Defending Against Denial of Service Attacks in Scout

Oliver Spatscheck
Department of Computer Science
University of Arizona

Larry L. Peterson
Department of Computer Science
Princeton University

Abstract

We describe a two-dimensional architecture for defending against denial of service attacks. In one dimension, the architecture accounts for all resources consumed by each I/O path in the system; this accounting mechanism is implemented as an extension to the path object in the Scout operating system. In the second dimension, the various modules that define each path can be configured in separate protection domains; we implement hardware enforced protection domains, although other implementations are possible. The resulting system—which we call Escort—is the first example of a system that simultaneously does end-to-end resource accounting (thereby protecting against resource based denial of service attacks where principals can be identified) and supports multiple protection domains (thereby allowing untrusted modules to be isolated from each other). The paper describes the Escort architecture and its implementation in Scout, and reports a collection of experiments that measure the costs and benefits of using Escort to protect a web server from denial of service attacks.

1 Introduction

It is becoming increasingly important that networked computing systems be able to protect themselves from denial of service attacks. For example, a web server needs to be able to detect and defend itself from an attacker that is consuming its resources by trying to initiate TCP connection establishment as rapidly as possible—the so called SYN attack [22]. Protecting against denial of service attacks involves three steps:

Accounting: A necessary first step is to account for all resources consumed by every principal.

Detection: A denial of service attack is detected when the resources consumed by a given principal exceed those allowed by some system policy.

Containment: Once an attack is detected, it must be possible to reclaim the consumed resources using as few additional resources as possible, otherwise, removal of an offending principal becomes a denial of service attack in its own right.

Attacks on traditional operating systems like Unix [18] frequently exploit the lack of accounting within the kernel, that is, before the work has been assigned to a particular user (principal). For example, it is possible for an attacker to consume all available TCP ports before a single message is dispatched to a user process which implements the policy and could detect the attack. Even if an attack is detected, it is often difficult, if not impossible, to reclaim all the resources consumed by the offending principal. Consider, for example, something as commonplace as a distributed file system: cached file blocks, NFS mount points, device buffers, and network connection state almost always have a longer lifetime than the user process that requested them. There is no direct way to account such resources towards a principal, and certainly no way to reclaim them when the principal is removed from the system because it has violated some usage policy.

Recent multimedia operating systems like Scout and Nemesis [13, 14] begin to address this problem by isolating data streams and minimizing cross talk between streams; cross talk is resource contention that interferes with the system's ability to make quality-of-service guarantees to each stream. Although these systems are successful in isolating streams, they do not provide the fine-grain accounting of resource usage needed to detect denial of service attacks. They are also limited in that their isolation mechanisms do not span multiple protection domains; they assume all resources used by a given data stream are confined to a single domain. Assuming a single protection domain is unrealistically restrictive, for example, it precludes a web server from running untrusted CGI scripts.

This situation points to a dilemma faced in designing a secure system: how to simultaneously support protection domains that allow untrusted components of the system to be isolated from each other, yet account for all system resources consumed (potentially across multiple domains) by a single principal. This paper addresses this dilemma by making two contributions. First, it presents a fine-grain resource accounting mechanism that has been implemented in the Scout operating system. The mechanism is able to account for virtually 100% of the resources used by a given principal at a low overhead of 8%. Second, it describes how this mechanism can be made to work across multiple protection domains. The paper does not offer any novel denial of service policies, but it does describe a working web server based on this architecture, and measures its performance while enforcing a representative set of usage policies.

The limitation of this work is that it is impossible to charge a piece of work to a particular principal until the principal has been identified. For incoming network packets, this means the system is vulnerable from the time a packet arrives until it has been demultiplexed and authenticated. The architecture we describe takes two steps to minimize the impact of this window of vulnerability. First, it pushes the demultiplexing/authentication decision as early as possible. Exactly how early depends on the protocols being used and the environment in which the system exists. For example, a WWW server using IPSEC [1] can authenticate an IPv6 datagrams cheaply using a secure hash function. This happens during demultiplexing, earlier than it would be possible using TLS [7]. In another example, a WWW server positioned behind a filtering router might use IP addresses from the local network for authentication, trusting the router to filter inappropriate datagrams. In a third, and more complex environment, IP addresses could be rated by an intrusion detection system, with resources allocated according to the trustworthiness of those addresses. In all three cases, it is important that the OS does not architecturally force a late demultiplexing decision.

The second way our architecture minimizes the impact of late authentication is that, even when the system has not yet determined precisely what principal is responsible for a particular packet, certain classes of packets can be aggregated and given only limited resources. For example, IPSEC allows early authentication by requiring a key exchange protocol to establish a shared key. In such an environment, the server is vulnerable to an attack by a new client that consumes server resources by sending the server bogus key exchange requests. Our architecture allows the WWW server to give preference to clients that already posses valid shared keys, thereby maintaining connectivity to the current set of clients while under attack. We will demonstrate this feature in a later section, showing how a web server might limit the cycles spent processing new connections (e.g., SYN packets) by giving preference to existing connections.

2 Architecture

This section defines Scout's security architecture. It begins with an overview of Scout, and then describes how we have extended Scout to support both fine-grain accounting and protection domains. It concludes with a brief discussion of how the resulting system—which we call Escort—facilitates the enforcement of different security policies.

2.1 Configurability

Modules are the unit of program development and configurability in Scout. Each Scout module provides a well-defined and independent function. Well-defined means that there is usually either a standard interface specification, or some existing practice that defines the exact function of a module. Independent means that each single module provides a useful, self-contained service. That is, the module should not depend on there being other specific modules connected to it. Typical examples are modules that implement networking protocols, such as HTTP, IP, UDP, or TCP; modules that implement storage system components, such as VFS, UFS, or SCSI; and modules that implement drivers for the various device types in the system.

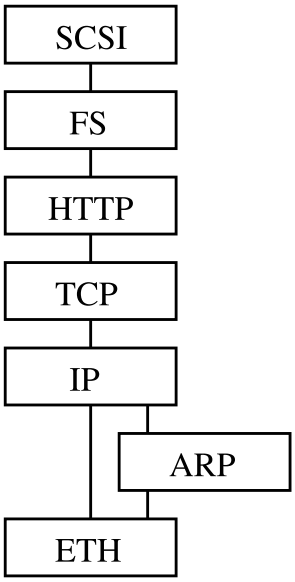

Figure 1: Example Scout Module Graph

To form a complete system, individual modules are connected into a *module graph*: the nodes of the graph correspond to the modules included in the system, and the edges denote the dependencies between these modules. Two modules can be connected by an edge if they support a common *service interface*. These interfaces are typed and enforced by Scout. By configuring Scout with different collections of modules, we can configure kernels for different purposes, including network-attached devices, web and file servers, firewalls and routers, and multimedia displays. For example, Figure 1 shows an extract of the module graph for a Scout kernel that implements a web server. The configuration includes device drivers for the network and disk devices (ETH and SCSI), four conventional network protocols (ARP, IP, TCP and HTTP), and a simple file system (FS). Such a configuration is specified at build time, and a set of configuration tools assemble the corresponding modules into an executable kernel.

2.2 Path Abstraction

Scout adds a communication-oriented abstraction—the *path*—to the configurable system just described. Intuitively, a path can be viewed as a logical channel through a modular system over which I/O data flows. In other words, the path abstraction defines a channel over which data moves through the system, for example, from input device to output device. Each path is an object that encapsulates two important elements: (1) it defines the sequence of code modules that are applied to the data as it moves through the system, and (2) it represents the entity that is scheduled for execution.

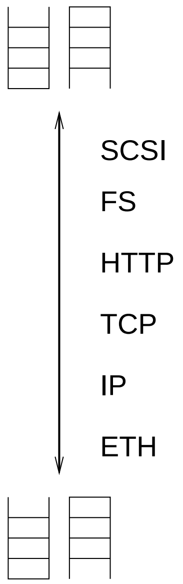

Figure 2: Example HTTP Path

Although the module graph is defined at system build time, paths are created and destroyed at run time as I/O connections are opened and closed. Figure 2 schematically depicts a path that traverses the module graph shown in Figure 1; it has source queues and sink queues, and is labeled with the sequence of software modules that define how the path "transforms" the data it carries. This particular path processes incoming HTTP requests by fetching web pages from disk.

The path-specific local state of each module is stored in a data structure called a *stage*. Stages from a sequence of modules are combined to form the path. In addition to this path-specific state, when executing code within a certain module, paths also have access to the state of the module. For example, a path executing code of the IP module has access to the routing tables stored in the IP module.

Each path goes through three phases during its lifetime. The first phase is path creation, during which the topology of the path—i.e., the sequence of modules it traverses—is determined, and the state of the path is initialized. Path creation is triggered by a pathCreate call to the kernel; the kernel limits path creation according to an access control list (ACL) specified by the system designer.

Specifically, the pathCreate operation takes six arguments: a set of attributes, the starting module, a subject, a subject class, the calling protection domain, and the calling owner. The first four arguments are explicitly given, while the last two are implicitly known from the calling thread. The attribute set defines invariants for the path, such as the port number and IP address for the peer. The kernel uses these invariants, plus the starting module, to determine the path's topology—the sequence of modules that the path traverses. Because only a certain small number of path topologies are useful in a given configuration, it is accurate to think of this process as determining the path's *type* (e.g., an "HTTP path"). Next, the kernel consults the ACL to determine if the entity trying to create the path is allowed to create a path of this type, and if so, what resource limits might be imposed on it. The entity creating the path is identified by the last four arguments to pathCreate: the subject (think of this as a user or a role), a subject class (this defines the availability level [16]), the calling protection domain (see Section 2.3), and the calling owner (see Section 2.4). At this point, the path exists and its resource limits are known.

Then the path enters its second phase, during which data is sent and received over it. Both send and receive work in the obvious way: data is enqueued at one end of the path and a thread is scheduled to execute the path. There is one complication, however. When data arrives on a device—e.g., a network packet arrives on the Ethernet—the kernel must determine to which path it belongs. This is done in a way that is analogous to path creation: the kernel identifies the path incrementally by invoking a demux operation on a sequence of modules. Each module's demux function has three choices: (1) it can determine that a unique path has not yet been identified and call the demux function of some adjacent module; (2) it can reject the request and drop the data; or (3) it can return a unique path. The demux function is side-effect free.

The last phase of a path is invoked by a pathDestroy or pathKill call to the kernel. In case of pathDestroy the kernel invokes a destroy function associated with each module along the path in the same order in which they were initialized before it frees all resources used by the path. pathKill frees all the path's resources, but does not invoke the destroy functions.

2.3 Protection Domains

Escort extends the basic Scout architecture by isolating the modules that have been configured into the system into separate protection domains. The kernel—which implements the path operations described above, as well as other objects described in the next section—runs in a privileged protection domain. The protection domain that each module is to run in is specified at configuration time. Trusted modules can be placed in the privileged domain. Modules can also be multiply instantiated, both across different protection domains, and in the same protection domain.

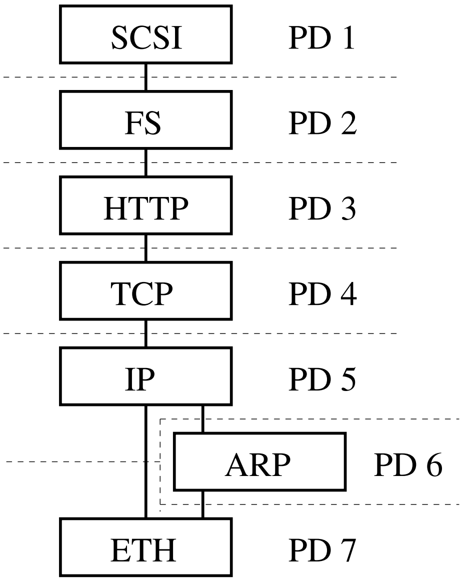

Figure 3: Modules Partitioned into Protection Domains

Figure 3 shows the module graph for our example web server partitioned into separate protection domains; one module per domain in this example. (The device drivers also have access to the memory regions used to access their devices.) This configuration represents the maximum possible separation. A less restrictive configuration might, for example, combine TCP, IP and ARP within one protection domain.

In addition to the kernel and the set of modules configured into the system, Escort also supports libraries that implement commonly used functions. Library code is trusted by their users, and so is mapped executable into all protection domains. Escort currently supplies libraries to manage messages, hash tables, participant addresses, attributes, queues, heaps, and time. It also includes a standard C library.

The current version of Escort runs in a single 64-bit address space and implements protection domains using hardware mechanisms available on the Alpha microprocessor. Modules not linked into the privileged domain invoke kernel services using a hardware trap. However, software fault isolation [24], type safe languages like Java, and proof carrying code [15] could be used instead.

Since the code for each module and library might be used by multiple protection domains, the calling environment for a given invocation of a library or module function must be specified. Furthermore, since modules can also be multiply instantiated within one protection domain, it is not sufficient to have one data segment per protection domain. Therefore, Escort explicitly passes the calling environment as the first argument to any procedure, optimizing for stateless libraries and libraries that access only protection domain state. This is similar to the approach described in [19].

Each module supports a well-known initialization function. When an Escort system boots, the kernel initializes every module by switching to the appropriate protection domain and calling the init function on each module in that domain. The modules initialize their global state and create an initial set of paths.

Finally, we return to the issue of demultiplexing incoming network packets, but this time in light of multiple protection domains. The base demux mechanism in Scout trusts the `demux` functions contributed by each module. Although not yet implemented in Escort, alternative mechanisms—e.g., pattern-based demultiplexers like PathFinder [3] and the current system augmented with Proof Carrying Code [15]—would be more appropriate since they do not trust the demultiplexing code to be correct and to not leak information via the demultiplexing decision.

2.4 Accounting for Resource Usage

A key goal of Escort is to account for all resource usage. Towards this end, all resources are charged to an *owner*, which can be either a path or a protection domain. Paths are the preferred choice since they most naturally correspond to the actual user of the resources. However, there are certain resources that cannot be accounted to a particular path. For example, an IP routing table cannot be directly associated with (charged to) any individual IP flow; the memory used by the routing table is associated with the protection domain that runs the IP module.

There are only a few differences between protection domains and paths in terms of ownership. One is that protection domains have a heap and paths do not. The reason for this is that the kernel allows memory allocation at the page level only. For paths this is extremely inefficient since it would require a path to allocate at least one page for each protection domain it crosses. To keep the accounting mechanism accurate, the protection domain can charge paths that cross it with memory usage. The memory charged toward a path is then deducted from the memory charged to the protection domain. In other words, the kernel gives memory pages to protection domains, which in turn implement a heap and hand out smaller memory objects to paths that traverse them.

To allow the automatic reclamation of this memory—

and other resources like the reservation of a TCP port—all modules can register destructor functions with a path. This function is called in the module's protection domain when a path is destroyed or killed, and results in charge for the memory being transfered back to the protection domain. The destructor function usually frees all memory charged toward the path. However, the domain is ultimately responsible for the freeing of the memory, that is, returning the page back to the kernel.

Another difference is that paths can be destroyed without destroying the modules or protection domains they cross. However, if a protection domain is destroyed, all paths crossing that protection domain are also destroyed. This is necessary since paths can access the global state of all modules they cross and this state will be removed if the protection domain is destroyed. For example after destroying the protection domain containing the IP module, IP's routing table will no longer be accessible by paths anymore.

```
struct Owner {
    OwnerType type; /* PATH or PD */

    /* Accounting */
    u_long kmem;
    u_long pages;
    u_long IoBuffer,
    u_long threads;
    u_long stacks;
    u_long cycle;
    u_long events;
    u_long semaphores;

    /* Tracking */
    PageList  pages;
    ThreadList threads;
    IoBufferLockList iobufferlock;
    EventList  event;
    SemaphoreList semaphore;

    /* Scheduling */
    Scheduler scheduler;

    /* Resource Monitoring */
    Resource limits;
};
```

Figure 4: Owner Data Structure

Figure 4 shows the Owner data structure; this structure is the first element of both the path and protection domain data structures. The Owner structure is divided into three parts. The first part keeps a count of the resources—kernel memory, memory pages, IOBuffer, threads, stacks, CPU cycles, events, and semaphores—used by this owner. The fields in this part are used to decide if the resource part of the security policy has been violated. Note that the kmem field counts the amount of memory used to store the kernel objects referenced in the second part of the data structure.

The second part contains doubly linked lists of the actual kernel objects associated with this owner; these objects are described in Section 3. These lists support the fast removal of the corresponding objects in event that the owner must be destroyed. The Scheduler object contains the information necessary to schedule threads belonging to this owner. The exact contents of this data structure depends on the scheduler used. The last part contains the resource limits of the owner. This object is more fully described in Section 2.5.

Whenever a new resource is requested, the owner is explicitly passed as an argument to the kernel allocator. Although not mandated by the architecture, many policies require that this argument must match the owner of the current thread.

2.5 Specifying Resource Limits

Owners are charged for resources they use, with any limits placed on this usage specified at system configuration time (for protection domains), and at path creation time (for paths). The resource limits for a particular owner are given by the Resource object of the Owner data structure. The action to be taken when a given limit is exceeded is specified in the Limit object; possible actions include destroying the path, denying the request, or preventing further demultiplexing of incoming data to the path. Figure 5 shows both the Resource and Limit data structures.

```
struct Limit {
    int val;
    Action action
};

struct Resource {
    Id subject;
    Id subject_class;

    Limit kmem;
    Limit pages;
    Limit IoBuffer;
    Limit threads;
    Limit stacks;
    Limit cycle;
    Limit events;
    Limit semaphores;
    Limit yield;
    Limit attribute[attr_count];
};
```

Figure 5: Limit and Resource Data Structures.

The Resource object defines limits for the very same resources as accounted for in the Owner object: kernel memory, pages, IOBuffers, threads, stacks, cycle, events, semaphores and attributes. In addition, the yield field limits the maximum number of cycles a thread can run without yielding the processor, attr_count is a system constant limiting the number of attributes which can be associated with a path and the attribute field limits the values of those attributes. The Resource object also contains identifiers for subjects, which correspond to users or roles and subject classes which represent availability levels in multi level availability systems. These identifiers are used to aggregate resource usage over multiple paths.

All resource limits, except for the yield and cycle restrictions, are enforced by a resource monitor. This monitor is called whenever resources are allocated or freed, or when attributes change. The resource monitor is also responsible for monitoring aggregated resource utilization for subjects and subject classes according to a given policy. To support multiple policies, Escort allows the appliance designer to configure different resource monitors into the system. Currently, Escort uses a simple resource monitor that compares the resources used against the stated limit, and performs the appropriate action when the limit is exceeded. It does not support aggregation of resources.

The yield and cycle restrictions are enforced directly by the kernel at clock interrupt time, and if a violation of policy occurs, the only action allowed is to destroy the associated owner.

2.6 Remarks

Although we have been focusing on how Escort accounts for resource usage, it is useful to place Escort's security mechanisms in a larger context. Specifically, Escort allows the system designer to enforce a security policy on four different levels.

- The kernel uses a conventional role-based ACL [2] to guard against unauthorized access. The role is determined by the owner of the thread and the current protection domain.
- The module graph defines the base channels of communication between protection domains, and therefore limits information flow between protection domains and those channels.
- The path object allows the system to always charge actions towards the principal that is ultimately responsible for them. Paths also allow us to perform certain complex access control decisions at path creation time instead of path execution time. In this way, a path is similar to a cache of capabilities for a specific owner, and as a consequence, the path creation process becomes an important part of the policy.
- It is possible to configure *filters* between modules in the module graph. Syntactically, filters are just like any other module, except their purpose is to enforce policy rather than to implement a specific function. For example, a filter between TCP and IP might restrict the TCP/IP interface from one that supports "receive packets" to one that supports only "receive packets to port 80". The filter enforces this more restricted interface by filtering data that does not adhere to this restriction. Such filters can be used along with a vanilla TCP module, and conversely, the same TCP module can be flanked by different filters. The important point is that the security policy need not be embedded in the TCP module.

3 Implementation

Escort currently implements 52 system calls that provide access to the following kernel objects: paths, IObuffers, threads, events, semaphores, memory pages, devices, and the console. This section describes the implementation of the first three of these objects in more detail.

3.1 Paths

As already described, paths are created and destroyed using pathCreate, pathDestroy and pathKill operations. The kernel also provides functions that allow data to be enqueued on either end of a path.

```
struct Path {
    struct Owner owner;
    Hash   allowed_pd_crossings;
    StageList stages;
    Queues[4] q;
    ThreadPool t;
    u_long refCnt;
};
```

Figure 6: Path Data Structure

The path data structure, as shown in Figure 6, is accessible only from within the kernel. It contains the owner state, a hash table of allowed protection domain crossings for this path, a list of the stages belonging to the path, pointers to the path input and output queues, a thread pool that provides threads for the path, and a reference counter used to delay pathDestroy but not pathKill calls.

The stages contained in the stage list represent the contribution of each module to the path. Stages communicate using predefined interfaces. The entry point of these interfaces are established during path creation and stored in the map of allowed protection domain crossings. Escort currently supports interfaces for asynchronous I/O, name resolution, and file access.

3.2 Threads

Threads, like any other resource in Escort, are owned by either a protection domain or a path. This means that the lifetime of a thread is bound by the lifetime of its owner, and as a consequence, threads cannot directly migrate between owners. Keep in mind that the motivation for migrating threads [5] is to allow a single execution context to cross multiple protection domains, but this is already supported in Escort by the explicit path abstraction. In a well designed configuration, thread migration between owners—e.g., from one path to another or from one protection domain to another—should be an uncommon event. Should such a need arise, Escort provides a handoff function that generates a new thread belonging to the target owner. Escort also synchronizes the threads, and wakes up any threads waiting for a thread belonging to an owner that has been destroyed.

Threads owned by a protection domain always execute within this domain and are implemented similar to regular UNIX threads. In contrast, threads owned by a path have the ability to cross the protection domains along the path. These threads have multiple stacks: one for each protection domain in which they can execute, plus a kernel-resident stack that records the protection domains currently being crossed. This is more efficient than assigning a new stack after each protection domain crossing since Escort threads are likely to switch into the same protection domain more than once. For example, a thread used to deliver an ICMP echo request datagram is also used to send the ICMP response, thereby crossing the protection domain containing IP twice.

To call from one domain to another, the call to the target function is executed, resulting in a memory access violation. The kernel then checks to see if the thread is owned by a path, and if the path data structure contains a mapping from the current protection domain to the target environment and function. If this mapping exists, the kernel switches to the appropriate protection domain and continues execution using the same thread. Since the mappings are maintained in a per-path hash table, access time is almost always constant. Upon return, a memory trap to a special address occurs, triggering the kernel to remove the last protection domain crossing from its stack and return to the caller that triggered the protection domain crossing.

Using the Alpha calling conventions, Escort passes integer arguments across protection domain boundaries in registers. Arguments passed by reference are either copied onto the stack that is mapped in the appropriated protection domain, or an IOBuffer (described in section 3.3) is used. This makes inter-domain calls indistinguishable from regular function calls, and allows the system builder to draw protection boundaries between modules as needed. In other words, whether a protection domain boundary sits between any pair of modules need not be known at the time the modules are implemented.

Escort threads cannot be preempted gracefully. They are similar to non-preemptive threads, with the exception that they can be preempted if they are destroyed immediately afterwards. The removal of a thread, however, most likely leaves its owner in an inconsistent state. Therefore, the owner of a removed thread is itself removed. Since Escort allows the kernel to specify a maximum thread runtime without yields for each owner, this mechanism is good enough to deal with runaway threads, but it does not impose the synchronization overhead within modules that would be necessary if preemptive threads were used.

In addition to `threadHandoff`, `threadYield` and `threadStop` operations, the kernel also supports events and semaphores. Again, these objects are owned by either paths or protection domains. Events allow modules to fork new threads that start executing a given function after a specified delay. Semaphores can be used to block threads. The threads that can be blocked on a semaphore are not limited to threads of the owner of the semaphore. If a semaphore is destroyed, however, all threads that do not belong to the owner of the semaphore are unblocked.

The thread scheduler is configured during configuration time. Escort currently supports a priority-based scheduler, a proportional share scheduler, and an EDF scheduler.

3.3 IOBuffers

Escort uses IOBuffers to pass blocks of data between protection domains. IOBuffers are similar to FBufs [8], except they use a more elaborate reference counting scheme and more restrictive mapping rules. IOBuffers are managed by the kernel and can be allocated, locked, unlocked, and associated with an owner. IOBuffers are always allocated as a multiple of the system's page size.

When an IOBuffer is allocated, it is associated with the owner that is specified as an argument. The owner argument is restricted to either the current protection domain, or a path that crosses the current protection domain. If the owner is the current protection domain, the IOBuffer is mapped read/write in that domain. If the IOBuffer is associated with a path, it is mapped read/write in the current protection domain, and read-only in all other protection domains along the path. The current direction that IOBuffer is flowing is also specified as an argument; direction is given by specifying the next stage along the path that will process the IOBuffer.

To allow paths to traverse multiple security levels, it is possible to designate certain protection domains along a path as termination domains. This limits the read mapping to the protection domains along the path from the current protection domain, up to an including the termination domain. An identifier for the protection domain that can write in an IOBuffer is stored as first long word in the IOBuffer.

The kernel keeps a reference count for each IOBuffer; a buffer's reference count is incremented by locking it.

Locking an IOBuffer removes all write privileges from the buffer; this is indicated by setting the protection domain id field in the IOBuffer to zero. The purpose of removing all write permission is that after locking an IOBuffer, the buffer can be checked for consistency and cannot be altered anymore by the original writer.

Unlocking an IOBuffer decrements the reference counter and removes all write mappings. If the reference counter reaches 0, the buffer is freed or added to a buffer cache. If an IOBuffer is allocated, and it has read mappings in the same protection domains as a cached buffer, the current protection domain mapping is changed to read/write and the buffer is reused. The advantage of this scheme is that cached IOBuffers do not have to be cleaned and a buffer allocation requires only changes in one protection domain's memory mapping.

A final kernel call associates a pre-existing IOBuffer with a second owner. The mapping directions and restrictions are specified in the same way as during IOBuffer allocation. This feature is useful for an application that implements a cache (e.g., a web cache): it allows the protection domain that manages the cache to allocate the IOBuffer, and later map the buffer into all protection domains traversed by paths that use (send/receive) the cached data. No copying is required and only one copy of each data item is stored. This association call includes locking the buffer for the second owner. The second owner is also fully charged for the buffer. This is necessary to avoid the case in which the original owner removes its lock and the second owner does not have enough resources to actually own the buffer. The disadvantage is that there are more resources charged for than actually used.

The message library [12] is used to efficiently manage the IOBuffer and offer a simple user interface tailored for manipulating network messages. All meta data used by the message library is stored in IOBuffers. The message library can deal with the possibility that it might lose write permission to an IOBuffer transparently. It also adds another layer of reference counting without involving the kernel. As a result, each protection domain holds at most one kernel lock on any IOBuffer reducing the number of kernel calls.

4 Performance

This section reports measurements of Escort designed to demonstrate the costs and benefits of accounting for resource usage across multiple protection domains. The example system we use for all our experiments is the web server introduced in Section 2.

4.1 Configurations

We measured Escort under a variety of configurations and loads, as outlined below.

4.1.1 Web Server

We tested four configurations of the web server. The first three run on Scout and implement the module graph shown in Figure 1. The fourth configuration runs on Linux. We denote the four configurations as follows:

Scout: All modules and the kernel are configured in a single, privileged protection domain. This configuration does no resource accounting, and so is equivalent to a base Scout kernel.

Accounting: Like Scout, all modules are implemented in a single protection domain, but the system accounts for all resources consumed by paths and protection domains.

Accounting_PD: Includes resource accounting, but each module is configured in its own protection domain. This is the worst-case scenario since each inter-module call implies a protection domain crossing. The module graph for this configuration is shown in Figure 3.

Linux: Apache 1.2.6 web server running on RedHat 5.1 with the 2.0.34 Linux kernel.

4.1.2 Load

The experiments place the following kinds of load on the web server:

Client: A regular client performs a sequence of requests to retrieve the same document. The document sizes used are 1-Byte, 1K-Byte and 10K-Byte. The small document sizes were chosen to minimize the effect of TCP congestion control on the experiment.

QoS Stream: A QoS Stream corresponding to one TCP connection with a guaranteed bandwidth of 1-MBps. A proportional share scheduler is used to ensure that the path responsible for this connection receives this bandwidth. The web server can only guarantee that enough resources for this stream are available on the server; it cannot guarantee sufficient bandwidth is available within the network.

CGI Attacker: A CGI Attacker performs a GET request at a rate of one every second. The request results in an an infinite-loop thread that emulates a runaway CGI script. This experiment simulates the impact a single user which is allowed to upload CGI scripts on a WWW server can have on the overall performance of the server. It also represents the most basic attack on an active network in which router and end hosts execute code associated with an active packet.

SYN Attacker: A SYN Attacker sends a SYN request to the server at a rate of 1000 every second.

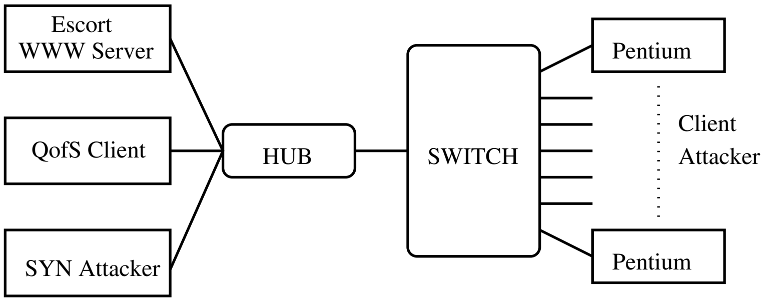

Figure 7: Experimental Setup

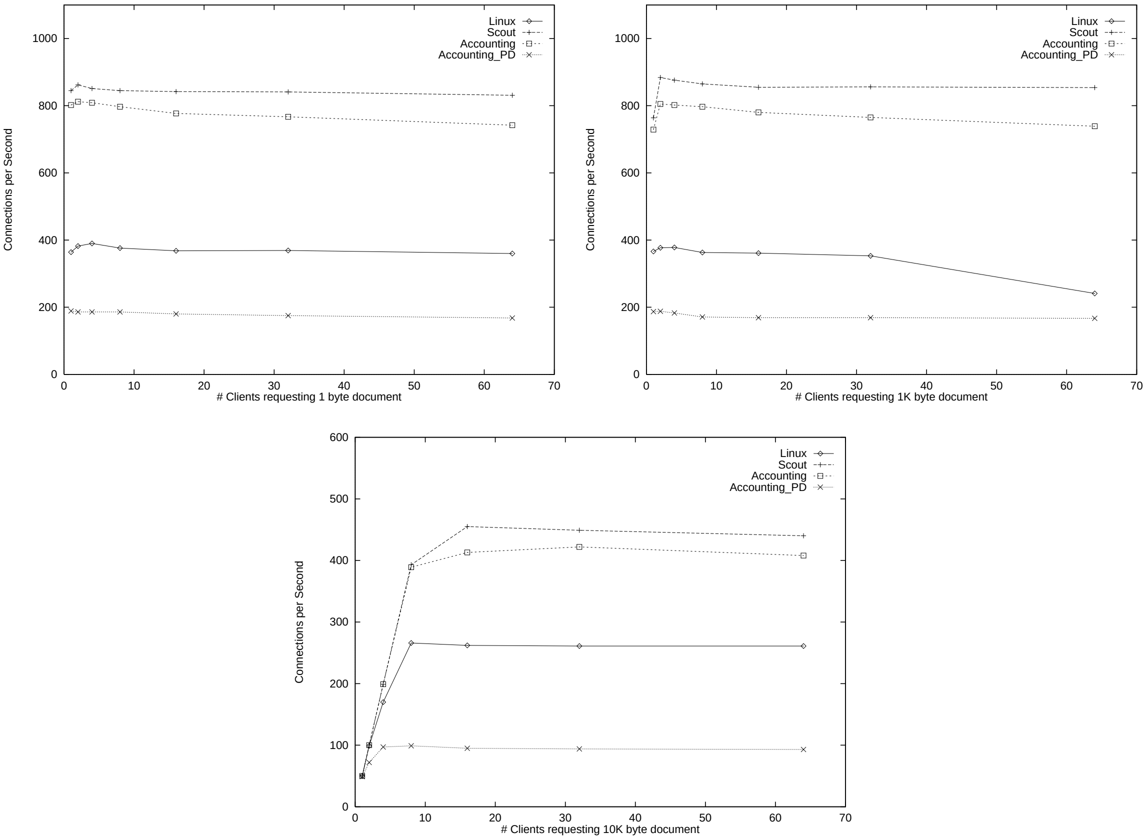

Figure 8: Basic performance of the different configurations in connection per second for a 1Byte document 1KByte document and 10KByte document.

4.1.3 Hardware

All four server configurations, as well as the QoS receiver and the SYN Attacker, run on 300MHz AlphaPC 21064 systems with Digital Fast EtherWORKS PCI 10/100 (DE500) Ethernet adapter connected to a 100Mbps Ethernet. The clients and CGI Attackers run on one to 64 200MHz PentiumPro workstations running Linux. These stations are connected by 100Mbps Ethernet cards to a CISCO Cat5500 switch. The switch is connected by a hub to the web server, the receiver of the QoS stream and the SYN Attacker.

The full configuration is shown in Figure 7. There are two reasons for this particular hardware configuration. First, it is possible to run a single Client and a single CGI Attacker on each PentiumPro, eliminating the effects of having overly loaded sources. Second, all Client and CGI Attacker traffic share one 100Mbps Ethernet link. This reduces the number of collisions on the hub and gives the QoS traffic enough network capacity to sustain the 1MBps rate.

4.2 Accounting and Protection Overhead

The first set of experiments measure the overhead imposed on the system by Escort's accounting and protection domain mechanisms. Specifically, Figure 8 reports the performance of the web server as it retrieves documents of size 1-byte, 1K-bytes, and 10K-bytes, respectively, from between 1 and 64 parallel clients. All measurements represent the ten-second average measured after the load had been applied for one minute.

The best performance is achieved by the base Scout kernel with Escort's accounting and protection domains disabled; the server is able to handle over two times as many requests as the Apache server running on Linux (800 versus 400 connections per second). This is not surprising considering that Linux is a general-purpose operating system with different design goals. It does, however, demonstrate that we used a competitive web server for our experiments.

Adding fine-grain accounting to the configuration decreases the server's performance by an average of 8%. This decrease in performance can be mostly attributed to keeping track of ownership for memory and CPU cycles.

Adding protection domains decreases the performance by an additional factor of over four. The impact of adding multiple protection domains is rather high, but keep in mind that we configured every module in its own protection domain so as to evaluate the worst-case scenario. In practice, it might be reasonable to combine TCP, IP, and ETH in one protection domain. Each additional domain adds, on average, a 25% performance penalty to the single domain case. We say "on average" because the actual cost depends on how much interaction there is between modules separated by a protection boundary.

Another contributing factor is a bug in our OSF1 Alpha PAL code that requires the kernel to invalidate the entire TLB at each protection domain crossing. Other single address space operating systems [14] have shown significant performance improvements by replacing the OSF1 PAL code with their own specialized PAL code. We plan to implement this fix, as well as modify the PAL code in two other ways: (1) to implement some of the system calls directly in PAL code, and (2) to replace the OSF1 page table with a simpler structure of our own. We expect these three optimizations to reduce the per-domain overhead by more than a factor of two.

The difference between 1-byte and 1K-byte documents is less than 3% in most cases, which is not surprising considering that the Ethernet MTU is 1460 bytes and our 100Mbps Ethernet has sufficient capacity. The 10K-byte document connection rate, however, is substantially slowed down by the TCP congestion control mechanisms if less than 16 parallel clients are present. If enough parallel clients are present, the connection rate is between 50-60% of the 1K-byte document case. This seems to be a reasonable slowdown to account for sending multiple TCP segments.

4.3 Micro-Experiments

The next set of experiments measure detailed aspects of the architecture.

4.3.1 Accounting Accuracy

Table 1 shows the results of a micro-experiment designed to demonstrate that Escort accounts for all resources consumed during a single HTTP request; here we focus on CPU cycles. The first row (Total Measured) reports the measured number of CPU cycles used during a request for a one-byte document. The measurement starts when the passive path accepts the SYN packet—resulting in the creation of an active path that serves the request—and concludes when the final FIN packet is acknowledged.[1] The next six rows report the total number of cycles accounted for by Escort; the last row (Total Accounted) corresponds to the sum of the preceding five.

We measured two configurations: the second column (**Accounting**) gives the results for a configuration that includes accounting but no protection domains, while the last column (**Accounting_PD**) includes both accounting and protection domains.

Owner	**Accounting**	**Acounting_PD**
Total Measured	402033	1123195
Idle	201493 (50%)	9825 (1%)
Passive SYN Path	11223 (3%)	78882 (7%)
Main Active Path	188685 (47%)	1033772 (92%)
TCP Master Event	38 (0%)	514 (0%)
Softclock	92 (0%)	200 (0%)
Total Accounted	402031 (100%)	1123193 (100%)

Table 1: Average number of cycles spent serving 100 serial requests of a one-byte web document.

There are two things to observe about this data. First, Escort accounts for virtually every cycle used, both with and without protection domains. Second, in both the **Accounting** and **Accounting_PD** cases, more then 92% of the non-idle cycles are charged to the active path serving the request. Most of the remaining cycles are accounted to the passive path that receives the SYN request and creates the active path. The number of cycles spent in this passive path is constant for each connection, and therefore its share of the overall time will decrease as the active path does more work.

All other cycles are charged to the TCP master event and the softclock. The TCP master event is responsible for scheduling timeouts of individual TCP connections. The softclock increments the system timer every millisecond and schedules the events. The time spent

[1] Passive and active paths are not an explicit part of the architecture; they are just a way to characterize paths according to their use. The former receive only connection setup messages (e.g., TCP SYN packets), while the latter correspond to open connections on which data messages are sent and received.

incrementing the timer and scheduling the softclock is charged to the kernel (it is constant per clock interrupt); the TCP master event is charged to the protection domain that contains TCP; and the cycles spent actually processing each TCP timeout is charged to the path that represents the connection.

4.3.2 Killing a Path

A second micro-experiment measures the time needed to remove all resources associated with a non-cooperating path. In the experiment, a client requests a document and the server enters an endless loop after the GET request is received. Escort then times out the thread after 2ms and destroys the owner.

	Accounting	**Accounting_PD**	Linux
Cycle	17951	111568	11003

Table 2: Cycle needed to destroy non cooperative path.

Table 2 shows the cycles needed to kill the path from the time the runaway thread is detected until all resources associated with the path in all protection domains are destroyed.

The Linux numbers are measured from the time a parent issues a kill signal until waitpid returns. The Linux number are only reported to give a general idea of the cost of destroying a process and should not be directly compared to the Escort numbers. In Escort, the pathKill operation reclaims all resources, including device buffers and other kernel objects. When protection domains are present, all resources associated with the path in every protection domain—as well as all IPC channels and IOBuffers along the path—are also destroyed. As a point of reference, the 111,568 cycles it takes to reclaim resources in a system with both accounting and protection domains represents approximately 10% of the cycles used to satisfy a single request to retrieve a 1-byte document. These numbers should improve as we optimize the inter-domain calls.

4.4 Defending Against Attacks

We conclude this section by considering three scenarios in which Escort can be used to enforce some resource usage policy. The examples we use were selected to illustrate the impact of policies Escort is able to support. We make no claims that the example policies are strong enough to protect against arbitrary attacks; they are merely representative of policies a system administrator might want to implement.

4.4.1 SYN Attack

The first example is a policy that protects against SYN attacks. We assume that there is a trusted part of the Internet and an untrusted part. The goal is to minimize the impact on HTTP requests from the trusted subnet during a SYN attack from the untrusted subnet.

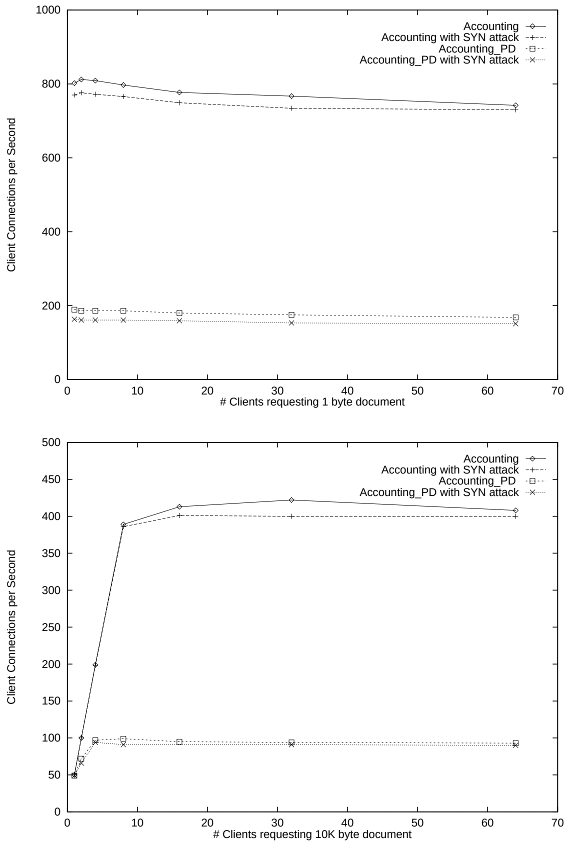

Figure 9: Performance for 1-Byte and 10K-Byte documents for Escort with and without protection domains, with one SYN Attacker generating 1000 SYN requests per second.

Escort implements this policy by providing different passive paths: one accepts SYN requests for the trusted subnet and the other from the untrusted subnet. Each passive path uses a path attribute to keep track of the number of active paths it has created which are in the SYN_RECVD state. This path attribute is monitored by the resource monitor and demultiplexing to the passive path is suspended as soon as 64 paths are in the SYN_RECVD state. Therefore, additional SYN requests are identified as such as early as possible and dropped instantly.

Figure 9 shows the impact on the best effort Client traffic of a SYN attack from the untrusted subnet. The best effort traffic of the **Accounting** kernel slows down by less than 5% for both document sizes. The **Accounting_PD** kernel slows down by less than 15%. Both slowdowns are caused by the interrupt handling and demultiplexing time spent on each incoming datagram. The higher slowdown for the **Accounting_PD** kernel is caused by a higher TLB miss rate during demultiplexing. This is because for each domain-crossing, the TLB is in-

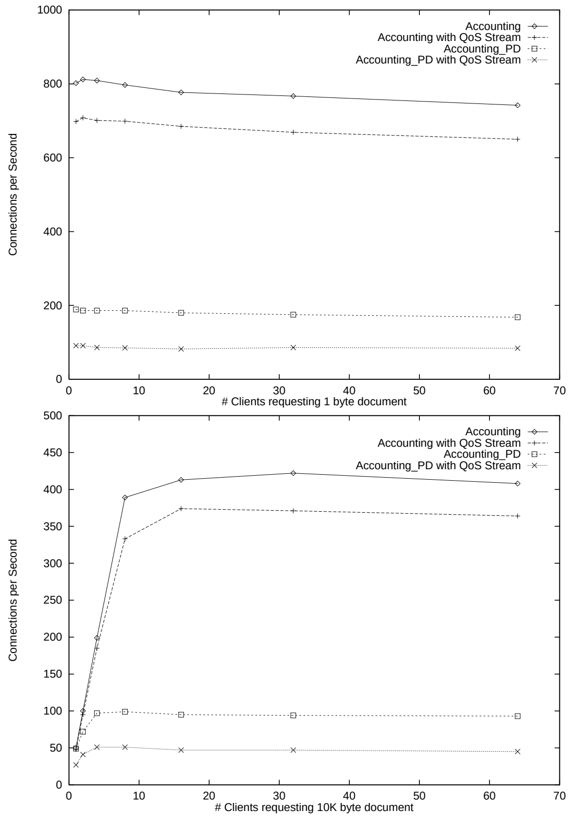

Figure 10: Performance of different configurations with and without a 1MByte/sec Qofs stream in connection per second.

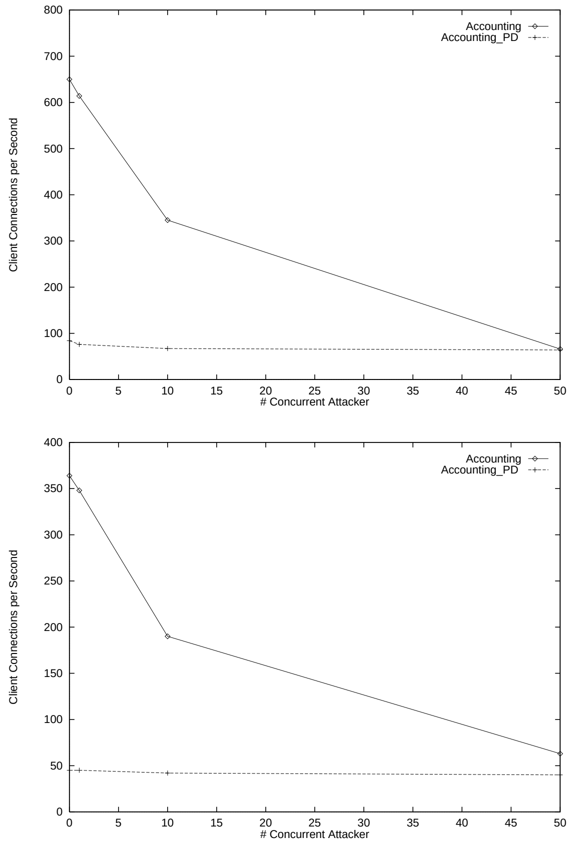

Figure 11: Performance for 1-Byte and 10K-Byte (top down) documents for Escort with and without protection domains, with one 1MBps QoS stream, 64 clients, and a variable number of attackers.

validated and, therefore, no mappings for demultiplexing are present.

The performance for of the 1K-byte documents are not shown but they are within 3% of the 1-byte document.

4.4.2 QoS Stream

In the next experiment we add one 1MBps TCP stream to the base experiment described in Section 4.2. The point of this experiment is to demonstrate that Escort is able to sustain a particular quality-of-service request in the face of substantial load. Figure 10 shows the impact on the best effort client traffic with and without protection domains. The results for the 1K-byte document are not shown but are again within 3% of the 1-byte document.

Although not shown in the figure, the ten-second average of the QoS stream is always within 1% of the target rate. The **Accounting** kernel slows down an average of 15%; the **Accounting_PD** kernel slows down by an average of 50%. This is not a surprising result since Escort with protection domains needs substantially more CPU cycles to sustain a 1MBps data stream.

Note that accounting is required to make QoS guarantees, therefore, we are not able to compare Escort with Linux in this case.

4.4.3 CGI Attack

In our final experiment we add 1, 10, or 50 CGI attackers to the previous experiment. As described earlier in this section, each attacker launches one attack per second. Our example policy realizes the attack within 2ms and removes the offending path. As before, we performed this experiment with 1 to 64 clients, document sizes of 1, 1K, and 10K bytes, and a 1MBps guaranteed data stream.

In all cases, the QoS traffic, as measured over ten-second intervals, stays within 1% of the target rate. Since for our example policy we do not distinguish between attackers and clients until the former has used 2ms of CPU time, the system allows connections from attackers with the same probability as from regular clients. This allows the attacker to slow the best effort traffic down substantially since each attacker consumes 2ms worth of CPU cycles before it is detected. This is shown in Figure 11 for the case of 64 concurrent clients. The advantage of Escort in this scenario is that after the attacker path has been detected and killed, all resources owned by the path have been reclaimed.

4.4.4 Remarks

Note that many alternative policies are possible and easily enforced in Escort. For example, the passive path that fields requests for new TCP connections can be given a limited share of the CPU, meaning that existing active paths are allowed to run in preference to starting new paths (creating new TCP connections). Similarly, clients that have previously violated some resource bound—e.g., the CGI attackers in our example—can be identified and their future connection request packets demultiplexed to a different distinct passive path with a very small resource allocation (or a very low priority). The possibility of IP spoofing, the presence of firewalls, and other aspects may also impact the policy that one chooses to implement. While we believe any such policy can be implemented in Escort, it is not clear that any single policy serves as a silver bullet for all possible denial of service attacks.

5 Related Work

Like Scout, Nemesis [14, 20] avoids cross talk by isolating data streams. It does not, however, take the additional step of accounting for all resource usage in a way that can be used to detect denial of service attacks. It also does not avoid cross talk when a data stream spans multiple protection domains. Escort's linkage and IPC model are also similar to Nemesis', as well as to other single address space operating systems [17, 11, 6].

Whereas Escort and Nemesis extend the operating system by moving functionality from the kernel to user space, Spin [4] and Vino [9] extend the OS by moving functionality into the kernel. However, all four systems face similar challenges. For example, [23] describes how transactions can be used in Vino to protect against misbehaving kernel extensions. The problem with this approach is that any single user of a kernel extension can consume all the extension's resources, even those allocated by other users. As a consequence, all the users of an extension have to trust each other.

Rushby [21] describes the security advantages of modeling a secure system after a distributed system. He argues that organizing an operating system in isolated protection domains which can only communicate via predefined channels as represented in our module graph makes arguing about and achieving high levels of security easier. We extend this idea by providing global QoS guarantees in the form of paths, and therefore enable such a system to deal with denial of service attacks.

LRPC [5] and migrating threads [10] are similar to Escort's thread model. Without the path abstraction, however, a migrating thread can be stopped only by destroying all the protection domains it crosses. This makes it substantially more difficult to defend against denial of service attacks.

6 Conclusions

This paper describes the Escort security architecture that we have implemented in the Scout operating system. Escort is novel in that it supports both end-to-end resource accounting (thereby protecting the system against denial of service attacks) and multiple hardware-enforced protection domains (thereby allowing untrusted modules to be isolated from each other).

We have used Escort to build a secure web server. Experiments with the server show that the accounting mechanism is highly accurate (accounting for virtually 100% of the cycles used to respond to HTTP requests), but imposes a relatively small overhead on the system (on the order of 8%). Enabling protection domains slows the system down by a factor of over four in the worst case measured. In practice, we expect the slowdown to be much less than a factor of two.

Finally, we demonstrate how Escort can be used to implement different denial of service policies. We measure three example policies and demonstrate that it is possible to detect and remove offending clients, while at the same time delivering quality-of-service guarantees to other clients. Although defining effective policies for various attacks is beyond the scope of this paper, we believe Escort provides the necessary mechanisms for implementing such policies.

References

[1] R. Atkinson. *RFC 1825 - Security Architecture for the Internet Protocol*. IETF, August 1995.

[2] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghigha. A domain and type enforcement UNIX prototype. In *Proceedings of the fifth USENIX UNIX Security Symposium: June 5–7, 1995, Salt Lake City, Utah, USA*, pages 127–140, Berkeley, CA, USA, June 1995. USENIX.

[3] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. PATHFINDER: A pattern-based packet classifier. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 115–123, 1994.

[4] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the spin operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, pages 267–284.

[5] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure calls. *ACM Transactions on Computer Systems, TOCS*, 8(1):37–55, Feb. 1990.

[6] J. Chase, H. Levy, M. Baker-Harvey, and E. Lazowska. Opal: A single address space system for 64-bit architectures. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 80–85, 1993.

[7] T. Dierks and C. Allen. *Internet Draft: The TLS Protocol Version 1.0*. IETF, November 1997.

[8] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 189–202, New York, NY, USA, Dec. 1993. ACM Press.

[9] Y. Endo, J. Gwertzman, M. Seltzer, C. Small, K. A. Smith, and D. Tang. VINO: The 1994 fall harvest. Technical Report TR-34-94, Harvard University, Cambridge, MA, 1994.

[10] B. Ford and J. Lepreau. Evolving mach 3.0 to a migrating thread model. In *Proceedings of the Winter 1994 USENIX Technical Conference and Exhibition*, pages 97–114, Jan. 1994.

[11] G. Heiser, K. Elphinstone, S. Russell, and J. Vochteloo. Mungi: A distributed single address-space operating system. Technical Report SCS&E Report 9314, University of New South Wales, Australia, Nov. 1993.

[12] D. Mosberger. Message library design notes. Technical Report TR97-19, The Department of Computer Science, University of Arizona, Tuesday, Nov. 25 1997.

[13] D. Mosberger and L. L. Peterson. Making paths explicit in the scout operating system. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 153–167, Berkeley, CA, USA, Oct. 1996. USENIX.

[14] S. J. Mullender, I. M. Leslie, and D. McAuley. Operating system support for distributed multimedia. In *Proceedings of the Summer 1994 USENIX Conference: June 6–10, 1994, Boston, Massachusetts, USA*, pages 209–219, Berkeley, CA, USA, Summer 1994. USENIX.

[15] G. C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, 15–17 Jan. 1997.

[16] P. G. Neumann. Architectures and formal representations for secure systems. Technical Report SRI-CSL-96-05, Computer Science Laboratory, SRI International, Menlo Park, CA, May 1996.

[17] D. Probert and J. Bruno. Building fundamentally extensible application-specific operating system in SPACE. Technical Report TR95-06, Computer Science Department, University of California Santa Barbara, Oakland, CA, May 1995.

[18] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[19] T. Roscoe. Linkage in the Nemesis single address space operating system. *Operating Systems Review*, 28(4):48–55, Oct. 1994.

[20] T. Roscoe. *The Structure of a Multi-Service Operating System*. PhD thesis, University of Cambridge Computer Laboratory, April 1995.

[21] J. Rushby. The design and verification of secure systems. In *Proceedings of the 8th ACM Symposium on Operating System Principles, Asilomar CA*, pages 12–21, 1981.

[22] 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.

[23] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 213–227, Berkeley, CA, USA, Oct. 1996. USENIX.

[24] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of 14th ACM SOSP*, pages 175–188, Asheville, NC, Dec. 1993.