Virtual Private Services: Coordinated Policy Enforcement for Distributed Applications

Sotiris Ioannidis[1], Steven M. Bellovin[2], John Ioannidis[2],
Angelos D. Keromytis[2], Kostas Anagnostakis[3], and Jonathan M. Smith[4]
(Corresponding author: Sotiris Ioannidis)

Departement of Computer Science, Stevens Institute of Technology[1]
Hoboken NJ, 07030, USA (Email: si@cs.stevens.edu)
Department of Computer Science, Columbia University[2]
MC 0401, 1214 Amsterdam Avenue, New York 10027-7003, USA (Email: {smb, ji, angelos}@cs.columbia.edu)
Institute for Infocomm Research[3]
21 Heng Mui Keng Terrace, 119613, Singapore (Email: kostas@i2r.a-star.edu.sg)
ICIS Dept., Levine Hall, 3330 Walnut St., Phila., PA 19104-6389, USA[4]

(Received Sept. 20, 2005; revised and accepted Nov. 5, 2005)

Abstract

Large scale distributed applications combine network access with multiple storage and computational elements. The distributed responsibility for resource control creates new security issues, caused by the complexity of the operating environment. In particular, policies at multiple layers and locations force conventional mechanisms such as firewalls and compartmented file storage into roles where they are clumsy and failure-prone. Our approach relies on two functional divisions. First, we split policy *specification* and policy *enforcement*, providing local autonomy within the constraints of the global security policy. Second, we create virtual security domains each with its own security policy. Every domain has an associated set of privileges and permissions restricting it to the resources it needs to use and the services it must perform. *Virtual private services* ensure security and privacy policies are adhered to through coordinated policy enforcement points.

Keywords: Distributed access control, security policy, trust management

1 Introduction

Security is an application-defined property, with some applications requiring very little assistance, while others require considerable infrastructure to support their privacy, integrity and availability requirements. When applications were confined to a single computer, the application programmer relied on the host operating system to support these requirements, but the addition of networking, and particularly the Internet, has introduced new challenges for applications with even moderately complex access control requirements. In particular, various network access control mechanisms such as firewalls are largely oblivious to applications (and vice versa), while file access privileges associated solely with users may not cope adequately with untrusted active content. Our proposed solution (introduced in [25]) is *virtual private services* (VPS), which captures the complete access control requirements of a service in a policy specification. This single policy specification can then be used by enforcement mechanisms in hosts, routers, firewalls and elsewhere to produce a coherent consistent environment for the service.

Before elaborating further on virtual private services, we believe an example application of the idea will help make the problem being addressed more concrete. Web services are often implemented with a medium to large-scale Web server, consisting of tens to hundreds of machines in a server "farm", with co-located auxiliary services such as a database, credit card transaction support, and Web mail service. Figure 1 shows the components of an example system, without elaborating on the replicas of each component used in a full scale implementation (for example, hundreds of servers per physical location, and replicated physical locations, each with a database replica and a credit card support system).

Other than what the administrator of such a system has configured manually, there is neither coordination among the nodes in the system, nor is there any coherent relationship between the network access control (achieved with firewalls and routers) and the node access control. The application components thus become very difficult to manage effectively, and misconfigurations and other administrative errors creep in. The causes can range from the difficulty of managing local components, the difficulty

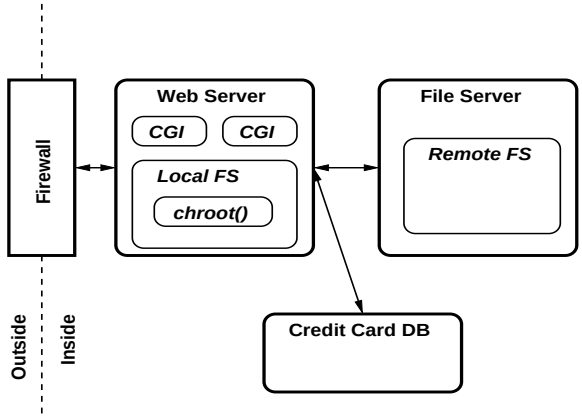

Figure 1: Current general architecture of distributed application clusters

of managing local scale, or the difficulty of coordinating across sites and administrators. Such systems call for an approach which can ensure consistent access control policies, as well as meet application-specific requirements using shared resources such as hosts and the network.

Virtual private services are distributed applications which require coordination amongst client, servers and networks to deliver a reliable, secure service to clients. We feel that the name captures the fact that we are combining ideas from the virtual private networks used to segregate groups of nodes, and the virtual machine models used to control resources in host operating systems. Our new contribution to this problem is designing and building a system architecture for a single ubiquitous *security policy*, which will be enforced everywhere (nodes, networks, *etc.*) to meet the service's access control requirements. Thus, in the Web server sketched earlier, network access and host resource access are managed consistently. For example, if an application or user are not permitted to access a service locally, they are prevented from accessing the same service remotely.

Organization The next section elaborates on the need for virtual private services. Sections 3 and 4 describe our approach, and a design and implementation for the OpenBSD operating system. Section 5 evaluates the system using both micro-benchmarks and in its application to the Flash Web server [39]. Section 6 discusses related work and Section 7 concludes the paper.

2 Motivation

Large distributed systems cannot be administered one machine at a time. This is not, of course, news to system administrators. Many tools (*i.e.*, ASD [26]) have been built to ease the task of administering multiple computers. But for the most part, these tools have been concerned with *files*, rather than policies. Policy configuration files can be centrally administered, but this is more a side-effect than a basic premise of the distribution tools.

The problem is that complex policies must be expressed in a variety of different ways. For example, consider again the network shown in Figure 1. Assume that there is a security policy barring "improper" access to the credit card database. How can this be implemented?

The first obvious step is a firewall rule blocking access from the outside. But more needs to be done to guard against attacks from the inside, either by insiders or from inside machines that have been compromised. Accordingly, the credit card database may have its own configuration and/or policy rules blocking most access from "inside" machines. Indeed, it may be protected by its own packet filter or firewall.

Even from machines authorized to connect to it, not all users can be trusted. Accordingly, other access rules may be needed as well. These may be lists of cryptographic credentials to be accepted, or they may be distributed firewall rules [5, 23], or both. For that matter, the database system may itself have access control mechanisms that need to be configured.

It is clearly impractical to try to configure each of these systems separately. While current tools can easily distribute policy files, the deeper problem—ensuring consistent access policies, across many different systems—is far more difficult. It is this problem that we are trying to solve. It is especially problematic when enforcement must be split across different layers. A rule that says "user A may access database column B on server C when coming from machine D via IPsec" should be specified in one place. Enforcement, however, could be split between a firewall that permits access to the database port from D, a distributed firewall rule on D that recognizes A's credentials, and enforcement of access to particular fields must be in the database server itself.

One attempt to solve this problem in a limited domain is the Firmato [4] firewall language. Firmato is a high-level language for specifying firewall policies. The administrator specifies a policy and a network topology; the policy is then compiled into rule-sets for different models of firewalls, and distributed to each firewall protecting the domain described in the topology.

While this is certainly a step in the right direction—a single policy statement can simultaneously control several different type of firewalls—it is limited to a single *class* of application. As noted above, complex—*i.e.*, realistic—security polices need to control many different *types* of applications. Furthermore, the policies must be enforceable *without* the co-operation of the applications, since they may be subverted.

We can thus list our requirements for an effective, multi-layer security mechanism:

1) The input language must be rich and extensible, in order to be able to express a wide variety of policies, for a wide variety of devices.

2) The input language must be high-level, to avoid unnecessary device-specific semantics.

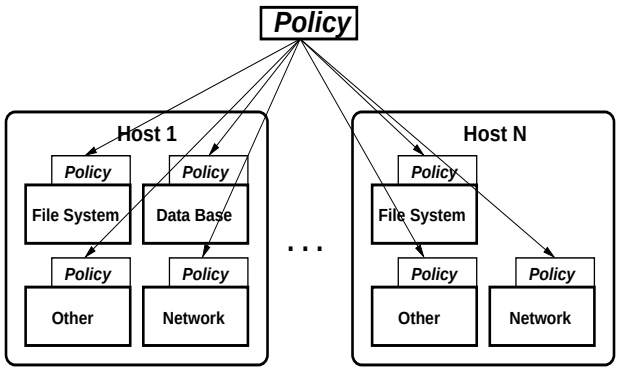

Figure 2: Policy flows from a central specification point to various services. Only the policy rules relevant to a specific service are pulled to that service. No redundant policy state is kept at the access points.

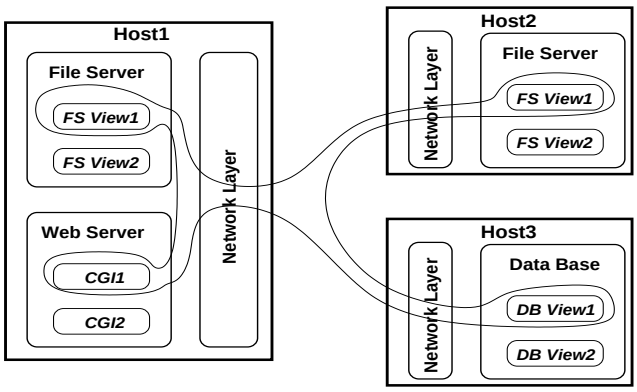

Figure 3: With virtual private services clients are granted access only to the resources they require to accomplish their task. In this example a CGI script running as part of a Web server is only given access to specific subtrees of a local and a remote file system, a part of a database, and can form network connections only to the machines that host the remote file system and data base.

3) There must be a reliable compilation and distribution mechanism that will send the policy to all relevant network nodes.

4) The policy must be completely enforceable by trusted components alone (*i.e.*, dedicated nodes, operating system kernels, *etc.*), without the co-operation of user-level processes on marginally-trusted machines.

The system we describe here accomplishes all of these goals.

3 A Multi-layer Architecture

A security system which can scale with Internet applications must handle growth in the number of clients, enforcement points, and rules pertaining to both, as well as an ability to support a variety of applications, services, and protocols. Policy updates must be as cheap as possible, since these are common and often-used operations in any system (adding/giving privileges to a user, removing/revoking privileges from a user). Security policies for a particular application should be specified in an application-specific language, and a single specification should be able to control the behavior of any needed security mechanism. Finally, administrators should be able to independently specify policies over their own domain: this should be true whether the administrator manages particular applications within a security domain, or manages a sub-domain of a larger administrative domain.

3.1 Separation of Management and Enforcement

The problems we discussed in the previous two sections are exhibited by practically all existing security architectures, and originate from the independent nature of each service. The components/applications that comprise these systems are viewed as independent pieces and are managed as such. Administrators have to configure a number of services *e.g.* Web servers, databases, and compute farms, each with its own security requirements. Conventional mechanisms such as firewalls and compartmented file storage lead to *ad hoc* solutions that often prove inadequate.

The problems originate from the independent nature of each service. Every application has a private notion of a security policy, performs access control according to that specification, and is oblivious to the security policies of other applications. This often causes configuration problems which lead to security violations.

Virtual private services are a promising approach to these challenges. Global security policies are specified for services, while enforcement of these policies remains distributed, local to the resource access points. Figure 2 shows how policy is managed in this scheme. The policy flows from a central specification point to the various services. Only the policy rules that are relevant to each specific service are pulled to that service, so no unneeded policy state is maintained at the various access points. In our architecture we implement policy *management* with the KeyNote [7, 8] trust-management system to express and distribute low-level security policy. Policy credentials are signed by the issuer and self-protected. Policy *enforcement* is carried out by an augmented host operating system (see Section 4).

In Figure 3 we demonstrate virtual private services in the context of a Web server application. A CGI script running as part of a Web server is only given access to specific subtrees of local and remote file systems, a part of a database, and can form network connections only to the machines that host the remote file system and database.

Four benefits accrue:

Scalability: Policy enforcement is distributed to lo-

```
Authorizer: ADMINISTRATOR_KEY
Licensees: USER_A Conditions:
 ((app_domain == "db access") &&
  (db_column == "column B") &&
  (permissions == "FULL_ACCESS") &&
  (dst_addr == "Server C") &&
  (src_address == "Host D") &&
  (ipsec_result == "YES"))
   -> "permit";
```

Figure 4: A simplified representation of the policies for the virtual private services of the database example from Section 2

cal access points, avoiding bottlenecks.

Flexibility: Addition and removal of policies is centralized. The modifications automatically propagate to the enforcement points making administration highly flexible.

Simplicity: Individualized administration of configuration is eliminated, simplifying management.

Consistency: Every service added remains consistent with the central security policy. New services cannot diverge from old policies.

3.2 Policy Translation and Composition

For our architecture to operate on multiple layers, and policy to be enforceable globally we include "referral" primitives; this is simply a reference to a decision made by another enforcement point (typically lower in the protocol stack). This primitive allows us to perform policy composition at enforcement time; decisions made by one enforcement mechanism (*e.g.,* IPsec) are made available to higher-level enforcement mechanisms and can be taken into consideration when making an access control decision.

To accomplish this, all that is necessary is a channel to propagate this information across enforcement layers. In our system, this is done on a case-by-case basis. For example, in our present system IPsec information can be propagated higher in the protocol stack by suitably modifying the Unix `getsockopt(2)` system call; in the case of a Web server and SSL, the information is readily available through the SSL data structures (since the SSL and the Web access control enforcement are both done in the context of a single process address space).

3.3 Sample Policies

In Section 2 we described an example of a policy for a user accessing a specific column in a database with some additional network constraints. Figure 4 shows how such a policy is described in our system. In this example, the administrator authorizes user A to have full access to the database column B, provided they access it on server C coming from host D over IPsec.

```
Authorizer: ADMINISTRATOR_KEY
Licensees: ANY_USER Conditions:
 ((app_domain == "net access") &&
  (src_addr == "ALICE") &&
  (dst_addr == "BOB")) -> "permit";
```

Figure 5: Sample policy for allowing network connections between two machines from Alice to Bob

```
Authorizer: ADMINISTRATOR_KEY
Licensees: ANY_USER Conditions:
 ((app_domain == "ftp access") &&
  (directory == "/ftpdir/*") &&
  (permissions == "READ") &&
  (dst_addr == "BOB")) -> "permit";
```

Figure 6: Specification for an FTP policy

In Section 3.1 we gave a brief example of a service provided by a CGI script (Figure 3). The script requires limited access to the file system (remote and local) and the database, and should not get all the privileges of the Web server. We accomplish this by setting up a distributed policy as seen in Figure 9. The first part of the policy guarantees that the script can only connect to either host2 or host3 from host1, the second part will limit file accesses to directories that only contain data for the script, and last part guarantees will only allow the script to access its own database records. The combination of these simple policies assures the properties of the service provided by the CGI script. These sub-policies are independently enforced by the firewall, filesystem, and database server respectively.

Finally, in Figures 5, 6, 7, and 8 we give examples of simple policies that define virtual private services for different users and applications. Administrators can customize services in their system by specifying such policies and guarantee consistency across all system components.

4 Implementation

Our architecture for virtual private services might appear somewhat complex. We have added a trust management

```
Authorizer: ADMINISTRATOR_KEY
Licensees: WEB_ADM Conditions:
 ((app_domain == "fs access") &&
  (directory == "/www*") &&
  (permissions == "FULL_ACCESS"))
   -> "permit";
```

Figure 7: Policy giving the Web administrator full access to the www directories

```
Authorizer: ADMINISTRATOR_KEY
Licensees: ANY_USER Conditions:
 ((app_domain == "Web access") &&
  (directory == "/www/webpages/*") &&
  (permissions == "READ") &&
  (dst_addr == "WEB_SERVER")
  (dst_port == "80")) -> "permit";
```

Figure 8: Policy allowing any user to access the Webserver pages

```
Authorizer: ADMINISTRATOR_KEY
Licensees: CGI1 Conditions:
 ((app_domain == "net access") &&
  (src_addr == "Host1") &&
  ((dst_addr == "host2") ||
   (dst_addr == "host3")))
   -> "permit";

Authorizer: ADMINISTRATOR_KEY
Licensees: CGI1 Conditions:
 ((app_domain == "fs access") &&
  (directory == "/www/cgi1data/*") &&
  (permissions == "FULL_ACCESS"))
  -> "permit";

Authorizer: ADMINISTRATOR_KEY
Licensees: CGI1 Conditions:
 ((app_domain == "db access") &&
  (records == "cgi1records) &&
  (permissions == "FULL_ACCESS"))
  -> "permit";
```

Figure 9: Set of polices that apply to the CGI script example from Section 3.1

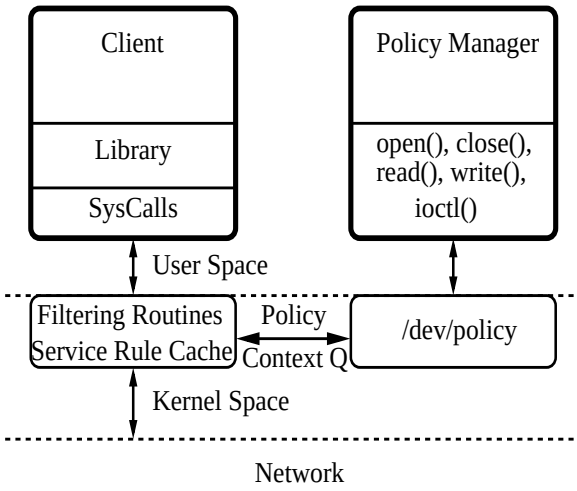

Figure 10: A graphical representation of the system, with all its components. The core of the enforcement mechanism lives in kernel space. Each service *e.g.* (file systems, network layer, *etc.*), has its own filtering routines as well as rule cache for storing policy rules. The policy specification and processing unit lives in user space inside the policy manager process. The two units communicate via a loadable pseudo device driver interface. Messages travel from the system call layer to the user level manager and back using the *policy context queue*.

system, modified host operating systems to control access rights using a policy specification, and added network access control using the same policy specification. We would argue in principle that, since policy dissemination occurs only when needed and the result is cached locally, that the overhead of the trust management system is low. The additional overhead for local enforcement of global policy, however, must be quantified. We used a prototype implementation of virtual private services to perform this evaluation, and were able to limit unwanted unknowns by using a well-understood operating system (4.4 BSD) as a starting point. Section 4 describes the implementation and Section 5 the measurements and evaluation.

We used the OpenBSD operating system [46]. OpenBSD provides well-integrated security features and libraries (an IPsec stack, SSL, KeyNote, *etc.*). Implementations of virtual private services are possible under other operating systems.

Our system has three components: (1) a set of kernel extensions, which implement the enforcement mechanisms at the various access points; (2) a user level daemon process, which implements the centralized policy manager; and (3) a device driver, which is used for two-way communication between the kernel and the policy manager.

Figure 10 shows a graphical representation of the system, with all its components. In the following three subsections we describe the various parts of the architecture, their functionality, and how they interact with each other.

4.1 Kernel Extensions

In the UNIX operating system users create outgoing and allow incoming connections, and access the file system using a number of provided system calls. Since any user has access to these system calls, some "filtering" mechanism is needed. This filtering should be based on a policy that is set by the administrator, and every incoming or outgoing packet as well as file system operation should be subject to it.

Network Access Points: To enforce our policy per-packet, we intercept network traffic at the IP layer and pass the packets to our filtering code, we have created two data structures to assist us in this process.

The first data structure, or *rules cache*, contains a set of rules that packets are compared against. If a match is found, the rule is followed to either accept or drop the packet. The second data structure is the *policy context queue*. A policy context is a container for all the information related to a specific packet. We associate a sequence number to each such context and then we start filling it with all the information the *policy manager* will need to make an access control decision. A request to the policy manager is comprised of the following fields: a sequence number uniquely identifying the request, the ID of the client the connection request belongs to, the number of information fields that will be included in the request, the lengths of those fields, and finally the fields themselves. This can include source and destination addresses, transport protocol and ports, *etc.* Any credentials acquired through IPsec may also be added to the context at this stage. There is no limit as to the kind or amount of information we can associate with a context. We can, for example, include the time of day or the number of other open connections of that user, if we want them to be considered by our decision–making strategy.

As we mentioned already every packet is intercepted at the IP layer and checked against the *rules cache*. If a match is found then the rule is enforced. If no match is found, we enqueue a new request to the *policy context queue*. If we have already enqueued a request for the same class of packets, no further action is necessary. Each entry in the context queue also contains the last packet from that packet flow; if a positive decision is received from the policy manager, the packet is re-queued for processing by the IP stack.

File System Access Points: File system access control works in a very similar fashion to network access control. We intercept file system requests and redirect them to our filtering code.

As in the network case we have another data structure that will hold a set of rules that apply to file accesses. When calls are intercepted we enqueue new requests in the *policy context queue*. The policy manager will receive the request and respond accordingly. Using this technique we can create arbitrary views of the file system, depending on the security policy. This is very much like `chroot(2)` but more like pruning the directory tree of the file system than plainly setting a new root.

Other Access Points: There are a number of other resources that can be easily managed using our architecture, for example memory management and CPU time allocation. However such controls are beyond the scope of this work and have not been implemented in the current prototype. To enable them, hooks require to be added in the memory manager as well as the CPU scheduler, and of course appropriate policies need to be specified in the security manager.

In the next section we discuss how messages are passed between the kernel and the policy manager.

4.2 Policy Device

To maximize the flexibility of our system and allow for easy experimentation, we decided to make the policy manager a user level process. To support this architecture, we implemented a *pseudo device driver*, `/dev/policy`, that serves as a communication path between the user–space policy manager, and the modified system calls in the kernel. Our device driver, implemented as a loadable module, supports the usual operations (`open(2)`, `close(2)`, `read(2)`, `write(2)`, and `ioctl(2)`).

The policy manager reads the device for pending requests in the policy context queue. It then handles the request and returns a new rule to the kernel by writing it to the device, as a result of which the appropriate entry is entered in the rules cache.

The `ioctl(2)` call is used for "house–keeping" tasks. This allows the kernel and the policy manager to re–synchronize in case of any errors in creating or parsing the request messages, and to also flush entries from the rule cache.

4.3 The Policy Manager

The last component of our system is the policy manager. The policy manager is part of the trusted computing base of our system. It is a user-level process responsible for making decisions, based on policies that are specified by some administrator and credentials retrieved remotely or provided by the kernel, on whether to allow or deny connections.

Policies are initially read in from a file. Addition and removal of policies can be done dynamically. The manager can simply flush one or more entries from the rules cache in the kernel. This way subsequent request will not match the existing rule set and the policy manager will be queried for the new policy. The manager receives each request from the kernel by reading the `policy` device. The request contains all the information relevant to that connection as described in Section 4.1. Processing of the request is done by the manager using our trust management system, and a decision to accept or deny it is

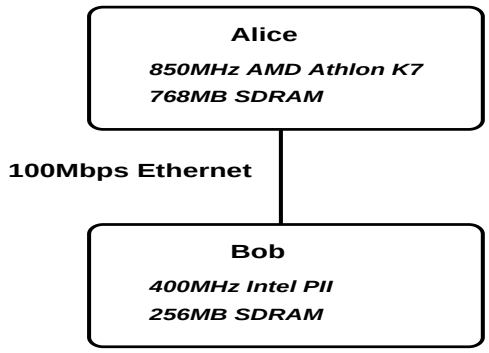

Figure 11: Experimental setup

reached. The decision is sent to the kernel, and the manager waits for the next request. While the information received in a particular message is application-dependent, the manager itself has no awareness of the specific application. Thus, it can be used to provide policy resolution services for many different applications, literally without any modifications.

4.4 Policy Revocation

In an evolving system security demands and constraints change over time. New users and applications are added, old ones are removed, and services are often moved from one host to another.

As we briefly mentioned in the previous section our systems offers policy revocation. Individual or sets of entries can be flushed from the rule cache associated with each access point. Subsequent requests will fail to match the cache resident rule set and will be redirected to the policy manager to acquire the new policy. The revocation functionality is provided by the `ioctl(2)` driver calls.

5 Evaluation

While the architectural discussion is largely qualitative, some estimates of the system performance are useful. To accomplish this we tested our system with the services for network connection, file system access and Web access, defined by the sample policies presented in Section 3.3. Even though the sample services are simple, small scale cases, we believe they provide an adequate picture of the *base* performance of the system. We are currently working on more complex and larger scale scenarios for a more complete evaluation. We performed several experiments, both micro-benchmarks and macro-benchmarks, to get a quantitative evaluation. The experiments are focused on any possible performance overheads introduced by our security features.

Our test machines are x86 architecture machines running OpenBSD 2.8 and interconnected by 100 Mbps Ethernet. More specifically, in the two-host tests that explore the network performance of our system, Alice is an 850 Mhz AMD K7 Athlon with 768MB of memory. Bob, is a 400 Mhz Intel PII with 256MB of memory (see Figure 11). The single host tests, that explore the storage performance of our system were performed on Bob.

Table 1: Average connection overhead measured in ms for 100 TCP connections between Alice and Bob

Network connect	Time (ms)
OpenBSD - Base	50.4
VPS - Cold cache	61.7
VPS - Warm cache	51.8
OpenBSD - IPF	63.1

Table 2: Average round-trip time (in ms) for 200 ICMP ECHO_REQUEST messages

Ping	Time (ms)
OpenBSD - Base	0.273 ± 0.091
VPS - Cold cache	0.283 ± 0.089
VPS - Warm cache	0.282 ± 0.077
OpenBSD - IPF	0.283 ± 0.124

In the following tables, *OpenBSD - Base* means that the measurements were taken on a standard OpenBSD system, where our policy management and enforcement architecture was inactive. *VPS - Cold cache* means that the system is active but the rules cache at the various enforcement points is empty, forcing a query of the policy manager to get the rules. *VPS - Warm cache* means that the rules are in the caches at the enforcement points. Finally *OpenBSD - IPF* means that `ipf(8)`, a standard IP packet filter was active. This represents the case where hosts are protected by firewalls.

5.1 Micro-benchmarks

In Table 1 we have a server application running on Alice; Bob runs a client which connects to the server 100 times using different ports. This generates 200 rules (for incoming and outgoing packets). In the *OpenBSD - IPF* case, those 200 rules are pre-loaded in the filter list. In the second experiment, Bob sent 200 ICMP ECHO_REQUEST messages to Alice; the results are shown in Table 2. We include the standard deviation, as the measurements did vary slightly. These two experiments show us that the cost of compliance checking in our architecture is very small (within 3% of an insecure system, except for the TCP cold cache case which is 20% more expensive), and typically better than *OpenBSD - IPF*. The reason we outperform IPF is due to our simpler design, *e.g.* no logging, IPv6 support, *etc.*, however IPF provides us with an upper bound on packet filtering. This means that an architecture with decentralized enforcement does not unduly affect end-system latency.

The previous experiments investigate network access overheads. In Table 3 we explore how our architecture

Table 3: Open of 100 files in the file system

File system open	Time (ms)
OpenBSD - Base	2.739 ± 0.011
VPS - Cold cache	5.592 ± 0.032
VPS - Warm cache	2.821 ± 0.003

Table 4: Transfer of a 100MB file over TCP, measured in ms

FTP	Time (ms)
OpenBSD - Base	11,131
VPS - Cold cache	11,196
VPS - Warm cache	11,178
IPF	11,151

performs when handling file system accesses. In the case of the cold cache, where no policy rules have yet been cached by the access points we notice a dramatic penalty in performance. This is expected since we have to pay for the additional cross domain call to the policy manager. As described in Section 4.1 when there is no applicable rule at the access point the operating system kernel enqueues a request to the policy manager and waits for the response. Once however the decision is cached, the overhead is minimal (less than 2%).

5.2 Macro-benchmarks

The measurements of Table 4 have a server application running on Alice; a client running on Bob connects to Alice and transfers 100MB, similar to FTP-ing a large file. It is clear that our system does not significantly affect network throughput (the difference is *ca.* 0.5%).

In the previous section, in Table 3, we identified a large disparity in performance, when file access rules are in place or not. To compose a more accurate picture we performed a more realistic experiment, running `make(1)` in the distribution of the Flash Web server. The distribution consist of 23 `.c` files and 22 `.h` files for a total of 12373 lines of code (including comments and white space), about 402KB total. The results are presented in Table 5. Thus, in a more realistic usage of the system than illustrated in Table 3, our architecture imposes very low overhead, as it benefits from the caches purposely frustrated in the micro-benchmark.

To simulate the usage of a search engine we designed a CGI script that performs file intensive operations. The

Table 5: Compilations and linking of the Flash Web server

Make	Time (sec)
OpenBSD - Base	10.4 ± 0.49
VPS - Cold cache	10.6 ± 0.49
VPS - Warm cache	10.6 ± 0.48

Table 6: CGI script counting the number of lines, words and bytes in the .c and .h files of the OpenBSD kernel

CGI query	Time (sec)
OpenBSD - Base	30.31
VPS - Cold cache	32.49
VPS - Warm cache	31.44

Table 7: Apache HTTP server benchmarking tool in two configurations, concurrency of 1 and 50. We report the number of requests per second that were serviced. Since the benchmark always makes the same connection and fetches the same file we only report the results for the warm cache case.

Apache benchmarking tool	Requests per second
`ab -n 500 -c 1`	
OpenBSD - Base	938 ± 31
VPS - Warm cache	930 ± 37
`ab -n 500 -c 50`	
OpenBSD - Base	1507 ± 97
VPS - Warm cache	1484 ± 83

script goes through every `.c` and `.h` file of the OpenBSD kernel and counts the number of lines, words and bytes. We executed the CGI script as part of the Flash Web server and we present the results in Table 6. Even in this intensive file processing script our warm cache performance falls within 4% of the ideal case.

For our final experiment we used `ab(8)`, the Apache HTTP server benchmarking tool. We run it for 500 requests with concurrency 1 and 50, the file transferred was 1024 bytes of static HTML. Since the benchmark always makes the same connection and fetches the same file in Table 7 we report the results for the warm cache case. We notice only a slight degradation in performance, approximately 1%, imposed by the additional access control performed at the network and file system layer.

6 Related Work

System security for large scale distributed applications is driven by the rapidly changing nature of those applications. Some distributed application environments propose use of type-safe languages [16, 17, 29, 32, 53], fault isolation [50] and code verification [36]. Other systems use operating system-specific permission mechanisms [30, 43], system call interception [1, 6, 13, 15], and firewalls [5, 9]. The use of firewalls, for example, illustrates our point about heterogeneity and loose autonomy; firewalls are to a large degree motivated by our inability to secure user hosts. Yet they have become essential tools in each system manager's toolbox. The environment we examine in this paper is one of heterogeneous systems, multiple layers of security mechanism, and great complexity; in that sense

it differs from research focused on single nodes, homogeneous nodes making up a distributed system, or single protocols.

The Flask system [43] extends the idea of capabilities and access control lists by the more generic notion of a *security policy*. The Flask micro kernel system relies on a security server for policy decisions and on an object server for enforcement. Every object in the system has an associated security identifier, requests coming from objects are bound by the permissions associated with their security identifier. However Flask does not address the issue of cooperation amongst clients, servers and networks to deliver reliable and secure services to clients. Its notion of the security identifier is very limiting, in our system we require any number of conditions to hold before we provide a service, for example user identification might not be enough to grant access to a service, the user might also be required to access the service over a secure channel. As a minor issue, we have demonstrated that our prototype can be implemented as part of a widely used, commodity operating system, as opposed to a more fluid experimental micro-kernel.

A different approach relies on the notion of call interposition. Systems like Janus [15], Consh [2], Mapbox [1], and the Mediating Connectors [3], operate at user level and confine applications by filtering access to system calls. To accomplish this they rely on `ptrace(2)`, the `/proc` file system, and special shared libraries. Another category of systems like SubOS [22, 24], Tron [6], SubDomain [10] and others [13, 14, 34, 52], goes a step further. They intercept system calls inside the kernel, and use policy engines to decide whether to permit the call or not. These systems present a number of ways to accomplish access control. Our architecture focuses on separation of policy enforcement and specification, and support for distributed compartmentalized services.

Capabilities and access control lists are the most common mechanisms operating systems use for access control. Such mechanisms expand the UNIX security model and are implemented in several popular operating systems, such as Solaris and Windows NT [11, 12]. The Hydra capability based operating system [28, 55] separated its access control mechanisms from the definition of its security policy. Follow up operating system such as KeyKOS [19, 40, 45] and EROS [41] divide a secure system into compartments. Communication between compartments is mediated by a reference monitor. Our system creates distributed compartments using a centralized policy specification.

The methods that we mentioned so far rely on the operating system to provide a mechanism to enforce security. There are, however, approaches that rely on safe languages, [21, 29, 44, 27] the most common example being Java [32]. In Java applets, all accesses to unsafe operations must be approved by the security manager. The default restrictions prevent accesses to the disk and network connections to computers other than the server the applet was down-loaded from. Our system is not restricted to users of a limited set of type safe languages. We can secure any service running on any host in the system.

Traditional firewall work [9, 18, 33, 35, 37, 42] has focused on nodes and enforcement mechanisms rather than overall system protection and policy coordination. There are however proposed firewall architectures [5, 23] that identify the need for flexible policy specification and distribution. Our system reaches beyond networking, extending the set of services that participate in the security domain.

Trust-based systems like Akenti [47, 48, 49], TPL [20], and others, use certificates to control access to resources. While this is similar to our credential-based approach built on KeyNote, VPS goes beyond simple access control It offers a way for multiple services to work together under a global security policy in a secure and consistent manner.

Our system assumes that the security policies are defined under a single administrative domain. Work by McDaniel et al., on security policy reconciliation, has shown how multiple organization can agree on a common policy that controls their interactions [31, 54]. By using their proposals, it is possible to extend our framework to make it apply on multiple administrative domains.

A security issue we do not address in this paper is Quality of Service (QoS) [38, 51]; here we have focused on resource access control. QoS is ultimately necessary as a defence against denial-of-service attacks; using our architecture it is possible to set up policies that address QoS issues. More specifically, we could use our framework to define policies that will place limits of resource usage at every participating access point, such as rate limits in network traffic, or upper bound in CPU usage.

7 Concluding Remarks

We have argued in this paper that an increasing number of applications are composed from heterogeneous software components interconnected by a network, and that this model introduces new security problems not easily addressed with a conventional set of tools such as compartmented file systems and firewalls. We proposed a new approach, *Virtual Private Services*, which unifies the management of all access control under a single global policy.

Our system copes with scale and heterogeneity, at a low cost in usability, by converting this global policy into a form in which it can be enforced locally. The remaining question was the impact on performance, which we addressed by implementing a prototype system under OpenBSD and then performing a set of micro- and macro-benchmarks selected to cover the space of uses of the server side in a distributed application setting. We used the Flash Web server for macro-benchmark measurements.

Highlights of the measurements (detailed in Section 5) were:

1) The performance effect of decentralized enforcement on end-system latency is small, discounting cache effects.

2) If the policy manager is called frequently, file access can be slowed considerably. When decisions are cached (a common case, as demonstrated by a build of the Flash system), overheads are less than 3%.

3) Large network file transfers are slowed *ca.* 0.5%, whereas small ones *ca.* 1%.

4) Execution of a file-access intensive Web script showed less than 10% overhead in the cold cache case and less than 5% overhead in the warm cache case.

The performance analysis ignored the security advantages of virtual private services. We believe that our hypotheses, that is that the cost of the centralized policy specification was low, and that the policy enforcement cost was low, have been demonstrated. More performance could be gained through recoding and better cache management. Our prototype was only deployed on two hosts as a proof of concept of our proposal, we are however interested in deploying the system in a realistic environment. The main goal of this deployment would be investigating the larger-scale (and unfortunately more qualitative) question of the value of a consistent global policy in real systems.

Acknowledgement

This work was supported by DARPA and NSF under Contracts F39502-99-1-0512-MOD P0001, DUE-0417085, CCR-TC-0208972 and ITR CNS-0426623.

References

[1] A. Acharya and M. Raje, "Mapbox: Using parameterized behavior classes to confine applications," in *Proceedings of the 2000 USENIX Security Symposium*, pp. 1-17, Denver, CO, Aug. 2000.

[2] A. Alexandrov, P. Kmiec, and K. Schauser, "Consh: A confined execution environment for internet computations," available at http://www.cs.ucsb.edu/, berto/papers/99-usenix-consh.ps, Dec. 1998.

[3] R. Balzer and N. Goldman, "Mediating connectors: A non-bypassable process wrapping technology," in *Proceeding of the 19th IEEE International Conference on Distributed Computing Systems*, pp. 73-77, June 1999.

[4] Y. Bartal, A. Mayer, K. Nissim, and A. Wool, "Firmato: A novel firewall management toolkit," in *Proceedings of IEEE Computer Society Symposium on Security and Privacy*, pp. 17–31, 1999.

[5] S. M. Bellovin, "Distributed firewalls," *Login: Magazine, special issue on security*, Nov. 1999.

[6] A. Berman, V. Bourassa, and E. Selberg, "TRON: Process-specific file protection for the UNIX operating system," in *Proceedings of the USENIX 1995 Technical Conference*, pp. 165–175, New Orleans, Louisiana, Jan. 1995.

[7] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis, "The role of trust management in distributed systems security," in *Secure Internet Programming*, LNCS 1603, pp. 185-210, Springer-Verlag, New York, NY, USA, 1999.

[8] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis, *The KeyNote Trust Management System Version 2*, RFC 2704, Sep. 1999.

[9] W. R. Cheswick and S. M. Bellovin, *Firewalls and Internet Security: Repelling the Wily Hacker*, Addison-Wesley, 1994.

[10] C. Cowan, S. Beattie, C. Pu, P. Wagle, and V. Gligor, "SubDomain: Parsimonious security for server appliances," in *Proceedings of the 14th USENIX System Administration Conference (LISA 2000)*, pp. 341–354, Mar. 2000.

[11] H. Custer, *Inside Windows NT*, Microsoft Press, 1993.

[12] H. Custer, *Inside the Windows NT File System*, Microsoft Press, 1994.

[13] T. Fraser, L. Badger, and M. Feldman, "Hardening COTS software with generic software wrappers," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 2-16, Oakland, CA, May 1999.

[14] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson, "SLIC: An extensibility system for commodity operating systems," in *Proceedings of the 1998 USENIX Annual Technical Conference*, pp. 39-52, June 1998.

[15] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A secure environment for untrusted helper applications," in *Procedings of the 1996 USENIX Annual Technical Conference*, pp. 1–13, 1996.

[16] L. Gong, *Inside Java 2 Platform Security*, Addison-Wesley, 1999.

[17] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison Wesley, Reading, 1996.

[18] M. Greenwald, S. K. Singhal, J. R. Stone, and D. R. Cheriton, "Designing an academic firewall. Policy, practice and experience with SURF," in *Proceedings of Network and Distributed System Security Symposium (NDSS)*, pp. 79-91, Feb. 1996.

[19] N. Hardy, "The KeyKOS architecture," in *Operating Systems Review*, pp. 8-25, Oct. 1985.

[20] A. Herzberg, Y. Mass, J. Michaeli, D. Naor, and Y. Ravid, "Access control meets public key infrastructure, or: Assigning roles to strangers," in *Proceedings of IEEE Symposium on Security and Privacy*, pp. 2-14, Apr. 2000.

[21] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles, *PLAN: A Programming Language for Active Networks*, Technical Report MS-CIS-98-25, Department of Computer and Information Science, University of Pennsylvania, Feb. 1998.

[22] S. Ioannidis, S. Bellovin, and J. M. Smith, "Sub-operating systems: A new approach to application security," in *Proceedings of 10th SIGOPS European Workshop*, pp. 108-115, Sept. 2002.

[23] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith, "Implementing a distributed firewall," in *Proceedings of Computer and Communications Security (CCS) 2000*, pp. 190-199, Nov. 2000.

[24] S. Ioannidis and S. M. Bellovin, "Building a secure browser," in *Proceedings of the Annual USENIX Technical Conference, Freenix Track*, pp. 127V134, June 2001.

[25] S. Ioannidis, S. M. Bellovin, J. Ioannidis, A. D. Keromytis, and J. M. Smith, "Design and implementation of virtual private services," in *Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Enterprise Security, Special Session on Trust Management in Collaborative Global Computing*, pp. 269–274, June 2003.

[26] A. Koenig, "Automatic software distribution," in *USENIX Conference Proceedings*, pp. 312-322, Salt Lake City, UT, Summer 1984.

[27] X. Leroy, *Le système Caml Special Light: modules et compilation efficace en Caml*, Research report 2721, INRIA, Nov. 1995.

[28] R. Levin, E. Cohen, W. Corwin, and W. Wulf, "Policy/mechanism separatino in hydra," in *Proceedings of the 5th ACM Symposium on Operating Systems Principles*, pp. 132-140, Nov. 1975.

[29] J. Y. Levy, L. Demailly, J. K. Ousterhout, and B. B. Welch, "The safe-tcl security model, in *USENIX 1998 Annual Technical Conference*, pp. 217-229, New Orleans, Louisiana, June 1998.

[30] D. Mazieres and M. F. Kaashoek, "Secure applications need flexible operating systems," in *The 6th Workshop on Hot Topics in Operating Systems*, pp. 56-61, May 1997.

[31] P. McDaniel and A. Prakash, "Methods and limitations of security policy reconciliation, in *2002 IEEE Symposium on Security and Privacy*, pp. 73-87, IEEE, Oakland, CA, May 2002.

[32] G. McGraw and E. W. Felten, *Java Security: Hostile Applets, Holes and Antidotes*, Wiley, New York, NY, 1997.

[33] B. McKenney, D. Woycke, and W. Lazear, "A network of firewalls: An implementation example," in *Proceedings of the 11th Anual Computer Security Applications Conference (ACSAC)*, pp. 3-13, Dec. 1995.

[34] T. Mitchem, R. Lu, and R. O'Brien, "Using kernel hypervisors to secure applications," in *Proceedings of the Annual Computer Security Applications Conference*, pp. 175-181, Dec. 1997.

[35] J. C. Mogul, "Simple and flexible datagram access controls for UNIX-based gateways," in *Proceedings of the USENIX Summer 1989 Conference*, pp. 203-221, 1989.

[36] G. C. Necula and P. Lee, "Safe, untrusted agents using proof-carrying code," in *Special Issue on Mobile Agents*, LNCS 1419, pp. 61-91, 1998.

[37] D. Nessett and P. Humenn, "The multilayer firewall," in *Proceeding of Network and Distributed System Security Symposium (NDSS)*, pp. 13-27, Mar. 1998.

[38] J. Nieh and M. S. Lam, "The design, implementation and evaluation of SMART: A scheduler for mult imedia applications," in *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pp. 184–197, Oct. 1997.

[39] V. S. Pai, P. Druschel, and W. Zwaenepoel, "Flash: An efficient and portable Web server, in *Proceedings of the USENIX Annual Technical Conference*, pp. 199-212, Aug. 1999.

[40] S. A. Rajunas, N. Hardy, A. C. Bomberger, W. S. Frantz, and C. R. Landau, "Security in KeyKOS," in *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, pp. 78-85, Apr. 1986.

[41] J. S. Shapiro, J. M. Smith, and D. J. Farber, "EROS: A fast capability system," in *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pp. 170-185, 1999.

[42] D. L. Sherman, D. F. Sterne, L. Badger, S. L. Murphy, K. M. Walker, and S. A. Haghighat, "Controlling network communication with domain and type enforcement," in *Proceedings of the 18th National Information Systems Security Conference*, pp. 211-220, Oct. 1995.

[43] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Anderson, and J. Lepreau, "The flask security architecture: System support for diverse security policies," in *Proceedings of the 2000 USENIX Security Symposium*, pp. 123-139, Denver, CO, Aug. 2000.

[44] J. Tardo and L. Valente, "Mobile agent security and telescript," in *Proceedings of the 41st IEEE Computer Society Conference (COMPCON)*, pp. 58-63, Feb. 1996.

[45] *The KeyKOS Operating System*, http://www.cis.upenn.edu/~KeyKOS/.

[46] *The OpenBSD Operating System*, http://www.openbsd.org/.

[47] M. Thompson, A. Essiari, and S. Mudumbai, "Certificate-based authorization policy in a PKI environment," *ACM Transaction on Information and System Security*, vol. 6, no. 4, pp. 566-588, Nov. 2003.

[48] M. Thompson, W. Johnston, S. Mudumbai, G. Hoo, K. Jackson, and A. Essiari, "Certificate-based access control for widely distributed resources," in *Proceedings of the USENIX Security Symposium*, pp. 215-228, Aug. 1999.

[49] M. Thompson, S. Mudumbai, A. Essiari, and W. Chin, "Authorization policy in a PKI environment,"in *1st Annual PKI Research Workshop*, pp. 149-161, 2002.

[50] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software–Based fault isolation," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 203-216, Dec. 1993.

[51] C. A. Waldspurger and W. E. Weihl, "Lottery scheduling: Flexible proportional-share resource management," in *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, pp. 1–11, Nov. 1994.

[52] K. M. Walker, D. F. Stern, L. Badger, K. A. Oosendorp, M. J. Petkac, and D. L. Sherman, "Confining root programs with domain and type enforcement," in *Proceedings of the 1996 USENIX Security Symposium*, pp. 21-36, July 1996.

[53] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten, "Extensible security architectures for Java," in *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pp. 116–128, Oct. 1997.

[54] H. B. Wang, S. Jha, P. D. McDaniel, and M. Livny, "Security policy reconciliation in distributed computing environments," in *IEEE 5th International Workshop on Policies for Distributed Systems and Networks*, pp. 137, 2004.

[55] W. Wulf, R. Levin, and P. Harbison, *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill, 1981.

Sotiris Ioannidis received his Ph.D. in Computer Science from the University in Pennsylvania, Philadelphia, PA, in 2005. He received his M.S. in Computer Science from the University of Rochester, NY, in 1997 and his B.S. in Mathematics from the University of Crete, Heraclion, Greece, in 1996. Currently he is a Reseach Scholar in the Computer Science of Stevens Institute of Technology. His research interests are in host and network security, operating and distributed systems, and security policies in large systems

Steven M. Bellovin is a professor of computer science at Columbia University, where he does research on networks, security, and especially why the two don't get along. He joined the faculty in 2005 after many years at Bell Labs and AT&T Labs Research, where he was an AT&T Fellow. He received a BA degree from Columbia University, and an MS and PhD in Computer Science from the University of North Carolina at Chapel Hill. While a graduate student, he helped create Netnews; for this, he and the other perpetrators were award the 1995 Usenix Lifetime Achievement Award. He is a member of the National Academy of Engineering and the Department of Homeland Security's Science and Technology Advisory Board.

Bellovin is the co-author of "Firewalls and Internet Security: Repelling the Wily Hacker", and holds several patents on cryptographic and network protocols. He has served on many National Research Council study committees, including those on information systems trustworthiness, the privacy implications of authentication technologies, and cybersecurity research needs; he was also a member of the information technology subcommittee of an NRC study group on science versus terrorism. He was a member of the Internet Architecture Board from 1996-2002; he was co-director of the Security Area of the IETF from 2002 through 2004.

John Ioannidis ("JI") is a Senior Research Scientist at Columbia University. The underlying theme of his research has been protecting large-scale infrastructures. In recent years, he was worked on ways of improving the state of interdomain routing with emphasis on scalable and incrementally-deployable protocols. He has also worked on methods to defend against distributed denial of service attacks, with emphasis on solutions that are incrementally deployable. He has also done extensive work on Trust Management and its applications. Older work includes the original Mobile-IP work, swIPe (the precursor to IPsec), and the original implementations of IPsec for BSD and FreeS/WAN.

Angelos D. Keromytis received his Ph.D. in Computer Science from the University in Pennsylvania, Philadelphia, PA, in 2001. He received his B.S. in Computer Science from the University of Crete, Heraclion, Greece, in 1996. Currently he is an Associate Professor of Computer Science at Columbia University, New York, since July 2001. His research interests include design and analysis of network and cryptographic protocols, software security and reliability, and denial of service protection.

Kostas Anagnostakis is currently an associate scientist at the Institute for Infocomm Research in Singapore. He holds a Ph.D. degree from the University of Pennsylvania, USA, and a B.Sc. from the University of Crete, Greece. His main research interests are in computer systems performance analysis and network security.

Jonathan M. Smith is on leave from the University of Pennsylvania, where he holds the Olga and Alberico Pompa Professorship in Engineering and Applied Science. Previous to Penn, Dr. Smith held positions at Bellcore (now Telcordia Technologies) and Bell Telephone Laboratories. He is a Fellow of the IEEE.