SCAR - Scattering, Concealing and Recovering data within a DHT

Bryan Mills
bmills@cs.pitt.edu

Taieb Znati
znati@cs.pitt.edu

Abstract

This paper describes a secure and reliable method for storing data in a distributed hash table (DHT) leveraging the inherent properties of the DHT to provide a secure storage substrate. The framework presented is referred to as "Scatter, Conceal, and Recover" (SCAR). The standard method of securing data in a DHT is to encrypt the data using symmetrical encryption before storing it in the network. SCAR provides this level of security, but also prevents any known cryptoanalysis from being performed. It does this by breaking the data into smaller blocks and scattering these blocks throughout the DHT. Hence, SCAR prevents any unauthorized user from obtaining the entire encrypted data block. SCAR uses hash chains to determine the storage locations for these blocks within the DHT. To ensure storage availability, SCAR uses an erasure coding scheme to provide full data recovery given only partial block recovery.

This paper will first present the SCAR framework providing details on the related protocols and mechanisms. This paper then presents explores the tradeoff between data security and data availability. This tradeoff is presented using analytical models developed to describe SCAR's behavior. In this examination we conclude that SCAR can effectively balance this tradeoff when the network nodes are "sufficiently" available. Lastly, there will be a discussion of the prototype implementation and a presentation of experimental results.

1 Introduction

Computer users continue to accumulate increasing amounts of data sparking the need for highly available and secure storage. Documents,such as medical files, family photos, and financial records,have become digitized and are now stored on desktop computers. This places these documents at risk of loss as the storage location is not resilient to disasters. A small fire or simple hard disk failure could cause a catastrophic data loss.

Additionally, users want the ability to reliably access their data from multiple locations. This results in users storing their data on laptops or compact flash drives which are less resilient to disasters and vulnerable to theft. The loss of data from theft further results in a breach in privacy that can potentially cause more damage to the owner of the data. These issues often lead users to rely upon a central service provider to store, access, and secure their data.

Such central service providers are not acceptable solutions because by definition they are centralized and suffer from a single point of failure. Additionally, because of security breaches and corporate privacy policies, the security of the data being stored can not be relied upon. Therefore, any solution that is centralized requires the users to implicitly trust the service providers. This illustrates the need for a solution that is fully distributed.

It becomes clear that a solution must have the following characteristics in order to meet the storage requirements of its users:

- Available - data must be available at all times and withstand failures and disasters.
- Accessible - data must be attainable from multiple locations.
- Distributed - the system should be independent of any centralized service.
- Private - data is only accessible to authorized users and owners, and those storing the data remain anonymous.

Peer-to-peer distributed hash tables (DHTs) provide an attractive solution to this problem. DHTs provide a distributed storage system that is decentralized yet provides a logically centralized hash table abstraction. These structures have been implemented in a variety of systems each with different purposes [9, 13, 16]. Many of these applications are concerned with the storage of only public data, such as file sharing, web caches, and content distribution. The proliferation of file sharing systems alone has proven that DHTs can successfully store publicly available data. However to satisfy the storage requirements, DHTs must have the ability to securely store private data. In addition to private storage, there is also a need for "closed" group storage where several people within a group need shared access to the data. Applications ranging from medical record

storage to family photo sharing expose the need for such a solution.

Research on DHT security has focused on the maintenance and stability of the DHT structure itself and has not adequately addressed the issue of data privacy within the network [1, 10, 15]. Systems requiring data privacy have relied upon data encryption or centralized services. The most common solution is to simply encrypt the data being stored, thereby making the data "self-secured". The DHT treats the data as it would any public data because it is assumed that data encryption is enough to protect the data from potential attackers. While encryption provides some level of security, this is not a complete solution. Encryption does not prevent an attacker from running brute force cryptanalysis tools to gain access to the data. This becomes even more apparent in systems designed to provide long term storage, such as Oceanstore [6]. In these systems, given enough time, any encryption could be broken. Another solution relies on a centralized service that acts as the gate-keeper granting or denying access. This centralized service creates a single point of attack thereby compromising both the privacy and availability constraints.

In this paper we present SCAR, a fully distributed DHT based scheme for providing highly available privacy preserving storage. The main tenet of SCAR is based on the observation that it is harder to break encrypted information if the attacker can not obtain the encrypted data. SCAR achieves this by using a simple, yet powerful concept of concealment through random distribution. The goal is to break data into pieces and randomly distribute the data throughout the network so that only authorized users can locate the pieces and recover the original data.

SCAR faces several crucial design challenges. First, how can one randomly distribute data so that the attacker can not find it, yet make the data easily accessible to authorized users? Can such a solution be used within the context of a DHT? Can the system ensure high availability in the face of node failures? Finally, how can a system provide a simple interface that enables the user to scatter, conceal, and recover data?

In order to randomly distribute data, envision using a password seeded hash function. This would produce a single hash location that is impractical to obtain without the password. This approach leads to one secret storage location; thus hiding the data in one location. However, this is not as secure as breaking the data into many pieces and hiding those pieces in multiple locations. To produce multiple locations using the scheme above, one would have multiple passwords, one for each storage location. It is not practical to require a user to remember multiple passwords. Therefore,to generate multiple secret storage locations, SCAR uses a given hash value and repeatedly hashes that value along with the same password to generate multiple random locations. This prevents the need for multiple passwords yet produces multiple storage locations that are computationally infeasible and yet easily determined for authorized users. This process is based upon the concept of hash chaining [4]. Because SCAR uses hash functions to produce storage locations this is easily implemented within a DHT.

Scattering data across multiple nodes increases the data security because it adds the requirement that the attacker needs to know where to look. By using password seeded hash functions knowing where to look for data is much harder for the attacker. The problem with this technique is that authorized users must depend upon multiple nodes to access their data. If anyone of the multiple nodes storing the users data is unavailable then that data is not available. To address the problem of data availability, SCAR does not simply split the data but instead generates the data pieces using of the erasure encoding schema called information dispersal algorithm (IDA) [8]. IDA splits a block of data into multiple pieces such that only a fraction of those generated pieces are needed for data recovery. Formally, IDA generates f pieces of data such that only k pieces are needed for recovery, such that $k < f$. Combining IDA with our hash chaining process enables SCAR to provide both privacy preserving scattering and availability of data.

In the rest of this paper we demonstrate the viability of this approach by building a prototype implementation and we present the implementation details in Section 2. Analytical models that describe the system behavior are provided and explored in Section 3. Then using the prototype implementation experimental results are explored in Section 4. In Section 5 this work is put in context to other related work. We then conclude in Section 6.

2 Design

SCAR assumes a structured peer-to-peer network, and uses the associated DHT to provide distributed data storage. Given this storage substrate SCAR builds a framework for providing secure, fully distributed and highly available storage of private data. The framework presented is not dependent upon any particular DHT implementation, but assumes that a *put/get* interface is available. With these assumptions, SCAR was then designed to meet the following goals:

- Provide a higher level of security than that provided by data encryption.
- Provide data availability that is at least as good as that provided by the underlying DHT implementation.
- Maintain independence with respect to the underlying distributed storage system.
- Provide a simple interface for users and application developers.

The process of storing data starts with the user providing three pieces of information: the data to be stored, a name for that data, and a password that will be used to grant access to that data. Given these pieces of information SCAR then stores the data ensuring that the availability and security requirements are met. The storage process can be broken into 3 logical steps: pre-processing the data, splitting the data into pieces, and storing the pieces. These steps are represented in Figure 1.

In order for SCAR to retrieve the stored data the user must provide the name of the data and a password. The process of retrieving data has 3 steps: finding the pieces, reassembling the data, and verifying the data. This process is depicted in Figure 2. The rest of this section will describe the details of each step in the storage and retrieval process.

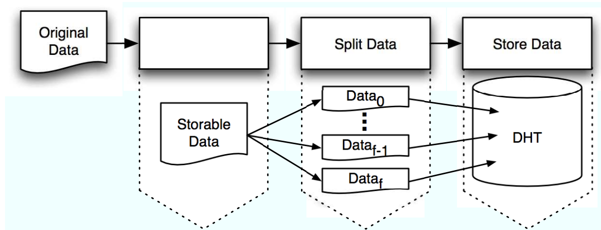

Figure 1: Overview of SCAR's Encoding Process

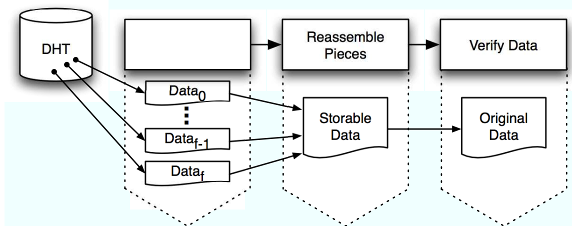

Figure 2: Overview of SCAR's Decoding Process

2.1 Data Pre-processing

As with most storage systems, there are no assumptions made about the data being stored. The first step in SCAR is to pre-process the data making it ready for storage. This pre-processing is used to ensure that the data is ready for storage and provide mechanisms for data verification after retrieving the data from storage.

First, the pre-processing step must ensure the data is large enough. One of SCAR's design principles is to use randomness as a means to make it hard for an attacker to reassemble the original data. SCAR divides the data it pieces; the size of these pieces is proportional to the size of the input data. Therefore, small input data size results in small pieces. These small sized pieces make it much easier for an attacker to reassemble the stored data. Furthermore, the data being split has to be at least as larger as the number of pieces to be generated. For example, the splitting process cannot generate 20 pieces given only 15 bytes of data. Therefore, this calls for the need to "augment" the original data with padding.

The security of SCAR is based upon the requirement that data is scattered such that an attacker would need to locate all required pieces to retrieve the data. This means that SCAR can not simply pad the data with null data but instead must add padding such that all the data, including the padding, is required for data retrieval. To obtain this property, SCAR uses secret sharing techniques [14] to generate the padding.

Let the required data size be S, this value is dependent upon the implementation but it must be greater than or equal to the number of pieces to be generated. To produce padding the lowest multiple of the original data size that results in a number greater than or equal the required data size is chosen, let this value be m. The value of m is determined by dividing the original data size by the required data size then taking the ceiling value of that result. Then $m-1$ random data blocks are generated. To produce the mth data block all the random data blocks and the original data block are combined using XOR. This produces m data blocks that when combined together using XOR produces the original data block. This process will always result in a data size greater than or equal to the required data size but less than 2 times the required data size.

For example, assume the required size is 100 bytes and the raw data is only 25 bytes. The m value would be 4, and SCAR would then generate three 25 byte random pieces of data. The last piece of data would be the XOR of each of the random pieces and the original data. By stringing each of these generated pieces of data together, the size of the data is the desired 100 bytes. To recover the original data, one must identify each generated piece of data and XOR each piece together. The result of this operation is the original data. Every generated piece is needed to recover the original data, meaning all 100 bytes are needed to recover the 25 bytes of original data.

The second pre-processing step constructs a fixed size header depicted in Figure 3. This header serves two functions: first it provides the retrieving process the ability to restore data that has been padded. This is done by adding the padding multiple, m, to the raw data header. The header includes a hash digest of the original raw data being stored, this allows the retrieving process to verify the recovered data.

The last pre-processing step is to encrypt the entire preprocessed data using symmetrical encryption. This encryption provides additional security preventing an attacker from viewing the header information and guaranteeing that

SCAR is just as secure as the chosen encryption mechanism.

<table>
<tr><td colspan="2">file size (64)</td></tr>
<tr><td>SHA-1 Hash (40)</td><td>padding multiple (10)</td></tr>
<tr><td colspan="2">...
data
...</td></tr>
</table>

Figure 3: Data Header, numbers represent size in bytes

2.2 Splitting Data

Once the data is ready for storage, SCAR must split the data into multiple pieces. The obvious way to do this is to simply divide the data into equally sized blocks such that all the pieces are required for data recovery. Because storage nodes join and then leave the network SCAR can not guarantee the availability of all the data pieces. SCAR addresses this problem by encoding the data during during the splitting process. SCAR implements Rabin's Information Dispersal Algorithm (IDA) [8], This process not only splits data into multiple pieces but also encodes the data to increase availability.

The main concept behind IDA is to split a given data object into f different pieces such that only k pieces are required for retrieving the original data, where $k \leq f$. To achieve this property, IDA uses principles of matrix multiplication to build a transform function from k original pieces of data into f encoded pieces of data. Then given any k pieces of the encoded data, it is possible to regenerate the original data using the inverse of the transform function. The inverse of the transform is guaranteed to exist because the transform is built using a linearly independent matrix thus the inverse of that transform exists. Algorithms for producing linearly independent matrixes can be found in [7].

Values of f and k are dependent upon the availability requirements for the data being encoded. Once values of f and k are chosen, SCAR divides the data into k pieces using a simple split function then executes IDA generating f encoded pieces. Each generated piece is unique and of the same size, these pieces are referred to as *storage units*.

Information dispersal algorithm (IDA) was chosen because of its ability to produce f distinct pieces of data from a data block of any size. In contrast, other erasure codes such as Reed-Solomon only produce a small set of fragments for a given block size. Other codes, such as Tornado, are focused on reducing computation speed and do not guarantee distinct pieces. Because IDA produces distinct pieces SCAR can guarantee that if any k pieces are retrieved the original data can be re-constructed.

2.2.1 Storage Unit Header

The user retrieving data is only required to provide SCAR with the data name and the password. SCAR must know how to reassemble the data given this information. In section 2.3 it is shown how the storage units can be found in the DHT, but for the data to be reassembled the values of f and k need to be known. This requires that each storage unit have additional header information. Adding headers to each storage unit enabled the retrieval process to determine f and k values after retrieving a single storage unit. In addition to providing required information, the header also enables verification of each storage unit before reassembly and the detection of hash collisions.

The header contains three values, the f and k values, a unit signature, and a checksum. The f and k values are just added to the header. The *unit signature* is a hash value of the previous and next storage location, the unit sequence number, and the user's password. By including the password in the unit signature, it is assured that an attacker could not determine the encoded information. The unit signature is used by the retrieval process to verify that the unit is part of the data being retrieved, detailed in section 2.4. A checksum is computed for the entire storage unit, including the f and k values and the unit signature previously computed.

checksum
unit signature
f and k values

Figure 4: Storage Unit Header

2.3 Scattering Storage Units

SCAR relies upon a hash chain seeded with the user's password to determine where the storage units will be placed within the DHT. Let the storage locations be called L_0, L_1, L_2, ..., L_n. Where L_1 is the location of the first unit, L_2 is the location of the second unit, and L_n is the location of nth unit. The first location in the sequence, L_0, is used as a base for the hash generation and does not represent an actual storage location. This process is depicted in Figure 5.

The chain is seeded with three pieces of information: a name identifying the data name and a secret password. This information is hashed, and the resulting value is then hashed again with the same seeding information (data name and password), to produce the first storage location (see Figure 5). The result of the previous hash, combined with the original seeding value is then used to generate the next storage location. This hash chaining is continued until all the units

$$
\begin{aligned}
L_0 &= \text{hash(password, data name)} \\
L_1 &= \text{hash}(L_0, \text{password, data name}) \\
L_2 &= \text{hash}(L_1, \text{password, data name}) \\
&\vdots \\
L_n &= \text{hash}(L_{n-1}, \text{password, data name})
\end{aligned}
$$

Figure 5: Storage Location Generation. Each argument to hash is to be concatenated together.

have been assigned a storage location. Once storage locations are determined, *put* requests can be issued to the DHT for each storage unit. This process can be done in parallel, and therefore the storage time bounded by the slowest time to insert one of the storage units.

A closer look at the above process reveals that the location procedure may result in storage location collisions. In order to handle collisions, a request is made to determine if another unit has been previously stored at that location before inserting the storage unit. If so, the algorithm skips that value in the hash chain and attempts to use the next value in the chain. This process is continued until a vacant location in the DHT is found. During the recovery process, the storage unit header is used to determine when such a collision has occurred. As a result, the value of n depicted in Figure 5, is dependent upon the number of the number of pieces generated, f, but is not directly correlated. This is because hash collisions may require additional storage locations.

If a collision occurs while trying to store the first storage bin, SCAR checks to make sure that the unit signature does does not match that of the block currently stored at that location. If it does, then the recovery process would not be able to detect the error and might retrieve the wrong data. In this case, an error must returned asking the user to select a different data name or password. If the unit signatures are different then the storage process can skip to the next storage location and being the data storage at that location. To limit the search space for the retrieval process if at any point f storage locations are skipped then storage process returns an error, see section 2.4 for details.

2.4 Data Retrieval

To retrieve data in SCAR, the user must provide the name of the requested data and a password. SCAR then generates the storage locations for each storage unit. SCAR can then issue *get* requests against the DHT to start retrieving the storage units. Each retrieved storage unit is verified using both the checksum and the unit signature. Once k units have been retrieved and verified the data is reconstructed using IDA. The retrieved data is then verified and any extra padding is removed as described in section 2.1. The output from this process will be the original stored data.

2.4.1 Retrieving Storage Units

Once a storage unit is retrieved it must verified that the retrieved unit is not corrupted and that unit is part of the requested data. The checksum inside the storage unit header is first verified, if this fails then the unit has been corrupted and therefore the unit stored at that location is assumed to be missing. By verifying the unit signature the retrieval process can verify that the unit is part of the requested data.

For the retrieval process to verify the unit signature the next and previous storage locations, the user's password, and the unit sequence must be known. The next and previous storage locations and user's password are fixed values with respect to this unit and therefore are known. However, the unit sequence can be a range of values. The range of unit sequence numbers that need to checked is equal to the number DHT locations skipped, if one location is skipped then there are 2 possible piece numbers, if two are skipped there are 3 possibilities, etc. For example assume the unit stored at the first storage location can not be verified. While verifying the unit stored in the second storage location there are two possible values for the unit sequence, 1 or 2. This is because the first unit might not have verified because of data corruption, making the second unit's sequence number 2. However, the first unit might not have verified because the storage process detected a hash collision and skipped that location, making the second unit's sequence number of 1.

Once any storage unit has been verified the retrieval process the f and k values are known. The retrieval process proceeds to retrieve and verify k out of the f storage units. There are two possible failures that can occur while recovering any of the f storage units.

The first failure that can occur if the checksum or unit signature fails to verify. This mismatch would indicate that there was data corruption within the DHT or a hash collision occurred during the storage process. In either case the retrieval process would skip that location and proceed to the next storage location specified by the hash chain. This is the the congruent with what the storage process does if a hash collision occurs. If at any point a max_f blocks are skipped the process assumes failure.

The second failure occurs if the storage location contains no data. This occurs when a node fails, removing all of it's data from the network thereby creating a void in the DHT. When this occurs there are two possibilities: the storage bin that used to be stored at this now vacant spot was a storage unit for the requested data; or the storage unit that was stored there was for different data, meaning a collision occurred. The retrieving process is able to check for this because the unit sequence number is part of the unit signature.

2.4.2 Reconstructing the Original Data

The retrieval process needs to find k storage units in order to retrieve the original data. Once these units are recovered, they are assembled and decoded using the information dispersal algorithm (IDA). The decoding matrix is known because the values of f and k are known. After IDA decoding occurs, the data can be decrypted using the symmetrical encryption performed during the storage process. The data header is then used to verify the entire data object. Once verified any padding is removed, as described in section 2.1. The original data has now been retrieved and is provided to the user.

2.5 Immutable storage

Most DHT's implement their storage as immutable objects. To ensure compatibility with these DHT's, a data object revision number can be added to SCAR. This revision number would become part of the hash chain used to generate storage locations. Every time a new revision is stored using SCAR, the revision number is incremented to the next available revision. This process can be hidden from the user by automatically querying the DHT to determine highest available revision number for the given data name and password. The process of determining the maximum revision number requires that a single storage unit from the other revision be retrieved to determine if that revision number already exists. Once the highest revision number is determined that number can be incremented by one. Finding the current revision number would involve a similar process.

In order to provide mutable storage some DHT systems allow multiple data entries for a single hash value. The addition of a data object revision number allows SCAR to handle this case. SCAR would also have to look at all storage units at a particular location rather than just a single entry.

3 Analytical Models

This section presents the analytical models developed to describe the availability and security of data stored using SCAR. These models are then used to analyze the inherent tradeoff between availability and security.

3.1 Data Availability Model

In SCAR, data availability is dependent upon the availability of nodes that are storing the data pieces. To understand data availability, we first explore the availability of a single node. Having an availability model of a single node, we can then develop a model that describes SCAR's data availability.

3.1.1 Node Availability

To model the node availability one must consider the behavior of the user of that node. The node's availability is based upon two user behaviors:

- the frequency of which the user visits the network, and
- the time the user resides in the network during each visit.

This behavior can be modeled by a process alternating between two states: Online and Offline. When the user is online, the node is available and therefore is able to provide peer-to-peer services. While offline, however, the node is unavailable and can no longer assume its peer-to-peer responsibilities.

At any given point in time, the node can be either available or unavailable. Because the user exhibits this behavior the node availability can be modeled as an ON/OFF process, where the duration in the ON and OFF states are reflective of a particular user behavior. To model this behavior, we define the following binary variable, which indicates if the node is available at a given time t, as follows:

$$S_i(t) = \begin{cases} 1 & \text{if node i is available at time t} \\ 0 & \text{otherwise} \end{cases}$$

SCAR is only concerned with the availability of the node. Consequently, node availability in the SCAR framework can be modeled as an ON/OFF process, where the average duration in the ON and OFF states are reflective of a particular user behavior. Notice that the reasons which drive a user to alternate between online and offline states are only relevant to the extent that they expose different classes of users. The node's availability is therefore defined as the probability that the value of $S_i(t)$ is 1.

Assuming that the nodes ON and OFF periods have a mean value, the node's asymptotic availability can be defined using $S_i(t)$. The ON and OFF cycles have a mean time therefore they have an expected value. Let $E[ON_i]$ and $E[OFF_i]$ be the expected ON and OFF time for node i respectively. Given this information, we can define A_i as the availability of node i.

$$A_i = \lim_{t\to\infty} Pr[S_i(t) = 1] = \frac{E[ON_i]}{E[ON_i] + E[OFF_i]}$$

Defining the values of $E[ON]$ and $E[OFF]$ is dependent upon the user's behaviors. These users can be broadly divided into two different groups, those that are profile driven and those that are task driven. Profile driven users can be described using their profile. Therefore, they login into a

peer-to-peer network and remain for a specified duration of time regardless of their activity in the network. However, task driven users join the network to perform a task and then leave the network after that task is complete. Therefore, the amount of time a task driven users is dependent upon their task and not their profile. However, the time the user spends off the network is never task dependent but always driven by the users profile. This reasoning leds to the conclusion that the nodes expected ON/OFF behavior should be modeled using two distributions. One distribution depicting the node's ON behavior and the other its OFF behavior. The only requirement for the analysis presented here is that these distributions must have expected values.

3.1.2 Node Types

Based on the node availability model discussed previously, peer nodes are characterized based on the parameters of an ON/OFF process, namely the expected duration of the ON and OFF periods. For this analysis we consider three different classes of users which correspond to three types of nodes: infrastructure, power, and peeper nodes.

Infrastructure nodes represent the core component of a peer-to-peer network. As such, these nodes are expected to remain connected to the network, barring physical failure. Power nodes represent a class of nodes which are not dedicated resources, but remain connected to the network for extended periods of time. Peepers, on the other hand, are short-lived visitors of the network who are task driven. Upon acquiring the needed resource or service, peeper nodes leave the network.

To determine the probability that a given node is in a particular class, the network is characterized by its class distribution. For example, the network might be composed of 50% infrastructure nodes, 30% power nodes, and the remaining 20% are peepers. This allows us to vary the class distribution of our network and see how this affects the average node availability.

Let $E[ON_c]$ and $E[OFF_c]$ be the mean ON and OFF times for each class c. Then, we can define the network node's average availability by performing a weighted summation over each of the node types.

$$\bar{A} = \sum_{c \in Classes} Pr[available|class = c] * Pr[class = c] = \sum_{c \in Classes} \frac{E[ON_c]}{E[ON_c] + E[OFF_c]} * Pr[class = c]$$

Conversely, let the node's average unavailability be defined as $\bar{U}$, which is the inverse of $\bar{A}$.

$$\bar{U} = 1 - \bar{A}$$

3.1.3 SCAR's Data Availability

Availability of data within the DHT is expressed as a probability that the data stored in the DHT will be available when requested. This probability is dependent upon the probability that at least k out of the f nodes storing the data are available. This model assumes that the availability of any node is independent of the availability of any other node. Based upon the model of node availability described previously, the average network node availability ($\bar{A}$) is used to determine the data availability of SCAR. The data availability in SCAR is described using the following formula [12]:

$$P_a(k, f, \bar{A}) = \sum_{i=k}^{f} \binom{f}{i} (\bar{A})^i (1 - \bar{A})^{f-i}$$

Looking at Figure 6, it is apparent that if the node's availability is less than 80%, then replication exhibits higher data availability than erasure coding based schemes. This is an important observation because it provides operational bounds upon the SCAR framework. The reason this bound exists is that erasure coding relies upon a quorum of nodes to be available, and as the nodes in the network become unavailable, achieving the necessary quorum becomes less likely. The advantages of erasure coding, however, is that it uses less storage and increases security; but if the node's availability is low, its effectiveness in enhancing data availability is significantly decreased.

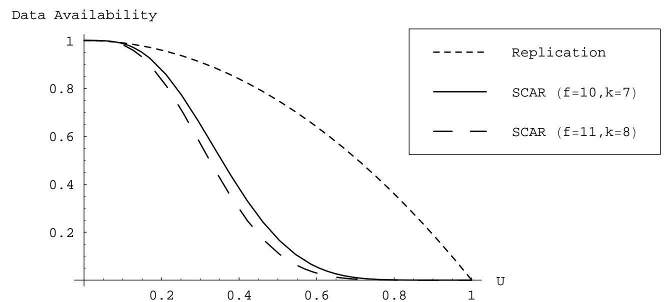

Figure 6: Comparison between replication and SCAR as the network nodes become increasingly unavailable.

3.2 Security Model

Because symmetrical encryption would be used on the raw data before SCAR is executed, any security provided by SCAR is in addition to this encryption. Therefore, instead of modeling the security of the entire system, we model only the additional security provided by SCAR. SCAR's security depends on the inability of an attacker to gather the pieces and reconstruct the original data. Therefore, to measure SCAR's security we define the probability that an

attacker could locate and reassemble the scattered pieces. This captures how *hard* it would be for the attacker to acquire the stored data.

This probability is based upon the number of combinations an attacker would have to assemble and test. This, however, is dependent upon the amount of information the attacker has access to. An attacker would need three pieces of information to recover the data stored with SCAR. First, the attacker needs to know the minimum number of pieces, k, required for reassembly. Second the attacker must determine the location of where each of these pieces are stored. Finally, the attacker must determine the encoding order of these pieces. SCAR's security is based upon the fact that the attacker could not easily obtain all three pieces of information.

Although the minimum number of required pieces, k, is variable we believe that in practice the value of k will be fixed or derived from a small range of values. Regardless of the size of the data being stored, the number of pieces required will be the same. By assuming the attacker knows the value of k but does not know the location or encoding order we can define the probability as follows:

$$P_s(k,f,n) = \frac{1}{k!} \times \frac{\binom{k}{f}}{\binom{k}{n}}$$

- n - the total number of pieces in the network
- k - the minimum number of required pieces
- f - the total number of pieces for the data in question

It should be noted that the size of the network is not a fixed value but is dependent upon the attackers *perceived capacity* of the network. The perceived capacity represents how large the network's storage capacity appears to the attacker. For example, if the attacker has no information about where the data pieces are scattered, the perceived capacity is the size of the entire network. On the other hand, if the attacker can monitor the network traffic, he potentially has a smaller perceived capacity. The more information the attacker has the smaller his perceived capacity, thus reducing his search space.

The conclusion is that to ensure SCAR's security, the attackers *perceived capacity* should be increased. Assuming the worst case the attacker can monitor all network accesses. One practical way of doing this is to batch data accesses across multiple data objects. This would increase the attackers perceived capacity thereby increasing the data security at the cost of increase storage latency.

3.3 Tradeoff Model

There is an tradeoff between data availability and data security within the SCAR framework. To increase data availability one wants to make the number of required pieces, k, as few as possible, the optimal value being $k = 1$. To increase security one wants to make the number of required pieces, k, large, the optimal value being $k = f$. In this analysis the two previously defined models are used to produce a metric to evaluate the tradeoff between availability and security.

The creation of a tradeoff metric results in a function with the following the parameters:

- n - the total number of pieces in the network
- k - the minimum number of required pieces
- f - the total number of pieces for the data in question
- $\bar{A}$ - average node un-availability
- c - an attackers computing capability

This function uses the availability metric as it was previously presented. The security metric, however, must be modified because it represents the likelihood that any one combination of fragments would result in a security breach. Because an attacker has the ability to try multiple combinations over time the tradeoff metric introduces the notion of *computing capability*. The computing capability represents the number of combinations an attacker could check given the attackers lifetime. For example if the attacker could check 1 combination every second and would be willing to do this for 5 years his computing capability would be 157,680,000 (the number of seconds in 5 years). The computing capability is then multiplied by the security model to produce the likelihood that an attacker could gain access to the data within the attackers lifetime. Combining the two models with the computing capability and a weighted sum produces the following tradeoff function:

$$P_t(n,k,f,\bar{A},c) = w_1 * P_a(k,f,\bar{A}) + w_2 * P_s(k,f,n) * c$$

In our analysis we assume that availability and security are equally important, and set the weights to 0.5. In Figure 7, we observe that the optimal value for k would be 8 when: there are total of 10 pieces ($f = 10$), two data accesses are batched together ($n = 20$); the average node unavailability is 10% ($p = 0.10$); and the computing capability is 1 per second for 5 years ($c = 157,680,000$). Intuitively this makes sense: for maximum availability, a small number of pieces are required, but for maximum security a large number of pieces should be required. The tradeoff model shows that given our current system configuration the balance point is not in the middle but shifted closer towards the total number of pieces generated (f).

The tradeoff function allows one to find the optimal values of f and k given some system state. The tradeoff model can be easily extended to include the amount of storage space, system latency, or other metrics one might be consider.

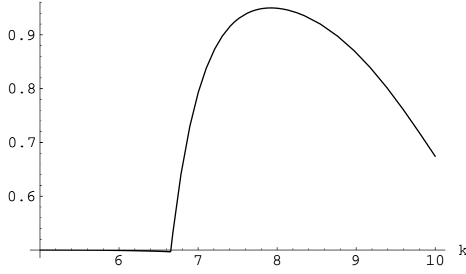

Figure 7: Tradeoff model varying k; $f = 10$, $max_k = 20$, $\bar{U} = 0.1$

4 Experiments and Implementation

SCAR was implemented as an application outside the DHT, and therefore is not bound to any specific DHT implementation. To accomplish this, there is an interface layer between the generation of the storage bins and the actual data accesses. Currently there is only two storage interfaces defined. One makes use of OpenDHT's web services API [11] and the other is a file system interface used for development.

SCAR was implemented using the Python programming language, and contains a command line interface. This command line utility allows the user to specify a file to store in the DHT. Here, the user is prompted for a password, which is used to store the data in the DHT. In addition, values for f and k can be specified on the command line, the default values are $k = 8$ and $f = 11$. The same command line can also be used to retrieve data from the DHT by specifying a filename, again prompting for a password.

4.1 Simulator

The simulator was built to allow stochastic evaluation of the SCAR system. It simulates the behavior of a DHT then executes SCAR's storage and retrieval processes. The DHT is simulated by creating nodes that are assigned a unique key using a randomly generated hash value. This emulates the way nodes are created in a DHT. A data object can then be stored within the DHT, the data object is also assigned a unique key using a hash function and stored on the node assigned the closest numerically matching key.

To simulate nodes ON and OFF behavior, nodes alternate between exiting and entering the network based on their node type. This ON or OFF probability is based upon a specified distribution, or can be set to fixed probability. At each simulation step, all nodes are given the opportunity to change their current state. Each node in the simulator is created as an independent object thus allowing the specification of the node properties independently of the other nodes. This can be used to simulate a network composed of different types of nodes.

Simulating the behavior of SCAR is accomplished by splitting data into f pieces such that k are required and then storing those within the DHT. The storage locations are determined using hash chains as described in section 2.3. At any simulation step, a query can be executed for a specific data object given the current network state. Any values of f and k can be simulated including replication, done by setting $f = 2$ and $k = 1$. During the simulations several standard network configurations were used, these are listed in Table 1.

Infrastructure	Power Users	Peepers	$\bar{A}$
100%	0%	0%	98.39%
80%	10%	10%	86.78%
70%	20%	10%	83.44%
50%	30%	20%	71.66%
40%	30%	30%	63.36%
20%	30%	50%	46.70%
10%	20%	70%	33.41%

Table 1: Simulated values of $\bar{A}$ for network configurations.

4.2 Data Availability

The goal of this experiment was to validate the theoretical models discussed in section 3. Using the simulator, we created a network with homogeneous nodes, all with the same fixed node availability. Data was inserted into the network using various values of f, k, n, and $\bar{A}$. We compared the simulation results with the results obtained using the analytical models. As can be seen in Figures 8 and 9, the simulation results show that the analytical models offer accurate estimates of SCAR's behavior.

In a few cases the experimentation resulted in two pieces of data being stored on one node, causing the data availability to be less than that predicted by the model. In the next experiment, we look at the likelihood that a single node might be assigned two pieces of data from the original data object.

4.3 Node Collision

SCAR's availability model assumes that each piece of data is distributed to a unique node location. This property is required because the metric assumes that data piece failures are independent from one another; however, if two or more pieces of data were stored on the same node, those failures would no longer be independent. To confirm our assumption, we simulated the storage of a 1000 data objects using SCAR and monitored the network for node collisions within the same data object. The obvious observation is that as the total number of nodes in the network increases, the

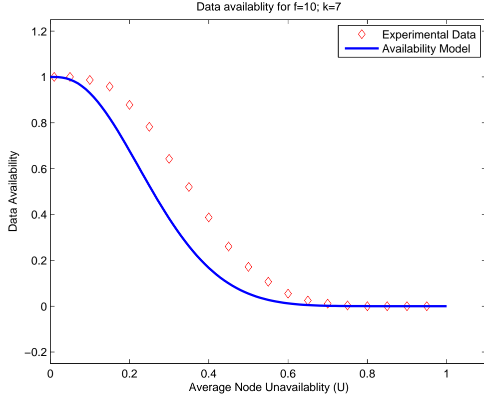

Figure 8: Comparing Model and Experimental Results varying $\bar{U}$ using $f = 10$, $k = 7$, $n = 1000$. Each experimental point represents mean recovery rate of 1000 samples.

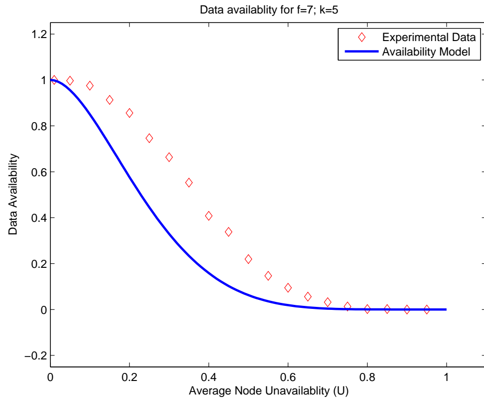

Figure 9: Comparing Model and Experimental Results varying $\bar{U}$ using $f = 7$, $k = 10$, $n = 1000$. Each experimental point represents mean recovery rate of 1000 samples.

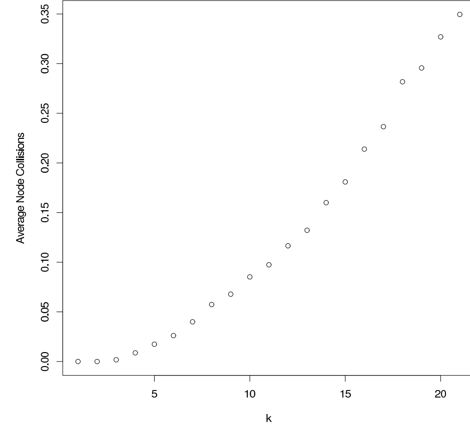

Figure 10: Average number of collisions varying number of fragments; $n = 1000$.

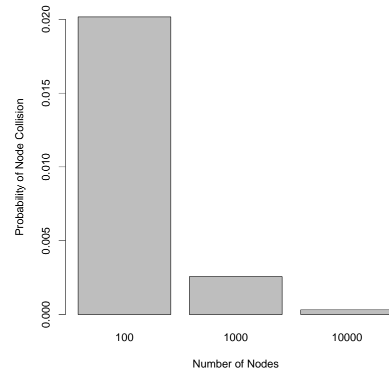

Figure 11: Maximum number of collisions to non-collisions varying number of nodes; $f = 11$.

probability of node collision decreases. Inversely, as f increases the probability of a node collision increases. This can be observed in Figure 10.

As can be observed in Figure 11, the probability of node collisions are very low when using a reasonably large network. In Figure 11 the value of f was fixed to 11 and the total number of nodes was varied to determine how many collisions resulted. The simulation was run multiple times for all 1000 data objects using randomly generated network nodes of the specified size. During each experiment the likelihood of a node collision was calculated by looking at the number of actual collisions divided by the total number of pieces being stored in the network $(1000 * 11)$. To characterize the worst case behavior, Figure 11 shows the maximum likelihood of collisions observed for a network of the specified size.

4.4 Network Sensitivity Analysis

In this section, we focus on SCAR's sensitivity to network changes. In particular, we focus on the data availability as the nodes in the network become less available. As mentioned in section 3, erasure coding is only effective when the nodes themselves are fairly reliable ($\sim 80\%$). To validate this assumption, we first define probabilities of nodes leaving and rejoining the network for the three defined node types, as described in section 3.1.2. The simulation was executed using an exponential distribution with mean probabilities set to the values listed in Table 2. The column labeled average node availability is the measured average percentage of time the nodes of that type were available across the lifetime of the network.

The purpose of this experiment is to determine how SCAR performs as the network becomes unstable. To simulate this behavior, we start with a network of only *Infras-*

tructure nodes and begin adding *Peepers* to the network, thus decreasing the average available of the DHT nodes. As can be observed in Figure 12, when the average node availability is above 80%, SCAR's data availability is comparable with replication; however when availability is below 80%, SCAR's data availability becomes un-acceptable. This is identical to the prediction made by our analytical models, and confirms that SCAR is only effective when the network is made of mostly available nodes.

type	$\bar{\alpha}$	$\bar{\beta}$	$\bar{A}$
Infrastructure	1.0%	95.0%	98.0%
Power Users	20.0%	40.0%	65.0%
Peepers	80.0%	10.0%	15.0%

Table 2: Network Sensitivity Analysis Settings. $\bar{\alpha}$ and $\bar{\beta}$ refers to ON/OFF model in Figure **??**.

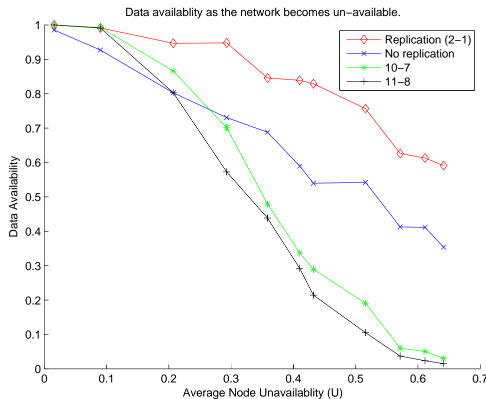

Figure 12: SCAR Simulation as network becomes unstable. Each data point represents the mean data availability of 10 objects over 200 simulation runs.

4.5 Recovery Likelihood and Availability

During the analysis and experiments, we have seen that SCAR does not perform well when the nodes in the network are unreliable. Observe that availability is dependent upon the user. When the user requests the data it must be available, but that doesn't mean the data has to be available at all time points. This reasoning led us to investigate the way SCAR performs if data only needs to be available a fraction of the time. Instead of verifying data availability at each time-point, we modified our simulation to check only a percentage of the time. In Figure 13, it can be seen that varying the probability of data checking has no effect upon the data availability. This is explainable because the overall availability would not be effected by the probability of accessing data. Therefore, to solve the data availability problem, one must make the data available at all times in order to increase the data availability for particular user.

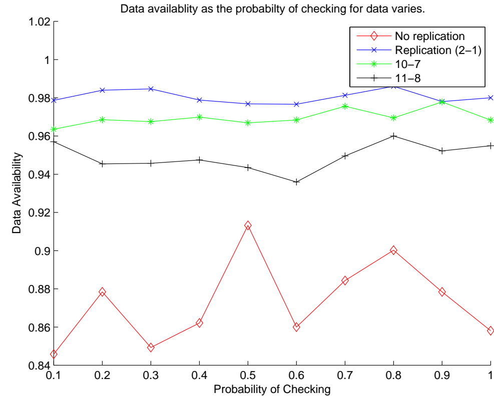

Figure 13: Data Availability as probability of checking for data varies. Network configuration (80% Infrastructure; 10% Power Users, 10% Peepers). Each data point represents the mean data availability for 10 data objects over 200 simulation runs.

5 Related Work

Research on DHT security has investigated both the maintenance of the DHT structure itself and the data storage system. A survey of the various routing attacks and other DHT infrastructure security concerns can be found in [1, 15]. At the storage level there are two security concerns, privacy of the user and the privacy of the users data. Privacy of the user is concerned with anonymity for both in what a user is viewing and posting within the network, this is addressed in systems such as Freenet [2]. Data privacy and security is concerned with access control and authorization of the data being stored within the DHT, which is the problem addressed in this paper. The typical solution to data privacy is to encrypt the data before inserting it into the DHT. While this does secure the data it is not adequate because data encryption could be broken. Other solutions rely upon a centralized authority to grant access to the stored data there are also systems that use multiple authorities [3]. SCAR addresses this problem in a unique way that is completely distributed and does not rely upon a cryptosystem or central authorities to provide data privacy.

Research is distributed storage has also addressed data privacy concerns. The work in POTSHARDS [17] makes use of secret sharing techniques to divide data but unlike SCAR the locations of these pieces are known, or approximately known. Also related is SafeStore [5] which uses erasure coding techniques to distribute data among multiple storage providers in order to increase reliability. SafeStore assumes that authentication and data privacy is handled by

the storage providers and focused on increasing data availability.

6 Conclusion

Peer-to-peer technologies promise to be a solution to the distributed storage problem, but current research has failed to fully address the problem of distributed data security. This paper presents a novel and elegant solution that provides both data security and availability using peer-to-peer networks. The framework presented combines hash chaining, erasure coding, and distributed hash tables to create a complete solution to a complex problem. A fully working implementation was developed thereby validating the protocols and design discussed. In addition, this paper provides an analysis of the proposed framework.

Analytical models were developed to model SCAR's security and availability characteristics. The analysis of these models showed us that the erasure coding techniques used by SCAR were only effective when the peer-to-peer node's availability is sufficiently high, $> 80\%$. This was a disappointment, however, this paper has provided the framework for evaluating new availability schemes. This work further emphasized the tradeoff between availability and security.

This paper presents a method for using peer-to-peer networks to securely and reliably store data using a novel combination of erasure coding and hash chaining. This paper also provides implementation details based upon the prototype that was developed showing the feasibility of SCAR. Beyond developing SCAR, this paper provides an analysis of the inherent tradeoff between security and availability. The exploration of this tradeoff lead to the conclusion that SCAR can effectively balance this tradeoff when the nodes of the network are sufficiently available.

References

[1] B. Awerbuch and C. Scheideler. Towards a scalable and robust dht. In *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 318–327, New York, NY, USA, 2006. ACM Press.

[2] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009, 2001.

[3] B. Crispo, S. Sivasubramanian, P. Mazzoleni, and E. Bertino. P-hera: Scalable fine-grained access control for p2p infrastructures. *icpads*, 01:585–591, 2005.

[4] N. M. Haller. The s/key one-time password system. In *In Proceedings of the Internet Society Symposium on Network and Distributed Systems*, 1994.

[5] R. Kotla, M. Dahlin, and L. Alvisi. Safestore: A durable and practical storage system. In *USENIX Annual Technical Conference ¡B¿Best paper award¡/B¿*, June 2007.

[6] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[7] F. MacWilliams and N. Sloane. *The Theory of Error-Correcting Codes*. North-Holland Publishing Company, 1977.

[8] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *J. ACM*, 36(2):335–348, 1989.

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM Press.

[10] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a dht. In *In Proc. of USENIX Technical Conference*, June 2004.

[11] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. Opendht: a public dht service and its uses. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 73–84, New York, NY, USA, 2005. ACM Press.

[12] R. Rodrigues and B. Liskov. High availability in dhts: Erasure coding vs. replication. In *IPTPS '05: 4th International Workshop on Peer-To-Peer Systems.*, February 2005.

[13] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.

[14] A. Shamir. How to share a secret. In *Communication of the ACM*. ACM Press, 1979.

[15] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash table. In *IPTPS '02*, 2002.

[16] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM Press.

[17] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. Secure, archival storage with potshards. In *FAST'07: Proceedings of the 5th conference on USENIX Conference on File and Storage Technologies*, pages 11–11, Berkeley, CA, USA, 2007. USENIX Association.