPU and 16GB Ram.

8.1 Lookup

In Figure 6, we plot the lookup time for a single resource when that resource is serialized into five, four, three, two, and one partition(s), as well as the lookup time for a single resource in Vanilla Chord[21] (which we also implemented). Rather than plotting large numbers (nanoseconds), we took the log of the recorded time to make the graph easier to read. For example, a difference from 22 to 21 is a $2x$ (or 100%) speedup.

We found that our encryption and authentication techniques inflicted a 10x loss to lookup performance speed when compared to Vanilla Chord. Although the overall performance degraded, we believe that the 10x degradation is an affordable cost because SE-DHT adds significant security guarantees. Moreover, we think that this amount of slowdown would be an acceptable cost to users who care for their data to be highly available, confidential, and maintain integrity. More importantly, we don't think that this slowdown would even be noticeable to the users as it would be extremely difficult for a user to notice the difference from a 0.004 second lookup (SE-DHT with 5 partitions) vs. a 0.0001 second lookup. Overall, for the additional, significant security guarantees SE-DHT is worth the 10x performance slowdown.

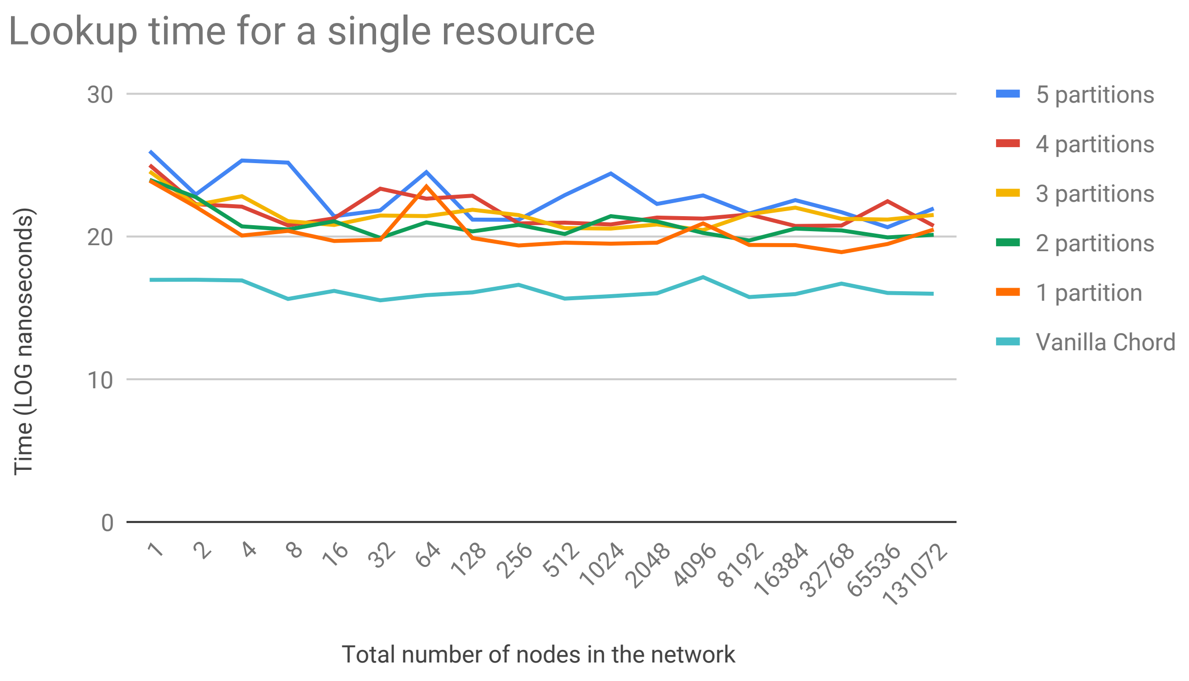


Figure 6: This figure shows the log time it takes in nanoseconds (ns) to lookup a single resource that is split into 5, 4, 3, 2, and 1 partitions. In addition, this figure shows the lookup time for a single resource in Vanilla Chord.

[20] We would like to acknowledge that the testing was done on a single device running SE-DHT. The lookup time in practice would be the time we present plus delta (network latency). Presumably, since delta would be constant for each node.

[21] Our implementation is located https://github.com/royshadmon/Distributed-Hash-Table

8.2 Load Balance

Figure 7 shows the load balance when inserting 2,751 resources into our SE-DHT network. The point of this test is to show how truly distributed the data is within the network. In this test, we plot the network's load balance when a resource is split into 5 to 1 partitions. The total number of nodes resembles the total number of nodes who are storing at least 1 partition.

We found that our load balancing of partitions is relatively linear to the total number of partitions each resource is split into (as expected). It is important for the load balance to be done well because good load balance ensures that users' data is highly available. To reiterate, this graph shows that if nodes do fail, SE-DHT will still likely be able to return enough partitions to the user so that they could reassemble their resource.

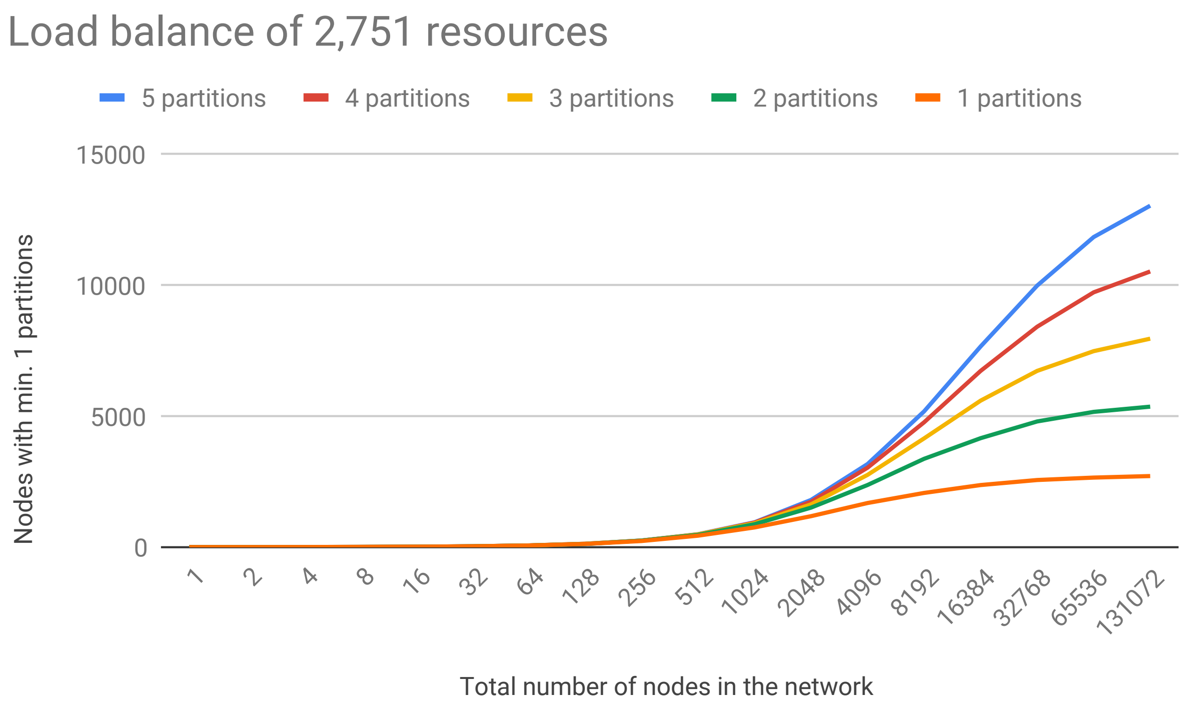


Figure 7: This figure shows the total number of nodes that are storing at least 1 partition given 2,751 resources are inserted to the network. In addition, we increase the total number of nodes to reflect how our networks behaves in scale.

9 Future Work

There are many paths to further improve SE-DHT, such as including an incentive layer and having users be able to specify the total partition's per resource and the threshold to reassemble the resource.

9.1 Incentive Layer

A big reason why nodes might act selfishly[22] is because nodes do not have any incentive to respond timely. In addition, nodes may choose to act maliciously when they please because there aren't any repercussions. An idea we

[22]Nodes responding to queries when they please

have is to add a financial incentive layer to SE-DHT using Smart Contracts that run on a decentralized trusted-third party such as the Ethereum blockchain. By using Smart Contracts, nodes who receive invalid partitions can appeal to the blockchain and verifiably prove that say node A returned the incorrect or a modified partition (verification using digital signatures). On the other hand, implementing Smart Contracts could be fairly expensive because every transaction to a Smart Contract costs a fee. To get around excessively paying fees, nodes could construct state channels[23] that would fall back to a Smart Contract in the case that a node believes another node acted maliciously.

The incentive mechanism would work as the following. Nodes participating in the network must deposit some money to discourage malicious actions. Nodes who query for their resource must pay a small fee to each node that responds and transmits back the correct partition. If a node were to not respond after a given amount of time or if a node returns an invalid partition, then that node would be punished some amount of money from their deposit, which would be sent to the node requesting the partition. Nodes would sign messages that provide the promise and guarantee that they would receive payment. Only when a node wants to leave the network would they make an on-chain transaction to withdraw their funds and any money they earned from responding to another node's query. Doing so would minimize the amount of money nodes would have to pay as transaction fees.

The reason to still use a SE-DHT rather than storing data on the actual blockchain is because storing encrypted data on the blockchain is extremely expensive and too public since data is inherently unencrypted and viewable to anyone. If the node does the encryption/decryption, partitioning, and reassembling locally, the node would therefore minimize the amount of interaction they would have with the blockchain (ultimately minimizing the amount of transaction fees it would have to pay).

9.2 Node Specification

One feature that we did not get the chance to approach is adding the functionality for a node to specify how many partitions it would like its resource to be split into and the minimum number of partitions it needs in return to reassemble the resource. Currently, all resources are split into 5 partitions and the minimum number of partitions required to reassemble the resource is 3 partitions. By giving a node some flexibility, it allows them to control the amount of security and availability [24] they want to have when inserting their resource, at the cost of lookup speed and potentially more faulty storage of partitions at malicious nodes.

9.3 Limitations & Short Comings

We found some bugs with our IDA implementation, where some combinations of total partitions to threshold does not work. In the future, we would like to improve our IDA implementation so it works with more combinations of total partitions to threshold, as this would provide more flexibility and security guarantees (on a node to node basis).

10 Conclusion

DHTs are great for file storage and retrieval over a network, but are unusable in a real-world scenario without proper security guarantees. With SE-DHT, we provide a DHT that has minor performance degradation compared to Vanilla Chord while significantly improving and ensuring security guarantees: authenticity, confidentiality, integrity, obliviousness, and high-availability. Our DHT scales well, as well. We plan to improve SE-DHT further by prevention of misbehaving nodes through an incentives layer, as well as few bug fixes and customizability options for nodes when partitioning their data.

[23] State Channels allow for users to sign verifiable messages and transmit partitions verifiably without actually having to interact with the blockchain. Doing so would only require nodes and users to pay transaction fees when they believe the other party acted maliciously.

[24] More partitions means more data to gather before fully uncovering a resource

References

[1] Y. Jia, T. Moataz, S. Tople, and P. Saxena, “Oblivp2p: An oblivious peer-to-peer content sharing system,” in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pp. 945–962, 2016.

[2] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, “Chord: a scalable peer-to-peer lookup protocol for internet applications,” *IEEE/ACM Transactions on Networking (TON)*, vol. 11, no. 1, pp. 17–32, 2003.

[3] M. O. Rabin, “Efficient dispersal of information for security, load balancing, and fault tolerance,” *Journal of the ACM (JACM)*, vol. 36, no. 2, pp. 335–348, 1989.