RC24589 (W0806-081) June 18, 2008
Computer Science

IBM Research Report

Mitigating Denial of Service Attacks on the Chord Overlay Network: A Location Hiding Approach

Mudhakar Srivatsa
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Ling Liu
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Mitigating Denial of Service Attacks on the Chord Overlay Network: A Location Hiding Approach

Mudhakar Srivatsa† and Ling Liu‡
IBM T.J. Watson Research Center, Yorktown Heights, NY 10598†
College of Computing, Georgia Institute of Technology, Atlanta, GA 30332‡
msrivats@us.ibm.com, lingliu@cc.gatech.edu

***Abstract*— [1] Serverless distributed computing has received significant attention from both the industry and the research community. Among the most popular applications are the wide area network file systems, exemplified by CFS, Farsite and OceanStore. 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 location inference guards, which refer to an extensible component of the LocationGuard. Our experimental results quantify the overhead of employing LocationGuard and demonstrate its effectiveness against DoS attacks, host compromise attacks and various location inference attacks.**

Keywords: File Systems, Overlay Networks, Denial of Service Attacks, Performance & Scalability, Location Hiding

I. Introduction

Several serverless file storage services, like CFS [6], Farsite [1], OceanStore [16] and SiRiUS [12], 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.

[1] A preliminary version of this paper appeared in USENIX Security Symposium 2005 [28]

Efficiency of the lookup protocol. There are two kinds of lookup protocol that have been commonly deployed: the Gnutella-like broadcast based lookup protocols [11] and the distributed hash table (DHT) based lookup protocols [29] [23] [25]. 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 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 denial of service (DoS) attack or a host compromise attack. A DoS 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, that is, given the file name f, an adversary (including users who may not have access to file f) can determine the IP-addresses of nodes that host f's replicas. 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 (BFT) 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 [5].

Bearing these issues in mind, 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. LocationGuard implements an efficient capability-based file access control mechanism through three essential components. The first component of LocationGuard is a location key, which is a random bit string (128 bits) used as a key to the location of a file in the overlay network, and addresses the capability revocation problem by periodic or conditional rekeying mechanisms. 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.

In addition to providing an efficient file access control mechanism with traditional cryptographic guarantees like file confidentiality and integrity, LocationGuard mitigates Denial-of-Service (DoS) and host compromise attacks, while adding minimal performance overhead and small 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 II provides terminology and background on overlay network and serverless file systems like CFS and Farsite. Section III describes our threat model in detail. We present the core techniques of LocationGuard in Sections IV, V, VI and VII. We present a concrete implementation and a thorough experimental evaluation of LocationGuard in Section VIII, related work in Section IX, and conclude the paper in Section X.

II. 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 [29], CAN [23], Pastry [25]). 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. 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) A 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 [29] is used as the overlay network's lookup protocol. However, the results presented in this paper are applicable to most DHT-based lookup protocols.

III. 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 also 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, denial-of-service attacks, and host compromise attacks. 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*). 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 node compromise times and recovery times follow an exponential distribution.

A. Targeted File Attacks

A 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

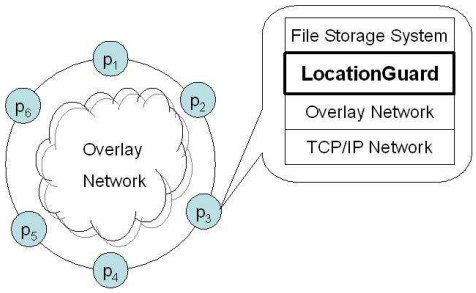

Fig. 1. LocationGuard: System Architecture

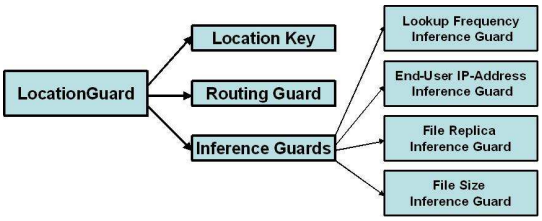

Fig. 2. LocationGuard: Conceptual Design

unavailable or corrupted. 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) [6]. Most P2P trust management systems such as the scheme in [30] use 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 [14] 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 all replica locations for 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. 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.

IV. LocationGuard

A. 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 the following subsections, we first present the main concepts behind location keys and location hiding (Section IV-B) and describe a reference model for serverless file systems that operate on LocationGuard (Section IV-C). Then we present the concrete design of LocationGuard's three core components: the location key (Section V), the routing guard (Section VI) and a suite of location inference guards (Section VII).

B. 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 transformation 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 the 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.

C. LocationGuard File System

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 [6], a read-only file system, may implement 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 an 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. 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 V 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 $r = \Psi(f, lk)$. User u gains access to replica f^i by presenting the token rlt^i to node r. Note that rlt^i acts as a pseudo-filename during lookup and a capability during access control.

Good Replica Holder. Assume that a node r is responsible for storing replica f^i. Internally, node r stores this file content under its pseudo-filename 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 V). 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 V).

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 a location rekeying technique discussed in Section VII-C.

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 V), 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 VI). It is important to note that the ability to guard the lookup from attacks like eavesdropping is critical to the 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.

V. 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 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 $rlt^i = E_{lk}(f \parallel i)$. Given a replica's identifier rlt^i, one can use the lookup protocol Ψ to locate it on the overlay network. We use a fast and efficient keyed-hash

function like HMAC-MD5 [15] since it satisfies the following conditions:

1a) Given $(f \parallel i)$ and lk it is easy to compute $E_{lk}(f \parallel i)$.
2a) Given $(f \parallel i)$ it is hard to guess $E_{lk}(f \parallel i)$ without knowing lk.
2b) Given $E_{lk}(f \parallel i)$ it is hard to guess the file name f.
2c) Given $E_{lk}(f \parallel i)$ and f 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 IV-B) respectively. In the remaining part of this paper, we use $khash$ to denote a keyed pseudo-random function that is used to derive a file's replica location tokens from its name and its secret location key.

VI. Routing guard

The second 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 lk. We need to secure the lookup algorithm $\Psi_{lk}(rlt^i)$ 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 IV-C).

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 [13] 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.

A. 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.

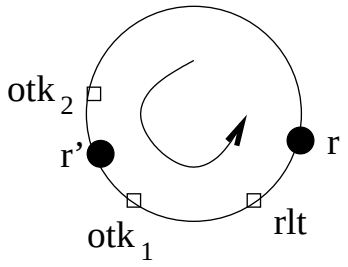

Fig. 3. Lookup Using File Identifier Obfuscation: r: hosting node; r': previous node to r on the Chord ring; $otk \cong rlt-rand(0, srg)$; $otk_1 \in (ID(r'), ID(r)]$ is safe while $otk_2 < ID(r')$ is unsafe; hence, safe obfuscation range $srg = rlt-ID(r')$

$1 - pr_{sq}$	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 I
Lookup Identifier obfuscation

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 identifiers 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 large collection of obfuscated identifiers $\{otk^i\}$ 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.

B. 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. Since 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 I 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'. 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 $0 \leq x < 1$.

Proof: Let Z be a random variable that denotes the distance between an identifier rlt^i and node r'. Let $f_Z(x)$ denote the probability distribution function (pdf) that the node r' is at a distance x from the identifier rlt^i, i.e., $dist(ID(r'), rlt^i) = x$. We first derive the probability distribution $f_Z(x)$ and use it to compute $\Pr(Z > x) = \Pr(dist(rlt^i, ID(r')) > x)$.

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. Hence, with probability $\triangle x$, a given node exists between a distance of $(x, x+\triangle x)$ from the identifier rlt^i (for any arbitrarily small region $\triangle x$). When there are N nodes in the system, the probability that one of them exists between a distance $(x, x + \triangle x)$ is $N * \triangle x$. Similarly, the probability that none of other node $N-1$ nodes lie within a distance rg from identifier rlt^i is $(1-x)^{N-1}$. Therefore, $f_Z(x)$ is given by Equation 1.

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

Now, using the probability density function in Equation 1 one can derive the cumulative distribution function (cdf), $\Pr(Z > x) = (1 - x)^N \approx e^{-x*N}$ (for small values of x) using standard techniques in probability theory. ∎

C. 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.

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 this information to corrupt the file replica actually stored at node r (using the capability rlt^i).

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. Thanks to the fact that the probability of an unsafe obfuscation can be extremely small, 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 I shows the expected number of retries required for a lookup operation for different values of pr_{sq}.

D. 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 publicly known. 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 randomly 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. In fact, the probability of observing otk^i_j for some $1 \leq j \leq nid$ given that the actual token is tk is $Pr(otk^i_j \mid tk) = \frac{1}{srg}$, $\forall\ tk \in RG_s$. Also, the probability of observing the obfuscated identifiers $\{otk^i_1, otk^i_2, \cdots, otk^i_{nid}\}$ given that the actual token is tk is $Pr(\{otk^i_1, otk^i_2, \cdots, otk^i_{nid}\} \mid tk) = \frac{1}{{}^{srg}C_{nid}}$, $\forall\ tk \in RG_s$. Note that ${}^{srg}C_{nid}$ denotes the number of ways of choosing nid balls from a pool of srg non-identical balls. 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 VI 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. Table I 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 $E[|RG_s|] = \frac{srg}{nid}$.

Proof: Let $otk^i_{min} = rlt^i - rand_{max}$ and $otk^i_{max} = rlt^i - rand_{min}$ denote the minimum and the maximum value of an obfuscated identifier that has been obtained by an adversary, where $rand_{max}$ and $rand_{min}$ are chosen from the safe range $(0, srg)$. Then, we have the sieved range $RG_s = (otk^i_{max}, otk^i_{min} + srg)$, 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 $RG_{min} = (otk^i_{max}, rlt^i)$ and $RG_{max} = (rlt^i, otk^i_{min} + srg)$. Thus we have $E[|RG_s|] = E[|RG_{min}|] + E[|RG_{max}|]$.

The size of the range RG_{min}, denoted as $|RG_{min}|$, equals to $rand_{min}$ since is $rlt^i - otk^i_{max} = rand_{min}$. We show that the cumulative distribution function of $rand_{min}$ is given by Equation 2.

$$Pr(rand_{min} > rg) = \left(1 - \frac{rg}{srg}\right)^{nid} \tag{2}$$

Since an obfuscation $rand$ is chosen uniformly and randomly over a range $(0, srg)$, for $0 \leq rg \leq srg$, the probability that any obfuscation $rand$ is smaller than rg, denoted by $Pr(rand \leq rg)$, is $\frac{rg}{srg}$. Hence, the probability that any obfuscation $rand$ is greater than rg is $Pr(rand > rg) = 1 - Pr(rand \leq rg) = 1 - \frac{rg}{srg}$. Now we compute the probability that $rand_{min} = min\{rand_1, rand_2, \cdots, rand_{nid}\}$ is greater than rg. We have $Pr(rand_{min} > rg)$ = Pr$((rand_1 > rg) \wedge (rand_2 > rg) \wedge \cdots \wedge (rand_{nid} > rg))$ = $\prod_{j=1}^{nid}$ Pr$(rand_j > rg) = \left(1 - \frac{rg}{srg}\right)^{nid}$.

Now, using standard techniques from probability theory and Equation 2, one can derive the expected value of $rand_{min}$: $E[|RG_{min}|] = E[rand_{min}] \approx \frac{srg}{nid}$. Symmetrically, one can show that the expected size of range RG_{max} is $E[|RG_{max}|] \approx \frac{srg}{nid}$. Hence the expected size of sieved range is $E[|RG_s|] = E[|RG_{min}|] + E[|RG_{max}|] \geq \frac{srg}{nid}$. ∎ ■

VII. Location Inference Guards

Location inference attacks refer to those attacks wherein an adversary attempts to infer the location of a file using *indirect* techniques that exploit file metadata information such as file access frequency, file size, and so forth. LocationGuard includes a suite of four fundamental and inexpensive inference guards: lookup frequency inference guard, end-user IP-address inference guard, file replica inference guard and file size inference guard. LocationGuard also includes a capability revocation based location rekeying mechanism as a general guard against any inference attack. In this section, we present the four fundamental inference guards and the location rekeying technique in detail.

A. Passive Inference Guards

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 present two inference guards: lookup frequency inference guard and end-user IP-address inference guard to guard the file system against two common passive inference attacks. 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. 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. The malicious node r could log and report this information to the adversary.

1) Lookup Frequency Inference Guard: 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 [30]. 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 7.1 formalizes the notion of perfectly hiding a file from a frequency inference attack.

Lemma 7.1: 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 is perfectly hidden from frequency inference attack if $\lambda'_f = c$: $\forall f \in F$ and some constant c.

Corollary 7.2: A collection of read-only files can be perfectly hidden from frequency inference attack.

Proof: 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 file replicas in the system is identical, namely, $\forall f\ \lambda'_{f^i} = \frac{\lambda_f}{R_f} = c$ ($1 \leq i \leq R_f$ for file f). By Lemma 7.1, an adversary would not be able to derive any useful information from a frequency inference attack. ∎ ■

Interestingly, the replication strategy used in Corollary 7.2 improves the performance and load balancing aspect of the file system as well. 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.

Guard by Result Caching. The first technique to mitigate the frequency inference attack is to obfuscate 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 $\lambda'_f = \frac{\lambda_f \mu_{dep}}{\lambda_f + \mu_{dep}}$ (assuming exponential distribution for λ_f and μ_{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 f's 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 the identities of files stored at good nodes.

Guard by File Identifier Obfuscation. The second technique that makes the frequency inference attack harder is based on the file identifier obfuscation technique described in Section VI. Let $f_1, f_2, \cdots, f_{nf}$ denote the files stored at some node n. Let the identifiers of these replicas be $rlt_1, rlt_2, \cdots rlt_{nf}$. Let the target file be f_1. The key idea is to obfuscate the identifiers such that an adversary would not be able to distinguish between an obfuscated 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 $R_1 = (rlt_1 - srg, rlt_1)$. 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 appropriately choosing a safe obfuscation range. Figure 4 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 $otk \in R_1 \cap R_2$, 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 obfuscation might introduce significant overlap between the ranges of different files stored at node n.

[2]This is analogous to performing a frequency analysis attack on old symmetric key ciphers like the Caesar's cipher [17]

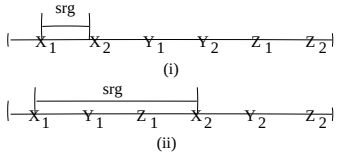

Fig. 4. Countering Frequency Analysis Attack by file identifier obfuscation. X_1X_2, Y_1Y_2 and Z_1Z_2 denote the ranges of the obfuscated identifiers of files f_1, f_2, f_3 stored at node n. Frequency inference attacks works in scenario (i), but not in scenario (ii). Given an identifier $otk \in Y_1Z_1$, it is hard for an adversary to guess whether the lookup was for file f_1 or f_2.

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 4) is given by Equation 3.

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

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 file identifier obfuscation on frequency inference attack in our experimental section VIII.

2) End-User IP-Address Inference Guard: In this section, we describe an end-user IP-address inference attack that assumes that the identity of an end-user can be inferred from his/her IP-address. 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. This is particularly true if the user obtains IP-address dynamically (DHCP [7]) from a large ISP (Internet Service Provider).

A 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 a 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 could itself become a viable target for the adversary.

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 non-trivial for an adversary to distinguish between the queries that *originated* at the user and those that were simply *routed* through it.

For the sake of simplicity, let us assume that q denotes the number of lookups issued per user per unit time. Assuming there are N users, the total lookup traffic is Nq lookups per unit time. Each lookup on an average requires $\frac{1}{2}\log_2 N$ hops on Chord. Hence, the total lookup traffic is Nq * $\frac{1}{2}\log_2 N$ hops per unit time. By the design of the overlay network, the lookup traffic is uniformly shared among all nodes in the system. Hence the number of lookup queries (per unit time) routed through any node u is $\frac{1}{N}$ * $\frac{1}{2}qN\log_2 N$ = q * $\frac{1}{2}\log_2 N$. Therefore, the ratio of lookup queries that originate at a node to that routed through it is $\frac{q}{q*\frac{1}{2}\log_2 N}$ = $\frac{2}{\log_2 N}$. For $N = 10^6$, this ratio is about 0.1, thereby making it hard for an adversary to selectively pick only those queries that originated at a particular node. Further, not all neighbors of a node are likely to be bad; hence, it is rather infeasible for an adversary to collect all lookup traffic flowing through an overlay node.

B. Host Compromise based Inference Guards

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 present two inference guards: file replica inference guard and file size inference guard to guard the file system against two common host compromise based inference attacks. 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 replica holder (unless the replica holder is malicious). The file size inference attack attempts to infer the identity of a file from its size. If the sizes of files stored on the overlay network are sufficiently skewed, the file size could by itself be sufficient to identify a target file.

1) File Replica Inference Guard: 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, 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 from the file replica inference attack.

We guard read-only files (and files updated very infrequently) by making their replicas non-identical; this is achieved by encrypting each replica with a different cryptographic key. We derive the cryptographic key for the i^{th} replica of file f using its location key lk as k^i = $khash_{lk}(f \parallel i \parallel$ 'cryptkey'). Further, if one uses a symmetric key encryption algorithm in cipher-block-chaining mode (CBC mode [19] [10]), then we could reduce the encryption cost by using the same cryptographic key, but a different initialization vector (iv) for encrypting different file replicas: k^i = $khash_{lk}(f \parallel$ 'cryptkey') and iv^i = $khash_{lk}(f \parallel i \parallel$ 'ivec').

We show in our experimental section that even a small update frequency on read-write files is sufficient to guard them the file replica inference attack. Additionally, one could also choose to encrypt read-write file replicas with different cryptographic keys (to make the replicas non-identical) to improve their resilience to file replica inference attack.

2) File Size Inference Guard: 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). We guard the file system from this attack by fragmenting files into multiple file blocks of equal size. For instance, CFS divides files into blocks of 8 KBytes each and stores each file block separately. We hide the location of the j^{th} block in the i^{th} replica of file f using its location key lk and token $rlt^{(i,j)} = khash_{lk}(f \parallel i \parallel j)$. Note that the last file block may have to be padded to make its size 8 KBytes. 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 dividing files into blocks is useful in minimizing the communication overhead for small reads/writes on large files.

C. 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*. This is analogous to periodic 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.

VIII. Experimental Evaluation

In this section, we report two sets of results. The first set of results is obtained from our prototype implementation of LocationGuard. The second of results is from simulation based experiments to evaluate the LocationGuard approach for building secure wide-area network file systems.

A. Implementation-Based Experiments

In this section we briefly sketch our implementation of LocationGuard and quantify the overhead added by LocationGuard to the file system.

Implementation. We have implemented a prototype of LocationGuard on a publicly available Java code for the Chord lookup protocol [26]. We used AspectJ [9] to modify the Chord lookup protocol to include routing guard and lookup result caching. The method *obfuscate* implements lookup identifier obfuscation. The method *check_safe_obfuscation* implements our check for safe obfuscation; if the check fails then it calls *obfuscate* followed by the Chord lookup protocol. The AspectJ compiler statically weaves the *obfuscate* method and the *check_safe_obfuscation* method before and after all method calls to the Chord lookup protocol respectively.

The file system is implemented on top of the overlay network. We split files into blocks of 8KBytes and store each block at a location determined by the file's location key. The file system is assumed to be flat (no directory hierarchy). The file names are simply the 32 Byte hexadecimal representation of the 128-bit file identifier. Access control in our system is implicit; if the file exists then the requested read/write operation is performed else an error is returned.

File Type	Description
T0	no cryptography
T1	integrity only
T2	confidentiality only
T3	confidentiality and integrity

TABLE II
LocationGuard File Types

LocationGuard permits files to be any one of the four types: no cryptographic security (T0), integrity only (T1), confidentiality only (T2), and confidentiality and integrity (T3). To ensure file integrity, the file includes a keyed message authentication code using the HMAC-MD5 keyed hash function. To ensure file confidentiality, the file is encrypted using the AES-128 encryption algorithm. Finally, adding message authentication code (using MD5 [24] or SHA1 [8]) followed by encryption (using AES-128) guarantees both file confidentiality and integrity. We assume that the file owners distribute location keys and cryptographic keys through a secure out-of-band mechanism. Figure 5 shows our implementation architecture and Table II shows the four file types.

Operational Overhead. We ran our prototype implementation on eight machines each with 8-processors (550MHz Intel Pentium III Xeon processor running RedHat Linux 9.0) connected via a high speed LAN. In reality the nodes would be distributed on a wide-area network. However, we believe that this setup would be equally insightful in providing us the percentage overhead added by LocationGuard.

We first quantify the performance and storage overheads incurred by LocationGuard. Let us consider a typical file read/write operation. The operation consists of the following steps: (i) generate the replica location tokens, (ii) lookup the replica holders on the overlay network, and (iii) process the request at replica holders. Step (i) requires computations 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 (a keyed pseudo-random function) and MD5 (a pseudo-random function) is negligibly small (order of a few microseconds) using the standard OpenSSL library [20]. 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 I) 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 pseudo-filename (token), the replica holder would honor a request on that file. Figures 6 and 7 shows the overhead of LocationGuard for file read and file write operations respectively. Each value reported in this experiment has been averaged over 64 runs. Note that file read/write operations of size greater than one block were parallelized, with each file block operation proceeding in parallel. Observe that the latency for file operations in a naive file system (FS) and LocationGuard (LGFS)

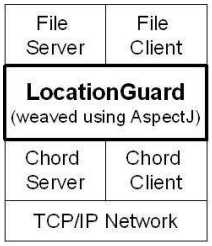

Fig. 5. Implementation Architecture

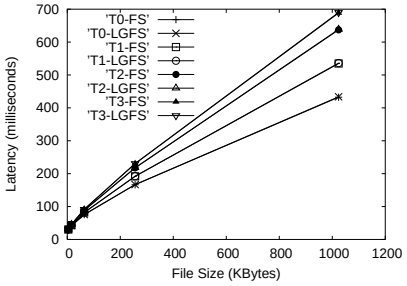

Fig. 6. File Read Overhead

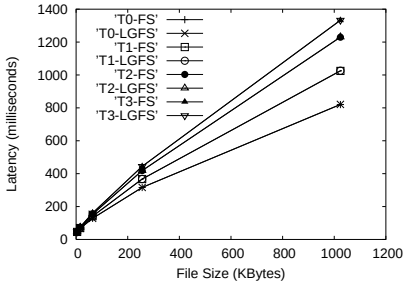

Fig. 7. File Write Overhead

is almost the same. For read operations maximum overhead due to LocationGuard was about 1.5ms (relative overhead of 0.4%) and that for write operation was 1.6ms (relative overhead of 0.3%).

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.

B. Simulation-Based Experiments

We implemented our simulator using a discrete event simulation [10] model. We simulate the Chord lookup protocol [29] 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 (the trends reported in this paper apply to all values of p). 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 III). We demonstrate the effectiveness of LocationGuard against DoS and host compromise based target file attacks.

Denial of Service Attacks. Figure 8 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, 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. Also, 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 LocationGuard, the effort required for an adversary to attack a target file is dependent only on R, but is independent of the number of good nodes (G) in the system. On the contrary, LocationGuard based techniques scale the hardness of an attack with the number of good nodes in the system. Thus even with a very small R, a LocationGuard based system can make it very hard for any adversary to launch a targeted file attack.

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 III
MEAN FRACTION OF GOOD NODES IN UNCOMPROMISED STATE (G')

Host Compromise Attacks. To further evaluate the effectiveness of LocationGuard against targeted file attacks, we evaluate LocationGuard 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 does 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 9 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 9 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 III). Table III shows the mean fraction of good nodes (G') that are in an uncompromised state for different values of ρ. Observe from Table III that when $\rho = 1$, most of the good nodes are in a compromised state.

As we have mentioned in Section IV-C, 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 probability of a attack assuming that the adversary collects the file tokens stored at compromised nodes. Figure 10 shows the mean effort required to locate all the replicas of a target file ($cr = R$). The effort required is expressed in terms

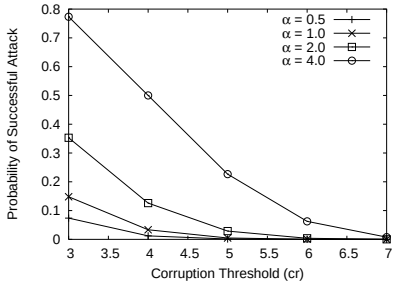

Fig. 8. Probability of a Target File Attack for $N = 1024$ nodes and $R = 7$ using DoS Attack

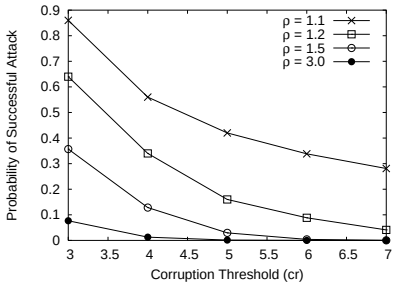

Fig. 9. Probability of a Target File Attack for $N = 1024$ nodes and $R = 7$ using Host Compromise Attack (with no token collection)

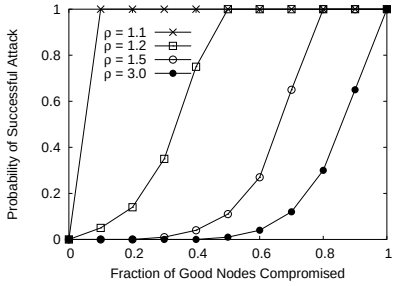

Fig. 10. Probability of a Target File Attack for $N = 1024$ nodes and $R = 7$ using Host Compromise Attack with token collection from compromised nodes

ρ	0.5	1.0	1.1	1.2	1.5	3.0
Rekeying Interval	0	0	0.43	1.8	4.5	6.6

TABLE IV
TIME INTERVAL BETWEEN LOCATION REKEYING (NORMALIZED BY $\frac{1}{\lambda}$ TIME UNITS)

μ_{dep}	0	1/256	1/16	1	16	256	4096	∞
S	15	12.64	11.30	10.63	10.00	9.71	9.57	9.55

TABLE V
COUNTERING LOOKUP FREQUENCY INFERENCE ATTACK APPROACH I: RESULT CACHING (WITH 32K FILES)

of the fraction of good nodes that need to be compromised by the adversary to attack the target file.

Note that in the absence of LocationGuard, an adversary needs to compromise at most R good nodes in order to succeed a targeted file attack. Clearly, LocationGuard 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 increase 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. Observe 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 VII-C we had proposed location rekeying to protect the file system from such attacks. The exact period of location rekeying can be derived from Figure 10. For instance, when $\rho = 3$, if a user wants to retain the attack probability below 0.1, the time interval between rekeying should equal the amount of time it takes for an adversary to compromise 70% of the good nodes in the system. Table IV 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 rekeying can be more and more infrequent.

C. Location Inference Guards

In this section we show the effectiveness of location inference guards against the lookup frequency inference attack, and the file replica inference attack.

Lookup Frequency Inference Guard. We have presented lookup result caching and file identifier obfuscation as two techniques to thwart the frequency inference attack. Recall that our solutions attempt to flatten the frequency profile of files stored in the system (see Lemma 7.1). 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 [30], 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* [18] 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 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 7.1). Table VI shows the maximum entropy (S_{max}) and the entropy of a zipf-like distribution (S_{zipf}) for different values of F. Note that 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.

Table V 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 V 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 VI 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 obfuscation in mitigating frequency inference attack. Figure 11 shows the entropy of the apparent file access frequency for varying values of pr_{sq} (the probability that obfuscated queries are safe, see Theorem 6.1) for different values of nf, the mean

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

TABLE VI
ENTROPY (IN NUMBER OF BITS) OF A ZIPF-DISTRIBUTION

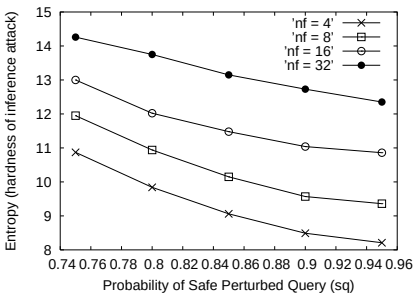

Fig. 11. Countering Lookup Frequency Inference Attack Approach II: File Identifier obfuscation

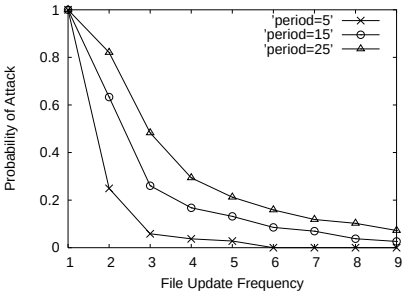

Fig. 12. Countering File Replica Frequency Inference Attack: Location Rekeying Frequency Vs File Update Frequency

number of files per node. Recall that an obfuscated identifier is safe if both the original identifier and the obfuscated identifier are assigned to the same node in the system. Higher the value pr_{sq}, smaller is the safe obfuscation range (srg); and thus, the lookup queries for a replica location token 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, there would be larger overlaps between the safe ranges of different files assigned to a node (see Figure 4). This evens out (partially) the differences between different apparent file access frequencies and thus, increases the entropy.

File Replica Inference Guard. 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. We assume that the adversary performs a host compromise attack with $\rho = 3$. Figure 12 shows the probability of a successful attack on a target file for different values of its update frequency and different values of rekeying 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 12; 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. Figure 12 also illustrates that lowering the time period between key changes lowers the attack probability significantly. This is because each time the location key of a file f is changed all the information collected by an adversary regarding f would be rendered entirely useless.

Inference Attacks Discussion. We have presented techniques to mitigate some popular inference attacks. There could be other inference attacks that have not been addressed in this paper. Even the location inference guards presented in this paper does not entirely rule out the possibility of an inference attack. 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 VI). Identifying other potential inference attacks and developing better defenses against the inference attacks that we have already pointed out in this paper is a part of our ongoing work.

IX. RELATED WORK

Serverless distributed file systems like CFS [6], Farsite [1], OceanStore [16] and SiRiUS [12] 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 secure files from malicious nodes. Unfortunately, cryptographic techniques cannot protect a file holder from DoS or host compromise attacks. LocationGuard presents low overhead and highly effective techniques to guard a distributed file system from such targeted file attacks.

The secure Overlay Services (SOS) paper [14] presents an architecture that proactively prevents DoS attacks using secure overlay tunneling and routing via consistent hashing. However, the assumptions and the applications in [14] 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 treats 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.

While we have described LocationGuard for a Chord [29] overlay network, we note that it also applies Pastry [25] and Tapestry [4]. An identity in Pastry (or Tapestry) can be obfuscated (using the same technique described in this paper) while being to preserve the lookup property. We note that within a small obfuscation range both identities id and $id + srg$ in Pastry (and Tapestry) are mapped to the same target node. On the other hand, it may be non-trivial to extend this scheme to d-dimensional CAN [23] network.

The Hydra OS [5] proposed a capability-based file access control mechanism. LocationGuard implements a simple and efficient capability-based access control on a wide-area network file system. 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 VI).

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 [21] 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.

A major overhead for LocationGuard arises from key distribution and key management. In particular, one can envision scenarios wherein (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 (and directories). Additionally, one could leverage literature on key management algorithms for access control hierarchies [2] to encode *nix-like access control policies. Note that *nix-like access control is essentially hierarchical since a user can read a file f only if it has execute (+x) permission on the parent directory containing file f. Inductively, the user must have execute permission on all directories in the absolute path to file f, and thus represented as hierarchical access control policies [2].

In the later scenario, one could use efficient group key management protocols to accommodate dynamic group membership updates (detailed survey in [22]). Such protocols typically incur a key update cost that is logarithmic in the number of users. Additionally, one could leverage recent advances in key management algorithms for temporal access control [3], wherein a user leases access to a file for some contracted time period (a, b). [3] requires no key update cost, a constant key distribution cost per lease and a public storage that is $O(T * \log\log T)$ (where T denotes number of time units of interest). We note that while LocationGuard mechanisms incur low overhead, the choice of key management protocols may significantly impact the file system's performance metrics.

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 [27] for detailed discussion on the robustness of lookup protocols on DHT based overlay networks.

X. Conclusion

We have described LocationGuard – a technique 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 hides the location of a file on an overlay network. LocationGuard protects a target file from DoS attacks, host compromise attacks, and file location inference attacks by providing a simple and efficient access control mechanism with minimal performance and storage overhead. The unique characteristics of LocationGuard approach is the careful combination of location key, routing guard, and an extensible package of location inference guards, which makes it very hard for an adversary to infer the location of a target file by either actively or passively observing the overlay network. Our experimental results quantify the overhead of employing location guards and demonstrate the effectiveness of the LocationGuard scheme against DoS attacks, host compromise attacks and various location inference attacks.

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 *5th OSDI*, 2002.

[2] M. Atallah, K. Frikken, and M. Blanton. Dynamic and efficient key management for access hierarchies. In *ACM CCS*, 2005.

[3] M. J. Atallah, M. Blanton, and K. B. Frikken. Incorporating temporal capabilities in existing key management schemes. In *ESORICS*, 2007.

[4] 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.

[5] E. Cohen and D. Jefferson. Protection in the hydra operating system. In *ACM SOSP*, 1975.

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

[7] R. Droms. RFC 2131: Dynamic host configuration protocol. http://www.faqs.org/rfcs/rfc2131.html.

[8] D. Eastlake and P. Jones. US secure hash algorithm I. http://www.ietf.org/rfc/rfc3174.txt, 2001.

[9] Eclipse. Aspectj compiler. http://eclipse.org/aspectj.

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

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

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

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

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

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

[16] 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 *9th ASPLOS*, November 2000.

[17] MathWorld. The caesar cipher. http://www.mathworld.com.

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

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

[20] OpenSSL. OpenSSL. http://www.openssl.org/.

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

[22] S. Rafaeli and D. Hutchison. A survey of key management for secure group communication. In *Journal of the ACM Computing Surveys, Vol 35, Issue 3*, 2003.

[23] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *ACM SIGCOMM*, Aug 2001.

[24] R. Rivest. The MD5 message-digest algorithm. http://www.ietf.org/rfc/rfc1321.txt, 1992.

[25] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *18th IFIP/ACM Middleware*, Nov 2001.

[26] A. Singh and M. Srivatsa. Apoidea: Decentralized P2P web crawling. In *SIGIR Workshop on Distributed Information Retrieval*, 2003.

[27] M. Srivatsa and L. Liu. Vulnerabilities and security issues in structured overlay networks: A quantitative analysis. In *In 20th ACSAC*, 2004.

[28] M. Srivatsa and L. Liu. Countering targeted file attacks using location keys. In *USENIX Security Symposium*, 2005.

[29] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM*, August 2001.

[30] 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.