Countering Targeted File Attacks Using Location Keys

Mudhakar Srivatsa and Ling Liu
College of Computing, Georgia Institute of Technology
{mudhakar, lingliu}@cc.gatech.edu

Abstract
Serverless distributed computing has received significant attention from both the industry and research community. One of its typical applications is wide area network file systems like CFS [4], Farsite [2] and OceanStore [9]. A unique feature of these file systems is that they are *serverless*. They store files on a large collection of untrusted nodes that form an overlay network. They use cryptographic techniques to secure files from malicious nodes. However, most of these distributed file systems are vulnerable to targeted file attacks, wherein an adversary attempts to attack a small (chosen) set of files in the system. This paper presents *location keys* as a technique for countering targeted file attacks. Location keys can be used to not only provide traditional cryptographic guarantees like file confidentiality and integrity, but also (i) mitigate Denial-of-Service (DoS) and host compromise attacks, (ii) construct an efficient file access control mechanism, and (iii) add almost zero performance overhead and very minimal storage overhead to the system. We also study several potential inference attacks on location keys and present solutions that guard the file system from such attacks.

1 Introduction

A new breed of serverless file storage services have recently emerged like CFS [4], Farsite [2] and OceanStore [9]. In contrast to traditional file systems, they harness the resources available at desktop workstations that are distributed over a wide-area network. As the hardware resources become cheaper and cheaper, these desktop personal computers are turning out to be more and more powerful. The collective resources available at these desktop workstations amount to several peta-flops of computing power and several hundred peta-bytes of storage space [2].

These emerging trends have motivated serverless file storage as one of the major applications on an overlay network. An overlay network is a virtual network formed by nodes (desktop workstations) on top of an existing IP-network. Overlay networks typically support a lookup protocol. A lookup operation identifies the location of a file (say, the IP-address of a node that hosts the file) given its filename. There are four important issues to be addressed by serverless file systems.

Efficiency of the lookup protocol. There are two kinds of lookup protocol that have been popularly deployed, namely, Gnutella-like broadcast based lookup protocol ([6]) and distributed hash table (DHT) based lookup protocols ([19], [16], [17]). File systems like CFS, Farsite and OceanStore use DHT-based lookup protocols because of their ability to locate the destination node in a small and bounded number of hops.

Malicious and unreliable nodes. Serverless file storage services are faced with the challenge of having to harness the collective resources of loosely coupled, insecure, and unreliable machines to provide a secure, and reliable file-storage service. To complicate matters further, some of the nodes in the overlay network could be malicious. Hence, wide-area file storage services must be capable of handling malicious behavior by a small fraction of nodes in the system. CFS employs cryptographic techniques to maintain file data confidentiality and integrity. Farsite permits file write and update operations by using a Byzantine fault-tolerant group of meta-data servers (directory service). Both CFS and Farsite use replication as a technique to provide higher fault-tolerance and availability.

Targeted Attacks (DoS and Host compromise attacks). A major drawback with serverless file systems like CFS, Farsite and OceanStore is that they are vulnerable to targeted attacks on files. In a targeted attack, an adversary is interested in compromising a small set of files (target files). The fundamental problem with these systems is that: (i) the number of replicas (R) maintained by the system is usually much smaller than the number of malicious nodes (B), and (ii) the replicas of a file are stored at *publicly known* locations. Hence, malicious nodes can easily attack the set of R replica holders of a target file ($R \ll B$). A denial-of-service attack would render the target file unavailable; a host compromise attack could corrupt all the replicas of a file thereby effectively wiping out the target file from the file system.

Efficient Access Control. Access control in large scale distributed systems has long been a very critical problem. A read-only file system like CFS can exercise access control by simply encrypting the files. Farsite, a read-write file system, exercises access control using access control list (ACL) that are maintained using a Byzantine-fault-tolerant protocol. Users are required to be authenticated and authorized by a directory-group before their write requests are executed. However, access control is not truly distributed in Farsite because all users need to be authenticated by a small collection of directory-group servers. Further, PKI (public-key Infrastructure) based authentication and Byzantine-fault-tolerance based authorization can be very expensive.

In this paper we present *location keys* as a technique that not only provides traditional cryptographic guarantees like data confidentiality and integrity, but also (i) mitigates DoS and host compromise attacks, (ii) provides a capability-based file access control mechanism, and (iii) incurs almost zero performance overhead and minimal storage overhead. The fundamental idea is to *hide* the very location of the file replicas such that, a legal user who possesses a file's location key can easily locate it; without knowing the file's location key, an adversary would not be able to even locate the file, let alone access it or attempt to attack it. We implement an efficient capability-based file access control mechanism [3] using location keys. A location key acts as a capability (handle or token) to access a file's replicas. We study several inference attacks on location keys, and present techniques to guard the file system from such attacks.

2 Background

In this section we briefly overview the vital properties of DHT-based overlay networks and their lookup protocols (Chord [19], CAN [16], Pastry [17]). All these systems are fundamentally based on distributed hash tables, but differ in algorithmic and implementation details. All of them store the mapping between a particular key and it's associated $data$ in a distributed manner across the network, rather than storing them at a single location like a conventional hash table. This is achieved by maintaining a small routing table at each node. Given a key, these techniques guarantee the location of it's associated $data$ in a small and bounded number of hops within the overlay network. To achieve this, each node is given an identifier and is made responsible for a certain set of keys. This assignment is typically done by normalizing the key and the node identifier to a common space (like hashing them using the same hash function) and having policies like numerical closeness or contiguous regions between two node identifiers, to identify the regions which each node will be responsible for.

For example, in the context of a file system, the key can be a filename and the identifier can be the IP address of a node. All the available node's IP addresses are hashed using a hash function and each of them store a small routing table (for example, Chord's routing table has only m entries for an m-bit hash function and typically $m = 128$) to locate other nodes. Now, to locate a particular file, its filename is hashed using the same hash function and depending upon the policy, the node responsible for that file is obtained. This operation of locating the appropriate node is called a *lookup*.

File system like CFS, Farsite and OceanStore are layered atop of DHT-based protocols. These file systems typically provide the following guarantees:

- A lookup operation for any file is guaranteed to succeed if and only if the file is present in the system.
- A lookup operation is guaranteed to terminate within a small and bounded number of hops.
- The files are uniformly (statistically) divided among all currently active nodes.
- The system is capable of handling dynamic node joins and leaves.

In the following sections of this paper, we use Chord [19] as the overlay network's lookup protocol. However, the results obtained in this paper are applicable for most DHT-based lookup protocols.

3 Threat Model

Adversary refers to a logical entity that controls and coordinates all actions by malicious nodes in the system. A node is said to be malicious if the node either intentionally or unintentionally fails to follow the system's protocols correctly. For example, a malicious node may corrupt the files assigned to them and incorrectly (maliciously) implement file read/write operations. By the very definition of our adversary, we permit collusions among malicious nodes.

We assume that the underlying IP-network layer is secure. Hence, (i) A malicious node has access only to the packets that have been addressed to it, (ii) All packets that can be accessed by a malicious node can be potentially modified (corrupted) by the malicious node. More specifically, if the packet is not encrypted (or does not include a message authentication code (MAC)) then the malicious node may modify the packet in its own interest, and (iii) The underlying domain name service (DNS), the network routers, and the related networking infrastructure are assumed to be completely secure, and hence cannot be compromised by an adversary.

An adversary is capable of performing two types of attacks on the file system, namely, the denial-of-service attack, and the host compromise attack. When a node is under denial-of-service attack, the files stored at that node are unavailable. When a node is compromised, the files stored at that node could be either unavailable or corrupted. We model the malicious nodes as having a large but bounded amount of physical resources at their disposal. More specifically, we assume that a malicious node may be able to perform a denial-of-service attack only on a finite and bounded number of good nodes, denoted by α. We limit the rate at which malicious nodes may compromise good nodes. We use λ to denote the mean rate per malicious node at which a good node can be compromised. Hence, when there are B malicious nodes in the system, the net rate at which good nodes are compromised is $\lambda * B$ *node compromises per unit time*. Every compromised node behaves maliciously. For instance, a compromised node may attempt to compromise other good nodes. Every good node that is compromised would independently recover at rate μ; hence, if C denotes the number of compromised nodes then the net rate of recovery is $\mu * C$ *node recoveries per unit time*. Note that recovery of a compromised node is analogous to cleaning up a virus or a worm from an infected node. When the recovery process ends, the node stops behaving maliciously. Unless and otherwise specified we assume that the rates λ and μ follow an exponential distribution.

Files on an overlay network have two primary attributes: (i) *content* and (ii) *location*. File content could be protected from an adversary using cryptographic techniques. However, if the location of a file on the overlay network is publicly known, then the file holder is susceptible to DoS and host compromise attacks. Location keys attempt to hide files in an overlay network such that only a legal user who possesses a file's location key can easily locate it. Thus, any previously known attacks on file contents would be applicable only after the adversary is successful in locating the file. Location keys are oblivious to whether or not file contents are encrypted. Hence, location keys can be used to protect files whose contents cannot be encrypted (say, to permit arbitrary keyword search on file contents).

4 Securing Files on the Overlay Network

In this section, we study targeted attacks on file storage applications on wide-area overlay networks. We present location keys based technique to secure the file system from targeted attacks by hiding the location of a file's replicas. We then present an extensive analysis of our proposed solution and study several inference attacks on the proposed solution.

4.1 Targeted File Attacks

Targeted file attack refers to an attack wherein an adversary attempts to attack a small (chosen) set of files in the system. An attack on a file is successful if the target file is either rendered unavailable or corrupted. A file is unavailable (or corrupted) if at least a threshold number of its replicas are unavailable (or corrupted). We use corruption threshold (cr) to denote the *minimum* number of replicas if attacked would make it impossible to operate on the file. For example, for read/write files maintained by a Byzantine quorum [2], $cr = \lceil R/3 \rceil$. For encrypted and authenticated files, the corruption threshold $cr = R$, since any file can be successfully recovered as long as at least one of the available replicas is uncorrupted [4]. A P2P trust management systems such as [1, 22] uses a simple majority vote on the replicas to compute the actual trust value ($cr = \lceil R/2 \rceil$).

Distributed file systems like CFS and Farsite are highly vulnerable to target file attacks since the target file can be rendered unavailable (or corrupted) by attacking a *very small* set of nodes in the system. The key problem arises from the fact that these systems store the replicas of a file f at *publicly known* locations [8]. For instance, CFS stores a file f at locations identified by the public-key of its owner. An adversary can attack any set of cr replica holders of file f, to render file f unavailable (or corrupted). Farsite utilizes a small collection of publicly known nodes for implementing a Byzantine fault-tolerant directory service. On compromising the directory service, an adversary could obtain the locations of a target file's replicas. Note that the directory service can be compromised by simply compromising one-third of the nodes participating in the directory service.

Yet another important challenge in building a distributed file systems, is how to perform efficient access control. A read-only file system like CFS can implement access control by simply encrypting the file contents, and distributing the keys only to legal users of that file. Farsite permits read/write operations by implementing access control through a small collection of directory servers. The crucial draw back with CFS is that it does not permit file write operation; and that with Farsite is that it performs access control using a Byzantine-fault-tolerant protocol on a small set of publicly known directory servers, and that it requires the end user to be authenticated to the file system, say using PKI based digital certificates.

4.2 Location Keys

One way to mitigate target file attacks is to hide the location of the replicas of a file from an adversary. If the location of a file were *perfectly* hidden then an adversary would not be able to isolate any small subset of good nodes, attacking whom guarantees that the target file is under attack. In the following portions of this section we define location keys and demonstrate the role of location keys in guarding a file system from targeted file attacks.

Definition Lookup Algorithm: Lookup algorithm $A : f \rightarrow loc$ maps a file f to its location loc on an overlay network.

Definition Location Key: A location key lk is a relatively small amount (m-bit binary string, typically m = 128) of information that is used by a Lookup algorithm $A : (f, lk) \rightarrow loc$ to customize the transformation of a file into its location such that,

1. Given the location key of a file, it is very *easy* to locate the replicas of that file, that is, the lookup algorithm A is inexpensive.

2. Without knowing the location key of a file, it is very *hard* for an adversary to locate its replicas. Any adversarial lookup algorithm $B : f \rightarrow \{locs\}$ such that $A(f, lk) \in B(f)$ is prohibitively expensive unless $|locs|$ is $O(G)$, where G denotes the total number of good nodes in the system.

Informally, location keys are *keys with location hiding property*. Each file in the system is associated with a location key that is kept secret by the users of that file. A location key for a file f determines the locations of its replicas in the overlay network. Analogous to traditional cryptographic keys which make a file's contents unintelligible, location keys make the location of a file unintelligible. Property 1 ensures that valid users can easily access a file f; and Property 2 ensures that illegal users would not even able to locate the file on the overlay network, let alone access it.

We denote the cost of an adversarial lookup algorithm as the minimum value of g such that $Adv(g) \geq thr$, where $Adv(g)$ denotes the probability that cr or more replicas of the target file are under attack when the adversary actually launches attacks on g chosen good nodes and $0 < thr < 1$ is a system wide security parameter. Note that larger the value of threshold thr lower is the system's security level. When the location of the target file is known, $Adv(g) = 1$ for $g \geq cr$. On the other hand, with a good location hiding algorithm, one can ensure that $Adv(g) \geq thr$ if and only if g is $O(G)$. Informally, the cost of an attack when file locations are publicly known is $O(1)$; while that using location keys scales linearly with G ($O(G)$.

In the following portions of this paper, we address the following issues. (i) How to choose a location key lk? (ii) How to map a file f using its location key lk to its replica holders? (iii) How to lookup the location of a file without revealing its true identifier to the overlay network? (iv) How to use location keys to perform efficient access control?

4.3 Designing Location Keys

Let user u be the owner of a file f. User u chooses a long random bit string (128-bits) lk as the location key for file f. The location key lk should be hard to guess; the key lk should not be semantically attached (or derived) from the file name (f) or the owner name (u). Owner u securely distributes the location key lk only to those users who may access the file f. We assume that the file name f, its owner u and the valid users of file f are known to the adversary. We discuss key security, key distribution and key management issues in Section 6. The rest of this section focuses on the definition and usage of location keys.

In order to hide file f, the user u derives the identifiers of the replicas of file f as a pseudo-random function of the filename f and the location key lk. Note that a DHT-based system is capable of mapping identifiers to nodes that participate in the overlay network. We propose that these replica's locations be identified by $\{E_{lk}(f \parallel 0), E_{lk}(f \parallel 1), \cdots, E_{lk}(f \parallel R-1)\}$, where $E_{lk}(x)$ denotes a keyed pseudo-random function with input x and a secret key lk; and $\parallel$ denotes string concatenation. We require that the function E satisfies the following properties:

1a. Given $(f \parallel i)$ and lk it is very easy to compute $E_{lk}(f \parallel i)$.

2a. Given $(f \parallel i)$ it is very hard to obtain the identifier $E_{lk}(f \parallel i)$ without knowing the location key lk.

2b. Given $E_{lk}(f \parallel i)$ it is very hard to obtain the identity of file f.

2c. Given $E_{lk}(f \parallel i)$ and f it is very hard to obtain the location key lk.

Note that the hardness in breaking a pseudo-random function is expressed in terms of its computational complexity. Property 1a ensures that it is very easy for a valid user to locate a file f as long as it is aware of the file's location key lk. From Property 2a, it is very hard for an adversary to guess the location of a target file f without knowing its location key. Property 2b ensures that even if an adversary obtains an identifier $E_{lk}(f \parallel i)$, he/she cannot deduce the identity of file f. Property 2c ensures that even if an adversary obtains the identifiers of one or more replicas of file f, he/she would not be able to derive the location key lk from them. Hence, the adversary would still have no clue about the remaining replicas of the target file f (Property 2a). Properties 2b and 2c play an important role because some of the replicas of any file could be stored at malicious nodes in the system; and hence, an adversary indeed could be aware of some of the replica identifiers. Finally, observe that Property 1a and Properties {2a, 2b, 2c} map to properties 1 and 2 respectively in Section 4.2.

There are a number of cryptographic tools that satisfies our requirements specified in properties 1a, 2a, 2b and 2c. Some possible candidates for the function E are (i) a keyed-hash function like HMAC [7] [1], (ii) a symmetric key

[1]A keyed-hash function is a keyed one-way pseudo-random function wherein the output of the function depends on the input and a secret key

encryption algorithm like DES [5] or AES [12], and (iii) a PKI based encryption algorithm like RSA [10]. We chose to use a keyed-hash function like HMAC because it can be computed very efficiently (HMAC hashes can be computed about 40 times faster than an AES encryption and about 1000 times faster than RSA encryption using the standard OpenSSL library [13]). Note that the reversible mappings (decryption algorithms) made possible by symmetric and PKI based algorithms are not required for location hiding. Keyed-hash functions are in some sense a minimal cover of the properties warranted by our design. In the following portions of this paper, we use $khash$ to denote a keyed-hash function that is used to derive a file's replica identifiers from its name and its secret location key.

4.4 Location Keys based File Access Control

In this section we show that one can use location keys to design an efficient access control mechanism that permits file read and write operations. Location key based access control system does not directly authenticate the user attempting to access a file. Instead, an user who presents the correct *token* (a capability or a handle in Hydra [3]) is permitted to access the file. We use the replica identifier $khash_{lk}(f \parallel i)$ as the token for the i^{th} replica of file f. The file identifier perturbation technique (Section 4.5) permits the users to perform lookups without revealing the file capability/token to other nodes to the adversary.

Let us suppose a node r is responsible for storing the i^{th} replica of a file f. Internally, node r stores this file data under a file name $tk = khash_{lk}(f \parallel i)$. Note that node r does not have to know the actual file name (f) of a locally stored file tk. Indeed, given the internal file name tk, node r cannot guess its actual file name (from Property 2b of a keyed-hash function).

When a good node r receives a read/write request on a file tk it does the following. First, it checks if a file named tk is present locally. If so, it *blindly* performs the requested operation on the local file tk. Access control follows from the fact that it is very hard for an adversary to *guess* correct file tokens or infer them by *observing* the lookup traffic on the overlay network (Section 4.5). If node r were bad, then its response to a file read/write request is undefined. Recall that we have always assumed that the replicas stored at malicious nodes are under attack; hence, the fact that the adversary is aware of the tokens of those file replicas stored at malicious nodes or that a bad node's action on file read/write requests is undefined does not significantly harm the system.

On the other hand, an adversary cannot access any replica of file f stored at a good node simply because it cannot guess the token $khash_{lk}(f \parallel i)$ (from Property 2a of the keyed-hash function). However, when a good node is compromised an adversary would be able to directly obtain the tokens for all files stored at that node. In general, an adversary could compile a list of tokens as it compromises more and more good nodes; and corrupt the file replicas corresponding to these tokens at any later point in time. We condone compromised file tokens based attacks using a location re-keying technique described in Section 5.3.

4.5 Lookup Using File Identifier Perturbation

We have so far presented techniques to derive a file's replica identifiers from its location key. In this section, we present the lookup algorithm, namely $A(f, lk)$, used by a location keys based file system. We cannot use the unmodified Chord lookup algorithm because the lookup operation proceeds through a sequence of hops which might include malicious nodes; and revealing the file capabilities to these malicious nodes is a serious security breach because we use them to perform access control (see Section 4.4). Note that capability-based access control in Operating Systems (like Hydra [3]) relies either on a trusted hardware or a trusted OS kernel to perform a lookup, that is, retrieve the object that is associated with the capability. This section develops techniques to locate a replica on the overlay network without revealing its actual identifier (its capability) to other nodes on the overlay network.

When an user intends to query for some identifier $id = khash_{lk}(f \parallel i)$ (for some $1 \leq i \leq R$), the user actually performs a lookup on a perturbed identifier $id' = id - \triangle$, where $\triangle$ denotes a random perturbation added to the identifier id. The probability distribution for the random variable $\triangle$ is chosen such that: (i) *with high probability* if the result of a lookup on identifier id is node r, then the result of a lookup on the identifier id' is also node r, and (ii) given a perturbed identifier id' it is very hard for an adversary to guess the actual identifier id. Recall that the mean size of the identifier space assigned to any node is approximately $2^{128}/N \approx 2^{108}$ for $N = 1$ million nodes (see Chord [19]). This makes it possible for a query identifier to be perturbed and yet yield the same result. Now, the user would present the actual identifier id only to node r and not to any intermediary node along the lookup path to node r. If node r were malicious then it already aware of the file identifier id and hence it does not obtain any more information. If node r were good then it becomes *almost impossible* for an adversary to collect file tokens by observing lookup queries on the overlay network.

Now, we focus our attention on the following issues. (i) What is a *safe* range from which the perturbation parameter $\triangle$ is chosen? (ii) What happens if a perturbation is unsafe? (iii) Given a safe range, how to choose $\triangle$ each time a node wishes to perturb an identifier? We use theorem 4.1 to answer the first issue.

Theorem 4.1 *Let node r be the node that is the immediate predecessor for an identifier id on the Chord ring. Let $dist(x, y)$ denote the distance between two identifiers x and y on a Chord's unit circle. Let $ID(r)$ denote the identifier of the node r. Let N denote the total number of nodes in the system. Then, the probability that the distance between identifiers id and $ID(r)$ exceeds $\frac{thr}{N}$ is given by $Pr(dist(k, ID(r)) > \frac{thr}{N}) = e^{-thr}$.*

Proof Let Z be a random variable that denotes the distance between a key k and node r be the immediate predecessor of the identifier id. Let $f_Z(x)$ denote the probability distribution function (pdf) that the node r is at a distance of x (on Chord's unit circle, $0 \leq x \leq 1$) from the identifier id, i.e., $dist(ID(r), id) = x$. Then $f_Z(x)$ is given by Equation 1.

$$f_Z(x) = N * (1 - x)^{N-1} \quad (1)$$

By the uniform and random distribution properties of the hash function the identifier of a node will be uniformly and randomly distributed between (0, 1). Hence, the probability that the identifier of any node falls in a segment of length x is equal to x. Equation 1 follows from the fact that the probability that there exists a node between distance x and $x + dx$ from identifier id is $N * dx$, and the probability that there is no other node within a distance x from identifier id is $(1 - x)^{N-1}$. Using the probability density function in Equation 1 one can derive the cumulative distribution function (cdf), $Pr(Z > \frac{thr}{N}) = e^{-thr}$. ∎

We use Theorem 4.1 to choose an appropriate threshold thr that bounds the perturbation $\triangle$. Let sq be a system defined parameter which denotes the minimum probability that a lookup on the actual identifier and the perturbed identifier results in the same physical node. Given sq, one could compute a *sq-safe* threshold thr_{safe} from $e^{-thr_{safe}} = sq$, i.e., $thr_{safe} = -\log_e(sq)$. The threshold thr_{safe} is said to be *sq-safe* since with a probability sq a perturbation $\triangle = \frac{thr_{safe}}{N}$ results in a perturbed identifier that is assigned to the same physical node as the actual identifier. Therefore, a safe range for file identifier perturbation would be $(0, \frac{thr_{safe}}{N})$. For instance, when we set $sq = 1 - 2^{-20}$ and $N = 1$ million nodes ($thr_{safe} = 2^{-20}$), $\triangle$ could be chosen from a range of size $rg = 2^{128} * \frac{thr_{safe}}{N} = 2^{128} * \frac{2^{-20}}{2^{20}} = 2^{88}$.

Now we address the second issue, namely, handling unsafe perturbations. Note that there is small (yet, non-zero) probability that perturbed identifier is not safe. If the user were not careful, the file identifier (token) might be exposed to some other node $r' \neq r$ (where r' is the result of a lookup operation on an unsafe perturbed identifier id'). In particular, if r' were malicious, then it could misuse this information to corrupt the file replica stored at node r. However, one can verify whether a perturbed identifier is safe or not using the following check: A perturbed identifier id' is safe if and only if $id \leq ID(r')$. Since $id' \leq id \leq ID(r')$, node r' should be the immediate successor of identifier id and thus be responsible for it. If $id > ID(r')$ then node r' is definitely not a successor of identifier id and can be flagged as an unsafe perturbation. If a perturbed identifier turns out to be unsafe, the user might have to retry the lookup operation with a new perturbed identifier. Since, the probability of an unsafe perturbation is extremely small, we found from our experiments that the number of retries required is almost always zero and seldom exceeds one. Figure 1 summarizes our *errorless probabilistic* algorithm for file identifier perturbation.

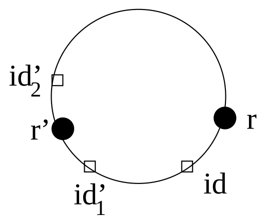


Figure 1: Lookup Using File Identifi er Perturbation: Let id be the actual fi le identifi er and the result of a lookup on identifi er id results in node r ($lookup(id) = r$). Identifi er id'_1 is a safe perturbation of identifi er id because, $lookup(id'_1) = lookup(id) = r$. Identifi er id'_2 is an unsafe perturbation of identifi er id because, $lookup(id'_2) = r' \neq r$. However, perturbed identifi er id'_2 can be detected to be unsafe because $ID(r') < id$.

Now, we address the third issue. Given a safe range $(0, rg)$ ($rg = 2^{88}$ in the example discussed above) $\triangle$ is chosen as follows. In general one could use any probability density function over the range to pick $\triangle$ each time a node wishes to perturb a file identifier. We propose that $\triangle$ be picked using a uniform and random distribution over the range rg. In fact, if one uses uniform and random distribution, given a perturbed identifier id' the adversary can infer nothing more than the fact that the actual file identifier lies in the range $(id', id' + rg)$. Using any other distribution would permit an adversary to statistically identify sub-ranges in $(id', id' + rg)$ that are more likely to contain the actual file identifier. For instance, if one used an Geometric distribution to draw $\triangle$ from $(0, rg)$ then, the probability that the actual file identifier equals x monotonically decreases as x varies from id' to $id' + rg$.

In summary, by choosing $\triangle$ uniformly and randomly over a huge safe range, we make it practically infeasible for an adversary to guess the actual identifier from a perturbed identifier. In the next section, we analytically show the hardness of breaking our file identifier perturbation technique.

4.6 Range Sieving Attack

In this section, we present a range sieving attack on perturbed file identifiers. The aim of this attack is to deduce the target file identifier (or a small range within which the target file identifier lies) from the perturbed identifiers. We use the range sieving attack to formally quantify the hardness of breaking our perturbation technique.

When an adversary obtains a perturbed identifier id', the adversary knows that the actual capability id is definitely within the range $RG = (id', id' + rg)$, where rg denotes the maximum perturbation that could be $sq-$safely added to a query identifier ($rg = 2^{88}$ in the example discussed in Section 4.5). In fact, given a perturbed identifier id', the adversary knows *nothing more* than the fact that the actual identifier id could be uniformly and distributed over the range $RG = (id', id' + rg)$. However, when the adversary obtains multiple perturbed identifiers that belong to a target file, the adversary can *sieve* the identifier space as follows. Let $RG_1, RG_2, \cdots, RG_{nid}$ denote the ranges corresponding to nid random perturbations on the identifier id. Then the capability of the target file is guaranteed to lie in the sieved range $RG_s = \cap_{j=1}^{nid} RG_j$. Intuitively, if the number of identifiers nid increases, the size of the sieved range RG_s decreases. Theorem 4.2 shows the relationship between the sieved range RG_s and the number of perturbed identifiers nid.

Theorem 4.2 *Let nid denote the number of perturbed identifiers that correspond to a target file. Let RG_s denote the sieved range using the range sieving attack. Let rg denote the maximum perturbation that could be $sq-$safely added to a file identifier. Then, the expected size of range RG_s is given by $E[|RG_s|] = \frac{rg}{nid}$.*

Proof Let $id'_{max} = id - \triangle_{max}$ and $id'_{min} = id - \triangle_{min}$ denote the minimum and the maximum value of a perturbed identifier that has been obtained by an adversary. Then, the sieved range $RG_s = (id'_{min}, id'_{max} + rg)$, namely, from the highest lower bound to the lowest upper bound. The sieved range RG_s can be partitioned into two ranges RG_{min} and RG_{max}, where $RG_{min} = (id'_{min}, id)$ and $RG_{max} = (id, id'_{max} + rg)$.

The size of the range RG_{min} is $|RG_{min}| = id - id'_{min} = \triangle_{min}$. The cumulative distribution function of $\triangle_{min}$ is given by Equation 2.

$$Pr(\triangle_{min} > x) = \left(1 - \frac{x}{rg}\right)^{nid} \qquad (2)$$

Since a perturbation $\triangle$ is chosen uniformly and randomly over a range $(0, rg)$, the probability any perturbation $\triangle$ is greater than x is $Pr(\triangle > x) = 1 - \frac{x}{rg}$. Hence, the probability that $\triangle_{min} = min\{\triangle_1, \triangle_2, \cdots, \triangle_{nid}\}$ is greater than x is $Pr(\triangle_{min} > x) = Pr((\triangle_1 > x) \wedge (\triangle_2 > x) \wedge \cdots \wedge (\triangle_{nid} > x)) = \prod_{j=1}^{nid} Pr(\triangle_j > x)$. Now, using standard techniques from probability theory, the expected value of $\triangle_{min}$ is $E[|RG_{min}|] = E[\triangle_{min}] \approx \frac{rg}{nid}$. Symmetrically, one can show that the expected size of range RG_{max} is $E[|RG_{max}|] \approx \frac{rg}{nid}$. Hence the expected size of sieved range is $E[RG_s] = E[RG_{min}] + E[RG_{max}] \approx \frac{rg}{nid}$. $\blacksquare$

$1 - sq$	2^{-10}	2^{-15}	2^{-20}	2^{-25}	2^{-30}
rg	2^{98}	2^{93}	2^{88}	2^{83}	2^{78}
hardness (years)	2^{38}	2^{33}	2^{28}	2^{23}	2^{18}

Table 1: Hardness of breaking the perturbation scheme with $N = 1$ million nodes, $sq = 1 - 2^{-20}$ safety and $nid = 2^{25}$ perturbed identifi ers

The range sieving attack makes it easier for an adversary to obtain the file identifier, when compared to a brute force attack on the entire range of size rg. Note that in a brute force attack, if an adversary obtains nid perturbed identifiers then the expected size of range left to be attacked is $|RG_s| = rg - nid$. However, even with range sieving attack it is practically infeasible for an adversary to obtain the file capability. Let the maximum perturbation $rg = 2^{88}$ (using the same settings for the probability of a safe query (sq) and the number of nodes (N) in Section 4.5). Let us suppose that the target file is accessed once a second for one year; this results in 2^{25} file accesses. An adversary who logs perturbed identifiers over an year could sieve the range to about $E[RG_s] = 2^{63}$. Assuming that the adversary performs a brute force attack on the sieved range, by attempting a file read operation at the rate of one a millisecond, then it would take the adversary about 2^{28} years to discover the actual file identifier. Table 1 summarizes the hardness of breaking the perturbation scheme for different values of sq (maximum probability of safe perturbation) assuming that the adversary has logged 2^{25} file accesses (one access per second for one year) and that the nodes permit not more one file access per millisecond.

An interesting observation that follows from the above discussion is that, amount of time taken to break the file identifier perturbation technique is almost independent of the number of attackers. This because the time taken for a brute force attack on a file identifier is fundamentally limited by the rate at which a hosting node permits accesses on files stored locally. On contrary, a brute force attack on a cryptographic key is inherently parallelizable and thus becomes more powerful as the number of attackers increases. Nonetheless, cryptographic algorithms like HMAC, AES and ElGamal are not vulnerable to the range sieving attack.

5 Countering Common Inference Attacks

We have discussed location key based schemes to effectively hide files on an overlay network. We achieve strong hiding properties by associating each file with a small location key that is kept secret by the users of that file. The properties of location keys ensure that an user who knows a file's location key can easily locate it; the location of a file is otherwise unintelligible for an adversary.

In this section, we present two common inference attacks: *passive inference attacks* and *host compromise based inference attacks*; and propose techniques to mitigate them. Inference attacks refer to those attacks wherein an adversary attempts to infer the location of a file using *indirect* techniques. Passive inference attacks refer to those attacks wherein an adversary attempts to infer the location of a target file by passively observing the file system. Host compromise based inference attacks require the adversary to perform an active host compromise attack before it can infer the location of a target file. In the following portions of this section, we study three passive inference attacks and two host compromise based inference attacks on a location keys based file system. It is very important to note that that none of the inference attacks described below would be effective in the absence of collusion among malicious nodes.

5.1 Passive Inference Attacks

Our first attack is based on the ability of malicious nodes to observe the frequency of lookup queries on the overlay network; more specifically, malicious nodes may log lookup queries routed through them and send them to an adversary. Assuming that the adversary knows the relative file popularity, it can perform a *lookup frequency inference attack*. We also study two other potentially possible inference attacks on location keys:end-user IP-address inference attack and file access pattern inference attack.

5.1.1 Lookup Frequency Inference Attack

In this section we present lookup frequency inference attack that would help a strategic adversary to infer the location of a target file on the overlay network. It has been observed that the general popularity of the web pages accessed over the Internet follows a Zipf-like distribution [22]. An adversary may study the frequency of file accesses by sniffing lookup queries and match the observed file access frequency profile with a actual (pre-determined) frequency profile to infer the location of a target file [2]. Note that if the frequency profile of the files stored in the file system is flat (all files are accessed with the same frequency) then an adversary will not be able to infer any information. Lemma 5.1 formalizes the notion of perfectly hiding a file from frequency inference attack.

[2]This is analogous to performing a frequency analysis attack on old symmetric key ciphers like the Caesar's cipher [21]

Lemma 5.1 *Let F denote the collection of files in the file system. Let λ'_f denote the apparent frequency of accesses to file f as perceived by an adversary. Then, the collection of files are perfectly hidden from frequency analysis attack if $\lambda'_f = c : \forall f \in F$ and some constant c.*

A collection of read-only files can be perfectly hidden from frequency inference attack. Let λ_f denote the actual frequency of accesses on a file f. Set the number replicas for file f to be proportional to its access frequency, namely $R_f = \frac{1}{c} * \lambda_f$ (for some constant $c > 0$). When a user wishes to read the file f, the user randomly chooses one replica of file f and issues a lookup query on it. From an adversary's point of view it would seem that the access frequency to all the read-only files in the system is identical ($\lambda'_f = \frac{\lambda_f}{R_f} = c$). By Lemma 5.1, an adversary would not be able to derive any useful information from a frequency inference attack. Interestingly, this replication strategy also improves the performance and load balancing aspects of the file system. However, it is not applicable to read-write files since an update operation on a file may need to update all the replicas of a file. In the following portions of this section, we propose two techniques to flatten the *apparent* frequency profile of read/write files.

Defense by Result Caching
The first technique to mitigate the frequency inference attack is to perturb the apparent file access frequency with lookup *result caching*. Lookup result caching, as the name indicates, refers to caching the results of a lookup query. Recall that wide-area network file systems like CFS, Farsite and OceanStore permit nodes to join and leave the overlay network. Let us for now consider only node departures. Consider a file f stored at node n. Let λ_f denote the rate at which users accesses the file f. Let μ_{dep} denote the rate at which a node leaves the overlay network (rates are assumed to be exponentially distributed). The first time the user accesses the file f, the lookup result (namely, node n) is cached. The lookup result is implicitly invalidated when the user attempts to access file f the first time after node n leaves the overlay network. When the lookup result is invalidated, the user issues a fresh lookup query for file f. One can show that the apparent frequency of file access as observed by an adversary is $\lambda'_f = \frac{\lambda_f \mu_{dep}}{\lambda_f + \mu_{dep}}$. The probability that any given file access results is a lookup is equal to the probability that the node responsible for the file leaves before the next access and is given by $Pr_{lookup} = \frac{\mu_{dep}}{\lambda_f + \mu_{dep}}$. Hence, the apparent file access frequency is equal to the product of the actual

file access frequency (λ_f) and the probability that a file access results in a lookup operation (Pr_{lookup}). Intuitively, in a static scenario where nodes never leave the network ($\mu_{dep} \ll \lambda_f$), $\lambda'_f \approx \mu_{dep}$; and when nodes leave the network very frequently ($\mu_{dep} \gg \lambda_f$), $\lambda'_f \approx \lambda_f$. Hence, more static the overlay network is, harder it is for an adversary to perform a frequency inference attack since it would appear as if all files in the system are accessed at an uniform frequency μ_{dep}.

It is very important to note that a node m storing a file f may infer its name since the user has to ultimately access node m to operate on file f. Hence, an adversary may infer the identities of files stored at malicious nodes. However, it would be very hard for an adversary to infer files stored at good nodes.

Defense by File Identifier Perturbation

The second technique that makes the frequency inference attack harder is based on the file identifier perturbation technique described in Section 4.5. Let $f_1, f_2, \cdots, f_{nf}$ denote the files stored at some node n. Let the identifiers of these replicas be $id_1, id_2, \cdots id_{nf}$. Let the target file be f_1. The key idea is to perturb the identifiers such that an adversary would not be able to distinguish between a perturbed identifier intended for locating file f_1 and that for some other file f_j ($2 \leq j \leq nf$) stored at node n.

More concretely, when a user performs a lookup for f_1, the user would choose some random identifier in the range $R_1 = (id_1 - \frac{thr_{safe}}{N}, id_1)$. A clever adversary may *cluster* identifiers based on their numerical closeness and perform a frequency inference attack on these clusters. However, one could defend the system against such a clustering technique by increasing the perturbation added to identifiers. Figure 2 presents the key intuition behind this idea diagrammatically. As the range R_1 overlaps with the ranges of more and more files stored at node n, the clustering technique and consequently the frequency inference attack would perform poorly. Let $R_1 \cap R_2$ denote the set of identifiers that belongs the intersection of ranges R_1 and R_2. Then, given an identifier $id \in R_1 \cap R_2$, an adversary would not able to distinguish whether the lookup was intended for file f_1 or f_2; but the adversary would definitely know that the lookup was intended either for file f_1 or f_2. Observe that amount of information inferred by an adversary becomes poorer and poorer as more and more ranges overlap. Also, as the number of files (nf) stored at node n increases, even a small perturbation might introduce significant overlap between ranges of different files stored at node n.

The apparent access frequency of a file f is computed as a weighted sum of the actual access frequencies of all files that share their range with file f. For instance, the apparent access frequency of file f_1 (see Figure 2) is given

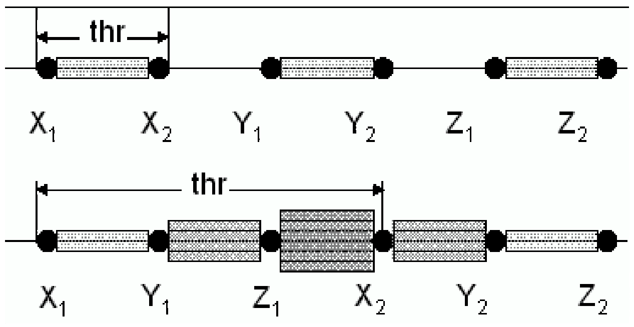


Figure 2: Countering Frequency Analysis Attack by adding more $\triangle$. X_1X_2, Y_1Y_2 and Z_1Z_2 denote the ranges of the perturbed identifi ers of fi les f, f_2, f_3 stored at node n. Frequency inference is very effective in scenario (i), but largely ineffective in scenario (ii). Given an identifi er $id \in Y_1Z_1$, it is hard for an adversary to guess whether the lookup was intended for fi le f_1 or f_2.

by Equation 3.

$$\lambda'_{f_1} = \frac{X_1Y_1 * \lambda_{f_1} + Y_1Z_1 * \left(\frac{\lambda_{f_1}+\lambda_{f_2}}{2}\right) + Z_1X_2 * \left(\frac{\lambda_{f_1}+\lambda_{f_2}+\lambda_{f_3}}{3}\right)}{thr_{safe}} \quad (3)$$

Clearly, the apparent access frequency of a file evens out the sharp variations between the frequencies of different files stored at a node, thereby making frequency inference attack significantly harder. We discuss more on how to quantify the effect of this perturbation on frequency inference attack in our experimental section 7.

5.1.2 Other Passive Inference Attacks

In this section we highlight two other potentially possible passive inference attacks: (i) end-user IP-address inference attack, and (ii) file access pattern inference attack.

The first attack is based on the assumption that the identity of an end-user can be inferred from his/her IP-address. An user typically locate their files on the overlay network by issuing a lookup query to some node r on the overlay network. If node r were malicious then it may log the file identifiers looked up by an user. Assuming that a user accesses only a small subset of the total number of files on the overlay network (including the target file) the adversary can narrow down the set of nodes on the overlay network that may potentially hold the target file.

One possible solution is for users to issue lookup queries through a trusted *anonymizer*. The anonymizer accepts lookup queries from users and dispatches it to the overlay network without revealing the user's IP-address. However, the anonymizer node could itself become a viable target for the adversary. The second and a more promising solution is for the user to join the overlay network (just like other nodes hosting files on the overlay network). When the user issues lookup queries, it is routed through some of its neighbors;

if some of its neighbors are malicious, then they may log these lookup queries. However, it is a non-trivial task for an adversary to distinguish between the queries that *originated* at the user and those that were simply *routed* through it.

The second attack is based on file access pattern. For instance, let us suppose that a target file is usually accessed at 9:30am every day (the target file may have its own *fingerprint* access pattern that uniquely identifies it). An adversary may log all lookups made between 9:28am and 9:30am. This would help an adversary to narrow down the possible identities of the target file. Further, if an adversary repeats this attack over a couple of days, then the chance of identifying the target file improves; while the target file is accessed everyday at 9:30am, the rest of the files accessed on one day at 9:30am are probably not accessed on other days. We acknowledge that such attacks are in general hard to be defended. However, we believe that such attacks are not applicable to a vast majority of files stored on the overlay network. We propose location re-keying 5.3 as a technique to mitigate both *known and unknown* inference attacks.

5.2 Host Compromise based Inference Attacks

In this section, we discuss two host compromise based inference attacks. Recall that when an adversary compromises a good node, the internal state of that node is completely exposed to the adversary. An adversary can pool information from such compromised nodes to perform inference attacks. In this section, we present file replica inference attack and file size inference attack

5.2.1 File Replica Inference Attack

Despite making the file capabilities and file access frequencies appear random to an adversary, the contents of a file could by itself reveal the identity of the file f. The file f could be encrypted to rule out the possibility of identifying a file from its contents. Even when the replicas are encrypted, the fact that all the replicas of file f are identical. When an adversary compromises a good node, it can extract a list of identifier and file content pairs (or a hash of the file contents) stored at that node. Note that an adversary could perform a frequency inference attack on the replicas stored at malicious nodes and infer their filenames. Hence, if an adversary were to obtain the encrypted contents of one of the replicas of a target file f, he/she could examine the extracted list of identifiers and file contents to obtain the identities of other replicas. This attack is especially more plausible on read-only files since their contents do not change over a long period of time. The update frequency on read-write files might guard them file replica inference attack. For read-only files, one could easily make the replicas non-identical by encrypting each replica with a different cryptographic key. In our experimental section, we study the probability of a successful attack using file replica inference analysis attack for varying file update frequencies.

5.2.2 File Size Inference Attack

File size inference attack is based on the assumption that an adversary might be aware of the target file's size. Malicious nodes (and compromised nodes) report the size of the files stored at them to an adversary. If the sizes of files stored on the overlay network follow a skewed distribution, the adversary would be able to identify the target file (much like the lookup frequency inference attack). One can mitigate this attack by fragmenting files into multiple file blocks of equal size. For instance, CFS divides files into blocks of 8KBytes each and stores each file block separately. We could hide the location of the j^{th} block in the i^{th} replica of file f using a location key lk as $khash_{lk}(f \parallel i \parallel j)$. Now, since all file blocks are of the same size, it would be vary hard for an adversary to perform file size inference attack. It is interesting to note that file blocks are also useful in minimizing the communication overhead for small reads/writes on large files.

5.3 Location Re-Keying

In addition to the inference attacks discussed in this paper, there could be other possible inference attacks on a location key based file system. Further, when a malicious node compromises a node, an adversary can obtain all file tokens stored at that node; the adversary may also use the compromised node to perform inference attacks. In course of time the adversary might be able to gather enough information to attack a target file.

To mitigate this virtually unpreventable leakage of information, users need to periodically choose new location keys so as to render *all* past inferences made by the adversary *useless*. This is analogous to periodic re-keying of cryptographic keys. Unfortunately, re-keying is an expensive operation: re-keying cryptographic keys requires data to be re-encrypted; re-keying location keys requires files to be relocated on the overlay network. Hence, it is important to keep the re-keying frequency small enough to reduce performance overheads and large enough to secure files on the overlay network.

There are two fundamentally different ways to implement location re-keying. (i) At periodic time instants, all the files stored on the overlay network are re-keyed. (ii) Each file in the system is independently re-keyed. The first technique causes heavy bursts in network traffic and it forces all files to be re-keyed with the same periodicity. On the other hand, the second technique does not cause huge bursts in network traffic. Also, it permits critical files to be re-keyed more frequently than relatively less important files. In our experiments section, we estimate the periodicity with which location keys have be changed in order to reduce the probability of a successful attack on a target file.

6 Discussion

In our discussion so far, we have defined and demonstrated the usage of location keys. In this section, we briefly explore the key security, distribution and management aspects of location keys.

Key Security. We have assumed that it is the responsibility of the users to secure location keys from an adversary. If an user has to access 1000s of files then the user must be responsible for the secrecy of 1000s of location keys. One viable solution would be compile all location keys into one *keys-file*, encrypt the file and store it on the overlay network. The user now needs to keep only one location key that corresponds to the *keys-file* secret. This 128-bit location key could be protected using untamparable hardware devices, smartcards, etc.

Key Distribution. Secure distribution of keys has been a major problem in large scale distributed systems. The problem of distributing location keys is very similar to that of distributing cryptographic keys. Traditionally, keys have been distributed using out-of-band techniques. For instance, one could use PGP [15] based secure email service to transfer location keys from a file owner to file users.

Key Management. Managing location keys becomes as important issue when (i) an owner owns several thousand files, and (ii) the set of legal users for a file varies significantly with time. When the number of files owned becomes very large, the file owner may choose to group files and assign one location key for a group of files. Frequent changes to group membership could seriously impede the system's performance.

The major overhead for location keys arises from key distribution and key management. Also, location re-keying could be an important factor; frequent changes to the list of legal users who are permitted to access a file can further exacerbate this situation. Key security, distribution and management that leverage additional properties of location keys are a part of our ongoing research work. There are other issues that we have not addressed in this paper. Using a capability-based access control mechanism we run into the problem of a valid user illegally distributing the capabilities (tokens) to an adversary. In this paper we assume that all valid users are well behaved. Also, we do not address the robustness of the lookup protocol or the overlay network in the presence of malicious nodes; these issues are addressed else where [18].

7 Experimental Evaluation

In this section, we report results from our simulation based experiments to evaluate location keys approach for building secure wide-area network file systems. We implemented our simulator using a discrete event simulation [5] model. Our system comprises of about $N = 1024$ nodes; a random $p = 10\%$ of them are chosen to behave maliciously. We implemented the Chord lookup protocol [19] on the overlay network compromising of these nodes. We set the number of replicas of a file to be $R = 7$ and vary the corruption threshold cr in our experiments. Recall that our adversary model in Section 3 models malicious nodes as powerful nodes but with bounded resources. We use the parameter α as an upper bound on the number of nodes on whom a malicious node can perform a DoS attack. The parameter λ (measured in number of node compromises per unit time) limits rate at which a malicious node can compromise other good nodes in the system. The parameter μ (measured in number of node recoveries per unit time) limits the rate at which a compromised node can recover to an uncompromised state. In the following portions of this section, we present results from two sets of experiments. The first set shows the robustness of location key based techniques (Section 4) in countering targeted attacks. We measure robustness in terms of the effort required for an adversary to attack a target file on the overlay network. The second set of experiments shows the effectiveness of our solutions that prevent inference attacks (Section 5).

7.1 Location Keys

Operational Overhead.[3] We first quantify the performance and storage overheads incurred by location keys. Let us consider a typical file read/write operation. The operation consists of the following steps: (i) generate the file replica identifiers, (ii) lookup the replica holders on the overlay network, and (iii) process the request at the replica holders. Step (i) using location keys requires computations using the keyed-hash function, which otherwise would have required computations using a normal hash function. We found that the computation time difference between HMAC (a keyed-hash function) and MD5 (normal hash function) is negligibly small (order of a few microseconds) using the standard OpenSSL library [13]. Step (ii) involves a pseudo-random number generation (few microseconds using the OpenSSL library) and may require lookups to be retried in the event that the perturbed identifier turns out to be unsafe. Given that unsafe perturbations are extremely rare (one in a million in our example in Section 4.5) retries are occasionally required and thus they add virtually no overhead to the system. Step (iii) adds no overhead because our access check

[3]As measured on a 900 MHz Intel Pentium III processor running RedHat Linux 9.0

is virtually free. As long as the user can present the correct filename (token), the replica holder would honor a request on that file.

Now, let us compare the storage overhead at the users and the nodes that are a part of the overlay network. Users need to store an additional 128-bit location key along with other file meta-data. Even an user who uses 1 million files on the overlay network needs to store an additional 16MBytes of location keys. Further, there is no additional storage overhead on nodes that are a part of the overlay network. Note that one can continue to use the same number of replicas for reliability purposes.

Denial of Service Attacks. Figure 3 shows the probability of an attack for varying α and different values of corruption threshold (cr) (see $Adv(g)$ in Section 4.2). Without the knowledge of the location of file replicas an adversary is forced to attack (DoS) a random collection of nodes in the system and *hope* that that at least cr replicas of the target file is attacked. Observe that if the malicious nodes are more powerful (larger α) or if the corruption threshold cr is very low, then the probability of an attack increases. If an adversary were aware of the R replica holders of a target file then: (i) a weak collection of B malicious nodes ($B = 10\%$ of $N = 102$) with $\alpha = \frac{R}{B} \approx \frac{7}{102} = 0.07$ can easily attack the target file, or (ii) for a file system to handle $\alpha = 1$, it would require as large 100+ replicas to be maintained for each file. In this case, the effort required by an adversary to attack a target file is dependent on R, but independent of the number of good nodes in the system. On contrary, location key based techniques scale the hardness of an attack with the number of good nodes in the system.

Host Compromise Attacks. Second, we evaluate location keys against host compromise attacks. Our first experiment on host compromise attack shows the probability of a successful attack on the target file assuming that the adversary does not collect capabilities (tokens) stored at the compromised nodes. Hence, the target file is successfully attacked if cr or more of its replicas are stored at either malicious nodes or compromised nodes. Figure 4 shows the probability of an attack for different values of corruption threshold (cr) and varying $\rho = \frac{\mu}{\lambda}$ (measured in number of node recoveres per node compromise). We ran the simulation for a duration of $\frac{100}{\lambda}$ time units. Recall that $\frac{1}{\lambda}$ denotes the mean time required for one malicious node to compromise a good node. Note that if the simulation were run for infinite time then the probability of attack is always one. This is because, at some point in time, cr or more replicas of a target file would be assigned to malicious nodes (or compromised nodes) in the system.

When $\rho \leq 1$ the system is highly vulnerable; with high probability the target file can be attacked by an adversary. In contrast to the DoS attack that could tolerate powerful malicious nodes ($\alpha > 1$), the host compromise attack cannot tolerate a situation wherein the node compromise rate is higher than its recover rate ($\rho \leq 1$). This is primarily because of the cascading effect of host compromise attack. Larger the number of compromised nodes, higher is the rate at which other good nodes are compromised (see the adversary model in Section 3). Table 2 shows the mean fraction of good nodes (G') that are in an uncompromised state for different values of ρ.

ρ	0.5	1.0	1.1	1.2	1.5	3.0
G'	0	0	0.05	0.44	0.77	0.96

Table 2: Mean Fraction of Good Nodes in an Uncompromised State (G')

ρ	0.5	1.0	1.1	1.2	1.5	3.0
Re-keying Interval	0	0	0.43	1.8	4.5	6.6

Table 3: Time Interval between Location Re-Keying (normalized by $\frac{1}{\lambda}$ time units)

As we have mentioned in Section 4.4, the adversary could collect the capabilities (tokens) of the files stored at compromised nodes; these tokens can be used by the adversary at any point in future to corrupt the files using a simple write operation. Hence, our second experiment on host compromise attack measures the probability of a attack assuming that the adversary collects the file tokens stored at compromised nodes. Figure 5 shows the mean effort required to locate all the replicas of a target file ($cr = R$). The effort required is expressed in terms of the fraction of good that need to be compromised by the adversary to attack the target file.

Note that in the absence of location keys, an adversary needs to compromise at most R good nodes. Clearly, location key based techniques increase the required effort by several orders of magnitude. For instance, when $\rho = 3$, an adversary has to compromise 70% of the good nodes in the system before improving the probability of a successful attack to a nominal value of 0.1 even under the assumption that an adversary collects file capabilities from compromised nodes. Note that if an adversary compromises every good node in the system once, it gets to know the tokens of all files stored on the overlay network. In Section 5.3 we had proposed location re-keying to protect the system from such attacks. Periodicity of location re-keying can be derived from Figure 5. For instance, when $\rho = 3$, if a user wants to retain the attack probability below 0.1, the time interval between re-keying should equal the amount of time it takes for an adversary to compromise 70% of the good nodes in the system. Table 3 shows the time taken (normalized by $\frac{1}{\lambda}$) for an adversary to increase the attack probability on a target file to 0.1 for different values of ρ. Observe that if ρ increases, location re-keying can be more and more infrequent.

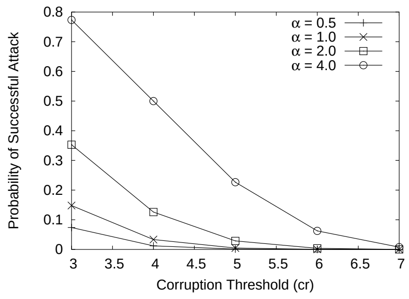


Figure 3: Probability of a Successful Target File Attack for $N = 1024$ nodes and $R = 7$ using DoS Attack

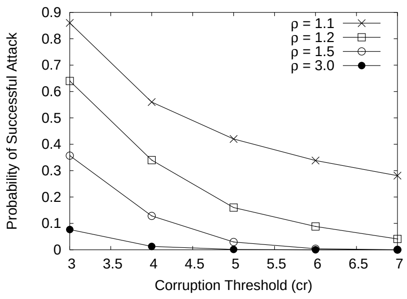


Figure 4: Probability of a Successful Target File Attack for $N = 1024$ nodes and $R = 7$ using Host Compromise Attack (with no token collection)

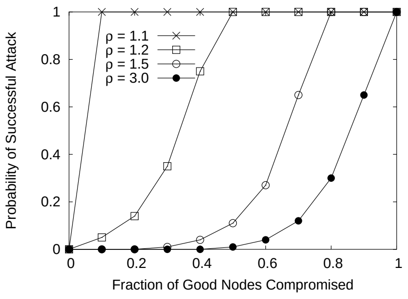


Figure 5: Probability of a Successful Target File Attack for $N = 1024$ nodes and $R = 7$ using Host Compromise Attack with token collection from compromised Nodes

7.2 Inference Attacks

In this section we experimentally measured the effectiveness of our solutions against the lookup frequency inference attack and the file replica inference attack.

Lookup Frequency Inference Attack. We have presented lookup result caching and file identifier perturbation as two techniques to thwart frequency inference attack. Recall that our solutions attempt to flatten the frequency profile of files stored in the system (see Lemma 5.1). Note that we do not change the actual frequency profile of files; instead we flatten the apparent frequency profile of files as perceived by an adversary. We assume that files are accessed in proportion to their popularity. File popularities are derived from a Zipf-like distribution [22], wherein, the popularity of the i^{th} most popular file in the system is proportional to $\frac{1}{i^\gamma}$ with $\gamma = 1$.

Our first experiment on inference attacks shows the effectiveness of lookup result caching in mitigating frequency analysis attack by measuring the *entropy* [11] of the apparent frequency profile (measured as number of bits of information). Given the apparent access frequencies of F files, namely, $\lambda'_{f_1}, \lambda'_{f_2}, \cdots, \lambda'_{f_F}$, the entropy S is computed as follows. First the frequencies are normalized such that $\sum_{i=1}^{F} \lambda'_{f_i} = 1$. Then, $S = -\sum_{i=1}^{F} \lambda'_{f_i} * \log_2 \lambda'_{f_i}$. When all files are accessed uniformly and randomly, that is, $\lambda'_{f_i} = \frac{1}{F}$ for $1 \leq i \leq F$, the entropy S is maximum $S_{max} = \log_2 F$. The entropy S decreases as the the access profile becomes more and more skewed. Note that if $S = \log_2 F$, no matter how clever the adversary is, he/she cannot derive any useful information about the files stored at good nodes (from Lemma 5.1). Table 4 shows the maximum entropy (S_{max}) and the entropy of a zipf-like distribution (S_{zipf}) for different values of F. Note that with every additional bit of entropy, doubles the effort required for a successful attack; hence, a frequency inference attack on a Zipf distributed 4K files is about 19 times ($2^{12-7.75}$) easier than the ideal scenario where all files are uniformly and randomly accessed.

F	4K	8K	16K	32K
S_{max}	12	13	14	15
S_{zipf}	7.75	8.36	8.95	9.55

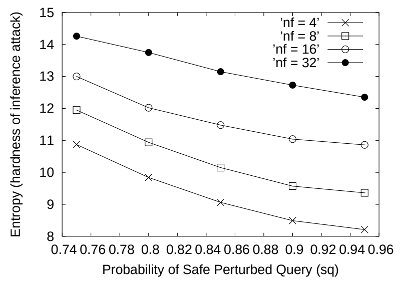
Table 4: Entropy (in number of bits) of a Zipf-distribution

μ_{dep}	0	1/256	1/16	1	16	256	∞
S	15	12.64	11.30	10.63	10.00	9.71	9.55

Table 5: Countering Lookup Frequency Inference Attacks Approach I: Result Caching (with 32K fi les)

Table 5 shows the entropy of apparent file access frequency as perceived by an adversary when lookup result caching is employed by the system for $F = 32K$ files. We assume that the actual access frequency profile of these files follows a Zipf distribution with the frequency of access to the most popular file (f_1) normalized to one access per unit time. Table 5 shows the entropy of the apparent lookup frequency for different values of μ_{dep} (the mean rate at which a node joins/leaves the system). Observe if μ_{dep} is large, the entropy of apparent file access frequency is quite close to that of Zipf-distribution (see Table 4 for 32K files); and if the nodes are more stable (μ_{dep} is small), then the apparent frequency of all files would appear to be identically equal to μ_{dep}.

In our second experiment, we show the effectiveness of file identifier perturbation in mitigating frequency inference attack. Figure 7.2 shows the entropy of the apparent file access frequency for varying values of sq (the probability that perturbed queries are safe, see Theorem 4.1) for different values of nf, the mean number of files per node. Recall that a perturbed query identifier is safe if both the original identifier and the perturbed identifier are assigned to the same node in the system. Higher the safety of queries, smaller is the perturbation threshold (thr_{safe}); and thus, the lookup queries for a file are distributed over a smaller region in the identifier space. This decreases the entropy of the apparent file access frequency. Also, as the number of files stored at a node increases larger would be overlap between the safe ranges of files assigned to a node (see Figure 2). This evens out (partially) the differences between different apparent file access frequencies and thus, increases its entropy.



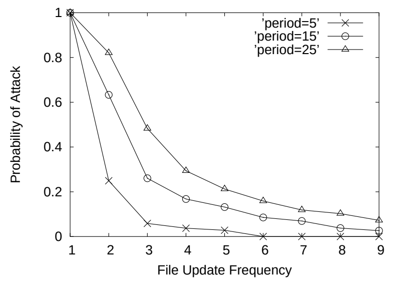


Figure 6: Countering File Replica Frequency Inference Attack: Location Re-Keying Frequency Vs File Update Frequency

File Replica Inference Attack. We study the severity of file replica inference attack with respect to the update frequency of files in the file system. We measured the probability that an adversary may be able to successfully locate all the replicas of a target file using the file replica inference attack when all the replicas of a file are encrypted with the same key. The adversary performs a host compromise attack with $\rho = 3$. Figure 7.2 shows the probability of a successful attack on a target file for different values of its update frequency and different values of re-keying durations. Note that the time period at which location keys are changed and the time period between file updates are normalized by $\frac{1}{\lambda}$ (mean time to compromise a good node). Observe the sharp knee in Figure 7.2; once the file update frequency increases beyond 3λ (thrice the node compromise rate) then probability of a successful attack is very small.

Note that λ, the rate at which a node can be compromised by one malicious node is likely to be quite small. Hence, even if a file is infrequently updated, it could survive a file replica inference attack. However, read-only files need to be encrypted with different cryptographic keys to make their replicas non-identical. Note that this adds the overhead of encrypting the replicas with different keys. Figure 7.2 also illustrates that lowering the periodicity of key changes lowers the probability of a successful attack significantly. This is because each time the location key is changed all the information collected by an adversary would be rendered entirely useless.

Inference Attacks Discussion. We have presented techniques to mitigate some popular inference attacks. There could be other inference attacks (such as file access pattern based attack) that are much harder to be condoned by our design. Even the solutions provided by us for frequency inference attack do not reach the theoretical optimum. For instance, even when we used result caching and file identifier perturbation in combination, we could not increase the entropy of apparent lookup frequency to the theoretical maximum (S_{max} in Table 4). Identifying other potential inference attacks and developing better defenses against the inference attacks that we have already pointed out in this paper would be a part of our future work.

8 Related Work

The secure Overlay Services (SOS) paper [8] presents an architecture that proactively prevents DoS attacks using secure overlay tunneling and routing via consistent hashing. However, the assumptions in this paper are markedly different from that of ours. For example, the SOS paper considers the overlay network as a black-box and assumes that all the participants in the overlay network are good. Also, the SOS architecture assumes that the client and server (service provider) are outside the overlay network. Nonetheless, our ideas were largely motivated by the SOS paper; we borrowed the idea of introducing randomness and anonymity into the architecture to make it difficult for malicious nodes to target their attacks on a small subset of nodes in the system.

The Hydra OS [3] proposed a capability-based file access control mechanism. Each object in Hydra is associated with a C-List (capability list). Each capability specifies an object that can be accessed and the corresponding access rights for that object. The OS Kernel is responsible for protecting a capability and retrieving (lookup) the objects pointed by it. Locations keys can be viewed as a technique to implement simple capability-based access control on a wide-area network. The most important challenge for location keys is that of keeping a file's capability secret and yet being able to perform a lookup on it (see Section 4.5).

Data obfuscation and steganography [20] present several interesting techniques to hide information by embedding them in seemingly harmless messages. Steganography (literally meaning *covered writing*) works by replacing bits of useless or unused data in regular computer files (such as graphics, sound, text, or HTML) with bits of different, invisible information. Similar to steganography, location keys embed files in an ocean of wide-area distributed workstations, and thus obfuscate the location of useful information in an overlay network.

Indirect attacks have long been the most popular technique to break cryptographic algorithms. Indirect attack, as the name suggests, does not directly break a cryptographic algorithm; instead, it may attempt to compromise cryptographic keys from system administrator or use fault attacks like RSA timing attacks, glitch attacks, hardware and software implementation bugs [14] to infer the encryption key. Similarly, a brute force attack (even with range sieving) on location keys is highly infeasible. Hence, attackers might resort to indirect or inference attacks like the lookup frequency, end-user IP-address and file access pattern based inference attacks. Unfortunately, a system designer is never aware of an exhaustive list of such inference attacks; hence, one is forced to develop custom solutions as and when a new inference attack surfaces.

9 Conclusion

In this paper we have proposed location keys. Analogous to traditional cryptographic keys that hide the contents of a file, location keys hide the location of a file on an overlay network. Location key guards a target file from DoS and host compromise attacks, provides a simple and efficient access control mechanism and adds minimal performance and storage overhead to the system (in addition, one could provide traditional guarantees like file confidentiality and integrity). We studied several inference attacks on the file system including file frequency inference attack. In conclusion, location keys based technique coupled with our guards to mitigate inference attacks, can effectively secure wide-area network file systems.

References

[1] K. Aberer and Z. Despotovic. Managing trust in a peer-2-peer information system. In *Proceedings of the 10th International Conference of Information and Knowledge Management*, 2001.

[2] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available and reliable storage for an incompletely trusted environment. In *5th Symposium on OSDI*, 2002.

[3] E. Cohen and D. Jefferson. Protection in the hydra operating system. In *ACM Symposium on Operating Systems Principles*, 1975.

[4] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the 18th SOSP*, October 2001.

[5] FIPS. Data encryption standard (des). http://www.itl.nist.gov/fi pspubs/fi p46-2.htm.

[6] Gnutella. The gnutella home page. http://gnutella.wego.com/, 2002.

[7] HMAC. Hmac: Keyed-hashing for message authentication. http://www.faqs.org/rfcs/rfc2104.html.

[8] A. Keromytis, V. Misra, and D. Rubenstein. Sos: Secure overlay services. In *ACM SIGCOMM*, 2002.

[9] J. Kubiatowics, D. Bindel, Y. Chen, S. Czerwinski, 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 the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[10] R. Laboratries. Rsa cryptography standard. http://www.rsasecurity.com/rsalabs/pkcs/.

[11] MathWorld. Shannon entropy. http://mathworld.wolfram.com/Entropy.html.

[12] NIST. Aes: Advanced encryption standard. http://csrc.nist.gov/CryptoToolkit/aes/.

[13] OpenSSL. Openssl. http://www.openssl.org/.

[14] OpenSSL. Timing-based attacks on rsa keys. http://www.openssl.org/news/secadv_20030317.txt.

[15] PGP. Pretty good privacy. http://www.pgp.com/.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of SIGCOMM Annual Conference on Data Communication*, August 2001.

[17] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, November 2001.

[18] M. Srivatsa and L. Liu. Vulnerabilities and security issues in structured overlay networks: A quantitative analysis. In *Proceedings of Annual Computer Security Applications Conference (ACSAC)*, 2004.

[19] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM Annual Conference on Data Communication*, August 2001.

[20] Webopedia. Steganography. http://www.webopedia.com/TERM/S/steganography.html.

[21] M. World. The caesar cipher. http://www.mathworld.com.

[22] L. Xiong and L. Liu. A reputation-based trust model for peer-to-peer ecommerce communities. In *IEEE Conference on E-Commerce (CEC'03)*, 2003.