P2Pass: A Distributed Password Storage System Using Peer-to-Peer Lookup with Chord

Brandon Bae
Duke University

Havish Malladi
Duke University

Edison Ooi
Duke University

Stuart Tsao
Duke University

Abstract

P2Pass is a novel distributed password storage system that uses the Chord framework to enhance the security of password storage. P2Pass incorporates two main components to ensure enhanced security - shredding, hashing, and salting the passwords and inserting different segments into different Chord nodes. The system is built using a Main Server which interacts with a Chord Wrapper, an API-like system to interface with the Chord network. The Chord Wrapper also connects with a DynamoDB table, which is used to store the passwords. The goal of P2Pass is to introduce a new system for password storage, mitigating the risks of password leaks. The results support that our system, when a hacker only breaks a part of the Chord network, is more secure than if a hacker purely brute forces a password in a centralized system. In this paper, we will dive into the implementation of our system and the analyze its effectiveness versus current procedures.

1 Introduction

In October of 2022, Meta Platforms informed its users that around 1 million Facebook passwords could potentially have gotten leaked [9]. Likewise, in June of 2012, nearly 6.5 million LinkedIn passwords were leaked in a major data breach [8]. Security leaks have now become the norm instead of an irregularity. The central problem with both of these password leaks is that these services, no matter how secure, have a central password system. If a hacker can manage to get into the system, they can brute force the hashes of the passwords in the system and gain access to millions of accounts.

Our goal with this paper is to provide an alternative solution to this problem. Instead of storing passwords in a centralized manner, we will use a decentralized, distributed, peer-2-peer lookup system with Chord to store our passwords. Our distributed systems exist, such as Blockchain, but in this paper, we explore how Chord can be utilized to build a secure system.

This paper proposes the use of the Chord distributed hash table (DHT) to store shredded and hashed password fragments. The primary objective was to evaluate the security and performance of the proposed method compared to traditional password storage techniques.

Using an existing Chord implementation, in this paper, we build a main server that shreds, hashes, and salts the passwords and interacts with our Chord network, a wrapper for the Chord network that facilitates this conversation and helps with the placement of information into our storage system, and a DynamoDB Wrapper that helps store our data and mimics and network of storage units.

Our hypothesis was that by shredding then storing passwords in a distributed peer-to-peer storage system, it would become harder for attackers to not only leak passwords from our system but also crack them as well. However, we predicted that this will come at the cost of higher password lookup times due to the peer-to-peer nature of our storage system.

The proposed system was implemented and evaluated based on the following metrics: storage and retrieval efficiency, system scalability, fault tolerance, and security against known attacks. The results were analyzed to determine the effectiveness of the proposed approach and its suitability for practical use cases.

This project aimed to address the ongoing challenge of secure password storage in distributed systems, where traditional methods have been prone to attacks and vulnerabilities. The proposed approach leveraged the benefits of Chord DHT and secret sharing schemes to provide better security and performance, making it a potential solution for a wide range of use cases.

The main contribution of this project was the implementation and evaluation of a novel password storage method that addresses the shortcomings of traditional methods. The results of the analysis provide valuable insights into the strengths and limitations of the proposed approach and inform future research in this area.

2 Related Work

Current best practices for secure, server-side password storage require servers to salt and hash passwords before storing them in a database [6]. In the event of an attack or a data leak, the salt and hash prevent malicious actors from directly obtaining user passwords for a given service. Scarfone and Souppaya [5] discuss salting as a means of ensuring different hashes for identical passwords. They also examine how salting can increase general password strength, as salts often make passwords longer and diversify their character sets, leading to even more diverse hashes. Gauravaram investigates how attack complexity must grow exponentially as length of hash values increases, suggesting that opting for longer hash values is favorable [3].

However, even with these precautions it is possible for malicious actors to crack hashed passwords. This can be seen in the 2012 LinkedIn hack where half of the 6.5 million leaked hashed passwords were cracked in 72 hours [8]. However, we believe that security can be improved by distributing a password over multiple nodes such that a leak of a single node does not reveal the entire password.

The inspiration for our paper stems from Shamir's paper on password management. Breaking away from the traditional methodology, Shamir suggests a method where data is broken up into n pieces, and any k of those pieces can be used to reconstruct the entire data. However, with $k-1$ pieces, no information about the data can be determined, creating a secure and robust system [1]. Fujiwara *et al.* describe flaws in Shamir's paper, as his scenario assumes that data transmission and authentication must be perfectly secure, a concept that is difficult to implement [2]. Their paper suggests another strategy for password storage, where they use a quantum distribution network to allow for secure transmission [2].

In order to facilitate the distributed storage of our password, we will be using an implementation of Chord as our storage system. Chord is a look up protocol that efficiently looks up nodes in a peer-to-peer distributed storage system [4]. We chose Chord as our storage system due to its peer to peer nature and efficient look up capabilities. The peer-to-peer nature of system allows the nodes storing our passwords to constantly change as nodes join and leave the system making it harder for attackers to identify specific nodes to compromise. Additionally, the efficient look up characteristic of Chord allows us to locate and reconstruct stored password in an efficient manner.

We provide a novel approach to the password management problem, drawing together components from these papers while also providing a unique perspective. Our strategy is to implement a peer-to-peer system, implemented using Chord, that stores our n pieces. With this framework, we utilize the security concepts described by Shamir [1] while also addressing the security concerns mentioned by Fujiwara *et al* [2].

3 System Design

At a high level our design has three main components: the Main Server, the Chord peer to peer storage, and the DynamoDB table. The Main Server acts as the interface for new password storage and password verification. It also handles the splitting and hashing of passwords before storage. The Chord peer-2-peer implementation handles the location of hashes and efficient lookup of previously stored hashes. The DynamoDB table acts as a simulation of the servers that each Chord node would be attached to. For simplicity reasons, we used one table to mimic this entire service.

3.1 Main Server

The Main Server is the interface through which users can create new password entries and verify password attempts. It handles the splitting, salting, and hashing of password segments. New password entries are created through the `createPasswordEntry(username, password)` operation. Verification of password attempts for use cases such as authenticating login attempts is accomplished with the `verifyPassword(username, password)` operation. Sections 3.1.1 and 3.1.2 describe their implementations.

3.1.1 `createPasswordEntry(username, password)`

1. Receive new account's (username, password) pair from user
2. Split password into N equal length parts. N is an arbitrary constant that is consistent throughout the Main Server
3. Assign random salt to each of the N parts and concatenate salt to end of part
4. Compute SHA3-256 hash of each part + salt combination
5. Insert all N hashes into chord with a key of format `{username}-hash-{i}`
6. Insert all N salts into chord with a key of format `{username}-salt-{i}`

3.1.2 `verifyPassword(username, password)`

1. Receive (username, password) pair to be validated
2. Query chord to get all N correct password hashes and associated salts
3. Split attempted password into N parts using same split rule that was used at creation time, and salt with salts retrieved from Chord
4. Compute SHA3-256 hash of each part + salt combination from attempted password
5. Concatenate all attempted hashes, concatenate all stored hashes, and compare for equality. Return true if both concatenations are equal, else return false

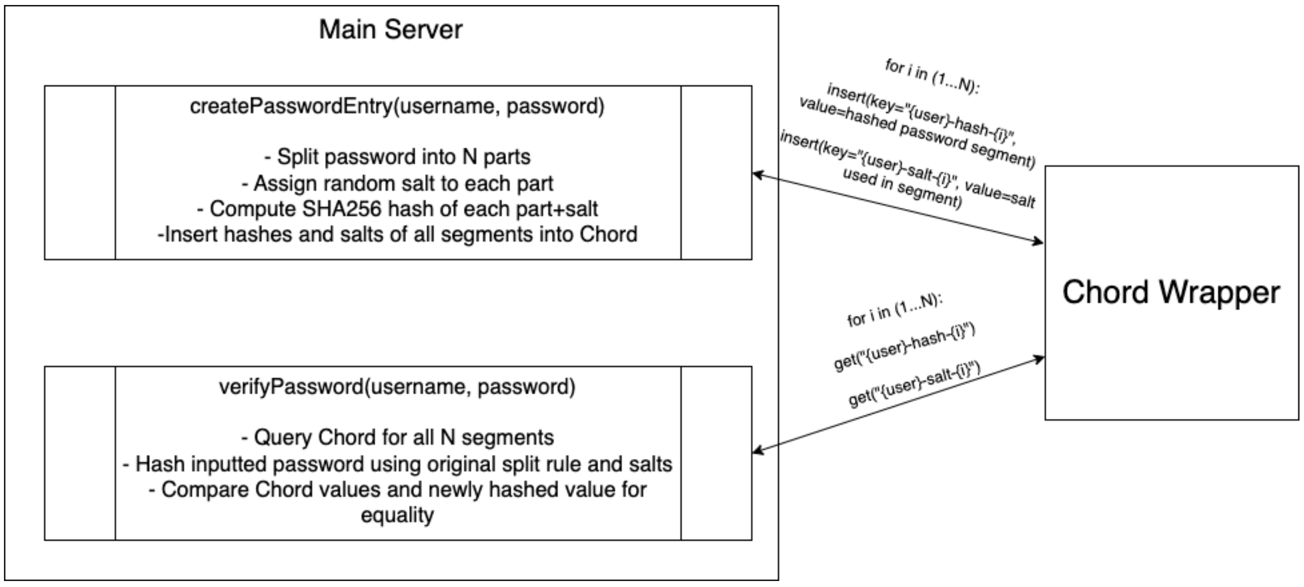
Main Server
createPasswordEntry(username, password)
- Split password into N parts
- Assign random salt to each part
- Compute SHA256 hash of each part+salt
-Insert hashes and salts of all segments into Chord
verifyPassword(username, password)
- Query Chord for all N segments
- Hash inputted password using original split rule and salts
- Compare Chord values and newly hashed value for equality
for i in (1...N):
insert(key="{user}-hash-{i}", value=hashed password segment)
insert(key="{user}-salt-{i}", value=salt used in segment)
for i in (1...N):
get("{user}-hash-{i}")
get("{user}-salt-{i}")
Chord Wrapper

Figure 1: High Level Workflow for Password Storage and Password Verification

3.2 Chord Storage

Our system relies on a Chord implementation which handles the math and lookup of machines, along with a Chord Wrapper which will handle the actual transfer of data with machines in the Chord network.

3.2.1 Chord Implementation

Our Chord implementation is built off of a simple Java implementation of Chord [10] created by Duke graduate Chuan Xia. In this implementation, the server provides the IP address that a user can query into, allowing for us to find the segments for the passwords. The implementation includes a 32-entry finger table in each node, which enables each node to maintain information about its successors and predecessors, allowing for efficient routing and lookup operations.

One key advantage of this implementation is its ability to support concurrent node joining and leaving, which is achieved by implementing the stabilization part outlined in the Chord paper. The system is designed to converge to the correct state even when dealing with nodes that fail or leave voluntarily. Every node constantly communicates with its successor and updates its finger table accordingly, ensuring that the system remains stable even when nodes enter or exit the network.

Overall, the Chord implementation by Xia provides a reliable and efficient framework for our password storage system, enabling us to efficiently search for password segments and maintain the stability of the network even when nodes enter or exit. However, it is important to note that this implementation only supports node management. The data storage system of individual nodes is not specified or implemented.

Furthermore, the hashing capability of the implementation is also limited. To address these limitations, we implemented a key-value data storage system and enhanced the hashing capabilities of the implementation. These modifications enabled us to store and retrieve password segments efficiently and securely, making our system more reliable and robust.

3.2.2 Chord Wrapper

As stated previously, the Chord implementation we built upon simply provides IP addresses of machines that should store or receive certain data based off ID's. Thus we must also implement a wrapper that handles the actual transferring and receiving of data from machines in chord. The purpose of the wrapper is two tasks:

- A consistent hash that maps a key to a Chord node ID
- Communicate with storage nodes to insert/get key-value pairs

The Chord Wrapper has two main endpoints: `insert(key, value)` and `get(key)`

- **`insert(key, value)`:** We hash the received key using a consistent hashing function to get a Chord ID to look up using the Chord implementation. The Chord implementation will then return the IP address for the machine that should store the given key-value pair. Our wrapper will then communicate with this machine to store this key-value pair.

- **`get(key)`:** We hash the key using the consistent hashing function to get a Chord ID to look up using the Chord implementation. The chord implementation will then return the IP address for the machine that should have the given key in storage. Our wrapper will then query whether or not this machine has the given key in storage. If it does, the machine returns the value to the wrapper. Otherwise it returns a key missing error.

3.2.3 Deterministic Keys

It is important to note that we do not have a need to have a separate metadata table for storing keys or salts related to a user due to the fact that we are able to construct the necessary password segment and salt keys to look up given a username. As seen previously our keys have the deterministic structure: `{username}-(hash/salt)-{i}`. Thus given that we know the username (which is provided by the user) and the number of segments (which we decide at a system level) we can construct any password segment or salt key. The main restriction here is that users cannot have the same username as that would lead to key collisions. However, this is a reasonable restriction to have as most products will force accounts to have unique usernames in the first place.

3.3 DynamoDB Wrapper

The role of the DynamoDB Wrapper is to serve as a connection between the Chord network and DynamoDB as in Figure 2. The DynamoDB table is used to store the hashed value and the associated node with that segment.

For added security, our strategy is to not only store the hashed and salted passwords in a Chord network, but to also shared the passwords and store the different components in different Chord nodes. Thus, whenever the Chord implementation determines where each segment of the password belongs, we insert this information, along with the hashed value and the ID of the segment into the DynamoDB table. Whenever we are verifying a password, we use the node and the ID of the hashed password to search in our table and cross check the expected hashed value of the segment with the hashed value of the segment of the user-inputted password. To ensure guaranteed success, as mentioned earlier, our sharding and salting processes are consistent.

3.3.1 DyanmoDB Table

To better understanding the insertion and verification processes, we will dive into the schema of the DynamoDB table. The table has a primary key, a sort key, and one attribute value:

- **Node** - This is the primary key of the DynamoDB table and the main key used to query it. This key is the IP address and port combined, in other words, the Chord ID. We use this key as the primary key instead of the segment's ID to easily query all segments associated with a node since DynamoDB requires a specific primary key to query and does not facilitate querying by just the sort key.

- **ID** - Our sort key in the table, this key is the ID of the segment. It is organized in the following way: `Chord.Node#{Node_ID}-{user}-{hash/salt}-{i}`. Since we assume that each username is unique, as is common in a username/password system, we will always have unique IDs for our segments.

- **Value** - The final component of our table is the hash value. This is the value of a segment salt or a segment of the hashed value of the password. Using both the primary key and sort key, the Chord Wrapper is able to insert or request for this value, which is important for insertion or verification, respectfully.

3.3.2 Key Migration

An important aspect to consider when creating the system is adding fault tolerance. One situation where issues may arise is when a Chord node fails or leaves the network. In the traditional Chord implementation, when such an event happens, the finger tables are updated and the data from the faulty node is transferred to its successor [4]. However, the Chord implementation we use does not control the storage of our nodes, just their locations, meaning, when a node leaves the network, it is our job to update the information in the DynamoDB table.

The process of migrating our keys when a node fails is a three-step process. First, we query our table for all the keys from the failed node. Since our primary key is just the node ID, this is a simple task. Next, using these queried values, we delete all entries in our table with the associated Node ID and store the entries in our DynamoDB Wrapper. Finally, we edit the entries to match the information of the successor node (such as the Node ID and editing the Segment ID) and input the values into our table.

Another situation to consider is when a dead node rejoins the network or a new node joins the network. In this case, our finger table is updated since this node is entered into the middle of the ring. Thus, we need to update our keys in the table. In this situation, we need to move all keys that are currently associated with the successor of the new node to the new node if they lie before the new node. Thus, our strategy was to the delete all the keys of the successor and re-enter them into the network. Of course, when entering into the network, we must also update the nodes with their new information. By doing this process, we guarantee that we are placing all these nodes in the right location since we have a consistent hash function, and thus, a consistent protocol on where to to place the nodes.

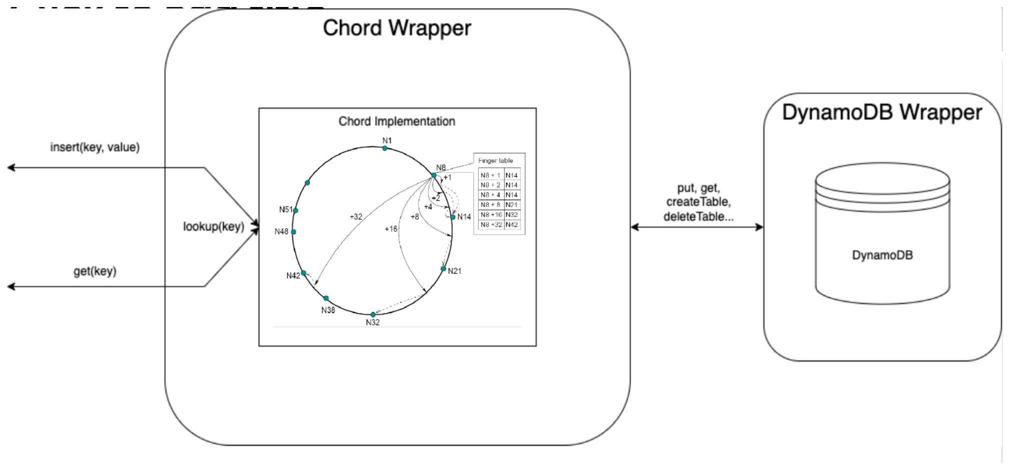

Figure 2: High Level Overview of Chord Wrapper and DynamoDB Connection

With our key migration procedure, we were able to build a system that was tolerant to node failure and node entry situations.

3.3.3 Implementation Simplifications

For our implementation of this design, we made some simplifications due to limited resources and time in regards to the storage aspect of our system. Ideally our system would use multiple key-value storage systems hosted on different physical machines that are connected to Chord to store our passwords in a distributed system. However, the main restriction that we have is that we are unable to obtain a large amount of physical machines to connect to Chord and utilize for storage. Even if we were to partition the devices we have currently using containers or VM's we would not be able to create enough nodes in our Chord system to accurately evaluate how our system would scale.

In order to address this issue we used DynamoDB in our evaluation implementation as the main way to store passwords. We utilized a single DynamoDB table where all nodes will insert/get password segments using the Main Server provided keys as the DynamoDB keys. In order to identify which node inserted a specific key, we appended the node's ID to the beginning of the key being inserted into DynamoDB table. Thus when querying for a node, our Chord implementation will return the IP as well as the DynamoDB URI and the node's ID.

The most important thing to consider is that, due to our integration of the DynamoDB table, we mimic having multiple servers and do not create a system with a central storage place. By storing the nodes information into the table, we create a simplistic system that is easily adaptable to unique servers. Our goal with this paper is to show that storing passwords using Chord is a secure process, and maintaining a singular storage system to mimic a network of servers does not infringe on that goal as the location of our passwords is insignificant as is where Chord places them and how to access them.

4 Results

In order to test our system we have ran a variety of experiments with a focus on system runtime and password security.

4.1 Runtime Evaluation

One goal we had while developing our distributed password storage was to make an efficient and scalable solution that could scale out to a large amount of distributed nodes in the system. In order to prove this we evaluated the runtime of our system.

4.1.1 Runtime Evaluation Setup

For our runtime evaluation we created various Chord Rings with an increasing number of nodes and ran the `createPasswordEntry` operation to insert a randomized 32 character password split into three parts. We then ran the `verifyPassword` operation to verify the inserted password. We ran these operations five times per Chord Ring and took the average runtime for our results. In all trials, we set $N = 3$, where N is the number of parts each password is split into.

4.1.2 Runtime Evaluation Results

As seen by Figure 3 and Figure 4 we can see that both the `createPasswordEntry` operation and `verifyPassword` operations follow a roughly logarithmic growth in runtime as the number of nodes in the Chord Ring increases. This is in line

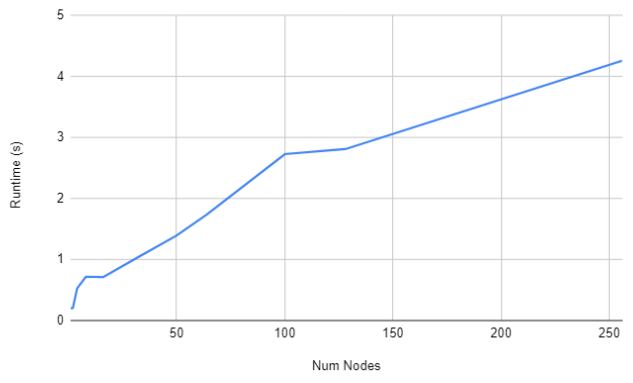

Figure 3: `createPasswordEntry` Runtime vs Number of Nodes in Chord Ring

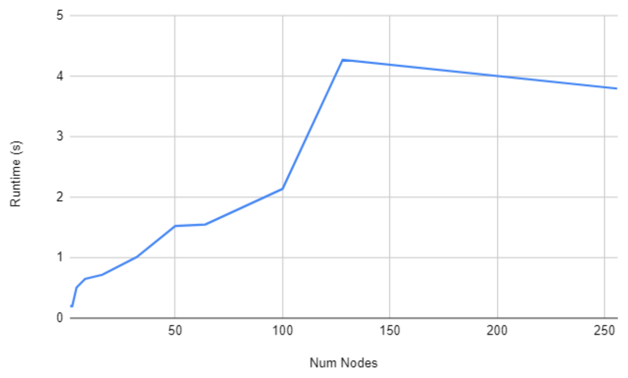

Figure 4: `verifyPassword` Runtime vs Number of Nodes in Chord Ring

with expectations as the Chord Paper[4] provided calculations and evaluations that proved the Chord lookup system scales in a logarithmic way to the number of nodes. This proves the scalability of our system as the more nodes we add to our distributed storage system, the runtime of operations does not grow in an uncontrolled manner.

It is important to note that after running operations in Chord Rings with more than 100 nodes, we observed a sharp spike in the volatility of operation runtimes. While we cannot are not confident in the exact reason behind this volatility, we believe it could be due to not all of the finger tables stabilizing. Additionally, some of the volatility could be attributed to the fact that some of the random password parts could be stored on nodes very far on the consistent hash circle from the node the Main Server is on.

4.2 Multiple Hash Security Evaluation

One of our initial hypotheses was that splitting up the password into multiple parts and hashing those individual parts would make the password harder to brute force.

4.2.1 Multiple Hash Security Evaluation Setup

In order to simulate this situation we created a program that would try to brute force random passwords given the hash, salt, and hash function used to create the password. We created random eight digit passwords made up of just numbers (to simplify the brute forcing algorithm) split into a varying amount of parts. Each of these parts would then be salted and hashed and provided to the brute forcing algorithm to solve. The brute forcing algorithm simply tries tries to increment numbers starting from zero and compares the hashed result to the actual password hash to verify if the password has been cracked. As our metric we measured the average time in milliseconds it took the brute forcing algorithm to crack an individual hashed part of the password.

4.2.2 Multiple Hash Security Results

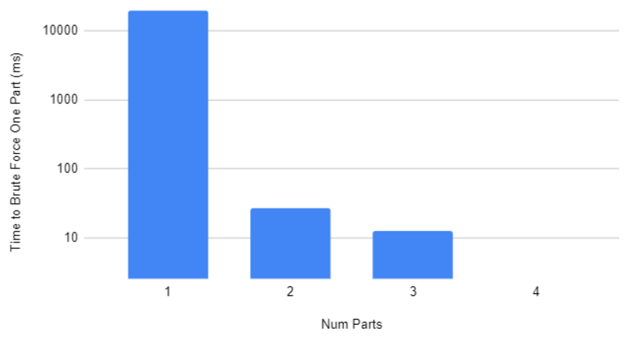

Figure 5: Average Brute Force Time Per Part vs Number of Password Parts

As mentioned previously we had hypothesized that splitting up the password into multiple hashed parts would make the password harder to brute force. However, as seen by Figure 5, the time to brute force individual parts exponentially decreases as the number of parts we split the password into increases. For example, the time it takes to brute force a password part when there is one part versus two parts is around a 10 times increase. This is due to the fact that each password part has less digits than the entire password. This means that the brute force algorithm will have a smaller number of password permutations to iterate through in order to figure out the password. Additionally, it is important to note that each password part could also be brute forced in parallel as each password part has its own hash part to verify against which could also significantly reduce the time necessary to brute force the entire password. Unfortunately this does mean that our hypothesis of increased security splitting a password into multiple parts was wrong.

However, there are ways to potentially fix this issue. Given that the brute forcing problem was simplified by the decreased number of digits per part, we could potentially pad each password part with more digits to make each password part's length greater than or equal to the actual password length. This would mean that each password part would have an equal or greater number of permutations to brute force. For example, for each password part we could create a padding hash by appending the salt, username, and password part sequence number and interlace the actual password part in between this padding hash.

4.3 Partial Leak Security Evaluation

Another hypothesis we assumed while developing this system was that it would be harder for an attacker to brute force the entire password given that they only have a subset of all password part hashes compared to brute forcing a password with only one hash.

4.3.1 Partial Leak Security Setup

In order to test this situation we utilized the same brute force algorithm as used in the Multiple Hash Security evaluation. The password would be a five digit password split into four parts spread across a three node chord distributed storage ring. However, in this situation we only used the brute force algorithm on a varying number of parts through the evaluation. The remaining digits would be brute forced by constantly polling different password guesses appended to the known password parts through our Main Server's `verifyPassword` operation. This setup mimics the situation where a malicious actor only compromises a small subset of nodes in the distributed password storage system and only knows a subset of all password part hashes.

4.3.2 Partial Leak Security Evaluation

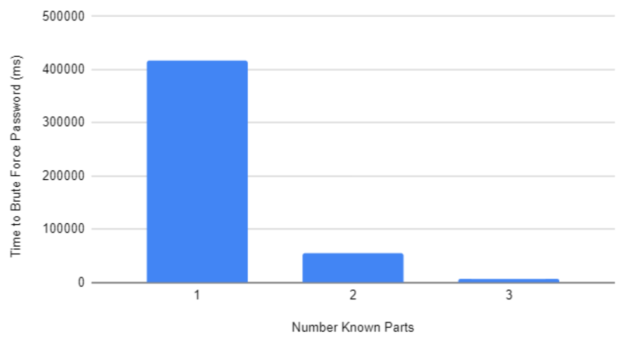

Figure 6: Entire Password Brute Force Time vs Number of Known Password Part Hashes

As mentioned previously we had hypothesized that it would be harder for a malicious actors to brute force a password given that they only had a subset of all password part hashes. As seen by Figure 6, we can see that as the number of known password part hashes decreases, the time it takes to brute force the entire password grows exponentially. In fact knowing only one part of the password increases the time to brute force the entire password considerably when compared to knowing the entire password hash. For example, in Figure 6 we can see that when the malicious actor only knows one password part it would take them around 400,000 milliseconds to crack the password. This is in comparison to Figure multi part security graph where knowing the singular hash for an entire eight digit password took our brute forcing algorithm around 20,000 milliseconds. Thus only knowing one password part hash increased the time to brute force the password by a factor of 20 even with Figure 6's password having less digits (five digits) compared to Figure 5's password (eight digits).

This increase in brute force time can be explained by a few factors. The most significant factor is that without all password part hashes, a malicious actors can only verify their guesses for the entire password through our main server. As shown in the Runtime Evaluation section our system does experience some delay introduced by Chord's ID lookup mechanism. Thus each individual guess for the malicious actor takes much longer to verify compared to running their guess through a known hash function and comparing the guess's hash value to the known password part hash. Additionally, the remaining unknown digits in the password parts the malicious actors do not have access to essentially become one long hash for the malicious actors to try and brute force. As shown in the Multiple Hash Security Evaluation section having one long hash takes much longer to brute force compared to multiple smaller hashes. Finally, it is important to note that since the malicious actors would have to verify their results through the Main Server, it would be easy for our system to keep track of repeated incorrect password guesses for a username and alert us of potential brute force attempts.

5 Discussion

The proposed P2Pass distributed password storage system provides a novel approach to password security by utilizing a secret sharing scheme to store hashed password fragments in a decentralized, distributed, peer-to-peer lookup system based on the Chord framework. The primary objective of this study was to evaluate the security and performance of P2Pass compared to traditional password storage techniques. The results show that P2Pass is more secure than traditional password storage methods, especially against partial leaks, when a hacker only breaks a part of the Chord network.

However, the hypothesis that splitting passwords into multiple hashes would make brute-forcing harder was found to be wrong. The results from the multi-hash security analysis revealed that splitting passwords into multiple hashes actually makes them easier to brute-force. One potential solution to this problem is to pad the password parts to make them longer. On the other hand, the hypothesis that brute-forcing passwords when only knowing a subset of all password part hashes is harder compared to knowing one long password hash was found to be correct. The results from the partial leak analysis demonstrated that it is harder to brute-force passwords when only a subset of the password part hashes are known.

Another important finding is that the number of password parts is a hyperparameter that needs to be set as it splits up the password among more nodes, making it less likely to leak all parts, but it also makes each individual part easier to brute-force. Therefore, a balance needs to be achieved between the number of password parts and the level of security required.

In terms of performance, the results suggest that the peer-to-peer nature of P2Pass may lead to higher password lookup times. However, the system scalability and fault tolerance were found to be satisfactory. Additionally, the security of the system against known attacks was evaluated, and the results suggest that P2Pass is secure against attacks such as dictionary attacks and rainbow table attacks.

Overall, the proposed P2Pass distributed password storage system provides a promising approach to password security by addressing the shortcomings of traditional password storage techniques. However, the limitations identified in this study, such as the trade-off between security and performance and the need to carefully balance the number of password parts, highlight the importance of continued research in this area. Future work could explore additional methods for improving the security and performance of distributed password storage systems, such as the use of encryption and alternative secret sharing schemes.

6 Future Work

Our current implementation of P2Pass is sufficient for testing the hypotheses outlined at the beginning of this paper. However, given more time and resources, we feel that we can improve upon P2Pass and turn it into a more complete system.

First, we would like to support heterogeneity of data storage methods. Ideally, each node in our Chord implementation should be able to store its data in arbitrary locations. For example, some nodes could utilize DynamoDB as we have, but others could store on Google Firestore, Amazon S3, Azure, or even directly on the node's disk. By having a diverse set of data locations, we can significantly reduce the chances of an attacker compromising all N hash segments.

Going along with the theme of security, we would like to increase difficulty for malicious nodes to access our Chord ring or act as a Main Server. One way to accomplish this would be fully flesh out an IP address whitelisting system, where nodes will not acknowledge another node in the ring if that node's IP address is not in some approved subnet or list of IP addresses. Another method could involve using public/private key pairs to verify that transactions between nodes are valid.

Finally, we would like to improve on our system's fault tolerance, specifically for lost hash segments, which would currently render the entire password unrecoverable. In a way our current implementation already accounts for this, by storing on fault-tolerant data stores such as DynamoDB. However, we would also like to explore the data reconstruction scheme outlined in Shamir's paper [1], which we discuss in Section 2.

7 Conclusion

This paper describes P2Pass, a distributed password storage system used to store user passwords in a more secure manner. P2Pass has been shown to increase password security against password leaks by splitting the password into separate parts. While the individual password parts are easier to brute force individually, the likelihood of leaking all parts of a password is reduced. P2Pass is also scalable with a logarithmic scaling pattern and allows for efficient lookup of all password parts utilizing Chord's lookup algorithm.

8 Group Contributions

8.1 Brandon Bae's Contributions

- Setup, ran, and analyzed security evaluation results
- Initial setup and experimentation of DynamoDB storage system
- Created helper scripts for easy mass Chord deployment for runtime analysis

8.2 Edison Ooi's Contributions

- Developed the Main Server, the primary interface through which users can create and verify password entries
- Researched and implemented salting and hashing schemes for secure password storage
- Designed and ran experiments for runtime analysis of `createPasswordEntry` and `verifyPassword` operations

8.3 Stuart Tsao's Contributions

- Conducted the preliminary setup and experimental phase of the previous Chord System.
- Developed a command-line interface tool to facilitate debugging and testing of new functionalities.
- Authored the Chord Wrapper, a software component that streamlines the integration of the Chord System with other applications.

8.4 Havish Malladi's Contributions

- Created the schema of DynamoDB table to fit our needs in terms of querying certain keys and authored the insert and get methods in the DynamoDB Wrapper to reflect these changes
- Updated the Chord Wrapper to switch from inserting and getting from a HashMap (initial design) to inserting and getting from Dynamo and allow for key migration
- Developed the key migration strategy, including editing the existing Chord implementation to transfer the dead node's keys and created methods in the DynamoDB Wrapper to contact Chord and the Chord Wrapper
- Fixed a few minor bugs in the existing Chord implementation that initially made the migration step difficult

9 Acknowledgements and Code Base

We would like to thank Dr. Danyang Zhuo for his guidance throughout this project and teaching us about distributed systems. We would also like to thank Chuan Xia, a former Duke student, for her Chord implementation, which was pivotal for the development of P2Pass.

To see code base, visit our GitHub.

References

[1] A. Shamir, "How to Share a Secret," Communications of the ACM, vol. 22, no. 11, pp. 612-613, 1979. Available: https://dl.acm.org/doi/10.1145/359168.359176

[2] Fujiwara, M., Waseda, A., Nojima, R. et al., "Unbreakable distributed storage with quantum key distribution network and password-authenticated secret sharing," Scientific Reports, vol. 6, 2016. Available: https://www.nature.com/articles/srep28988

[3] P. Gauravaram, "Security Analysis of salt||password Hashes," 2012 International Conference on Advanced Computer Science Applications and Technologies (ACSAT), Kuala Lumpur, Malaysia, 2012, pp. 25-30, doi: 10.1109/ACSAT.2012.49.

[4] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," in Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, 2001, pp. 149-160. Available: https://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf

[5] K. Scarfone, M. Souppaya, "Guide to Enterprise Password Management (Draft)" National Institute of Standards and Technology, 2009. Available: https://csrc.nist.gov/csrc/media/publications/sp/800-118/archive/2009-04-21/documents/draft-sp800-118.pdf

[6] S. Whited, "XEP-0438: Password Storage Best Practices," XMPP Standards Foundation, 2017. Available: https://xmpp.org/extensions/xep-0438.html

[7] Chung, J. Joe-Wong, C. Ha, S. et al, "On the Feasibility of Secure Distributed Storage in the Cloud," in Proceedings of the 2015 ACM Conference on Computer and Communications Security, 2015, pp. 90-101. Available: https://dl.acm.org/doi/10.1145/2741948.2741951

[8] Mirante, D., amp; Cappos, J. (2013). (tech.). Understanding Password Database Compromises. New York City, NY: NYU.

[9] Gillum, J. (2022, October 7). 1 million Facebook usernames, passwords may have been stolen (meta). Bloomberg.com. Retrieved April 18, 2023.

[10] Xia, C. (n.d.). Chuanxia/chord: Implementation of chord - A distributed hash table. GitHub.