FileWall: A Firewall for Network File Systems*

Stephen Smaldone, Aniruddha Bohra† and Liviu Iftode
Department of Computer Science, Rutgers University, Piscataway, NJ 08854
{smaldone,bohra,iftode}@cs.rutgers.edu

Abstract

Access control in network file systems relies on primitive mechanisms like Access Control Lists and permission bits, which are not enough when operating in a hostile network environment. Network middleboxes, e.g., firewalls, completely ignore file system semantics when defining policies. Therefore, implementing simple context-aware access policies requires modifications to file servers and/or clients, which is impractical.

We present FileWall, a network middlebox that allows administrators to define context-aware access policies for file systems using both the **network context** *and the* **file system context**. *FileWall interposes on the client-server network path and implements administrator defined policies through message transformation without modifying either clients or servers. In this paper, we present the design and implementation of FileWall for the NFS protocol. Our evaluation demonstrates that FileWall imposes minimal overheads for common file system operations, even under heavy loads.*

1. Introduction

Network middleboxes, for example, Firewalls, Network Address Translators (NATs), Network Intrusion Detection Systems (NIDS), Virtual Private Network Concentrators (VPNs), etc., are an integral part of the network infrastructure today. These systems interpose on the network path between critical infrastructure services and the untrusted, possibly vulnerable clients of these services. Through interposition, these middleboxes can: identify and discard unwanted, malformed, or malicious traffic; transform packet contents to map hosts with private addresses to the public address space; and generate traffic monitoring and profiling data based on administrator defined policies and the state accumulated during operation.

Network file systems, e.g., NFS, CIFS, etc., provide centralized storage for data sharing and are an important part of the *critical network infrastructure* protected by network middleboxes. While several advances have improved network file system performance, access control has remained *identical* to that of local file systems. However, such traditional access control is insufficient when files are accessed over the network, due to the presence of malicious clients and its susceptibility to attacks.

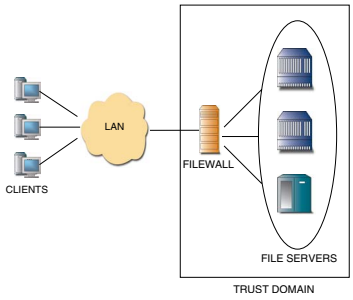

Figure 1. FileWall architecture.

Unfortunately, both network file systems and network middleboxes cannot independently implement context-aware policies on file system accesses. On the one hand, network middleboxes are limited to using the network context information, for example IP addresses, to implement *network* access policies. On the other hand, network file systems rely on the limited local access control mechanisms, e.g., Access Control Lists and rwx permission bits, to implement access policies, and ignore the network context. As a result, simple access control policies to protect the file system, for example, preventing user accesses to the file server simultaneously from multiple machines, are difficult, if not impossible to implement in network file systems.

We believe that a combination of file system context and network context is necessary to successfully enforce context aware file system access policies. In a network file system, the file system context includes naming, file hierarchy, and the semantics of the file system operations, while the network context includes the client identities, e.g., IP addresses and hostnames, the network characteristics, e.g., bandwidth, delay, loss rate, and the network topology.

In this paper, we present FileWall, a middlebox that combines network and file system contexts to implement

*This work is partially supported by National Science Foundation Grant No. CCR 0133366.

†Also with NEC Labs America

file system access policies. It interposes between file system clients and servers and *mediates* their interaction. It captures, modifies, and generates network file system messages, provides a persistent state storage mechanism, *access context*, and defines an execution engine for rules that implement the policies. Figure 1 shows the FileWall architecture. Analogous to a firewall, the file servers and FileWall are trusted and reside in the same network domain, whereas the clients are external. FileWall shares the keys with the file servers to handle encrypted and signed traffic. In this model, administrators have exclusive access to FileWall.

FileWall provides a single point of administration for imposing access policies on network file system communication. While a similar functionality can also be implemented at the servers, there are several benefits of interposition. First, it provides isolation by separating out the monitoring and control functionality from the file servers. This leads to a separation of concerns and allows file servers to evolve independent of the access policies. Second, by restricting access only to administrators, and allowing no user programs or daemons to execute on it, FileWall reduces the chances of subversion of the file server by rootkits. Third, interposition enables FileWall to virtualize the network endpoint visible to clients, allowing file system federation, failure mitigation, and system upgrades to be handled transparent to the clients. Finally, FileWall requires no modification to existing file servers and is readily deployable.

Realizing FileWall as a network middlebox is challenging. While clients and servers have a complete knowledge of the file system state, only the state updates are visible over the network. Therefore, the file system state must be inferred and maintained externally, using message history and protocol specifications. Network file systems are built on top of transport protocols, which implement their own semantics. Therefore, a FileWall cannot make arbitrary modifications to messages and flows, and must adhere to the semantics of the underlying transport protocols. An additional challenge for implementing FileWall in the network is usability. For administrators to use the system, it must be easy to configure, monitor, debug, and extend.

We have implemented FileWall for the NFS protocol, using the Click modular router framework and have evaluated our system using microbenchmarks, latency sensitive workloads, and real-world workloads. Our results demonstrate that FileWall imposes small interposition overheads of less than $100\mu s$ and these overheads increase linearly with the number of policies executed by FileWall. We also demonstrate overheads of less than 12% for the real-world workloads compared to the default NFS for the same setup.

In summary, this paper makes the following contributions. First, we present a system that implements network file system access policies by combining network and file system contexts. Second, we present the design and implementation of a readily deployable network middlebox that operates transparent to both client and servers to implement file system policies. Third, we demonstrate through our evaluation that FileWall is practical and imposes minimal overhead.

	Firewall / NAT	**FileWall**
Attributes	Packet	File System Message
State	Flow State	Access Context
Event	Packet Arrival, Timeout	Message Arr, Timeout
Condition	Firewall Rules	FileWall Rules
Action	Forward, Rewrite, Discard	Forward, Rewrite, Discard, Generate

Table 1. Firewall vs. FileWall

This paper is organized as follows. Section 2 describes the FileWall model. Section 3 presents the FileWall design. We present the implementation in Section 4, and the evaluation of our system in Section 5. Section 6 discusses open issues and limitations, and Section 7 summarizes the related work. Finally, Section 8 concludes the paper.

2 Background and Motivation

Network middleboxes, for example firewalls, NATs, VPNs, etc. interpose on the network path between *private* networks and the *public* Internet to prevent unauthorized or malicious network packets from crossing the trust boundary. Additionally, these middleboxes provide a mapping between private network identifiers and public identifiers, clearly separating the administration domains.

Administrators define the access policies by programming these middleboxes, which operate on all packets crossing the network boundary and evaluate *rules* that constitute the policies. All rules are specified using the Event-Condition-Action programming model. There are three type of events: network events (e.g., packet arrivals/departures), monitoring events (e.g., SNMP traps), and administrative events. Each event is associated with a set of conditions, evaluated when the event occurs. Each matching condition results in an action, for example, drop a packet, rewrite an IP header, forward a packet, etc. Additionally, middleboxes maintain state, which is built up during system operation and is used during the evaluation of rules.

FileWall combines file system context information with network context information to evaluate rules that operate on file system messages. Table 1 compares the features of a typical firewall/NAT with FileWall. As shown in the table, both firewalls and FileWall utilize attributes contained in messages. Additionally, both are event-driven, evaluating conditions (rules) on message arrival or when a timeout occurs. Both maintain state, either network flow state or access context. A firewall can either forward, rewrite, or discard a packet, while FileWall may choose to forward, rewrite, discard, or generate a new message. Finally, both are configured through a restricted interface, which is ac-

cessible only to administrators.

In the following, we describe a security policy that we use as a running example throughout the paper to illustrate FileWall. This example policy demonstrates the necessity of using network and file system context to implement file system access policies. It also identifies the limitations of existing mechanisms at file servers and middleboxes to implement such policies. However, this policy is deliberately simplistic for illustration and more complex conditions on who and when to allow access can easily be defined.

2.1 Example

Users of network file systems may access their files from multiple client systems. However, some files containing confidential or valuable information must be protected and access to them must be restricted. In such scenarios, the user must be allowed access to protected files from exactly one client system. Formally, a user may not access a subset of files, ($\mathcal{F}$), concurrently from two clients ($C_{last} = C_{cur}$) at any time (T_{cur}). If the user wishes to move to a different client ($C_{last} \neq C_{cur}$), then he must wait for a set threshold (T_{thresh}) to expire before the policy will grant him access to files in $\mathcal{F}$ from the new client. Equation 1 formally defines our example security policy $\mathcal{P}$.

$$\mathcal{P} = \begin{cases} ALLOW & \text{if } object \notin \mathcal{F} \\ ALLOW & \text{if } C_{last} = C_{cur} \vee T_{cur} - T_{last} > T_{thresh} \\ DENY & \text{if } C_{last} \neq C_{cur} \wedge T_{cur} - T_{last} \leq T_{thresh} \end{cases} \quad (1)$$

Note that even our simplistic example policy cannot be implemented from within a file server since a file server is unaware of the network layer and cannot differentiate between file system accesses from different clients. This policy cannot be implemented at a firewall due to the lack of understanding of file system semantics there. The best a firewall can do is to deny file system accesses by dropping packets, but this will cause a file system client to hang waiting for a response that will never arrive.

3. FileWall Design

The core functionality of FileWall is message construction, attribute extraction, scheduling, and message forwarding. The minimal functionality of the FileWall core leads to a small and easy-to-maintain code base and encourages reuse of code. In the following, we describe policy specification, access context, message transformation, and policy execution and scheduling.

3.1. Policy Specification

FileWall policies are specified as a sequence of *rules*. Rules in FileWall are analogous to firewall rules and are used to implement the policy logic. A convenient macro-like language (FWL) for specifying new policies is provided. FWL is first translated into C or C++, which is then compiled into object code using the native compiler. However, developers (or power users) who wish low-level access can bypass the macro language completely and work directly with the native language policy implementations.

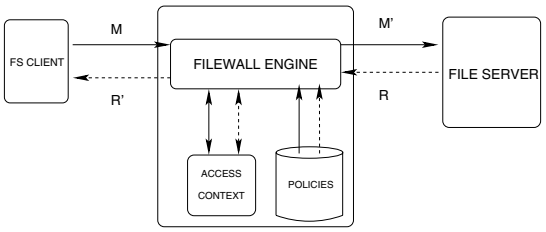

Figure 2. FileWall Architecture showing the request (solid) and response (dotted) paths. FileWall transforms the request from M to M', and the response from R to R'.

A FileWall policy consists of *sections* that correspond to the respective processing in the policy. Administrators define new policies using the following format:

```
@POLICY::PolicyName {
    @CONFIG {
        /* Declarations, Configuration variables */ }
    @RULE::RuleName {
        /* The first rule in the policy chain */
        @CLItoSRV {
            /* Client-side message handling code */ }
        @SRVtoCLI {
            /* Server-side message handling code */ }
    }
    /* The remaining rules in the policy */
    ...
}
```

Configuration information is declared in the `@CONFIG` section and is followed by a list of individual FWL rules (`@RULE` sections). The `@CONFIG` section contains all of the configuration variables and references to externally-defined functions or libraries. A rule is specified in two subsections: client-side (`@CLItoSRV`) and server-side (`@SRVtoCLI`), which handle the corresponding FileWall processing.

3.2. Access Context

Policy rules are evaluated using the attributes contained in the file system message, in the context of state maintained by the policy or present in the execution environment. We call this state the *access context*, and it represents the aggregation of the following four components: *(i)* network state, *(ii)* file system state, *(iii)* environmental state, and *(iv)* private policy state. Table 2 shows examples of state stored in different access context components. Persistent state components can be stored in a transactional database, which guarantees the ACID properties (Atomicity, Consistency, Integrity, and Durability) of the stored state.

Component	State Type	Examples
Network	Temporary	IP Addresses, Ports
File System	Temp/Pers	File Ids, FS Ops
Environment	Temporary	Time, CPU Load
Private	Persistent	Message Arrival Rate, Keys

Table 2. Access Context Components

Returning to the example from Section 2, the following FWL code demonstrates the use of access context:

```
@CONFIG {
  int Tcur, Ccur;
  int climap[string][int];
  string object;
}
@CLItoSRV {
  object = fileid@$MSG;
  ...
  TCur = $TIME;
  climap[object][Ccur] = Tcur;
  ...
}
```

The definition of private policy state (variables and an associative array) occurs in the `@CONFIG` section. For each client-server message (`$MSG`), the file identifier (`fileid`) is extracted (using the `@` attribute extraction operator) and the current time (`Tcur`) is queried from the environmental state (`$TIME`). Finally, `Tcur` is stored in an associative array (`climap`) indexed by the file `object` being accessed and the origin client identifier `Ccur`. The server-client subsection is omitted since it simply forwards the server response messages back to the client.

3.3. Message Transformation

FileWall is external to clients and servers, therefore all file system access policies are implemented using a combination of *attribute transformation*, *flow transformation*, and *flow coordination*. These primitives represent the minimal set of functionality supported by FileWall for message transformation.

To illustrate message transformation, Figure 2 shows the flow of a client request message (M) and the response message (R) through FileWall. The client sends its request to the server. FileWall intercepts this message and invokes policies that use access context to transform this message to M' and forwards M' to the server. The server responds with a message R, which is transformed by FileWall to R' and sent to the client as the file system response.

Attribute Transformation: File system messages are constructed as a set of arguments and updates in a request, or as a response to a previously received request. Upon receiving a message, the destination (client or server) executes the request, or updates its state based on the response. The attributes in the message determine the corresponding action.

Policy rules may modify the attributes to affect the action performed at the destination, or to request additional information. Attribute transformation can be used to remap attributes, adding a layer of indirection between the client and the server. This transformation can also be used to modify the file system operation identifier, included in all messages. In general, any message attribute can be transformed by policy rules, including the file system operation arguments. For example, data query and update attributes (read/write arguments) may be modified to perform data transformations.

To demonstrate attribute transformation, we present an extension to the example policy from Section 2. The following code represents an additional rule that enables revocation of file identifiers. We call the file identifiers presented to clients *virtual* file identifiers. We maintain two persistent maps (forward and reverse file identifier to virtual file identifier maps) in the access context to implement such transformation.

```
@CLItoSRV {
  flowid = flowid@$MSG;
  vfid = fileid@$MSG;
  fid = fwdmap[flowid][vfid];
  ...
  fileid@$MSG = fid;
  forward($MSG);
}
```

For each client-server message, the flow identifier (`flowid`) and the virtual file identifier (`vfid`) are extracted from the message and are used to perform a lookup in the access context (`fwdmap`), to determine the real server file identifier (`fid`). Once found, the client-server message is transformed to replace the `vfid` with the `fid`. Finally, any request that does not contain a previously encountered virtual file identifier is denied (code not shown). This prevents malformed or malicious requests from reaching the server leading to a potential crash or information disclosure.

The process is similar for any server-client message (code not shown), but in that direction, the `fid` is replaced with the `vfid`. If the `vfid` does not already exist, one is generated and the new mapping is inserted into the forward and reverse maps.

Flow Transformation: A network flow represents a client-server channel where all file system messages are exchanged. One or more messages in a sequence determine the operations performed and the state updated at the clients and servers. Flow transformation includes filtering, reordering, and injecting messages within a flow to control file system accesses.

Filtering removes messages sent by one end of the flow from being received at the other. Such transformations are used to implement access control and security policies, where only messages representing permitted operations are forwarded to the server, removing all other messages from the flow.

Message injection is the dual of filtering and introduces additional messages in the flow. Since all file system requests must generate a response, for every filtered request, a response must be injected in the flow to maintain protocol semantics. Together, message injection and subsequent

filtering can also be used to retrieve state information (not previously seen in the message stream) from the server, required for policy evaluation.

Finally, message reordering modifies the sequence of messages within a flow. Such transformation may be used to modify the caching and read-ahead mechanisms at the file server by changing the inputs to the existing mechanisms.

Returning to the example from Section 2, the following FWL code demonstrates the use of flow transformation:

```
@CLItoSRV {
  ...
  if (protected_objs[object])
    if(Ccurrent == Clast)
        ...
        forward($MSG);
    else if (Tcur - Tlast > Tthresh)
        ...
        forward($MSG);
    else
        ...
        rsp = denyrsp($MSG);
        forward(rsp);
        discard($MSG);
  else
    forward($MSG);
}
```

For each client-server message received, the `@CLItoSRV` section above is evaluated. Although a portion of the code is omitted, the listing shows that all file system accesses are checked to determine if they are for a file in the protected set (`protected_objs` map). If the file access is for one of these objects, then the secondary conditions apply and are evaluated. If the conditions hold, then the access is allowed and the message is forwarded, otherwise a new *DENY* message is generated by `denyrsp()` and forwarded back to the client, discarding the original message. The server-client section (not shown) simply forwards messages it receives.

Flow Coordination: For a single client-server pair, all FileWall rules may be implemented using attribute and flow transformation. However, when policies affect more than one client-server pair, additional mechanisms to transform messages across a collection of flows is necessary. Flow coordination includes multiplexing/serialization and demultiplexing/fan-out across a group of flows.

Multiplexing or fan-in operates on a collection of flows and generates a single flow of messages delivered to the client or the server. This may be used to implement novel consistency semantics by controlling the sequence of messages across flows, atomic updates by serializing all operations performed on a file system object or a group of objects, etc. Demultiplexing or fan-out are the duals of multiplexing and separate a single flow of messages into multiple flows. This includes demultiplexing aggregated flows and flow generation (e.g., flow replication). File server replication can be implemented through a combination of flow multiplexing and demultiplexing. Each request must be replicated using fan-out and the responses combined using fan-in.

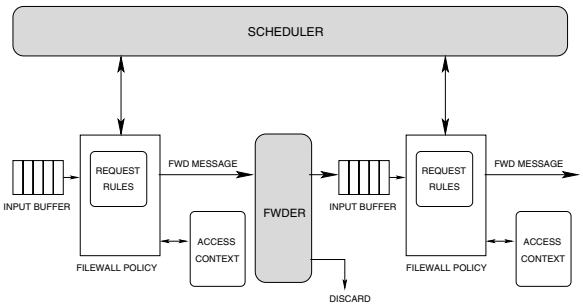

Figure 3. Policy chains with the FileWall scheduler and forwarder.

3.4. Execution and Scheduling

A policy is the minimum unit of FileWall processing. Policies may be linked into *policy chains*, as shown in Figure 3, and FileWall manages fixed-sized input buffers for each policy in a chain. A policy interacts with FileWall through either reading/writing the access context, or through explicitly forwarding a message. FileWall places the forwarded messages in the input buffer of the next policy in the chain.

Two FileWall components, *Scheduler* and *Forwarder* (shaded boxes in Figure 3), enable policy chains. The forwarder is invoked by policies with a message, it classifies the message as a request or response, finds the next policy in the chain, and places a reference to the message in the next policy's buffer. If this buffer has no empty slots, the message is dropped. Messages dropped by FileWall are no different than messages dropped by the network and do not affect the correctness of the file system protocol.

Policy chains are organized as two ordered queues: the input and the output queue, which are mirror images of each other. That is, the policy at the head of the input queue is at the tail of the output queue. Intuitively, if a policy, P0, sees a request R0 before another policy P1, then the response for R0 will be processed first by P1 and then by P0. The start (*RECV*) and the end (*SEND*) of the policy chains are fixed, and receive and send messages over the network, respectively. Policy chains are defined by the administrator using a plain-text configuration file that is parsed by FileWall. This file lists the policies in the order they must see the client request messages.

FileWall gains control of execution either asynchronously, on receiving a new message, or synchronously, on completion of a policy evaluation. On receiving a new message, FileWall simply inserts it in the input buffer of the RECV end of the policy chain. No scheduling decisions are made during the execution of a policy. FileWall policies are scheduled round-robin, from RECV-to-SEND,

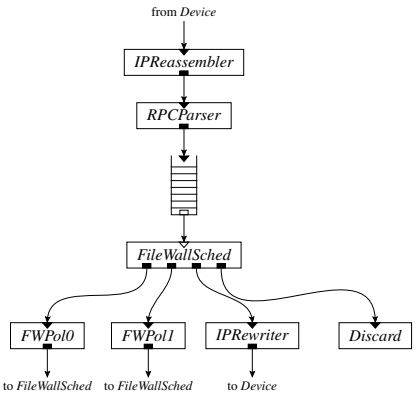

Figure 4. FileWall implementation.

in the chain specified during configuration. Scheduling is performed at the granularity of a policy, and execution continues for a policy until the current message is evaluated against a policy's sequence of rules. More complicated scheduling algorithms, for example proportional fair share or lottery scheduling, can be used as replacements for our default implementation.

4. Implementation

We implemented FileWall in the Click modular router [12] framework as an external user-level package. Click element classes define input and output ports, which represent connections to other elements in the router. Elements are classified as *push* and *pull*. Push elements are invoked in the context of a network packet and run to completion. Optionally, these elements output a packet to one or more ports through a synchronous call. Pull elements, on the other hand, wait for packets to be generated by upstream elements and execute when packets are available. Explicit queues are used to buffer packets between push and pull elements.

Figure 4 shows the Click configuration that realizes FileWall with two policies. Each policy is implemented as an autonomous Click push element, which runs to completion. The FileWall scheduler is a pull element and chooses the policy to execute. As shown in the figure, *FileWallSched* interposes between all FileWall policies to schedule and maintain global state across policies.

We implemented the private policy state store component of the access context using an open-source database – BerkeleyDB [21]. This database shares a process's address space allowing direct function calls to be made to the DB. We use the default DB configuration where the data is stored as B-Trees. Separate databases are created, for each policy in the system, to maintain isolation between policies.

The FileWall templates define the skeleton of the classes and the initialization code required to construct a Click element class. The FWL compiler is implemented using treecc [24], which converts input files in the treecc syntax into source code that permits creating and walking abstract syntax trees. We use treecc along with standard compiler generation tools, lex and yacc, to generate Click element code from the FWL code.

5. Evaluation

Our goal in this evaluation is to measure FileWall *overheads* and to characterize FileWall *behavior*, under varying network conditions and workloads.

Setup: In our experimental setup, all systems are Dell Poweredge 2600 SMP systems with two 2.4GHz Intel Xeon II CPUs, 2GB of RAM, and 36GB 15K RPM SCSI drives. All systems run the Fedora Core 3 distribution with a Linux-2.6.16 kernel and are connected using a Gigabit Ethernet switch. The average round-trip time between any two hosts on the switch is $30\mu s$. FileWall is configured to interpose on all NFS requests and responses.

Microbenchmark: To study the behavior of the file system, with and without FileWall, we developed our own microbenchmark. This benchmark is an RPC client and issues NFS requests *without* relying on the client file system interface. Using this benchmark eliminates the noise due to the client buffer cache and other file system optimizations, and allows fine-grained measurements to be collected. The benchmark measures the CPU cycles between a request and the corresponding response.

5.1. Latency

In this section, we study the effect, on the client observed latency, of adding FileWall to the network file system message path. To isolate the FileWall overheads, we study three systems: *(i)* Tunnel, which is a pass-through network tunnel implemented at the user level and does not include FileWall, *(ii)* Kernel Tunnel, which is identical to Tunnel implemented in the kernel, and *(iii)* FileWall, which is built on top of the Tunnel system with the policy described in Section 2. As a baseline, we present the latency of the default NFS protocol.

Interposition Overheads: The latency of an operation includes the network delays and the server processing overheads. Interposition overheads manifest themselves as increased latency for each network file system message. Increased latency has three components: *(i)* the packet capture and injection latency, *(ii)* the FileWall processing latency, and *(iii)* the policy latency. Formally, the base NFS request-response latency and the latency with interposition is given by

$$L_{NFS} = 2 \times D_{CS} + T_S \quad (2)$$

$$L_{Tun} = 2 \times (D_{CF} + T_P + D_{FS}) + T_S \quad (3)$$

$$L_{FileWall} = 2 \times (D_{CF} + T_P + T_F + D_{FS}) + T_S \quad (4)$$

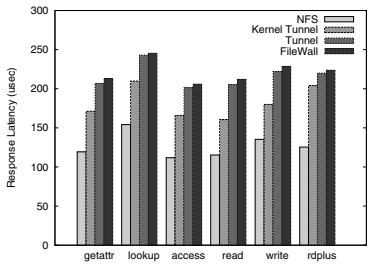

Figure 5. Interposition Overheads.

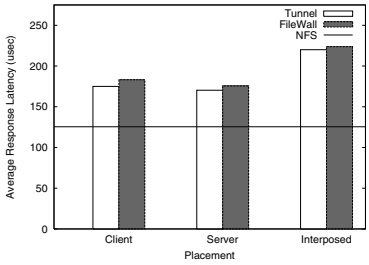

Figure 6. Overheads of placing FileWall at client, server, and interposed

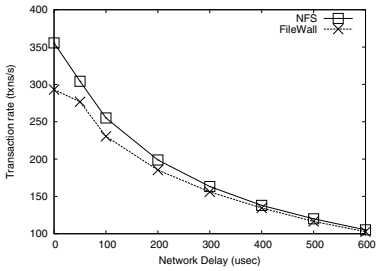

Figure 7. Postmark with varying network delay

where L represents the client observed latency, D represents the one-way network delay, and T is the processing overhead of each component. The subscripts C, S, and F represent the client, server, and FileWall respectively. T_P is the per-message processing overhead due to the packet capture and injection.

Our first goal is to isolate the overheads imposed by FileWall (T_F) and per-message processing (T_P), eliminating the impact of network delays. From Eqs. 3 and 4, we observe that the difference between $L_{FileWall}$ and L_{Tun} represents the FileWall and policy overheads.

$$T_F = \frac{L_{FileWall} - L_{Tun}}{2} \quad (5)$$

Figure 5 shows the client observed latency for different NFS operations. For clarity, we show only the most common operations, as reported by various file system workload studies [5] in the figure. We observed similar results for other policies as well as for the NFS operations not shown. In the figure, each group of bars has 4 members, base NFS, tunnel, kernel tunnel, and FileWall. The height of each bar shows the average response latency for 1000 instances of the call. We observe that FileWall imposes minimal overhead compared to the Tunnel overhead, as shown in Eq. 5. As expected, the kernel-tunnel is more efficient than the user level implementation, but the difference between them is less than $60\mu s$.

To isolate per-message processing overheads (T_P), we observe from Eqs. 2 and 3 that

$$T_P = \frac{L_{Tun} - L_{NFS}}{2} - (D_{CF} + D_{FS} - D_{CS}) \quad (6)$$

Therefore, if $(D_{CF} + D_{FS} - D_{CS}) \rightarrow 0$, we can identify the interposition overheads using operation latencies over the two systems. To accomplish this, we instantiate the network tunnel at the client, where $(D_{CF} \ll D_{FS})$ and $D_{FS} \approx D_{CS}$, and at the server $(D_{CF} \gg D_{FS})$ and $D_{CF} \approx D_{CS}$.

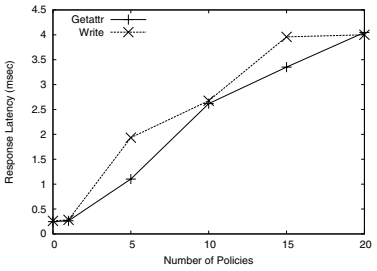

Figure 8. FileWall scalability with increasing number of policies.

Figure 6 shows the results for our microbenchmark for the READDIRPLUS call. The figure shows the average time between the call and its response at the client when the tunnel and FileWall are instantiated at the client, server, and on a separate machine on the network. As a baseline, we include the latency for the default NFS.

We observe that FileWall instantiation on a separate node (our model) performs well. The interposition overheads (T_P) are within $100\mu sec$ and can be improved further using an in-kernel implementation.

Network Delays: To study the effect of network delays on client-observed latency with a metadata intensive workload mix, we vary the network delays and study the effect of this variation using Postmark.

Postmark [10] is a synthetic benchmark that measures file system performance with a workload composed of many short-lived, relatively small files. In our experiments, we use 8KB block sizes for read and write operations. The initial file set consists of 5,000 files with sizes distributed randomly between 1KB and 16KB. For each run of the benchmark, we perform 20,000 transactions and report the transaction rate.

Figure 7 shows the Postmark results for varying the one-way network delay between the client and the server (D_{CS}) for base NFS and the client-FileWall delay (D_{CF}) for FileWall. The FileWall-server link is unmodified.

We observe that for low delays ($< 100\mu s$), FileWall and the base NFS differ significantly in the supported transaction rate. However, for networks with larger delays, the difference between FileWall and NFS performance is minimal. FileWall introduces delays on the order of 10s of μs, therefore, for comparable network delays, the performance of latency sensitive operations is greatly affected. However, these delays are hidden when the network delays dominate the overall delay between the client and the server and the operations are pipelined.

For typical deployments, file servers are physically separated from clients and delays of up to $300\mu s$ are common. Therefore, in realistic scenarios, we believe that FileWall will not affect performance and will be transparent to the clients.

Scalability: In this experiment, our goal is to study the effect of the processing overheads due to FileWall policies (T_F in Eq. 4). As we increase the number of policies, we expect this overhead to increase. However, this increase should not be exponential and ideally should be linear.

The number of FileWall policies vary across deployments. Therefore, to understand the client perceived performance as the number of policies increases is difficult. We define a synthetic policy that captures the tasks common across a wide range of policies and vary the number of instances of this policy to study FileWall behavior.

Each policy performs the following tasks: On a request message, it stores a fixed number (20) of keys, each of size 50 bytes in the access context (DB insert). On a response, it looks up all the keys inserted by the corresponding request and deletes them (DB lookup and delete). Since DB access is the most CPU intensive task performed by a FileWall policy, our results capture the expected behavior. However, these results may vary with different policies.

Figure 8 shows the average response latency for our benchmark as the number of policies is varied. The two curves are for GETATTR and WRITE requests, which illustrate the FileWall performance for metadata and data operations respectively. We observe that FileWall overheads increase linearly in the worst case and even with 20 policies, the response time is within 5 ms.

5.2. Real Workloads

Emacs Compilation: In the following, we compare the performance of FileWall with the context-aware policy described in Section 2, default NFS, and a pass-through tunnel, for a multi-stage software build similar to a modified Andrew Benchmark [8]. We measure the time taken to *untar*, *configure*, *compile*, *install*, and *remove* an Emacs 20.7 distribution. We performed one run of the benchmark to load the compiler binaries and associated libraries that are external to the file system under test. We discarded this result and performed ten further runs. Between each run of the benchmark, we unmounted the file system and mounted it to start with a cold cache on the file system under test at the client.

Figure 9 shows the time taken for each phase of the Emacs compilation benchmark from left to right. The bars in each group are NFS, Pass-through tunnel, and FileWall with the context-aware security policy (Section 2).

We observe that FileWall imposes a modest overhead of around 12% for the sum of all phases of the benchmark. The most expensive data intensive phases have small $(< 10\%)$ overheads. However, the metadata intensive untar, install, and remove phases show significant degradation. These results are pessimistic for FileWall due to the extremely low network delay without interposition ($30\mu s$). Recall from our microbenchmark that interposition imposes tens of μs of additional latency, which is comparable to the network delays in our environment. These overheads reflect in the extra time required for the metadata intensive phases of the benchmark. For networks with larger delays, the performance of all three – NFS, tunnel, and FileWall is identical.

Fstress: Fstress [1] is a synthetic, self-scaling benchmark that measures server scalability. We use the canned SpecSFS97-like workload distributed with Fstress, which performs random read-write accesses with file sizes varying from 1KB–1MB. The benchmark increases the offered load by issuing NFS requests according to the workload mix. The metric used is the latency of these operations. For an overloaded server, the client observed latencies grow rapidly, whereas for an underloaded server, these latencies are small. For our experiments, we use three load generators (clients) that have the configurations described in the beginning of the section.

Figure 10 shows the results from the Fstress benchmark. We observe that FileWall latencies are comparable to both the default NFS and the network tunnels (user and kernel) up to an offered load of 2,500 requests/s. The systems diverge rapidly beyond this point. However, in all cases, the NFS server is overloaded at around 3,000 requests/s. The sharp increase in observed latency with FileWall is due to the CPU at FileWall being overloaded. At overload, the small processing overheads imposed by FileWall become significant and the performance drops. However, the overheads are small (around 15%) and the enhanced functionality provided in exchange by FileWall make them acceptable.

FileWall does not necessarily saturate before the NFS server. The impact of FileWall under heavy load is due to the relative performance between the FileWall machine and the NFS server machine. This is illustrated in Figure 11. For this experiment we run Fstress, while varying the CPU speed of the NFS server. We use the Intel Speedstep voltage scaling to reduce the CPU speed to 300MHz and 1200MHz, while maintaining all other system parameters constant. The curves in the Figure represent the base

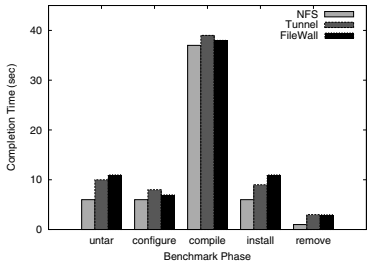

Figure 9. Results of emacs compilation benchmark.

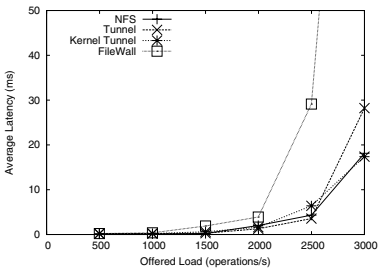

Figure 10. Fstress performance with 2.4GHz CPU.

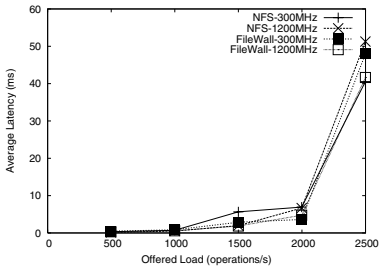

Figure 11. Fstress performance with different NFS server CPU speeds.

NFS and FileWall performance for the different NFS server CPU speeds. In all cases, we observe that the performance is similar with and without FileWall, and the NFS server saturates beyond 2000 requests/s. We conclude that given sufficient resources relative to the NFS server, FileWall does not impose significant overheads even under heavy workloads.

6. Discussion and Limitations

While we focus on context-aware access policies, FileWall can also be used to implement additional functionality for network file systems. Apart from access control, we have used FileWall to implement virtual namespaces, user behavior modeling, rate based admission control, and real-time monitoring of file system traffic. These enhancements to network file systems are made possible by the ability of FileWall to identify network context, build up a view of the file server state, and transform messages based on administrator defined policies.

In this paper, we describe a proof of concept implementation for the NFSv3 protocol. The choice of this protocol is due to its popularity, and due to its limited support for callbacks and per-file open and close messages. This allows us to study the limitations of implementing access policies with the least support from the protocol. However, this is not a fundamental limitation of our system. Over the years, most systems that extend NFS functionality have introduced callbacks [8, 11] and support for OPEN and CLOSE [6, 18]. With such support, we can easily implement file system extensions, within FileWall, to provide novel consistency semantics, atomic updates, application aware caching, etc.

FileWall provides an interface for administrators to define file system access policies. However, this interface is protocol dependent. That is, the administrator must have knowledge of and FileWall must understand the underlying file system protocols. FileWall is a first step towards the ultimate goal of a policy specification and enforcement platform where administrators can describe system-wide policies, and the low-level implementation of the policy is generated automatically. In the future, we plan to design a protocol-independent high-level language that eases policy specification and enables verification and debugging of policies. Such a language could also be used to enforce and extract invariants, which can be used by administrators for writing more robust and effective FileWall policies.

A limitation of our system is the lack of support for fault-tolerance and reliability. Since we introduce an additional host in the network file system message path, its failure would lead to loss of service even when both clients and servers are active. However, we believe we can introduce a primary-backup failover scheme for FileWall. Since FileWall state is built up during execution by observing messages, a backup system that snoops all traffic entering and leaving FileWall can build up its own copy of the state. For stateful protocols, e.g., TCP, we can preserve live connections at FileWall using existing connection migration techniques [23, 22].

7. Related Work

FileWall is related to a variety of work in the areas of *(i)* Distributed and Extensible File Systems and *(ii)* Composable Network Processing. The following section is a survey of the work related to FileWall in these areas.

Distributed and Extensible File Systems: Several methods for extending local file systems by interposing on the request path have been proposed [25, 3, 19, 20, 15]. These systems interpose between the file systems and transform the vnode operations to enhance functionality. Such systems have been used to trace file system calls [3], virtualized namespaces [19, 20], virus protection [15], etc. Unlike the above, FileWall targets network file systems to implement policies, through message transformation, external to both clients and servers.

Network storage systems have been extended for functional decomposition [2], enhanced security [13], saving bandwidth [17], repair and rollback [26], extended access control [7, 14], etc. These systems interpose on the client-server path to implement a solution for a specific problem. In contrast, FileWall provides a platform for implementing network file system policies and builds mechanisms, which

go beyond a specific problem instance. In fact, each of the above solutions can be implemented using FileWall.

Composable Network Processing: FileWall is inspired by packet filters [4], network address translators, and firewalls, which implement network access policies by interposing on network traffic and mapping private addresses to public Internet addresses.

Several systems provide composable network processing frameworks [9, 16, 12]. FileWall is implemented within the Click [12] modular router framework. In contrast to the above, which process network packets, FileWall processing is in the context of file system messages, which may be composed of multiple packets. FileWall also supports persistent state using a database, which is not required for a router.

8. Conclusions

We presented the design, implementation, and evaluation of FileWall, a network middlebox that implements context-aware network file system access policies. Using FileWall, we demonstrate that policies that cannot otherwise be implemented without significant modification of clients and servers can easily be realized through interposition. In FileWall, these policies combine network and file system contexts, are feasible, are easy to specify, and do not impose significant overheads. FileWall is the first step towards defining a platform where administrators can easily define and extend the functionality of network file systems to implement organization-wide disciplines that control the way file systems are accessed.

References

[1] D. Anderson and J. Chase. Fstress: A flexible network file service benchmark. Technical report, CS-2002-01, Duke University, 2002.

[2] D. C. Anderson, J. S. Chase, and A. Vahdat. Interposed request routing for scalable network storage. *ACM Trans. Comput. Syst.*, 20(1):25–48, 2002.

[3] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A File System to Trace Them All. In *Proc. of FAST*, San Francisco, CA, March/April 2004. USENIX Association.

[4] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis. FFPF: Fairly Fast Packet Filters. In *Proc. of OSDI*, 2004.

[5] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS Tracing of Email and Research Workloads. In *Proc. FAST*, San Francisco, CA, 2003.

[6] K. Fu, M. F. Kaashoek, and D. Mazieres. Fast and Secure Distributed Read-Only File System. *ACM Trans. Comput. Syst.*, 20(1):1–24, 2002.

[7] Z. He, T. Phan, and T. D. Nguyen. Enforcing enterprise-wide policies over standard client-server interactions. In *Proc. of SRDS*, oct 2005.

[8] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[9] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. In *IEEE Trans. on Soft. Eng.*, 1991.

[10] J. Katcher. Postmark: A new file system benchmark. Technical Report TR3022, Network Appliance Inc., October 1997.

[11] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Trans. Comput. Syst.*, 10(1), 1992.

[12] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3), August 2000.

[13] D. Mazieres, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proc. of SOSP*, dec 1999.

[14] N. H. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Trans. Softw. Eng. Methodol.*, 9(3), 2000.

[15] Y. Miretskiy, A. Das, C. P. Wright, and E. Zadok. Avfs: An On-Access Anti-Virus File System. In *Proc. of Security 2004*, San Diego, CA, August 2004.

[16] A. Montz, D. Mosberger, S. O'Malley, L. Peterson, T. Proebsting, and J. Hartman:. Scout: A communications-oriented operating system (abstract). In *Proc. of OSDI'94*, 1994.

[17] A. Muthitacharoen, B. Chen, and D. Maziéres. A low-bandwidth network file system. In *Proc. of SOSP*, New York, NY, USA, 2001.

[18] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. of OSDI*, Boston, MA, December 2002.

[19] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottora. The use of name spaces in Plan 9. *Operating Systems Review*, 27(2):72–76, 1993.

[20] H. C. Rao and L. L. Peterson. Accessing files in an internet: The jade file system. *IEEE Trans on Software Eng.*, 19(6), 1993.

[21] Sleepycat Software Inc. Berkeley DB. `http://dev.sleepycat.com/`.

[22] A. C. Snoeren and H. Balakrishnan. An End-to-End Approach to Host Mobility. In *Proc. 6th ACM MOBICOM*, Aug. 2000.

[23] F. Sultan, A. Bohra, and L. Iftode. Service Continuations: An Operating System Mechanism for Dynamic Migration of Internet Service Sessions. In *Proc. SRDS*, Oct. 2003.

[24] R. Weatherley. An aspect oriented approach to writing compilers. `http://www.southern-storm.com.au/treecc.html`.

[25] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proc. of USENIX*, San Diego, CA, June 2000.

[26] N. Zhu, D. Ellard, and T.-C. Chieuh. TBBT: Scalable and Accurate Trace Replay for File Server Evaluation. In *Proc. of FAST*, San Francisco, CA, December 2005.