Countering Targeted File Attacks using LocationGuard

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

Abstract

Serverless file systems, exemplified by CFS, Farsite and OceanStore, have received significant attention from both the industry and the research community. These file systems store files on a large collection of untrusted nodes that form an overlay network. They use cryptographic techniques to maintain file confidentiality and integrity from malicious nodes. Unfortunately, cryptographic techniques cannot protect a file holder from a Denial-of-Service (DoS) or a host compromise attack. Hence, most of these distributed file systems are vulnerable to targeted file attacks, wherein an adversary attempts to attack a small (chosen) set of files by attacking the nodes that host them. This paper presents LocationGuard – a location hiding technique for securing overlay file storage systems from targeted file attacks. LocationGuard has three essential components: (i) location key, consisting of a random bit string (e.g., 128 bits) that serves as the key to the location of a file, (ii) routing guard, a secure algorithm that protects accesses to a file in the overlay network given its location key such that neither its key nor its location is revealed to an adversary, and (iii) a set of four location inference guards. Our experimental results quantify the overhead of employing LocationGuard and demonstrate its effectiveness against DoS attacks, host compromise attacks and various location inference attacks.

1 Introduction

A new breed of serverless file storage services, like CFS [7], Farsite [1], OceanStore [15] and SiRiUS [10], have recently emerged. In contrast to traditional file systems, they harness the resources available at desktop workstations that are distributed over a wide-area network. The collective resources available at these desktop workstations amount to several peta-flops of computing power and several hundred peta-bytes of storage space [1].

These emerging trends have motivated serverless file storage as one of the most popular application over decentralized overlay networks. An overlay network is a virtual network formed by nodes (desktop workstations) on top of an existing TCP/IP-network. Overlay networks typically support a lookup protocol. A lookup operation identifies the location of a file given its filename. Location of a file denotes the IP-address of the node that currently hosts the file.

There are four important issues that need to be addressed to enable wide deployment of serverless file systems for mission critical applications.

Efficiency of the lookup protocol. There are two kinds of lookup protocol that have been commonly deployed: the Gnutella-like broadcast based lookup protocols [9] and the distributed hash table (DHT) based lookup protocols [25] [19] [20]. File systems like CFS, Farsite and OceanStore use DHT-based lookup protocols because of their ability to locate any file 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. 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 File Attacks. A major drawback with serverless file systems like CFS, Farsite and Ocean-Store is that they are vulnerable to targeted attacks on files. In

a targeted attack, an adversary is interested in compromising a small set of target files through a DoS attack or a host compromise attack. 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. 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 launch DoS or host compromise attacks on the set of R replica holders of a target file ($R \ll B$).

Efficient Access Control. A read-only file system like CFS can exercise access control by simply encrypting the contents of each file, and distributing the keys only to the legal users of that file. Farsite, a read-write file system, exercises access control using access control lists (ACL) that are maintained using a Byzantine-fault-tolerant protocol. However, access control is not truly distributed in Farsite because all users are authenticated by a small collection of directory-group servers. Further, PKI (public-key Infrastructure) based authentication and Byzantine fault tolerance based authorization are known to be more expensive than a simple and fast capability-based access control mechanism [6].

In this paper we present *LocationGuard* as an effective technique for countering targeted file attacks. The fundamental idea behind LocationGuard is to *hide* the very location of a file and its replicas such that, a legal user who possesses a file's *location key* can easily and securely locate the file on the overlay network; but 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. Further, an adversary would not be even able to learn if a particular file exists in the file system or not. LocationGuard comprises of three essential components. The first component of LocationGuard is a location key, which is a 128-bit string used as a key to the location of a file in the overlay network. A file's location key is used to generate legal capabilities (tokens) that can be used to access its replicas. The second component is the routing guard, a secure algorithm to locate a file in the overlay network given its location key such that neither the key nor the location is revealed to an adversary. The third component is an extensible collection of location inference guards, which protect the system from traffic analysis based inference attacks, such as lookup frequency inference attacks, end-user IP-address inference attacks, file replica inference attacks, and file size inference attacks. LocationGuard presents a careful combination of location key, routing guard, and location inference guards, aiming at making it very hard for an adversary to infer the location of a target file by either actively or passively observing the overlay network.

In addition traditional cryptographic guarantees like file confidentiality and integrity, LocationGuard mitigates denial-of-service (DoS) and host compromise attacks, while adding very little performance overhead and very minimal storage overhead to the file system. Our initial experiments quantify the overhead of employing LocationGuard and demonstrate its effectiveness against DoS attacks, host compromise attacks and various location inference attacks.

The rest of the paper is organized as follows. Section 2 provides terminology and background on overlay network and serverless file systems like CFS and Farsite. Section 3 describes our threat model in detail. We present an abstract functional description of LocationGuard in Section 4. Section 4, 5 describes the design of our location keys and Section 6 presents a detailed description of the routing guard. We outline a brief discussion on overall system management in Section 8 and present a thorough experimental evaluation of LocationGuard in Section 9. Finally, we present some related work in Section 10, and conclude the paper in Section 11.

2 Background and Terminology

In this section, we give a brief overview on the vital properties of DHT-based overlay networks and their lookup protocols (e.g., Chord [25], CAN [19], Pastry [20], Tapestry [3]). All these lookup protocols are fundamentally based on distributed hash tables, but differ in algorithmic and implementation details. All of them store the mapping between a particular *search key* and its associated $data$ (file) in a distributed manner across the network, rather than storing them at a single location like a conventional hash table. Given a *search key*, these techniques locate its associated $data$ (file) in a small and bounded number of hops within the overlay network. This is realized using three main steps. First, nodes and search keys are hashed to a common identifier space such that each node is given a unique identifier and is made responsible for a certain set of search keys. Second, the mapping of

search keys to nodes uses policies like numerical closeness or contiguous regions between two node identifiers to determine the (non-overlapping) region (segment) that each node will be responsible for. Third, a small and bounded lookup cost is guaranteed by maintaining a tiny routing table and a neighbor list at each node.

In the context of a file system, the search 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 the node responsible for that file is obtained using the concrete mapping policy. This operation of locating the appropriate node is called a *lookup*.

Serverless file system like CFS, Farsite and OceanStore are layered on top of DHT-based protocols. These file systems typically provide the following properties: (1) A file lookup is guaranteed to succeed if and only if the file is present in the system, (2) File lookup terminates in a small and bounded number of hops, (3) The files are uniformly distributed among all active nodes, and (4) The system handles dynamic node joins and leaves.

In the rest of this paper, we assume that Chord [25] is used as the overlay network's lookup protocol. However, the results presented 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. This definition of adversary permits collusions among malicious nodes.

We assume that the underlying IP-network layer may be insecure. However, we assume that the underlying IP-network infrastructure such as domain name service (DNS), and the network routers cannot be subverted by the 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 and use λ to denote the mean rate per malicious node at which a good node can be compromised. For instance, 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*). Note that it does not really help for one adversary to pose as multiple nodes (say using a virtualization technology) since the effective compromise rate depends only on the aggregate strength of the adversary. 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. Note that the 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.

3.1 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. Let f_n denote the name of a file f and f_d denote the data in file f. Given R replicas of a file f, file f is unavailable (or corrupted) if at least a threshold cr number of its replicas are unavailable (or corrupted). For example, for read/write files maintained by a Byzantine quorum [1], $cr = \lceil R/3 \rceil$. For encrypted and authenticated files, $cr = R$, since the file can be successfully recovered as long as at least one of its replicas is available (and uncorrupt) [7]. Most P2P trust management systems such as [27] uses a simple majority vote on the replicas to compute the actual trust values of peers, thus we have $cr = \lceil R/2 \rceil$.

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 [13] for easy lookup. For instance, CFS stores a file f at locations derivable from 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 all the replicas of a target file.

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. LocationGuard provides mechanisms to hide files in an overlay network such that only a legal user who possesses a file's location key can easily locate it. Further, an adversary would not even be able to learn whether a particular file exists in the file system or not. Thus, any previously known attacks on file contents would not be applicable unless the adversary succeeds in locating the file. It is important to note that LocationGuard is oblivious to whether or not file contents are encrypted. Hence, LocationGuard can be used to protect files whose contents cannot be encrypted, say, to permit regular expression based keyword search on file contents.

4 LocationGuard

4.1 Overview

We first present a high level overview of LocationGuard. Figure 1 shows an architectural overview of a file system powered by LocationGuard. LocationGuard operates on top of an overlay network of N nodes. Figure 2 provides a sketch of the conceptual design of LocationGuard. LocationGuard scheme guards the location of each file and its access with two objectives: (1) to hide the actual location of a file and its replicas such that only legal users who hold the file's location key can easily locate the file on the overlay network, and (2) to guard lookups on the overlay network from being eavesdropped by an adversary. LocationGuard consists of three core components. The first component is *location key*, which controls the transformation of a filename into its location on the overlay network, analogous to a traditional *cryptographic key* that controls the transformation of plaintext into ciphertext. The second component is the *routing guard*, which makes the location of a file unintelligible. The routing guard is, to some extent, analogous to a traditional *cryptographic algorithm* which makes a file's contents unintelligible. The third component of LocationGuard includes an extensible package of location inference guards that protect the file system from indirect attacks. Indirect attacks are those attacks that exploit a file's metadata information such as file access frequency, end-user IP-address, equivalence of file replica contents and file size to infer the location of a target file on the overlay network. In this paper we focus only on the first two components, namely, the location key and the routing guard. For a detailed discussion on location inference guards refer to our tech-report [23].

In the following subsections, we first present the main concepts behind location keys and location hiding (Section 4.2) and describe a reference model for serverless file systems that operate on LocationGuard (Section 4.3). Then we present the concrete design of LocationGuard's core components: the location key (Section 5), and the routing guard (Section 6).

4.2 Concepts and Definitions

In this section we define the concept of location keys and its location hiding properties. We discuss the concrete design of location key implementation and how location keys and location guards protect a file system from targeted file attacks in the subsequent sections.

Consider an overlay network of size N with a Chord-like lookup protocol Γ. Let $f^1, f^2, \cdots, f^R$ denote the R replicas of a file f. Location of a replica f^i refers to the IP-address of the node (replica holder) that stores replica f^i. A file lookup algorithm is defined as a function that accepts f^i and outputs its location on the overlay network. Formally we have $\Gamma : f^i \rightarrow loc$ maps a replica f^i to its location loc on the overlay network P.

Definition 1 *Location Key:* A location key lk of a file f is a relatively small amount (m-bit binary string, typically m = 128) of information that is used by a Lookup algorithm $\Psi : (f, lk) \rightarrow loc$ to customize the transfor-

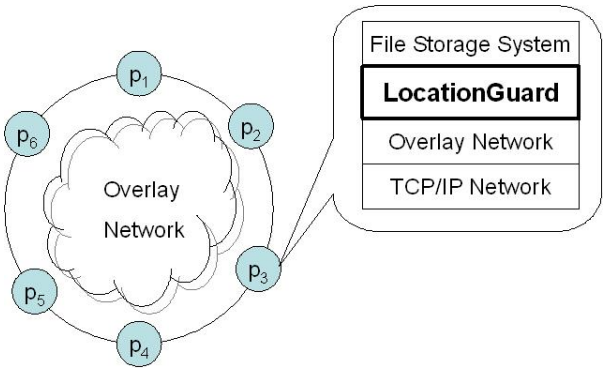

Figure 1: LocationGuard: System Architecture

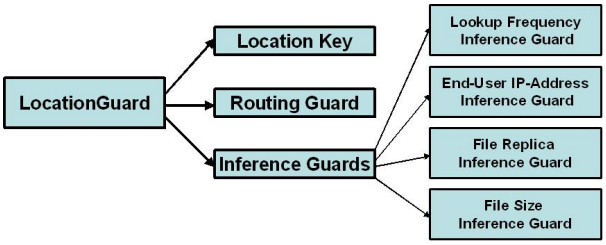

Figure 2: LocationGuard: Conceptual Design

mation of a file into its location such that the following three properties are satisfied:

1. Given the location key of a file f, it is *easy* to locate the R replicas of file f.
2. Without knowing the location key of a file f, it is *hard* for an adversary to locate any of its replicas.
3. The location key lk of a file f should not be exposed to an adversary when it is used to access the file f.

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. Note that the lookup algorithm Ψ is publicly known; only a file's location key is kept secret.

Property 1 ensures that valid users of a file f can easily access it provided they know its location key lk. Property 2 guarantees that illegal users who do not have the correct location key will not be able to locate the file on the overlay network, making it harder for an adversary to launch a targeted file attack. Property 3 warrants that no information about the location key lk of a file f is revealed to an adversary when executing the lookup algorithm Ψ.

Having defined the concept of location key, we present a reference model for a file system that operates on LocationGuard. We use this reference model to present a concrete design of LocationGuard's three core components: the location key, the routing guard and the location inference guards.

4.3 Reference Model

A serverless file system may implement read/write operations by exercising access control in a number of ways. For example, Farsite [1] uses an access control list maintained among a small number of directory servers through a Byzantine fault tolerant protocol. CFS [7], a read-only file system, implements access control by encrypting the files and distributing the file encryption keys only to the legal users of a file. In this section we show how a LocationGuard based file system exercises access control.

In contrast to other serverless file systems, a LocationGuard based file system does not directly authenticate any user attempting to access a file. Instead, it uses location keys to implement a capability-based access control mechanism, that is, any user who presents the correct file capability (token) is permitted access to that file. In addition to file token based access control, LocationGuard derives the encryption key for a file from its location key. This makes it very hard for an adversary to read file data from compromised nodes. Furthermore, it utilizes routing guard and location inference guards to secure the locations of files being accessed on the overlay network. Our access control policy is simple: *if you can name a file, then you can access it*. However, we do not use a file name directly; instead, we use a pseudo-filename (128-bit binary string) generated from a file's name and its location key (see Section 5 for detail). The responsibility of access control is divided among the file owner, the legal file users, and the file replica holders and is managed in a decentralized manner.

File Owner. Given a file f, its owner u is responsible for securely distributing f's location key lk (only) to those users who are authorized to access the file f.

Legal User. A user u who has obtained the valid location key of file f is called a legal user of f. Legal users are authorized to access any replica of file f. Given a file f's location key lk, a legal user u can generate the replica location token rlt^i for its i^{th} replica. Note that we use rlt^i as both the pseudo-filename and the capability of f^i. The user u now uses the lookup algorithm Ψ to obtain the IP-address of node $r = \Psi(rlt^i)$ (pseudo-filename rlt^i). User u gains access to replica f^i by presenting the token rlt^i to node r (capability rlt^i).

Non-malicious Replica Holder. Assume that a node r is responsible for storing replica f^i. Internally, node r stores this file content under a file name rlt^i. Note that node r does not need to know the actual file name (f) of a locally stored file rlt^i. Also, by design, given the internal file name rlt^i, node r cannot guess its actual file name (see Section 5). When a node r receives a read/write request on a file rlt^i it checks if a file named rlt^i is present locally. If so, it *directly* performs the requested operation on the local file rlt^i. Access control follows from the fact that it is very hard for an adversary to guess correct file tokens.

Malicious Replica Holder. Let us consider the case where the node r that stores a replica f^i is malicious. Note that node r's response to a file read/write request can be undefined. Note that we have assumed that the replicas stored at malicious nodes are always under attack (recall that up to $cr - 1$ out of R file replicas could be unavailable or corrupted). Hence, the fact that a malicious replica holder incorrectly implements file read/write operation or that the adversary is aware of the tokens of those file replicas stored at malicious nodes does not harm the system. Also, by design, an adversary who knows one token rlt^i for replica f^i would not be able to guess the file name f or its location key lk or the tokens for others replicas of file f (see Section 5).

Adversary. An adversary cannot access any replica of file f stored at a good node simply because it cannot guess the token rlt^i without knowing its location key. 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 good nodes, and corrupt the file replicas corresponding to these tokens at any later point in time. Eventually, the adversary would succeed in corrupting cr or more replicas of a file f without knowing its location key. LocationGuard addresses such attacks using location rekeying technique discussed in Section 7.3.

In the subsequent sections, we show how to generate a replica location token rlt^i ($1 \leq i \leq R$) from a file f and its location key (Section 5), and how the lookup algorithm Ψ performs a lookup on a pseudo-filename rlt^i without revealing the capability rlt^i to malicious nodes in the overlay network (Section 6). It is important to note that the ability of guarding the lookup from attacks like eavesdropping is critical to the ultimate goal of file location hiding scheme, since a lookup operation using a lookup protocol (such as Chord) on identifier rlt^i typically proceeds in plain-text through a sequence of nodes on the overlay network. Hence, an adversary may collect file tokens by simply sniffing lookup queries over the overlay network. The adversary could use these stolen file tokens to perform write operations on the corresponding file replicas, and thus corrupt them, without the knowledge of their location keys.

5 Location Keys

The first and most simplistic component of LocationGuard is the concept of location keys. The design of location key needs to address the following two questions: (1) How to choose a location key? (2) How to use a location key to generate a replica location token − the capability to access a file replica?

The first step in designing location keys is *to determining the type of string used as the identifier of a location key.* Let user u be the owner of a file f. User u should choose a long random bit string (128-bits) lk as the location key for file f. The location key lk should be hard to guess. For example, the key lk should not be semantically attached to or derived from the file name (f) or the owner name (u).

The second step is *to find a pseudo-random function* to derive the replica location tokens rlt^i ($1 \leq i \leq R$) from the filename f and its location key lk. The pseudo-filename rlt^i is used as a file replica identifier to locate the i^{th} replica of file f on the overlay network. Let $E_{lk}(x)$ denote a keyed pseudo-random function with input x and a secret key lk and $\|$ denotes string concatenation. We derive the location token $rlt^i = E_{lk}(f_n \parallel i)$ (recall that f_n denotes the name of file f). Given a replica's identifier rlt^i, one can use the lookup protocol Ψ to locate it on the overlay network. The function E should satisfy the following conditions:

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

2a) Given $(f_n \parallel i)$ it is hard to guess $E_{lk}(f_n \parallel i)$ without knowing lk.

2b) Given $E_{lk}(f_n \parallel i)$ it is hard to guess the file name f_n.

2c) Given $E_{lk}(f_n \parallel i)$ and the file name f_n it is hard to guess lk.

Condition 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. Condition 2a), states that it should be very hard for an adversary to guess the location of a target file f without knowing its location key. Condition 2b) ensures that even if an adversary obtains the identifier rlt^i of replica f^i, he/she cannot deduce the file name f. Finally, Condition 2c) requires 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 still has no clue about the remaining replicas of the file f (by Condition 2a). Conditions 2b) and 2c) play an important role in ensuring good location hiding property. This is because for any given file f, some of the replicas of file f could be stored at malicious nodes. Thus an adversary could be aware of some of the replica identifiers. Finally, observe that Condition 1a) and Conditions $\{$2a), 2b), 2c)$\}$ map to Property 1 and Property 2 in Definition 1 (in Section 4.2) respectively.

There are a number of cryptographic tools that satisfies our requirements specified in Conditions 1a), 2a), 2b) and 2c). Some possible candidates for the function E are (i) a keyed-hash function like HMAC-MD5 [14], (ii) a symmetric key encryption algorithm like DES [8] or AES [16], and (iii) a PKI based encryption algorithm like RSA [21]. We chose to use a keyed-hash function like HMAC-MD5 because it can be computed very efficiently. HMAC-MD5 computation is about 40 times faster than AES encryption and about 1000 times faster than RSA encryption using the standard OpenSSL library [17]. In the remaining part of this paper, we use $khash$ to denote a keyed-hash function that is used to derive a file's replica location tokens from its name and its secret location key.

6 Routing guard

The second and fundamental component of LocationGuard is the routing guard. The design of routing guard aims at securing the lookup of file f such that it will be very hard for an adversary to obtain the replica location tokens by eavesdropping on the overlay network. Concretely, let rlt^i ($1 \leq i \leq R$) denote a replica location token derived from the file name f, the replica number i, and f's location key identifier lk. We need to secure the lookup algorithm $\Psi(rlt^i)$ such that the lookup on pseudo-filename rlt^i does not reveal the capability rlt^i to other nodes on the overlay network. Note that a file's capability rlt^i does not reveal the file's name; but it allows an adversary to write on the file and thus corrupt it (see reference file system in Section 4.3).

There are two possible approaches to implement a secure lookup algorithm: (1) centralized approach and (2) decentralized approach. In the centralized approach, one could use a trusted location server [12] to return the location of any file on the overlay network. However, such a location server would become a viable target for DoS and host compromise attacks.

In this section, we present a decentralized secure lookup protocol that is built on top of the Chord protocol. Note that a naive Chord-like lookup protocol $\Gamma(rlt^i)$ cannot be directly used because it reveals the token rlt^i to other nodes on the overlay network.

6.1 Overview

The fundamental idea behind the routing guard is as follows. Given a file f's location key lk and replica number i, we want to find a safe region in the identifier space where we can obtain a huge collection of *obfuscated tokens*, denoted by $\{OTK^i\}$, such that, with high probability, $\Gamma(otk^i) = \Gamma(rlt^i), \forall otk^i \in OTK^i$. We call $otk^i \in OTK^i$ an obfuscated identifier of the token rlt^i. Each time a user u wishes to lookup a token rlt^i, it performs a lookup on some randomly chosen token otk^i from the obfuscated identifier set OTK^i. Routing guard ensures that even if an adversary were to observe obfuscated identifier from the set OTK^i for one full year, it would be highly infeasible for the adversary to guess the token rlt^i.

We now describe the concrete implementation of the routing guard. For the sake of simplicity, we assume a

unit circle for the Chord's identifier space; that is, node identifiers and file identifiers are real values from 0 to 1 that are arranged on the Chord ring in the anti-clockwise direction. Let $ID(r)$ denote the identifier of node r. If r is the destination node of a lookup on file identifier rlt^i, i.e., $r = \Gamma(rlt^i)$, then r is the node that immediately succeeds rlt^i in the anti-clockwise direction on the Chord ring. Formally, $r = \Gamma(rlt^i)$ if $ID(r) \geq rlt^i$ and there exists no other nodes, say v, on the Chord ring such that $ID(r) > ID(v) \geq rlt^i$.

We first introduce the concept of *safe obfuscation* to guide us in finding an obfuscated identifier set OTK^i for a given replica location token rlt^i. We say that an obfuscated identifier otk^i is a safe obfuscation of identifier rlt^i if and only if a lookup on both rlt^i and otk^i result in the same physical node r. For example, in Figure 3, identifier otk_1^i is a safe obfuscation of identifier rlt^i ($\Gamma(rlt^i) = \Gamma(otk_1^i) = r$), while identifier otk_2^i is unsafe ($\Gamma(otk_2^i) = r' \neq r$).

We define the set OTK^i as a set of all identifiers in the range ($rlt^i - srg$, rlt^i), where srg denotes a safe obfuscation range ($0 \leq srg < 1$). When a user intends to query for a replica location token rlt^i, the user actually performs a lookup on an obfuscated identifier $otk^i = obfuscate(rlt^i) = rlt^i - random(0, srg)$. The function $random(0, srg)$ returns a number chosen uniformly and randomly in the range $(0, srg)$.

We choose a safe value srg such that:

(C1) With high probability, any obfuscated identifier otk^i is a safe obfuscation of the token rlt^i.

(C2) Given a set of obfuscated identifier otk^i it is very hard for an adversary to guess the actual identifier rlt^i.

Note that if srg is too small condition C1 is more likely to hold, while condition C2 is more likely to fail. In contrast, if srg is too big, condition C2 is more likely to hold but condition C1 is more likely to fail. In our first prototype development of LocationGuard, we introduce a system defined parameter pr_{sq} to denote the minimum probability that any obfuscation is required to be safe. In the subsequent sections, we present a technique to derive srg as a function of pr_{sq}. This permits us to quantify the tradeoff between condition C1 and condition C2.

6.2 Determining the Safe Obfuscation Range

Observe from Figure 3 that a obfuscation $rand$ on identifier rlt^i is safe if $rlt^i - rand > ID(r')$, where r' is the immediate predecessor of node r on the Chord ring. Thus, we have $rand < rlt^i - ID(r')$. The expression $rlt^i - ID(r')$ denotes the distance between identifiers rlt^i and $ID(r')$ on the Chord identifier ring, denoted by $dist(rlt^i, ID(r'))$. Hence, we say that a obfuscation $rand$ is safe with respect to identifier rlt^i if and only if $rand < dist(rlt^i, ID(r'))$, or equivalently, $rand$ is chosen from the range $(0, dist(rlt^i, ID(r')))$.

We use Theorem 6.1 to show that $\Pr(dist(rlt^i, ID(r')) > x) = e^{-x*N}$, where N denotes the number of nodes on the overlay network and x denotes any value satisfying $0 \leq x < 1$. Informally, the theorem states that the probability that the predecessor node r' is further away from the identifier rlt^i decreases exponentially with the distance. For a detailed proof please refer to our tech-report [23].

Observe that an obfuscation $rand$ is safe with respect to rlt^i if $dist(rlt^i, ID(r')) > rand$, the probability that a obfuscation $rand$ is safe can be calculated using $e^{-rand*N}$.

Now, one can ensure that the minimum probability of any obfuscation being safe is pr_{sq} as follows. We first use pr_{sq} to obtain an upper bound on $rand$: By $e^{-rand*N} \geq pr_{sq}$, we have, $rand \leq \frac{-log_e(pr_{sq})}{N}$. Hence, if $rand$ is chosen from a safe range $(0, srg)$, where $srg = \frac{-log_e(pr_{sq})}{N}$, then all obfuscations are guaranteed to be safe with a probability greater than or equal to pr_{sq}.

For instance, when we set $pr_{sq} = 1 - 2^{-20}$ and $N =$ 1 million nodes, $srg = -\frac{log_e(pr_{sq})}{N} = 2^{-40}$. Hence, on a 128-bit Chord ring $rand$ could be chosen from a range of size $srg = 2^{128} * 2^{-40} = 2^{88}$. Table 1 shows the size of a $pr_{sq}-$safe obfuscation range srg for different values of pr_{sq}. Observe that if we set $pr_{sq} = 1$, then $srg = -\frac{log_e(pr_{sq})}{N} = 0$. Hence, if we want 100% safety, the obfuscation range srg must be zero, i.e., the token rlt^i cannot be obfuscated.

Theorem 6.1 *Let N denote the total number of nodes in the system. Let $dist(x, y)$ denote the distance between two identifiers x and y on a Chord's unit circle. Let node r' be the node that is the immediate predecessor for an identifier rlt^i on the anti-clockwise unit circle Chord ring. Let $ID(r')$ denote the identifier of the node r'.*

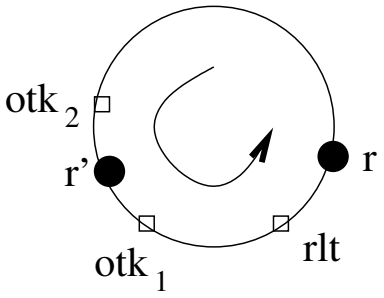

Figure 3: Lookup Using File Identifier Obfuscation: Illustration

$1 - pr_{sq}$	2^{-10}	2^{-15}	2^{-20}	2^{-25}	2^{-30}
srg	2^{98}	2^{93}	2^{88}	2^{83}	2^{78}
$E[retries]$	2^{-10}	2^{-15}	2^{-20}	2^{-25}	2^{-30}
hardness (years)	2^{38}	2^{33}	2^{28}	2^{23}	2^{18}

Table 1: Lookup Identifier obfuscation

Then, the probability that the distance between identifiers rlt^i and $ID(r')$ exceeds rg is given by Pr(dist(rlt^i, $ID(r')) > x$) = e^{-x*N} for some $0 \leq x < 1$.

6.3 Ensuring Safe Obfuscation

Given that when $pr_{sq} < 1$, there is small probability that an obfuscated identifier is not safe, i.e., $1-pr_{sq} > 0$. We first discuss the motivation for detecting and repairing unsafe obfuscations and then describe how to guarantee good safety by our routing guard through a self-detection and self-healing process.

We first motivate the need for ensuring safe obfuscations. Let node r be the result of a lookup on identifier rlt^i and node v ($v \neq r$) be the result of a lookup on an unsafe obfuscated identifier otk^i. To perform a file read/write operation after locating the node that stores the file f, the user has to present the location token rlt^i to node v. If a user does not check for unsafe obfuscation, then the file token rlt^i would be exposed to some other node $v \neq r$. If node v were malicious, then it could misuse the capability rlt^i to corrupt the file replica actually stored at node r.

We require a user to verify whether an obfuscated identifier is safe or not using the following check: An obfuscated identifier otk^i is considered *safe* if and only if $rlt^i \in (otk^i, ID(v))$, where $v = \Gamma(otk^i)$. By the definition of v and otk^i, we have $otk^i \leq ID(v)$ and $otk^i \leq rlt^i$ ($rand \geq 0$). By $otk^i \leq rlt^i \leq ID(v)$, node v should be the immediate successor of the identifier rlt^i and thus be responsible for it. If the check failed, i.e., $rlt^i > ID(v)$, then node v is definitely not a successor of the identifier rlt^i. Hence, the user can flag otk^i as an unsafe obfuscation of rlt^i. For example, referring Figure 3, otk_1^i is safe because, $rlt^i \in (otk_1^i, ID(r))$ and $r = \Gamma(otk_1^i)$, and otk_2^i is unsafe because, $rlt^i \notin (otk_2^i, ID(r'))$ and $r' = \Gamma(otk_2^i)$.

When an obfuscated identifier is flagged as unsafe, the user needs to retry the lookup operation with a new obfuscated identifier. This retry process continues until *max_retries* rounds or until a safe obfuscation is found. Since the probability of an unsafe obfuscation is extremely small over multiple random choices of obfuscated tokens (otk^i), the call for retry rarely happens. We also found from our experiments that the number of retries required is almost always zero and seldom exceeds one. We believe that using *max_retries* equal to two would suffice even in a highly conservative setting. Table 1 shows the expected number of retries required for a lookup operation for different values of pr_{sq}.

6.4 Strength of Routing guard

The strength of a routing guard refers to its ability to counter lookup sniffing based attacks. A typical lookup sniffing attack is called the *range sieving attack*. Informally, in a range sieving attack, an adversary sniffs lookup queries on the overlay network, and attempts to deduce the actual identifier rlt^i from its multiple obfuscated identifiers. We show that an adversary would have to expend 2^{28} years to discover a replica location token rlt^i even if it has observed 2^{25} obfuscated identifiers of rlt^i. Note that 2^{25} obfuscated identifiers would be available to an adversary if the file replica f^i was accessed once a second for one full year by some legal user of the file f.

One can show that given multiple obfuscated identifiers it is non-trivial for an adversary to categorize them into groups such that all obfuscated identifiers in a group are actually obfuscations of one identifier. To simplify

the description of a range sieving attack, we consider the worst case scenario where an adversary is capable of categorizing obfuscated identifiers (say, based on their numerical proximity).

We first concretely describe the range sieving attack assuming that pr_{sq} and srg (from Theorem 6.1) are public knowledge. When an adversary obtains an obfuscated identifier otk^i, the adversary knows that the actual capability rlt^i is definitely within the range $RG = (otk^i, otk^i + srg)$, where $(0, srg)$ denotes a $pr_{sq}-$safe range. In fact, if obfuscations are uniformly and randomly chosen from $(0, srg)$, then given an obfuscated identifier otk^i, the adversary knows *nothing more* than the fact that the actual identifier rlt^i could be uniformly and distributed over the range $RG = (otk^i, otk^i + srg)$. However, if a persistent adversary obtains multiple obfuscated identifiers $\{otk^i_1, otk^i_2, \cdots, otk^i_{nid}\}$ that belong to the same 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 obfuscations on the identifier rlt^i. 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 obfuscated identifiers (nid) increases, the size of the sieved range RG_s decreases. For all tokens $tk \in RG_s$, the likelihood that the obfuscated identifiers $\{otk^i_1, otk^i_2, \cdots, otk^i_{nid}\}$ are obfuscations of the identifier tk is equal. Hence, the adversary is left with no smart strategy for searching the sieved range RG_s other than performing a brute force attack on some random enumeration of identifiers $tk \in RG_s$.

Let $E[RG_s]$ denote the expected size of the sieved range. Theorem 6.2 shows that $E[RG_s] = \frac{srg}{nid}$. Hence, if the safe range srg is significantly larger than nid then the routing guard can tolerate the range sieving attack. Recall the example in Section 6 where $pr_{sq} = 1 - 2^{-20}$, $N = 10^6$, the safe range $srg = 2^{88}$. Suppose that a target file is accessed once per second for one year; this results in 2^{25} file accesses. An adversary who logs all obfuscated identifiers over a 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 read per millisecond, the adversary would have tried 2^{35} read operations per year. Thus, it would take the adversary about $2^{63}/2^{35} = 2^{28}$ years to discover the actual file identifier. For a detailed proof of Theorem 6.2 refer to our tech-report [23].

Table 1 summarizes the hardness of breaking the obfuscation scheme for different values of pr_{sq} (minimum probability of safe obfuscation), assuming that the adversary has logged 2^{25} file accesses (one access per second for one year) and that the nodes permit at most one file access per millisecond.

Discussion. An interesting observation follows from the above discussion: the amount of time taken to break the file identifier obfuscation technique is almost independent of the number of attackers. This is a desirable property. It implies that as the number of attackers increases in the system, the hardness of breaking the file capabilities will not decrease. The reason for location key based systems to have this property is 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 the contrary, a brute force attack on a cryptographic key is inherently parallelizable and thus becomes more powerful as the number of attackers increases.

Theorem 6.2 *Let nid denote the number of obfuscated identifiers that correspond to a target file. Let RG_s denote the sieved range using the range sieving attack. Let srg denote the maximum amount of obfuscation that could be $pr_{sq}-$safely added to a file identifier. Then, the expected size of range RG_s can be calculated by $E[|RG_s|] = \frac{srg}{nid}$.*

7 Location Inference Guards

Inference attacks over location keys refer to those attacks wherein an adversary attempts to infer the location of a file using *indirect* techniques. We broadly classify inference attacks on location keys into two categories: *passive inference attacks* and *host compromise based inference attacks*. It is important to note that none of the inference attacks described below would be effective in the absence of collusion among malicious nodes.

7.1 Passive inference attacks

Passive inference attacks refer to those attacks wherein an adversary attempts to infer the location of a target file by passively observing the overlay network. We studied two passive inference attacks on location keys.

The lookup frequency inference attack is based on the ability of malicious nodes to observe the frequency of lookup queries on the overlay network. Assuming that the adversary knows the relative file popularity, it can use the target file's lookup frequency to infer its location. It has been observed that the general popularity of the web pages accessed over the Internet follows a Zipf-like distribution [27]. 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. This is analogous to performing a frequency analysis attack on old symmetric key ciphers like the Caesar's cipher [26].

The end-user IP-address inference attack is based on assumption that the identity of the end-user can be inferred from its IP-address by an overlay network node r, when the user requests node r to perform a lookup on its behalf. A malicious node r could log and report this information to the adversary. Recall that we have assumed that an adversary could be aware of the owner and the legal users of a target file. 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. Note that this is a worst-case-assumption; in most cases it may not possible to associate a user with one or a small number IP-addresses (say, when the user obtains IP-address dynamically (DHCP [2]) from a large ISP (Internet Service Provider)).

7.2 Host compromise based inference attacks

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. We studied two host compromise based inference attacks on location keys.

The file replica inference attack attempts to infer the identity of a file from its contents; note that an adversary can reach the contents of a file only after it compromises the file holder (unless the file holder is malicious). The file f could be encrypted to rule out the possibility of identifying a file from its contents. Even when the replicas are encrypted, an adversary can exploit 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, it could examine the extracted list of identifiers and file contents to obtain the identities of other replicas. Once, the adversary has the locations of cr copies of a file f, the f could be attacked easily. This attack is especially more plausible on read-only files since their contents do not change over a long period of time. On the other hand, the update frequency on read-write files might guard them file replica 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 size of files stored on the overlay network follows a skewed distribution, the adversary would be able to identify the target file (much like the lookup frequency inference attack).

For a detailed discussion on inference attacks and techniques to curb them please refer to our technical report [23]. Identifying other potential inference attacks and developing defenses against them is a part of our ongoing work.

7.3 Location Rekeying

In addition to the inference attacks listed above, there could be other possible inference attacks on a LocationGuard based file system. In due course of time, the adversary might be able to gather enough information to infer the location of a target file. Location rekeying is a general defense against both *known and unknown* inference attacks. Users can periodically choose new location keys so as to render *all* past inferences made by an adversary *useless*. In order to secure the system from Biham's key collision attacks [4], one may associate an initialization vector (IV) with the location key and change IV rather than the location key itself.

Location rekeying is analogous to rekeying of cryptographic keys. Unfortunately, rekeying is an expensive operation: rekeying cryptographic keys requires data to be re-encrypted; rekeying location keys requires files to

be relocated on the overlay network. Hence, it is important to keep the rekeying frequency small enough to reduce performance overheads and large enough to secure files on the overlay network. In our experiments section, we estimate the periodicity with which location keys have to be changed in order to reduce the probability of an attack on a target file.

8 Discussion

In this section, we briefly discuss a number of issues related to security, distribution and management of LocationGuard.

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

Key Distribution. Secure distribution of keys has been a major challenge in large scale distributed systems. The problem of distributing location keys is very similar to that of distributing cryptographic keys. Typically, keys are distributed using out-of-band techniques. For instance, one could use PGP [18] based secure email service to transfer location keys from a file owner to file users.

Key Management. Managing location keys efficiently becomes an important issue when (i) an owner owns several thousand files, and (ii) the set of legal users for a file vary significantly over time. In the former scenario, the file owner could reduce the key management cost by assigning one location key for a group of files. Any user who obtains the location key for a file f would implicitly be authorized to access the group of files to which f belong. However, the later scenario may seriously impede the system's performance in situations where it may require location key to be changed each time the group membership changes.

The major overhead for LocationGuard arises from key distribution and key management. Also, location rekeying could be an important factor. Key security, distribution and management in LocationGuard using group key management protocols [11] are a part of our ongoing research work.

Other issues that are not discussed in this paper include the problem of a valid user illegally distributing the capabilities (tokens) to an adversary, and the robustness of the lookup protocol and the overlay network in the presence of malicious nodes. In this paper we assume that all valid users are well behaved and the lookup protocol is robust. Readers may refer to [24] for detailed discussion on the robustness of lookup protocols on DHT based overlay networks.

9 Experimental Evaluation

In this section, we report results from our simulation based experiments to evaluate the LocationGuard approach for building secure wide-area network file systems. We implemented our simulator using a discrete event simulation [8] model, and implemented the Chord lookup protocol [25] on the overlay network compromising of $N = 1024$ nodes. In all experiments reported in this paper, a random $p = 10\%$ of N nodes are chosen to behave maliciously. We set the number of replicas of a file to be $R = 7$ and vary the corruption threshold cr in our experiments. We simulated the bad nodes as having large but bounded power based on the parameters α (DoS attack strength), λ (node compromise rate) and μ (node recovery rate) (see the threat model in Section 3). We study the cost of using location key technique by quantifying the overhead of using location keys and evaluate the benefits of the location key technique by measuring the effectiveness of location keys against DoS and host compromise based target file attacks.

9.1 LocationGuard

Operational Overhead. We first quantify the performance and storage overheads incurred by location keys. All measurements presented in this section were obtained on a 900 MHz Intel Pentium III processor running RedHat Linux 9.0. 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 replica holders. Step (i) requires computa-

tions using the keyed-hash function with location keys, which otherwise would have required computations using a normal hash function. We found that the computation time difference between HMAC-MD5 (a keyed-hash function) and MD5 (normal hash function) is negligibly small (order of a few microseconds) using the standard OpenSSL library [17]. 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 obfuscated identifier turns out to be unsafe. Given that unsafe obfuscations are extremely rare (see Table 1) retries are only required occasionally and thus this overhead is negligible. Step (iii) adds no overhead because our access check is almost 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 only an additional 128-bit location key (16 Bytes) along with other file meta-data for each file they want to access. Even a user who uses 1 million files on the overlay network needs to store only an additional 16MBytes of location keys. Further, there is no extra storage overhead on the rest of the nodes on the overlay network. For a detailed description of our implementation of LocationGuard and benchmark results for file read and write operations refer to our tech-report [23].

Denial of Service Attacks. Figure 4 shows the probability of an attack for varying α and different values of corruption threshold (cr). 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 with respect to the size (N) of the network, then the probability of an attack is higher. If an adversary were aware of the R replica holders of a target file then a weak collection of B malicious nodes, such as $B = 102$ (i.e., 10% of N) with $\alpha = \frac{R}{B} = \frac{7}{102} = 0.07$, can easily attack the target file. It is also true that for a file system to handle the DoS attacks on a file with $\alpha = 1$, it would require a large number of replicas (R close to B) to be maintained for each file. For example, in the case where $B = 10\% \times N$ and $N = 1024$, the system needs to maintain as large as 100+ replicas for each file. Clearly, without location keys, the effort required for an adversary to attack a target file (i.e., make it unavailable) is dependent only on R, but is independent of the number of good nodes (G) in the system. On the contrary, the location key based techniques scale the hardness of an attack with the number of good nodes in the system. Thus even with a very small R, the location key based system can make it very hard for any adversary to launch a targeted file attack.

Host Compromise Attacks. To further evaluate the effectiveness of location keys against targeted file attacks, we evaluate location keys against host compromise attacks. Our first experiment on host compromise attack shows the probability of an attack on the target file assuming that the adversary can not collect capabilities (tokens) stored at the compromised nodes. Hence, the target file is attacked if cr or more of its replicas are stored at either malicious nodes or compromised nodes. Figure 5 shows the probability of an attack for different values of corruption threshold (cr) and varying $\rho = \frac{\mu}{\lambda}$ (measured in number of node recoveries 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.

From Figure 5 we observe that when $\rho \leq 1$, the system is highly vulnerable since the node recovery rate is lower than the node compromise rate. Note that while a DoS attack could tolerate powerful malicious nodes ($\alpha > 1$), the host compromise attack cannot tolerate the situation where the node compromise rate is higher than their recovery rate ($\rho \leq 1$). This is primarily because of the cascading effect of host compromise attack. The larger the number of compromised nodes we have, the 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. Observe from Table 2 that when $\rho = 1$, most of the good nodes are in compromised state.

As we have mentioned in Section 4.3, the adversary could collect the capabilities (tokens) of the file replicas stored at compromised nodes; these tokens can be used by the adversary at any point in future to corrupt these replicas using a simple write operation. Hence, our second experiment on host compromise attack measures the

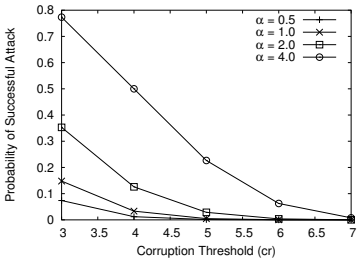

Figure 4: Probability of a Target File Attack for $N = 1024$ nodes and $R = 7$ using DoS Attack

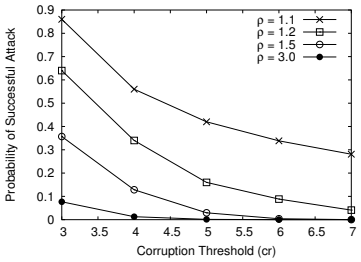

Figure 5: Probability of a Target File Attack for $N = 1024$ nodes and $R = 7$ using Host Compromise Attack (with no token collection)

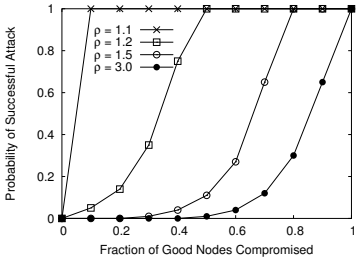

Figure 6: Probability of a Target File Attack for $N = 1024$ nodes and $R = 7$ using Host Compromise Attack with token collection from compromised nodes

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 Uncompromised State (G')

probability of a attack assuming that the adversary collects the file tokens stored at compromised nodes. Figure 6 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 in order to succeed a targeted file attack. 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 in order to attain the probability of an attack to a nominal value of 0.1, even under the assumption that an adversary collects file capabilities from compromised nodes. 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 7.3 we had proposed location re-keying to protect the file system from such attacks. The exact period of location re-keying can be derived from Figure 6. 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 as ρ increases, location re-keying can be more and more infrequent.

Lookup Guard. We performed the following experiments on our lookup identifier obfuscation technique (see Section 6): (i) studied the effect of obfuscation range on the probability of a safe obfuscation, (ii) measured the number of lookup retries, and (iii) measured the expected size of the sieved range (using the range sieving attack). We found that a safe range for identifier obfuscation is very large even for large values of sq (very close to 1); we observed that number of lookup retries is almost zero and seldom exceeds one; we found that the size of the sieved range is too large to attempt a brute force attack even if file accesses over one full year were logged. Finally, our experimental results very closely match the analytical results shown in Table 1. For more details on our experimental results refer to our tech-report [23].

10 Related Work

Serverless distributed file systems like CFS [7], Farsite [1], OceanStore [15] and SiRiUS [10] have received significant attention from both the industry and the research community. These file systems store files on a large collection of untrusted nodes that form an overlay network. They use cryptographic techniques to ensure file data confidentiality and integrity. Unfortunately, cryptographic techniques cannot protect a file holder from DoS or host compromise attacks. LocationGuard presents low overhead and highly effective techniques to guard

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)

a distributed file system from such targeted file attacks.

The secure Overlay Services (SOS) paper [13] describes an architecture that proactively prevents DoS attacks using secure overlay tunneling and routing via consistent hashing. However, the assumptions and the applications in [13] are noticeably different from that of ours. For example, the SOS paper uses the overlay network for introducing randomness and anonymity into the SOS architecture to make it difficult for malicious nodes to attack target applications of interest. LocationGuard techniques treat the overlay network as a part of the target applications we are interested in and introduce randomness and anonymity through location key based hashing and lookup based file identifier obfuscation, making it difficult for malicious nodes to target their attacks on a small subset of nodes in the system, who are the replica holders of the target file of interest.

The Hydra OS [6], [22] proposed a capability-based file access control mechanism. LocationGuard can be viewed as an implementation of capability-based access control on a wide-area network. The most important challenge for LocationGuard is that of keeping a file's capability secret and yet being able to perform a lookup on it (see Section 6).

Indirect attacks such as attempts to compromise cryptographic keys from the system administrator or use fault attacks like RSA timing attacks, glitch attacks, hardware and software implementation bugs [5] have been the most popular techniques to attack cryptographic algorithms. Similarly, attackers might resort to inference attacks on LocationGuard since a brute force attack (even with range sieving) on location keys is highly infeasible. Due to space restrictions we have not been able to include location inference guards in this paper. For details on location inference guards, refer to our techreport [23].

11 Conclusion

In this paper we have proposed LocationGuard for securing wide area serverless file sharing systems from targeted file attacks. Analogous to traditional cryptographic keys that hide the contents of a file, LocationGuard hide the location of a file on an overlay network. LocationGuard retains traditional cryptographic guarantees like file data confidentiality and integrity. In addition, LocationGuard 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. We presented experimental results that demonstrate the effectiveness of our techniques against targeted file attacks. In conclusion, LocationGuard mechanisms make it possible to build simple and secure wide-area network file systems.

Acknowledgements

This research is partially supported by NSF CNS, NSF ITR, IBM SUR grant, and HP Equipment Grant. Any opinions, findings, and conclusions or recommendations expressed in the project material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] 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 *Proceedings of the 5th International Symposium on OSDI*, 2002.

[2] I. R. Archives. RFC 2131: Dynamic host configuration protocol. http://www.faqs.org/rfcs/rfc2131.html.

[3] J. K. B. Zhao and A. Joseph. Tapestry: An infrastructure for fault-tolerance wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California, Berkeley, 2001.

[4] E. Biham. How to decrypt or even substitute DES-encrypted messages in 2^{28} steps. In *Information Processing Letters, 84*, 2002.

[5] D. Boneh and D. Brumley. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

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

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

[8] FIPS. Data encryption standard (DES). http://www.itl.nist.gov/fipspubs/fip46-2.htm.

[9] Gnutella. The gnutella home page. http://gnutella.wego.com/.

[10] E. J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *Proceedings of NDSS*, 2003.

[11] H. Harney and C. Muckenhirn. Group key management protocol (GKMP) architecture. http://www.rfc-archive.org/getrfc.php?rfc=2094.

[12] T. Jaeger and A. D. Rubin. Preserving integrity in remote file location and retrieval. In *Proceedings of NDSS*, 1996.

[13] A. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure overlay services. In *Proceedings of the ACM SIGCOMM*, 2002.

[14] H. Krawczyk, M. Bellare, and R. Canetti. RFC 2104 - HMAC: Keyed-hashing for message authentication. http://www.faqs.org/rfcs/rfc2104.html.

[15] 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.

[16] NIST. AES: Advanced encryption standard. http://csrc.nist.gov/CryptoToolkit/aes/.

[17] OpenSSL. OpenSSL: The open source toolkit for ssl/tls. http://www.openssl.org/.

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

[19] 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*, Aug 2001.

[20] 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)*, Nov 2001.

[21] RSA. RSA security - public-key cryptography standards (pkcs). http://www.rsasecurity.com/rsalabs/pkcs/.

[22] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A fast capability system. In *Proceedings of 17th ACM Symposium on Operating Systems Principles*, 1999.

[23] M. Srivatsa and L. Liu. Countering targeted file attacks using location keys. Technical Report GIT-CERCS-04-31, Georgia Institute of Technology, 2004.

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

[25] 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.

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

[27] L. Xiong and L. Liu. Peertrust: Supporting reputation-based trust for peer-to-peer electronic communities. In *Proceedings of IEEE TKDE, Vol. 16, No. 7*, 2004.