The Mirage NFS Router

Scott Baker and John H. Hartman

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

Technical Report TR02-04

November 26, 2002

Abstract

Mirage is a system that aggregates multiple NFS servers into a single, virtual NFS file server. It is interposed between the NFS clients and servers, making the clients believe that they are communicating with a single, large server. Mirage is an *NFS router* because it routes an NFS request from a client to the proper NFS server, and routes the reply back to the proper client. Mirage also prevents DoS attacks on the NFS protocol, ensuring that all clients receive a fair share of the servers' resources. Mirage is designed to run on an IP router, providing virtualized NFS file service without affecting other network traffic. Experiments with a Mirage prototype show that Mirage effectively virtualizes an NFS server using unmodified clients and servers, and it ensures that legitimate clients receive a fair share of the NFS server even during a DoS attack. Mirage imposes an overhead of only 7% on a realistic NFS workload.

1 Introduction

Mirage provides NFS [Sandberg85] clients with the illusion that a collection of NFS servers is actually a single, virtual NFS server. Mirage is not an NFS server -- it is an *NFS router*, which operates by routing client requests to the appropriate NFS servers, and routing replies back to the appropriate clients (Figure 1). Mirage exports a set of file systems that is the union of the file systems exported by the NFS servers. Clients mount a file system from the virtual server and access its contents as if Mirage were a real, physical NFS server. Mirage is thus fully transparent to the clients and servers, allowing unmodified client and server implementations to use Mirage.

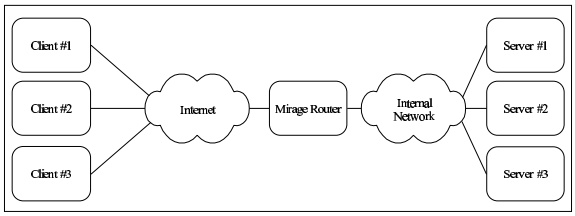

Figure 1: Mirage routes requests and replies between the NFS clients and the NFS servers.

An NFS file system is stored on a single NFS server. NFS has no provision for spreading a file system over multiple servers. Due to this limitation, as the system scales to more users, the NFS server will become overloaded with the increased demand. The system administrator has few solutions to this problem, including upgrading the NFS server or introducing an additional server and moving some files from the old server to the new. The former solution is expensive and disruptive, while the second requires reconfiguring the clients to be aware of the new server. Similarly, if an NFS server runs out of storage, the only way to gain additional space is to upgrade or add an additional server.

Mirage implements a *Virtual Server* abstraction that solves these system administration problems. Clients perceive a single NFS server, unaware that it is actually a virtual server that is the aggregation of the real

This work was supported in part by DARPA contract F30602-00-2-0560 and NSF grant CDA-9500991.

NFS servers. Files and file systems can be moved from one NFS server to another without reconfiguring the clients. Similarly, new NFS servers can be deployed without the clients' knowledge. All that is required is to reconfigure Mirage to include the new server in its virtualization. By hiding the server configuration details from the clients, Mirage allows NFS to scale to large numbers of servers without a overwhelming increase in system administration complexity.

Mirage virtualizes an NFS server such that the IP address of the virtual server is the IP address of the Mirage router. Client NFS requests are delivered to the Mirage router, which rewrites the requests and forwards them to the appropriate NFS server. The clients are unaware that Mirage is a router, or that it is aggregating multiple NFS servers into a single virtual server. No client modifications are necessary, which is advantageous because the NFS client protocol is usually implemented as an integral part of the host operating system on the client computer. To modify the NFS implementation would require installing a custom kernel on the client computer, a task that is beyond the abilities of most end users, and inconvenient for most system administrators.

Mirage also supports unmodified servers. Server modification is infeasible because many NFS servers are commercial products and contain proprietary code. Commercial servers also provide additional features such as snapshots, which not only are complicated to implement, but protected by patents. Furthermore, since Mirage doesn't require server modifications and exists outside of the server, it is always possible to access the NFS servers directly if Mirage suffers a failure or is otherwise unavailable.

Mirage contains support for preventing Denial of Service (DoS) attacks on the NFS protocol. A DoS attack is one in which a malicious client overwhelms the server with requests, preventing legitimate clients from accessing the server. These attacks may not be detected by existing DoS mechanisms because they may generate very little network traffic, yet induce a high load on the server. The request might be quite small, but require the NFS server a lot of effort to process. Mirage mitigates DoS attacks by ensuring that each NFS client receives a fair share of the servers' resources.

Mirage is designed to run on a programmable network router [Gottlieb02]. It has a minimal amount of state, and is able to recover its state automatically after a crash. Mirage operates by re-writing packet contents based on table-lookups, and therefore introduces a minimal amount of overhead to the packet processing. We have developed two prototype implementations, one as a user-level NFS proxy, and another as a Linux kernel module. The kernel module version has only a 7% slowdown of a large compilation benchmark over that of a simple IP router. We believe this to be an acceptable overhead for the benefits that Mirage provides.

2 NFS

Mirage implements the NFS protocol, version 2. The NFS protocol is based upon SUN Remote Procedure Call (RPC) [Sun88], and therefore uses a request/reply paradigm for communication between the clients and the servers. The protocol uses *handles* to represent files, directories, and other file system objects. An NFS handle is a 32-byte quantity that is generated by the server and opaque to the client. The client receives the handle for the root directory of a file system when it mounts that file system. The mount request includes the name of the file system to be mounted (the mount point). The NFS server checks the mount request for the appropriate security and authentication and then issues a reply containing the handle for the root directory of the file system. A client uses a handle to access the contents of its associated object, as well as to obtain handles for additional objects.

An NFS client obtains a handle for a desired object in an iterative fashion. It works its way through the desired pathname one component at a time, sending a lookup request to the server containing the handle of the current directory and the name of the desired component to the server, and receiving back a handle for the component. For example, to get a handle for the file /foo/bar, the client first sends a lookup request to the server containing root file handle and the string "foo", and receives back a handle for that directory. The client then sends then a lookup request containing the handle for /foo and the string "bar", and receives back the handle for /foo/bar. The client then uses the handle to read and write the file. Figure 2 illustrates the sequence of events required to read the file /home/fred/photo.

Client Request	Server Reply
Mount("/home")	$handle_{home}$
Lookup($handle_{home}$, "fred")	$handle_{fred}$
Lookup($handle_{fred}$, "photo")	$handle_{photo}$
Read($handle_{photo}$, 0-1024)	First 1024 bytes
Read($handle_{photo}$, 1024-2048)	Next 1024 bytes

Figure 2: Sequence of NFS client requests and server replies to read the first 2048 bytes of /home/fred/photo.

The use of handles in the NFS protocol poses two problems for Mirage. First, the NFS server generates handles however it desires, as long as different objects have different handles. Typically, the NFS server will encode information in the handle that identifies the location of the object in the server's internal file system, improving access performance. The handles are opaque to the client, however, so it is unaware of what information the handle holds. Since Mirage virtualizes multiple NFS servers into a single server, it must ensure that different objects have different handles, even if the objects reside on different NFS servers. Mirage has no control over how the servers create their handles, so there is no way for Mirage to ensure that the handles generated by the servers do not conflict. As a result, Mirage must virtualize the NFS handle space by creating its own (virtual) handles for objects, and mapping between its virtual handles and the physical handles used by the servers. This requires Mirage to maintain state about the mapping, increasing Mirage's complexity. Virtual handles and how they are managed in Mirage is described in more detail in Section 3.2.

The second handle-related issue that affects Mirage is once a client has a handle, it may use the handle indefinitely to access the associated object. The client never needs to refresh the handle, therefore the server cannot subsequently change the handle contents. If the server crashes and reboots, it may no longer have information about handles issued before the crash, causing it to return "stale handle" errors to the client. This causes the client to re-lookup the handle, if possible. Handle longevity is an issue for Mirage because it means that not only must Mirage maintain the mapping from a virtual handle to a physical handle indefinitely, but Mirage must also be able to reconstruct that mapping after a crash. If Mirage did not reconstruct the mapping, then Mirage would have to return a "stale handle" error to the client, causing Mirage to no longer be transparent to the client. Mirage's crash recovery mechanism is described in more detail in Section 3.3.

3 Mirage

Mirage addresses the NFS scalability problem by virtualizing multiple NFS servers as a single NFS server. This virtualization is entirely transparent to the client and the servers, so the clients are not aware they are dealing with a collection of servers. There are several issues that Mirage must resolve, including the abstraction presented to the clients, how virtual handles are maintained and validated, and how Mirage recovers from crashes.

3.1 Virtual Server Abstraction

There are at least two possibilities for the virtual server abstraction presented by Mirage. The simplest is that the set of mount points exported by the virtual server is the union of the sets of mount points exported by the underlying NFS servers (Figure 3). When the Mirage router receives a mount request it forwards the request to all of the NFS servers that it virtualizes. The server that exports the desired mount point will return its root handle, while the other servers return errors. Mirage generates a virtual handle, associates it with the root handle provided by the server, and returns it to the client. In this way the client can access the entire union of the mount points and believes it is communicating with one larger server. The advantage of this approach is that it requires a relatively small amount of state and processing on Mirage to implement. The Mirage router must simply associate different virtual handles with different servers, so that requests are directed properly. When a new virtual handle is created via a lookup on an existing handle, the new handle is associated with the same server as the existing handle.

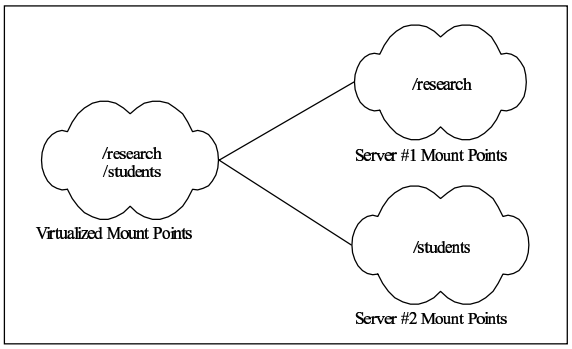

Figure 3: The set of mount points exported by the Mirage virtual server is the union of the sets of mount points exported by the individual NFS servers.

The downside of this approach is that the sets of mount points exported by the servers must be disjoint. Suppose two servers export mount points with the same name. What should the virtual server export? An alternative virtual server abstraction that addresses this issue is to export the union of the file system namespaces (Figure 4). For mount points that overlap, the virtual server exports a single mount point whose name space merges the name spaces of the individual mount points. When the client mounts an overlapping mount point, subsequent accesses to that file system must be directed to the proper NFS server on a case-by-case basis. This approach allows for more flexibility in the organization of the underlying servers, but increases the virtual server's com-

plexity because it must map virtual handles to NFS servers individually. If a new virtual handle is created using a lookup on an existing handle, it is possible that the new handle should be associated with a different server than the existing handle. Furthermore, this approach introduces semantic problems if there are name conflicts in the underlying file systems. Suppose a directory on one NFS server has the same name as a file on another server. What type of object should the clients see? It isn't obvious that there is a correct answer to this problem.

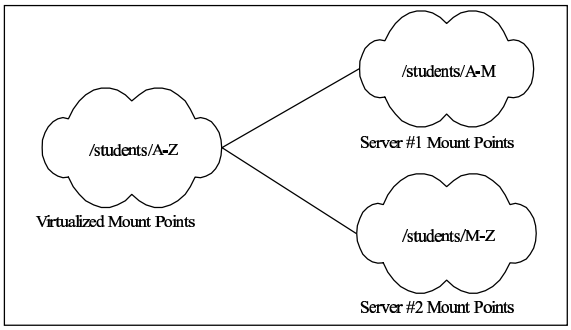

Figure 4: The virtual server exports file systems whose namespaces merge the namespaces of the underlying exported file systems.

The current Mirage prototype uses the former approach of exporting the union of the mount points exported by the NFS servers. This approach is simpler to implement, demonstrates the most of the advantages of a virtual NFS server, and doesn't have the semantic problem of naming conflicts within file namespaces. We are currently experimenting with the "union of file namespaces" approach and have a partial implementation in Mirage. Its viability and value is an area of future work, however.

3.2 Virtual Handle Mapping

One of the core functions of the Mirage router is to map between the file handles produced by the Mirage router and the file handles produced by the NFS servers (Figure 5). The clients cannot be presented with the NFS server handles directly because there is no guarantee that the NFS servers won't generate the same handle for different objects. For this reason, Mirage generates its own file handles that uniquely identify objects. We refer to the file handles generated by Mirage as *virtual file handles* (VFH), and those generated by the NFS servers as *physical file handles* (PFH). Mirage stores the mapping between a VFH and a PFH in a memory-resident *handle table*. When Mirage receives an NFS request from a client, Mirage looks up the VFH in the handle table to determine the proper server and PFH for the request. Mirage then rewrites NFS request using the PFH and forwards it to the server.

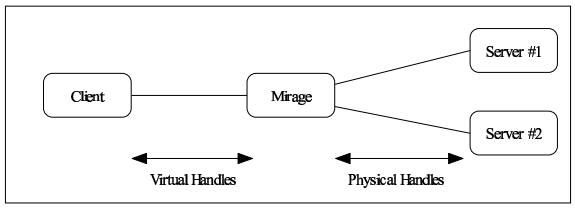

Figure 5: The Mirage router maps between the virtual file handles used by the clients and the physical file handles used by the NFS servers.

Mirage must perform a reverse mapping on NFS replies. Each PFH in a reply must be mapped to the appropriate VFH before the reply is forwarded to the client. The most common reply to contain a PFH is the reply to the Lookup request that is used to resolve a file name into a file handle. Mirage looks up the PFH contained in the reply in the handle table and rewrites the NFS reply with the correct VFH before forwarding the reply to the client. If this is the first time the PFH has been used, then Mirage generates the new VFH and stores it in the handle table.

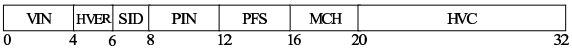

Figure 6: Format of a virtual file handle. Offsets are in bytes.

The handle table is a concern because it represents state on the Mirage router that consumes memory resources and must be recovered after a crash. It also requires processing to look up handles in the table. Mirage minimizes the state and processing resources of the table by encoding information in the VFH (Figure 6). The VFH is a 32-byte quantity that is opaque to the client, make all 32 bytes available for Mirage's use. Mirage encodes the following information in the VFH (Figure 6):

Virtual Inode Number (VIN): Every object has a unique VIN.

Server ID (SID): The SID identifies the NFS server that stores the object associated with the VFH.

Handle Version (HVER): The handle version number allows multiple handles to be issued for the same object. This is used by the DoS prevention mechanism to invalidate compromised handles (Section 4.4).

Physical Inode Number (PIN) and Physical File System ID (PFS): The PIN is the inode number of

the object as assigned by the NFS server, and the PFS is the file system ID on the server. These two fields thereby uniquely identify an object within the physical file server and are used during recovery (Section 3.3). The triple (SID, PIN, PFS) uniquely identifies an object within all of Mirage.

Mount Checksum (MCH): The MCH is a 32-bit checksum of the name of the mount point associated with the VFH. The MCH speeds up Mirage recovery (Section 3.3).

Handle Validation Checksum (HVC): The HVC is a cryptographically secure checksum that includes the other fields of the VFH and prevents clients from forging VFHs (Section 3.4).

3.3 Mirage Crash Recovery

If a traditional IP router fails, the Internet routes packets around it. This process is transparent and the communicating parties are not aware that their traffic has shifted from one router to another. Since Mirage is an NFS router, Mirage must provide the same degree of transparency. Clients and servers should not be aware of Mirage router failures, in the sense that NFS service will be unavailable while Mirage is down, but will resume once Mirage reboots without any special processing on the part of the clients or the servers.

The biggest issue with recovering from a Mirage crash is recovering the contents of the handle table. Mirage simplifies this process by ensuring that the contents of the handle table represent soft state. Soft state is state that can be discarded or lost because it can subsequently be regenerated. Hard state, on the other hand, must be protected at all costs because it cannot be regenerated if lost and will affect the correct functioning of the system. This usually means maintaining multiple copies of the hard state, either on redundant computers or on backup media. Mirage avoids the problems of hard state by ensuring that the contents of the handle table can be regenerated by existing mechanisms in the NFS protocol. This allows Mirage to lose the handle table during a crash, or discard part of the table if it gets too large, without affecting correctness.

Mirage uses a uniform mechanism for dealing with lost handle table state, that deals with handle table information lost in a crash, as well as handle table entries discarded because the table grew too large. If a client presents a VFH that is not found in the handle table, Mirage initiates handle recovery. The first step is to ensure that the VFH has not been forged (Section 3.4). Once the VFH has been authenticated, the fields it contains are used to recover the proper server and PFH to which it should be mapped.

The Server ID (SID) allows Mirage to determine which server the file came from, and the Physical File System (PFS) and Physical Inode (PIN) uniquely identify the file within that server. Ideally, the Mirage router would present this information to the NFS server and obtain a handle for the object. Unfortunately, the NFS protocol does not include such a function. Therefore, Mirage is forced to search the server until it finds an object with the correct PFS and PIN.

Although searching a server for the correct PFS and PIN can be a lengthy process, Mirage uses two techniques to speed up the search. First the mount checksum (MCH) reduces the file systems that must be searched. The MCH is a simple 32-bit checksum of the mount point for the handle. When the client originally mounted the file system, Mirage computed the MCH and stored it in the root file handle. From the point on, every time a name lookup was performed, the MCH was propagated from parent handle to child handle. Therefore, every handle includes a valid MCH that will identify the file system to which the VFH belongs.

During handle recovery, Mirage uses the "exports" function of the Mount protocol to get a list of all exported mount points from the NFS server. The checksum of each mount point is computed, and only those matching the MCH in the VFH are searched. If more than one mount point matches, then multiple file systems may have to be searched, but this should be infrequent. Normally, the MCH should remove the vast majority of file systems from consideration.

Once Mirage has used the MCH to determine that a file system may contain the desired object, Mirage enumerates all files and directories contained within that file system looking for one whose PFS and PID matches. The enumeration is performed by issuing "Read Directory" NFS requests. The Read Directory calls are issued recursively in a depth-first search order. The Lookup NFS function is used to get the PFH, PFS and PIN for every object until the desired object is found, allowing the handle table to be updated with the VFH to PFH mapping.

Mirage speeds up the handle recovery process by caching the PFH, PFS, and PIN information obtained during the recursive search. This information is used during subsequent handle recoveries to avoid con-

tacting the server, and substantially speed up recovery.

Although handle recovery is potentially a slow process, it is guaranteed to find the correct PFH for a VFH, allowing Mirage crashes to be transparent to the clients and servers. Handle recovery could be made nearly instantaneous if the NFS server provided a function that mapped from PFS/PIN to PFH. Such a function would probably be simple to implement, but would obviously require server modifications, which violates Mirage's goal of using unmodified NFS servers.

3.4 Handle Validation

Mirage's handle recovery mechanism is itself a potential target for a DoS attack because every time Mirage is presented with a VFH that is not in the handle table it begins the expensive handle recovery process. A malicious client could wreak havoc on Mirage by flooding it with invalid file handles. Mirage uses the HVC field in the VFH to protect itself from such an attack. The HVC stored in a VFH is a cryptographically secure hash of a Mirage secret, the client's IP address, and the other fields of the VFH. The MD5 algorithm [Rivest92] is used to generate the cryptographically secure hash. MD5 has the property that it is computationally infeasible to generate two blocks of data that hash to the same value. When Mirage generates a VFH it computes the associated HVC and stores it in the VFH.

When a Mirage router receives a VFH it first computes the HVC for the VFH, source IP address, and Mirage secret, and compares the computed HVC with the HVC stored in the VFH. If they match, the handle is valid. Otherwise the handle is invalid, and the request discarded. By including a Mirage secret in the hash, Mirage ensures that no one but a legitimate Mirage router can generate a VFH with a valid HVC. The system administrator configures the Mirage secret into the Mirage router, much like the "root" password is configured on a Unix computer. This secret can either be stored in persistent storage on the router (preferably), or typed every time the Mirage router reboots.

The HVC also includes the IP address of the client associated with the VFH. Including the IP address in the HVC ensures that one client is not able to use another client's VFH, perhaps obtained by sniffing on the network. This does not prevent IP spoofing, however, which requires a separate mechanism to prevent (Section 4.4).

Finally, the HVC includes the other fields of the VFH to prevent modification by a malicious client. If these fields could be modified, then the client could change the mapping of VFH to PFH and access objects improperly.

4 Denial of Service Prevention

A Denial of Service (DoS) attack occurs when malicious clients prevent a resource from being used by legitimate clients. Attacks range from destructive attacks that could render the contents of a server useless, to attacks that merely consume resources in an attempt to inconvenience other users. Most techniques for preventing DoS attacks focus on the low-level network protocols, such as TCP. For example, the TCP SYN attack [Schuba97] is a very popular form of distributed denial of service attack that targets TCP's connection mechanism. Mirage, on the other hand, focuses on preventing attacks on the NFS protocol, since Mirage is an NFS router. Mirage's NFS DoS prevention mechanism should be combined with DoS prevention mechanisms for the lower levels of the protocol stack, although we have not done so in our prototype.

A destructive DoS attack is one that denies service by destroying the underlying resource. Not much can be done about destructive attacks, other than to ensure timely backups of important information. For example, if an attacker is able to steal the password to an account on a file server, then the attacker is able to delete or overwrite files at will. The only way such an attack can even be detected is when the legitimate user shows up and notices that data have been compromised.

The best way to prevent destructive attacks is through proper security – ensuring users choose good passwords and protect the passwords from theft. The saving grace of a destructive attack is that only the files that are owned by the compromised account can be destroyed. However, once an attacker has gained access to an account, he is able to carry out a resource-consumption attack.

Resource-consumption DoS attacks have the ability to affect all users of the file server, not just the account (or accounts) that the attacker may have broken into. For example, an attacker could flood the server with useless write requests that consume server resources by causing the server to repeatedly write to its disk. By causing a bottleneck on the server's disk, all legitimate clients will have their productivity affected.

4.1 NFS DoS Scenarios

Mirage's DoS prevention mechanism is designed to mitigate DoS attacks on the NFS protocol itself. To help in understanding what types of attacks Mirage can and cannot prevent, it is useful to categorize the different scenarios under which a NFS DoS attack can occur (Figure 7).

Client Compromised ?	Authentication	Result
Yes	AUTH_UNIX	Attacker can impersonate any user and delete any object on the server.
	DES or Kerberos	Attacker can only impersonate users that he can authenticate with the third party server.
No	AUTH_UNIX	Attacker can only impersonate users for whom he knows the username and password.
	DES or Kerberos	Attacker can only impersonate users for whom he knows the username and password and can authenticate with the third party server.

Figure 7: Classification of NFS DoS scenarios.

The first classification of scenarios is whether or not the operating system on the client computer issuing the attack has been compromised. If a client computer hasn't been compromised, then the attacker is running a user-level program that accesses NFS through system calls. This means that the operating system is issuing valid NFS requests from valid ports using valid credentials. An attacker can use an uncompromised client for an attack, but he must do so within the framework that the operating system provides.

The degree of an attack from an uncompromised computer is bounded by the rules that are enforced by the operating system. For example, Linux computers typically issue a maximum of four NFS requests at once. To issue more than four requests, the computer must wait for the requests that are outstanding to be resolved first, leading to an implicit method of flow control. Furthermore, since the operating system is creating the NFS requests, it's safe to assume that the requests are properly formulated and contain valid parameters. For example, the user identification that is sent in the request will pertain to the actual user account that is logged into the machine.

An attack from a compromised client is more difficult to handle, but probably more likely. The attack is more difficult to prevent because there is no longer any guarantee that the NFS requests issued will properly follow the NFS protocol, or even be formatted properly. The attacker can issue bogus requests from secure port numbers, and present falsified credentials in the NFS request, making it difficult to determine the attacker's true identity. The attacker can issue hundreds or even thousands of requests in parallel. The attacker can even spoof IP addresses and impersonate other computers.

The second classification of NFS DoS scenarios is by the type of authentication used by the NFS protocol itself. By default, NFS uses AUTH_UNIX, a very weak authentication method in which the client operating system presents the user id and group id of the current user as the authentication credentials. The server compares these credentials against the user id and group id of the desired object before granting access. If a client computer is compromised, then it is very easy for the attacker to use a forged user id and thus impersonate any user on the system.

More secure authentication methods such as DES [DES88] and Kerberos [Steiner88] are available for NFS and are typically used when high security is desired. DES and Kerberos use opaque tokens for authentication that are very difficult to forge. DES and Kerberos use a trusted third party authentication server to independently authenticate the clients.

Mirage's DoS mechanism does not prevent attacks if the client is compromised and AUTH_UNIX authentication is used. This is because security is so minimal that an attacker can delete or corrupt any or all of the files on the file server, rendering the server useless. Mirage can prevent DoS attacks in the other three scenarios, however, assuming that the attacker only knows a subset of the valid credentials (i.e. name and password pairs). If the attacker knows all of valid credentials, then it is be simple for the attacker to implement a destructive attack that destroys all files on the server.

4.2 Invalid NFS Request Attack

In an invalid NFS request attack the attacker floods the NFS with bogus requests that consume server resources. For example, an attacker could generate

random file handles, causing the server to repeatedly resolve the handles and bypassing the server's internal caches. An attacker could repeatedly attempt to exceed his disk quota, read past the end of a file, or write to a file that is read-only. Invalid request attacks fall into two categories: those that contain invalid file handles and those that contain valid handles but invalid operations. Requests that contain forged file handles are easy for Mirage to detect because the HVC field will be invalid. Mirage simply computes the HVC for the presented VFH, and drops the request if the HVC stored in the VFH does not match the computed HVC.

Requests that contain valid handles but invalid operations are more difficult to detect. There are various scenarios in which an invalid operation could be issued but does not constitute an attack. Consider a situation where one client truncates a file while another client is reading from that same file. Clearly, this is an example of an error condition, but not necessarily a DoS attack.

One way to deal with invalid request attacks is to cache the results of all invalid requests so that the router responds directly to an invalid request rather than forwarding the invalid request to the server. Consider the read beyond end of file scenario. If a client tries to read beyond the end of the file once, and the file is not modified, then if the client attempts to read it again, the result will certainly be another read past end of file error. As long as the router knows that a request is invalid, it is safe to send the error reply directly from the router, without bothering the file server.

In an initial Mirage prototype, we constructed a cache-based invalid request responding system. The results of any invalid requests were cached and if the identical request showed up in the future, Mirage replayed the same error code that was issued the first time. The exception was in the case that an operation occurred that would invalidate an error condition. Continuing the read past end of file example, any write operation on the same file caused the "read past end of file" rejection that was stored in the cache to be discarded, so that future reads would again be sent to the server.

We found that this mechanism worked very well for preventing invalid request attacks. Mirage could be deployed at the periphery of the network and would limit the number of invalid requests that made it through to the file servers. However, we found in our experimentation that valid request attacks posed a much greater threat than invalid request attacks, and that we could develop a mechanism that would handle both cases.

4.3 Valid NFS Request Attacks

In a valid NFS request attack the attacker floods the server with valid, but spurious, requests. The requests are designed to consume resources by executing expensive operations such as creating files or writing data. For example, an attacker could repeatedly write to a file causing the server to continually flush data to its disk. This is loads the server, and diminishes other clients' access to the server.

The trouble with a valid request attacks is that it is difficult to detect. It is difficult to determine that one client's stream of requests is doing useful work while another client's stream of requests is serving no useful purpose. Mirage finesses this issue by treating it as a scheduling problem. Rather than detecting and prevent DoS attacks, Mirage simply ensures that each client receives a fair share of the server. If there are N clients using the file server, then each client should receive 1/N of the server's resources. If any one client attempts to use more than its fair share, then it is throttled back to enforce fairness.

Allocating a fair share of the server's resources to each client is made difficult because different NFS requests consume different amounts of server resources. For example, a write request will likely consume more server resources than a simple Lookup request. It isn't sufficient to simply ensure that all clients issue requests at the same rate; Mirage must also take the type of NFS requests into account. For this reason, the Mirage scheduler does not weight all requests equally, but rather assigns a higher cost to more costly requests such as writes and file creations. In our prototype, the Mirage assigns static costs to different requests. The costs that are used are a combination of the size of the packet and a penalty for expensive operations such as creates and writes (Figure 8).

Request Type	Cost
Getattr, Lookup, ReadLink, Read, ReadDir, StatFs	packet size
Symlink, Rename, Link, Rmdir, Remove	packet size + 500
Write	packet size + 1000
Create	packet size + 2000

Figure 8: NFS request costs.

The current Mirage prototype uses static costs because they are simple to implement. Ideally, the costs would be computed dynamically, by monitoring the time it takes different types of requests to be serviced by the server. This is an area of ongoing research.

4.4 Stolen Handles

One way for a malicious client to use more than its fair share of the server's resources is to issue requests using handles stolen from other clients, such as by sniffing the network. In order to use the stolen handle, however, the attacker also has to spoof the IP address of the owner of the handle. The handle is protected by the HVC and the HVC includes client's IP address. If the attacker doesn't spoof the IP address, then Mirage would detect the invalid HVC and drop the request. If the attacker does spoof the IP address, then Mirage cannot detect the invalid handle directly. If DES or Kerberos authentication is used, however, then authentication will fail on the server and the server will return an error. Mirage keeps track of handles that generated authentication errors and considers them potentially stolen. Such handles are stored in a *stolen handle table*, and a request containing a stolen handle is assigned a higher cost than a request that does not. The cost associated with a stolen handle is increased every time the handle generates an authentication error.

As a result, only the performance of requests issued using a stolen handle is degraded. Once Mirage has flagged a handle as stolen, it never returns the same handle in an NFS reply. Instead, Mirage increments the Handle Version Number (HVER) in the handle so the next time a valid client performs a lookup, a new handle is generated. This causes legitimate clients to stop using the stolen handle.

It is possible for a legitimate client to lose time synchronization with the NFS server and cause a DES or Kerberos authentication error to occur. This is not a denial of service attack, but is a rarely occurring side-effect of the DES and Kerberos authentication methods. Since the weight penalty assigned to stolen handles is proportional to the number of authentication errors a handle receives, the effect of a single authentication error has a minimal impact on a valid client's performance.

5 Implementation

We have developed two versions of Mirage – a user-mode Unix process, and a Linux kernel module[1]. The user-mode version is used for rapid prototyping, while the kernel-mode version is used for experimentation. Both are written in C and use the same set of functions for the core Mirage functionality, but use different mechanisms for sending and receiving packets. The core code is responsible for maintaining the handle table, rewriting packets, and performing handle recovery. The user-mode process uses sockets bound to local port numbers to read and write packets. One implication of this is that the user-mode Mirage cannot modify the UDP and IP headers on the packets it sends, so that it cannot put the client's addresses in the headers. As a result, packets forwarded to the NFS servers by the user-mode Mirage appear to come from the Mirage router itself, not the clients. This means that Mirage is not transparent to the servers, but is an unavoidable result of using a user-mode process.

The kernel-mode version of Mirage is implemented as a Linux kernel module that can be loaded into Silk [Peterson01]. Silk stands for "Scout in the Linux Kernel" and is a port of the Scout operating system [Peterson94] to Linux. Silk provides a flexible message passing mechanism that separates packet classification from protocol processing. Silk uses a *path* abstraction for scheduling and resource management. A path represents a source and sink of packets, along with the protocol processing necessary when moving the packets from the source to the sink. A path can be specialized according to any invariants in its packets. For example, a path can be created for UDP packets from a specific source IP address to a specific destination IP address and port. Since all packets on the path have the same type (UDP) and the same source and destination addresses and ports, the path can be optimized to improve overall performance.

The Silk version of Mirage is implemented as a module that sits on top of the UDP module. Mirage uses three types of paths: client/Mirage, Mirage/server, and client/server. Client requests arrive on the client/Mirage path. The first time Mirage receives a request from client it creates a path for the requests from that client. The path is optimized for that client, and has per-path state that includes the handle table for the client.

---

[1] The Mirage source code is available at http://www.cs.arizona.edu/mirage/.

Once Mirage has decoded the server ID from the VFH in the request, it rewrites the handle and other fields of the request appropriately and sends the packet on the Mirage/server path associated with the server ID. This path is optimized to send packets to the particular NFS server, and rewrites the destination address in the request to be the address of the NFS server. The source address is left unchanged, however, so that to the server the request appears to come from the client.

Replies from the server arrive on the client/server path. Mirage changes the source address in the reply to Mirage's IP address. Thus it appears to the client that the reply came from Mirage, as is necessary for the virtual server abstraction.

Mirage's DoS prevention mechanism uses "charge-based proportional scheduling" [Maheshwari95]. Each client has a bank balance, and each client has a queue of pending requests (up to 16 maximum). The scheduler iterates through the clients in a round-robin fashion, and if the current client has a pending request and a positive bank balance, then the request is sent to the NFS server and the bank balance reduced by the cost of the request. If there are pending requests but all clients have negative bank balances, then all bank balances are increased by an equal amount.

6 Performance

We measured Mirage performance using both micro-benchmarks that show the raw Mirage performance, and macro-benchmarks that use an end-to-end test to show the overall performance in a real-world situation. We ran the tests on the kernel-mode version of Mirage, running on Linux 2.4.7 and Silk 3.0. The configuration included three computers consisting of an NFS client and an NFS server connected to a Mirage router via 100 Mbs Ethernet. The client and router were 1.7 GHz Pentium 4 machines, while the server was a 2x600 MHz Pentium 3. Each machine had 256 MB of memory. All results reported are the average of 5 runs.

6.1 Micro-Benchmark

The micro-benchmark measures the latency and bandwidth of the Mirage router. The benchmark consists of sending 128-byte[2] Lookup requests from the client to the server. For comparison purposes, the performance of a standard IP router implemented in Silk is also presented. Two versions of the IP router were measured. The standard IP router provided by Silk uses "raw paths" and a level-3 cache to forward packets from the input network interface to the output interface. The details of this aren't relevant to this paper, but suffice it to say that these two mechanisms bypass most of the Silk functionality and simply forward packets between network interfaces at interrupt level. Although the "raw" IP forwarding numbers are indicative of a real IP router, we also measured Mirage against a Silk-based IP router that uses the standard Silk path mechanism to forward packets. This allows Mirage's overhead to be measured. In either case, it is important to note that Mirage's performance does not affect the performance of forwarding other types of packets in the same router.

The latency of the micro-benchmark was measured as the time from when a packet was received by Silk to the time that Silk sent the packet, and is shown in Figure 9. As can be seen, Mirage processing introduces a significant latency as compared to IP forwarding using "raw" paths, but a more acceptable amount for IP forwarding using standard paths. From this we conclude that 10 us of Mirage's request latency is due to path overhead, and only 5 us is due to Mirage processing. Improving Mirage's latency is an area of ongoing work.

	Request (us)	Reply (us)
Silk IPFW (raw)	2	2
Silk IPFW (path)	12	11
Mirage	17	11

Figure 9: Time to forward a 128-byte Lookup request and reply.

The throughput test measures the rate at which the different routers forward packets. The client generates an offered load on the router by sending packets of various sizes to the router at a specific rate. The server counts how many packets are successfully received, and the difference between the packets sent and the packets received is the number of packets dropped by the router. The offered load is varied to determine at what point the router begins to drop packets, and how the forwarding rate behaves as the offered load is increased beyond that point.

[2] All sizes reported for NFS packets are the size of the NFS header, RPC header, and payload.

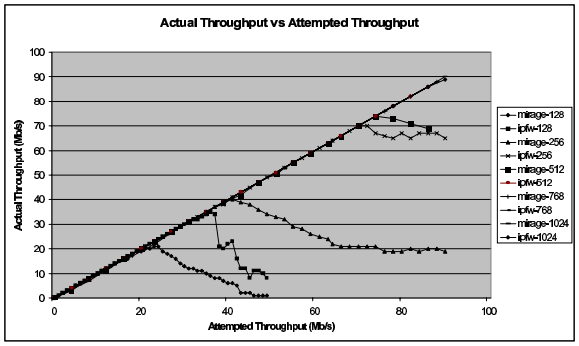

Figure 10: Router throughput as a function of attempted throughput (offered load).

Figure 10 shows the results of the throughput test. If the router doesn't drop packets then the actual throughput of the packets that reach the server matches the attempted throughput of the packets sent by the client, producing a diagonal line. Once the router starts to drop packets, the actual throughput falls below the diagonal. With 128-byte packets, Mirage starts to drop packets at an offered load of about 21 Mb/s, whereas IP forwarding doesn't drop packets until 37 Mb/s. Similarly, with 256-byte packets Mirage starts to drop packets at about 40 and IP forwarding starts to drop packets at about 71 Mb/s. The points at which the two systems begin to drop packets correspond to packet rates of about 20,000 packets/second for Mirage and about 40,000 packets/second for IP forwarding. Mirage is only able to sustain about half the packet rate of IP forwarding, due to the overhead of Mirage's processing.

As the attempted throughput is increased beyond the point that the router drops packets the actual throughput begins to drop off as the router spends more and more time handling the incoming packets, only to drop them. As a result, the actual throughput drops towards zero as the router approaches livelock.

At larger packet sizes the 100 Mbps Ethernet link is the bottleneck rather than the router. For example, for 768-byte packets, the physical link is only capable of transporting approximately 15,000 packets per second. Since Mirage can sustain 20,000 packets/second, it can operate at full line speed.

6.2 Macro-Benchmarks

Although the micro-benchmark is illustrative of Mirage's performance as compared to an IP router. A real NFS server cannot process NFS requests at the same rate than an IP router can route them, so Mirage's effect on NFS performance is likely to be much less pronounced.

The first macro-benchmark we used to measure Mirage's effect on NFS performance is an end-to-end bandwidth that consists of copying a 256 MB file from /dev/zero to an NFS file server and reading it back. The block size used for reads and writes is 1024 bytes.

	Write (MBs)	Read (MBs)
Silk IPFW	2.46	2.60
Mirage	2.35	2.41

Figure 11: Bandwidth of reading and writing a 256 MB file.

The end-to-end bandwidth test shows that Mirage's bandwidth is about 5% lower on writes and 7% lower on reads (Figure 11). The decrease in performance is due to the additional latency that Mirage requires on the router, which prevents the client and server from keeping as much data in flight as with IP forwarding. Since Linux limits NFS to four requests at once, the time taken to receive a reply delays the next request, and Mirage's higher delay therefore reduces bandwidth.

The second macro-benchmark represents a more realistic use of Mirage, and consists of unpacking and compiling the Linux 2.4 kernel distribution. This test creates over 10,000 individual files, and causes the client to issue approximately 472,000 NFS requests. The experiment took 7:52 minutes to complete using Silk IPFW, and 8:24 to complete on Mirage, a difference of 32 seconds or 7%. Thus, on a real-world workload Mirage induces an overhead of 7%, which we consider acceptable given the benefits of using Mirage.

6.3 DoS Benchmark

The DoS benchmark measures Mirage's ability to prevent DoS attacks. In this experiment, two client machines send request to Mirage. One client is a "good" client performing simulated useful work consisting of a mix of Lookup, GetAttr, and Write requests. The other client is an attacker that executes a rapid mix of simultaneous write requests. Figure 12 shows the throughput of the good client (in terms of successful requests per second) as the number of the attacker's simultaneous requests is varied, both on Mirage with DoS prevention enabled and with it disabled for comparison.

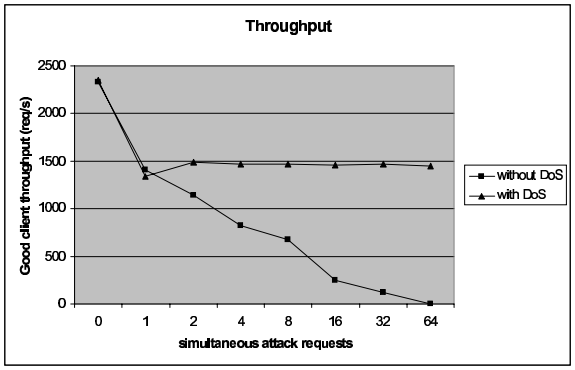

Figure 12: Throughput of "good" client while Mirage undergoes DoS attacks. Mirage's DoS mechanism prevents the good client's throughput from degrading. Note that the x-axis is logarithmic.

As can be seen, Mirage's DoS prevention mechanism allows the good client to maintain about 1500 successful requests independent of the size of the attack, a little more than half of its rate with no attack. This is to be expected, since the good client must share the server with the attacker. If DoS prevention is turned off, however, the number of successful requests drops off precipitously as the number of simultaneous attacks increases, reaching almost zero when there are 64 simultaneous attacks.

7 Related Work

Base [Rodrigues01] provides Byzantine fault tolerance for the NFS protocol by replicating objects on multiple NFS servers. Base is implemented via a user level relay process that mediates communication between the NFS client built into the operating system and the NFS servers. The idea is to use a heterogeneous collection of NFS servers, so that implementation-specific software bugs do not cause all the servers to fail. Base uses a handle translation mechanism similar to Mirage where the relay process translates client handles to server handles via a table. The Mirage handle recovery mechanism of using an exhaustive search is derived from Base. Unlike Mirage, multiple Base relay processes coordinate their actions, so that a client can access its files through any of them.

Deceit [Siegal90] is a distributed file system that combines multiple servers to provide the illusion of a single large NFS server. Deceit differs from Mirage in that it requires modified servers. The Deceit servers provide a superset of the NFS protocol and require the ISIS protocol for server-to-server communication. Deceit requires each server to function as both a server for its own files and as a router to forward requests from a client to another server's files. To use the automatic failover and version control features, Deceit requires client modification as well.

Slice [Anderson] provides request routing in which a micro-proxy functions as a switch that routes client requests to a group of bulk data servers and file managers. Slice uses a proprietary OBSD protocol for communication between the micro-proxy and the bulk data servers. The file managers are distinct from the bulk data servers and manage the directory structure and metadata of the file system. A drawback of Slice is that standard NFS servers cannot be used.

Locality-Aware Request Distribution [Pai98] is a mechanism for routing requests to servers to enhance load distribution and locality. LARD uses a front-end that performs routing, and a collection of back-end servers that each export identical data. The key observation with LARD is that the collection of servers will be most efficient if the requests always hit in the cache of the back-end servers. If the current working set were divided amongst the back-end servers in round-robin manner, then the back-end servers would each have to cache the entire working set. Instead, the LARD front-end divides the working set into a collection of subsets and assigns each subset to one back-end server. This allows the collection of servers to support a working set that is the sum of their individual caches. Both LARD and Mirage use application-level data within the requests to determine which back-end server the request will be routed to, and both LARD and Mirage operate transparently to the clients and servers.

8 Future Work

We are currently working on a full implementation of the namespace union. The namespace union presents many issues that are not present in the mount point union. While the mount point union allows the decision of which server contains a file to be made at mount time, the namespace union requires the decision to be made potentially on a file-by-file basis.

Currently Mirage routes packets between different subnets. In the real world, switches are used much more often than routers, and we are experimenting with developing a version of Mirage that functions more like a switch than a router. A Mirage switch would be more lightweight than the Mirage router, thus being able to handle more traffic and producing less latency. We hope to implement the Mirage switch using the Intel IXP1200 hardware [Intel00], which should allow Mirage to operate at line speeds and without measurable latency.

Multiple Mirage routers could coordinate to increase performance and mask failures. If a Mirage router were to fail, then automatic failover mechanisms would divert traffic to a backup Mirage router. Similarly, if one Mirage router undergoes a DoS attack, legitimate traffic would be routed to a different router. The major difficulty with multiple Mirage routers is the communication between them. For example, if a client migrates from one router to another, the VFH to PFH cache on the new router may not contain the entries that the client needs. Although the Mirage recovery mechanism would be able to ensure correctness, it would be inefficient to be constantly performing recovery every time a client was migrated. Similarly, the DoS scheduler would need to divide bandwidth between routers in addition to dividing it between clients.

8 Conclusion

Mirage is an NFS router that provides a virtual NFS server abstraction, aggregating the resources of multiple unmodified NFS servers for use by unmodified clients. Both the clients and the servers are unaware of Mirage's presence, yet Mirage allows files and file systems to be moved between NFS servers, and new NFS servers to be deployed, without reconfiguring the clients. In this way it greatly simplifies NFS system administration.

Mirage also protects against DoS attacks. Mirage ensures that each client is given an equal share of the servers' resources. Malicious clients can waste server resources, but never more than their share, and thus they are unable to prevent legitimate clients from accessing the NFS servers.

Experiments with Mirage show that packet latencies are significantly higher than in a standard IP router on the same hardware, but that bandwidth is only reduced by 5-7%. A realistic workload is only 7% slower using Mirage than using the NFS server directly. Mirage's low overhead, combined with the benefits of the virtualized server abstraction, demonstrate the viability of using an NFS router to build a scalable and versatile NFS system.

References

[Anderson00] Darrell C. Anderson, Jeffrey S. Chase, and Amin M. Vahdat. "Interposed Request Routing for Scalable Network Storage," in *Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI),* October 2000.

[DES88] Data Encryption Standard, Federal Information Processing Standard Publication 46-1, National Bureau of Standards, U.S. Department of Commerce, January 22, 1988.

[Gottlieb02] Yitzchak Gottlieb and Larry Peterson. "A Comparative Study of Extensible Routers," in *2002 IEEE Open Architectures and Network Programming Proceedings*, 51-62, June 2002.

[Intel00] Intel Corporation. IXP1200 Network Processor Datasheet, September 2000

[Maheshwari95] U. Maheshwari. "Charge-Based Proportional Scheduling," *Working Draft, MIT Laboratory for Computer Science,* Cambridge, MA, February 1995.

[Pai98] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum, "Locality Aware Request Distribution in Cluster-based Network Servers," in *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII),* San Jose, CA, Oct. 1998.

[Peterson94] Allen B. Montz, David Mosberger, Sean W. O'Malley, Larry L. Peterson, Todd A. Proebsting. John H. Hartman. "Scout: A Communications-Oriented Operating System," *Operating Systems Design and Implementation*. 1994

[Peterson01] Andy Bavier, Thiemo Voigt, Mike Wawrzoniak, Larry Peterson, Per Gunningberg. "SILK: Scout Paths in the Linux Kernel ," *Technical Report 2002-009, Department of Information Technology, Uppsala University*, Uppsala Sweden, 2002

[Rivest92] R. Rivest, "RFC 1321: The MD5 Message-Digest Algorithm," http://www.alternic.net/rfcs/1300/rfc1321.txt.html

[Rodrigues01] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. "BASE: Using Abstraction to Improve Fault Tolerance," in *Proceedings of the 18th {ACM} Symposium on Operating System Principles*, Banff, Canada, Oct. 2001.

[Sandberg85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. "Design and implementation of the Sun network filesystem," in *Proceedings of the Summer 1985 USENIX Conference*, pages 119-130, June 1985.

[Schuba97] C. L. Schuba, I. Krsul, M. Kuhn, E. Spafford, A. Sundaram, and D. Zamboni. “Analysis of denial of service attacks on TCP,” in *Proceedings of the 1997 IEEE Symposium on Security and Privacy, May 4-7, 1997. Oakland, California*, pages 208–223, Los Alamitos, CA, USA, May 1997. IEEE Computer Society Press.

[Siegal90] K. Birman and A. Siegal. “Deceit: A Flexible Distributed File System,” *in Proceedings of the Summer 1990 Usenix Conference*. June 1990.

[Steiner88] Steiner, Jennifer G, Neuman, Clifford, and Schiller, Jeffrey J. "Kerberos: An Authentication Service for Open Network Systems," *USENIX Conference Proceedings*, USENIX Association, Berkeley, CA, June 1988.

[Sun88] Sun Microsystems, “RPC: Remote Procedure Call, Protocol Specification, Version 2”, Request for Comments 1057, 1988.