

Escort: Securing Scout Paths

by

Oliver Spatscheck

Copyright© Oliver Spatscheck 1999

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF COMPUTER SCIENCE
In Partial Fulfillment of the Requirements
For the Degree of
DOCTOR OF PHILOSOPHY
WITH A MAJOR IN COMPUTER SCIENCE
In the Graduate College
THE UNIVERSITY OF ARIZONA

1 9 9 9

Escort: Securing Scout Paths

Oliver Spatscheck, Ph. D.

The University of Arizona, 1999

Director: Larry L. Peterson

It is becoming increasingly common to find special-purpose communication devices—*Information Appliances*—attached to the Internet. Information appliances include network-attached disks, cameras, and displays; web and file servers; set-top boxes; application routers and firewalls. Many of these systems perform mission critical functions, like company web servers or firewalls, but are built on general purpose operating systems that do not protect them with adequate security measures.

This work introduces Escort, a security architecture for the Scout operating system. Escort provides a set of mechanisms designed to protect information appliances. It uses Scout's path abstraction to provide accurate accounting over multiple protection domains, thereby protecting privacy and integrity while enabling the defense against denial of service attacks. Escort also provides a configuration interface

that allows the designer of the Information Appliance to configure the functional specification and security policy needed for a given environment.

The performance penalty of many secure systems is a deterrent for their deployment. Therefore, an additional goal of Escort is to provide high performance. To achieve this goal, Escort introduces novel mechanisms for shared buffer management and thread migration without introducing security holes. Again, the path abstraction is a major enabling factor for these mechanisms.

This work also presents two example Information Appliances, a web server and a TCP forwarder (firewall). They show how secure high performance systems can be built using Escort's mechanisms. The web server shows, in particular, how to deal with denial of service attacks using a path-based resource revocation mechanism, while the firewall demonstrates a path-based optimization enabled by Escort.

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowed without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the copyright holder.

SIGNED: _____

ACKNOWLEDGEMENTS

I would like to thank my advisor Larry L. Peterson, who was instrumental in completing this work. He not only took time to discuss many ideas and problems with me, he also helped improve my English.

I also would like to thank my committee members John Hartman and Richard Schlichting. They helped me through the process of writing this dissertation. Hilarie Orman, Rich Schroepel and Sean O'Malley were responsible for getting me interested in security in the first place and motivated me to join the PhD program at the University of Arizona. In general, all members of the staff and faculty of the Computer Science Department were extremely helpful in supporting me to achieve my goals.

Over the years I also had many interesting discussions with fellow graduate students and members of the Scout team. To name a few: David Mosberger, Brady Montz, Jørgen S. Hansen and Robert Muth—who suggested Escort as name of my project—were extremely helpful.

Last but not least I would like to thank my parents Bruni and Helmut and my wife Inge. They provided moral and financial support throughout my studies enabling my achievements.

This work was supported in part by DARPA Contract DABT63-95-C-0075 and NSF grant CCR-9415932.

TABLE OF CONTENTS

LIST OF FIGURES	10
LIST OF TABLES	13
ABSTRACT	14
CHAPTER 1: INTRODUCTION	16
1.1 Information Appliances	17
1.2 Securing Information Appliances	20
1.2.1 Security Threats	21
1.2.2 Policy	23
1.2.3 Mechanisms	24
1.3 Securing Scout	26
1.3.1 Modularity	26
1.3.2 Paths	27
1.3.3 Escort: Securing Paths	28
1.4 Thesis Statement and Contributions	30
1.5 Dissertation Overview	31

CHAPTER 2: ESCORT ARCHITECTURE	32
2.1 Overview	32
2.2 The Module Graph	35
2.2.1 Multiple Instantiation	37
2.2.2 Filters	37
2.3 Path	39
2.4 Protection Domains	43
2.5 Owners	45
2.6 Accounting	46
2.7 Kernel ACL	51
2.8 Threads	54
2.9 IOBuffer	56
2.10 Demultiplexing	60
2.10.1 Privacy and Integrity	61
2.10.2 Denial of Service	62
2.11 Configuring Escort	63
2.11.1 Overview	64
2.11.2 Definitions and Macros	66
2.11.3 Owner	70
2.11.4 Path Management	73
2.11.5 Initialization	76

	7
2.12 Related Work	77
CHAPTER 3: SECURITY IMPLICATIONS	81
3.1 Managing Trust	81
3.1.1 Requirements	82
3.1.2 Trust	84
3.2 Security Implications of Escort's Mechanisms	86
3.2.1 Modularity	86
3.2.2 Multiple Instantiation	88
3.2.3 Filters	90
3.2.4 Protection Domains	91
3.2.5 Kernel Access Control	92
3.2.6 Paths	94
3.2.7 IOBuffer	96
3.3 Related Work	98
CHAPTER 4: WEB SERVER	100
4.1 Experimental Setup	101
4.1.1 Configuration	102
4.1.2 Load	102
4.1.3 Hardware	104
4.2 Results	105

4.2.1	Accounting and Protection Overhead	105
4.2.2	Accounting Accuracy	107
4.2.3	Killing a Path	109
4.3	Defending Against Attacks	110
4.3.1	SYN Attack	110
4.3.2	QoS Stream	112
4.3.3	CGI Attack	113
4.3.4	Remarks	114
4.4	Related Work	115
CHAPTER 5: TCP FORWARDER		117
5.1	TCP Forwarding	117
5.1.1	Firewall	121
5.1.2	HTTP Server Proxy	122
5.1.3	Mobile Computing	123
5.2	Connection Splicing	124
5.2.1	Overview	124
5.2.2	Forwarding	126
5.2.3	Splicing	130
5.2.4	Unsplicing	133
5.2.5	Flow Control	135

5.2.6	Additional Optimizations	136
5.2.7	Other Issues	138
5.3	Connection Splicing in Escort	139
5.4	Experimental Setup	143
5.5	Results	147
5.5.1	Processing Overhead	147
5.5.2	Aggregate Throughput	149
5.5.3	Cost of Splicing	151
5.5.4	Buffer Requirements	153
5.6	Related Work	154
CHAPTER 6: CONCLUSION		157
6.1	Contribution	157
6.2	Future Work	158
REFERENCES		161

LIST OF FIGURES

2.1	Escort Overview: A simple WWW server.	33
2.2	Example HTTP Path	39
2.3	Path Data Structure	42
2.4	Modules Partitioned into Protection Domains	44
2.5	Owner Data Structure	48
2.6	Accounting Data Structure	49
2.7	Tracking Data Structure	49
2.8	Limit and Resource Data Structures.	50
2.9	ACL Data Structure	52
2.10	Escort Design Process	64
2.11	Example config.acl.	66
2.12	Example config.resource.	68
2.13	Example config.iface.	70
2.14	Example config.pd.	71
2.15	Example config.path.	72
2.16	Example config.graph.	74

	11
2.17 Example config.pathmanager.	75
2.18 Example config.module.	77
3.1 Module graph for user authentication.	87
3.2 Module graph for user authentication with multiple applications.. .	89
3.3 Module graph for user authentication with filter.	90
3.4 Module graph with two networks.	95
4.1 Router graph for WWW server with all routers configured in separate protection domains.	101
4.2 Experimental Setup	104
4.3 Basic performance of the different configurations in connection per second for a 1Byte document 1KByte document and 10KByte doc- ument.	105
4.4 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.	111
4.5 Performance of different configurations with and without a 1MByte/sec Qofs stream in connection per second.	112
4.6 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.	113

5.1	TCP forwarding via a proxy.	118
5.2	Overview of a application-level firewall. Data from one network passes through the proxy which forwards them to the other network if the desired security guarantees are not violated.	121
5.3	Optimizing two TCP connections into a single spliced connection. .	125
5.4	TCP segment header with fields modified by FWD in bold.	127
5.5	TCP buffers potentially containing acknowledged data.	131
5.6	Further optimizing the spliced connection when there is no fragmen- tation.	137
5.7	TCP forwarding implemented in two Escort paths.	140
5.8	TCP forwarding implemented in a single Escort path.	141
5.9	Connection spliced paths in Escort.	142
5.10	Test Setup	146
5.11	TCP Sequence Number Trace showing the effects of splicing	152

LIST OF TABLES

4.1	Average number of cycles spent serving 100 serial requests of a one-byte web document.	108
4.2	Cycle needed to destroy non cooperative path.	109
5.1	Non-processing related overhead removed from latency measurements.	147
5.2	Firewall and router processing per TCP segment.	149
5.3	Estimated maximum throughput of firewall in MB/s	151

ABSTRACT

It is becoming increasingly common to find special-purpose communication devices—*Information Appliances*—attached to the Internet. Information appliances include network-attached disks, cameras, and displays; web and file servers; set-top boxes; application routers and firewalls. Many of these systems perform mission critical functions, like company web servers or firewalls, but are built on general purpose operating systems that do not protect them with adequate security measures.

This work introduces Escort, a security architecture for the Scout operating system. Escort provides a set of mechanisms designed to protect information appliances. It uses Scout's path abstraction to provide accurate accounting over multiple protection domains, thereby protecting privacy and integrity while enabling the defense against denial of service attacks. Escort also provides a configuration interface that allows the designer of the Information Appliance to configure the functional specification and security policy needed for a given environment.

The performance penalty of many secure systems is a deterrent for their deployment. Therefore, an additional goal of Escort is to provide high performance.

To achieve this goal, Escort introduces novel mechanisms for shared buffer management and thread migration without introducing security holes. Again, the path abstraction is a major enabling factor for these mechanisms.

This work also presents two example Information Appliances, a web server and a TCP forwarder (firewall). They show how secure high performance systems can be built using Escort's mechanisms. The web server shows, in particular, how to deal with denial of service attacks using a path-based resource revocation mechanism, while the firewall demonstrates a path-based optimization enabled by Escort.

CHAPTER 1

INTRODUCTION

It is becoming increasingly common to find special-purpose communication devices—*Information Appliances*—attached to the Internet. Information Appliances include network-attached disks, cameras, and displays; web and file servers; set-top boxes; application routers and firewalls. All these devices have in common that they provide sophisticated communication capabilities.

This trend is fueled by three developments. First, the increase in communication capabilities allows the use of many dedicated systems to achieve a single shared goal. Second, increases in performance of embedded systems [69] allow them to inexpensively provide advanced communication features in devices like pocket computers. Third, the price drop in PCs allows users to dedicate a PC to a single purpose, avoiding difficulties in running a heterogeneous workload.

Mosberger [47] recognized that these dedicated network-centric systems provide certain challenges and opportunities that are not addressed by general-purpose operating systems. This led to the development of Scout [47, 44, 46], an operating system specifically developed for dedicated communication-oriented devices.

However, Scout fails to provide an adequate set of mechanisms for policy-driven security on such systems. The security of these systems becomes increasingly important as they are connected to an ever more hostile Internet.

This work extends Mosberger's work by adding a security architecture, called Escort to Scout. Escort exploits opportunities provided by the dedicated nature of the systems identified above, and addresses the challenge of insecure communication channels among such systems. It uses Scout's path abstraction and supports the possibility that many different dedicated systems can be constructed using Scout.

1.1 Information Appliances

The term Information Appliance was introduced to describe the kind of special-purpose network-centric systems in which we are interested [51]. Mosberger described the major characterization of an Information Appliance [47]. The following section summarizes and interprets this characterization; the special security related aspects of Information Appliances are discussed in Section 1.2.

Communication: Information Appliances are communication-centric. Information Appliances might still perform computation, but most computation is triggered by, or results in, communication. For example, a simple WWW server retrieves documents from a disk and sends them to a client via a communication network. Therefore, the information appliance only performs protocol translations from the disk (SCSI) to the network (IP), triggered by an

incoming HTTP request.

Another implication of the communication-centric nature of an Information Appliance is that the notion of a single application running on a single computer is no longer a satisfying concept. For example, if a client requests a document from a WWW server, multiple computers are involved to fulfill the request. From the client's view, however, a single application-level action has been performed. In this sense, all computers along the communication path of an Information Appliance can be considered part of the application. In the previous example the application would reach from the WWW browser running on the client's machine, through the routers within the network, to the WWW server fulfilling the request.

Specialization: Information Appliances are specialized systems, meaning that each system is dedicated to providing one function. For example, a WWW server does not perform arbitrary computations—it serves WWW documents. This characterization does not imply that an Information Appliance is a trivially simple system. A full blown WWW server, for example, supports CGI scripts, TLS, and other complicated features.

The trend to specialization is mainly fueled by the complexity of today's system. A general purpose operating system (OS) like Unix [62] or WindowsNT is difficult to maintain so that it will run reliably and securely. It is, for

example, trivial to crash WindowsNT with certain software. If, therefore, a WWW server would share one WindowsNT machine with regular users, the reliability would significantly suffer. The same is true for security and resource allocation. The understanding of useful security and resource allocation policies for a single task is hardly understood. The implication of the combination of such policies is even less clear.

This trend to specialized systems was made possible by the reduction in PC hardware costs and the performance increase of embedded systems.

Diversity: One reason for the invention of general purpose systems was to allow a wide variety of applications like, for example, a WWW server, file server and WWW browser to share a common interface useful to all of them. This allows the simple reuse of the functionality provided by this interface. Information Appliances still have to provide the functionality of all those diverse applications. However, each single Information Appliance will only provide a small part of the functionality. In the previous example there will most likely be one Information Appliance for each of the WWW Server, the File Server, and WWW browser.

To make such systems economical, an operating system used to build Information Appliances has to be general enough to allow the reuse of components in multiple highly diverse Information Appliances.

Predictability: Reliability is one of the forces driving the trend towards dedicated systems. However, reliable operation usually requires that we can predict the behavior of the system under different workloads. By using a specialized Information Appliance, the chance of predicting the system's behavior has been drastically improved.

Predictability is usually found in the context of realtime systems that provide strong guarantees. However, this is only one end of the spectrum. In general, it is not economical to use hard realtime systems which provide strong guarantees for most Information Appliances. Soft realtime systems are more economical; however, they have a certain, presumably low, probability that a prediction will fail.

1.2 Securing Information Appliances

The main topic of this dissertation is to provide mechanisms to secure Information Appliances, where security is defined as privacy, integrity and availability [2]. A system supports privacy if access to information adheres to a given policy; it supports integrity if the alteration of data can be limited by a policy; and it supports availability if an unauthorized user cannot prevent an authorized user from making sufficient progress. There are finer grain classification of security properties [77, 85], but, privacy, integrity and availability provide the most commonly used classification, and are sufficient for the arguments presented in this dissertation.

The mechanisms presented—while addressing all three aspects of security—focuses on availability, which is becoming increasingly important and is not sufficiently supported in current systems.

1.2.1 Security Threats

The security threats encountered by a Information Appliance can be categorized into three groups: attacks from within, attacks from the network, and attacks against a group of Information Appliances.

Attacks from within are attacks originating from authorized users against the privacy and integrity of other users' information, or the availability of the Information Appliance. Attacks against the availability of the network appliance are called *denial of service attacks* since the goal of the attack is to deny service to some other authorized user. Due to the specialization of Information Appliances, the defense against this kind of attack is much simpler than in a general-purpose system. Specialization provides the much needed information about the normal operation of the system, and allows for many Information Appliances to enumerate all modes of operations. If all modes of operations are enumerated policy can be based on a specific role a user plays in a certain mode, rather than on the identity of the user. This leads to very fine grain role-based access control decisions [4, 52, 28, 12, 68]. For example, a user might be allowed to append to a certain database file on disk while performing a database transaction, but not be allowed to alter the

file arbitrarily. This kind of fine grain access control can be used to enforce a Clark Wilson integrity policy [20], which relies on the assumption that allowed changes to the database always leaves the database in a consistent state.

The second threat, attacks from the network, is both important and difficult to prevent. The network threat is especially important since an Information Appliance does not provide a stand-alone service, but rather provides its service while communicating with other Information Appliances. Therefore, the network is the lifeblood of any Information Appliance. Without a functioning network, no application level functionality can be provided to the user, resulting in a total denial of service attack.

Attacks from the network are difficult to prevent for two reasons. First the currently used network infrastructure is unreliable and insecure. The TCP/IP [56, 55] protocol stack widely used in the Internet, for example, forces late demultiplexing since reassembly has to be provided. Also, it does not provide strong source authentication. This leads to the widely known TCP SYN flooding attack [71], and other attacks, based on the lack of accountability within the network. Some of these attacks were described by Bellare as early as 1989 [9]. Second, even if the network provides accountability as, for example, IPSEC [3] does in IPv6 [22], there is always computation required to verify the authenticity of incoming data before the responsible user can be identified. This enables an attacker to perform computation on an Information Appliance before the attack can be detected.

The last threat is against a group of Information Appliances, and is currently a topic of research in the active network community. The basic problem is that each individual Information Appliance does not know the global state of a distributed application, but is supposed to provide local protection to ensure the global security of the application. For example, an active network node might allow an incoming active datagram to transmit two outgoing active datagrams, which might not seem to result in a local policy violation, but if every Information Appliance enforces such a policy, one active datagram could reproduce itself with exponential growth, thereby overloading the global network in a short period of time.

1.2.2 Policy

Policy in a network of Information Appliances has to be defined at multiple levels. On the highest level a policy defines the allowed behavior of a distributed application running on multiple Information Appliances. For example, an organization might want to enforce a Bell/LaPadula [8] or Biba [13] policy on its data. The data might be stored on a WWW server, delivered via multiple routers, and viewed using a WebTV. All these Information Appliances have to collaborate to enforce this global policy.

On the next level, this global policy has to be translated into a local policy for each Information Appliance. This translation not only has to consider the overall policy, but it also has to deal with the issues of how much trust can be placed in

each Information Appliance involved. For example, what level of security can be provided by the network protocols? What level of security can be provided by a specific algorithm? In the previous example, it would be a realistic decomposition of the high level policy to entrust the enforcement of the Bell/LaPadula or Biba policy to the WWW server and WebTV, and use secure network protocols to protect against malicious routers.

The lowest level policy is concerned with restricting the operation of an Information Appliance. For example, if the overall policy requires the WWW server to separate data from two different users, this level of policy will be responsible for restricting access to memory, disks and the network to enforce this policy. This level of policy is usually enforced by the kernel on each Information Appliance

1.2.3 Mechanisms

Most secure operating systems rely on a *trusted code base* (TCB) [23]: a code base that is completely trusted and responsible for enforcing a given security policy. This approach, when implemented on a single processor, received substantial criticism by Proctor[58], who argues that a cost-effective TCB without covert channels does not exist.

Sterne and Benson [76] take this argument further, arguing that not only does a covert channel free system not exist, but also that the TCB approach does not address accountability and integrity. Therefore, they put certain trust in individual

applications; in particular, they trust each application to not exploit covert channels. Their approach, which they call *Controlled Application Set* (CAS), is limited, however, in that applications cannot be installed by arbitrary users. Fortunately, this is not a problem for Information Appliances, which are limited to a single function or service, by definition.

Additional requirements of Information Appliances make the TCB-based approach even less suitable. Specifically, Information Appliances not only require privacy and integrity guarantees, but denial of service (DOS) guarantees are in many cases more important than either privacy or integrity. A web-based newspaper, for example, would probably rather give its contents away to everybody than to deny access to their paying subscribers. That is, they very likely might prioritize denial of service over privacy. The important point is that the TCB, by itself, cannot guarantee such a service since the correctness of the application is an essential part of providing this guarantee.

Another characteristic of Information Appliances is that they must use secure communication protocols to authenticate their users. If a TCB is used, then all such protocols would have to be part of the TCB since the TCB is the single point of policy enforcement. This makes the TCB, in all likelihood, the largest piece of software in the system. For example, a system containing a typical TCP/IP protocol stack and basic kernel services (e.g., threads and memory management) contains more than 17,000 lines of code in the Scout operating system [46]. A

simple HTTP server running as an application on this foundation consists of less than 1,000 lines of code. In terms of assurance, the TCB does not provide any benefits, since we have to trust a big piece of code just in order for us to avoid trusting a small piece of code.

Unfortunately, the CAS approach proposed by Stern and Benson is also not sufficient. Since CAS still trusts a TCB-like code base to perform most of the security functionality, it does not eliminate the assurance problem mentioned above. The CAS approach also trusts applications as a whole, which sometimes requires assuring large software components.

1.3 Securing Scout

Scout [47, 44, 46] was introduced to address the specific challenges an operating system encounters trying to support Information Appliances. It provides support for modularity and introduces the path abstraction. The work described in this dissertation adds to Scout the security features needed to address the security challenges outlined in the previous section.

1.3.1 Modularity

Modularity is a fundamental abstraction in system design. Its main advantages are the reuse of software (which makes modular systems more economical) and the management of complexity (which makes modular systems more reliable and

understandable).

Modularity is especially important for Information Appliances that are specialized, but diverse. Therefore, many different Information Appliances have to be supported. Modularity allows us to generate an Information Appliance by combining modules from a tool-box. This allows us the reuse of common modules like the IP network protocol without installing unnecessary functionality.

1.3.2 Paths

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 encapsulates data as it 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 and charged for resource usage.

The main motivations for the path abstraction are to eliminate the performance penalty introduced by modularity and to increase the predictability of Information Appliances by reducing cross talk—resource contention that interferes with the system’s ability to make quality-of-service guarantees to each stream.

1.3.3 Escort: Securing Paths

From a security perspective, the path abstraction represents the information flow within a modular system. Rushby [65] argues that this is a very desirable structure to assure a secure system.

The main problem with the path abstraction is that Scout does not ensure that **all** information flows are represented by paths, which is a necessary requirement to ensure security of a system. If some information flows are not accounted for, an attacker will exploit them. The attacker could either use those unidentified information flows to leak information violating privacy constraints, to alter information violating integrity constraints or to consume resources that are not accurately accounted for violating availability constraints. This brings us back to the argument presented in section 1.2, which argues that no system can provide such functionality since a covert channel free TCB does not exist.

Therefore, Escort isolates and identifies information flows as far as reasonable and offers a design process to cope with the remaining risks. Specifically, Escort added the following features to Scout.

Memory safety: Escort provides hardware enforced protection domains within a single address space. One of the novel features is the transparency of this protection—an appliance designer is allowed to draw protection domains arbitrarily between modules, depending on the security policy, without any

additional programming.

Escort also provides IOBuffers to mediate the overhead introduced by transferring data between multiple protection domains. IOBuffer are similar to fbufs [25], except they use a more elaborate reference counting scheme and more restrictive mapping rules.

IPC restricted to path: Inter process communication (IPC) is restricted to paths.

Since the interfaces of the modules are well defined, Escort enforces that direct communication can only occur along a path using those interfaces.

Accountability of all resources: Scout is based on the voluntary accounting of

most resources. Escort adds mandatory accounting of all resources consumed by a user. It also enforces strict resource limits for single users or group of users based on a security policy that deals with denial of service attacks.

Using the path abstraction, Escort can account virtually all resources to the ultimate user and not to a generic daemon process, as in more conventional operating systems like Unix.

Fine grain access control: Escort restricts access to all kernel provided abstractions by limiting the access of users in a certain role represented by a path.

In this sense, a path can be seen as a cache of capabilities, representing all capabilities a user has within different protection domains.

Path creation is also controlled by policy. A new path not only represents a new information flow, it also represents access rights and resource limits of a certain user in a certain role. Therefore, the path creation policy is one of the most fundamental parts of the system.

1.4 Thesis Statement and Contributions

This dissertation introduces a novel architecture in the fields of operating system security, denial of service prevention, the design of secure Information Appliances and the optimization of secure Information Appliances. The informal arguments and collaborating experiments presented in this dissertation provide strong evidence of their validity.

1. Information Appliances benefit from the novel Escort security architecture.
2. The design process allows a structured risk-based approach to the design of Information Appliances.
3. The prevention of denial of service attacks requires detailed accounting provided by Escort.
4. Information Appliances can be optimized without compromising security using the path abstraction.

1.5 Dissertation Overview

Chapter 2 describes the Escort architecture with a focus on the newly introduced security mechanisms: fine grain accounting, access control, protection domains and shared buffers. It also describes the configuration process used to build and secure an Escort kernel.

Chapter 3 gives a short introduction to the threats a Information Appliance faces and the importance of carefully distributing trust within a Information Appliance. It focuses on the impact the mechanisms introduced in Chapter 2 have on the overall security of the Information Appliance, and how they can be used to reduce the trust placed in low assurance components.

Chapter 4 and Chapter 5 present example Information Appliances to demonstrate and validate Escort's features. Chapter 4 shows a simple WEB server and is used to evaluate the overall performance of Escort. It also shows how a set of simple denial of service policies can be enforced using Escort's mechanisms. Chapter 5 discusses a simple TCP forwarder that can be used for firewalls or level 4 routers. This chapter focuses especially on the optimization opportunities enabled by using Escort instead of a more general purpose operating system.

Chapter 6 concludes the dissertation with some final thoughts and identifies future research topics that became apparent during the course of this work.

CHAPTER 2

ESCORT ARCHITECTURE

This chapter describes the mechanisms Escort provides to support security. It especially focuses on how these mechanisms benefit from Scout’s *path* abstraction. The chapter concludes by describing the configuration process used to build Information Appliances with Escort.

2.1 Overview

Figure 2.1 depicts the major components of Escort described in detail within this chapter. The left side shows a module graph for a WWW server. It contains the Ethernet and SCSI device drive modules (ETH, SCSI), the networking protocol modules (ARP, IP, TCP), the WWW server module (HTTP) and a simple file system module (FS). A single path used to serve a HTTP request is shown (as dark line).

The right side shows the Escort Kernel. The reason for depicting the kernel on the side of the module graph instead of between the hardware and the modules is that device drivers have direct hardware access to the hardware they control and the kernel might not be involved in all hardware accesses. The kernel in Escort

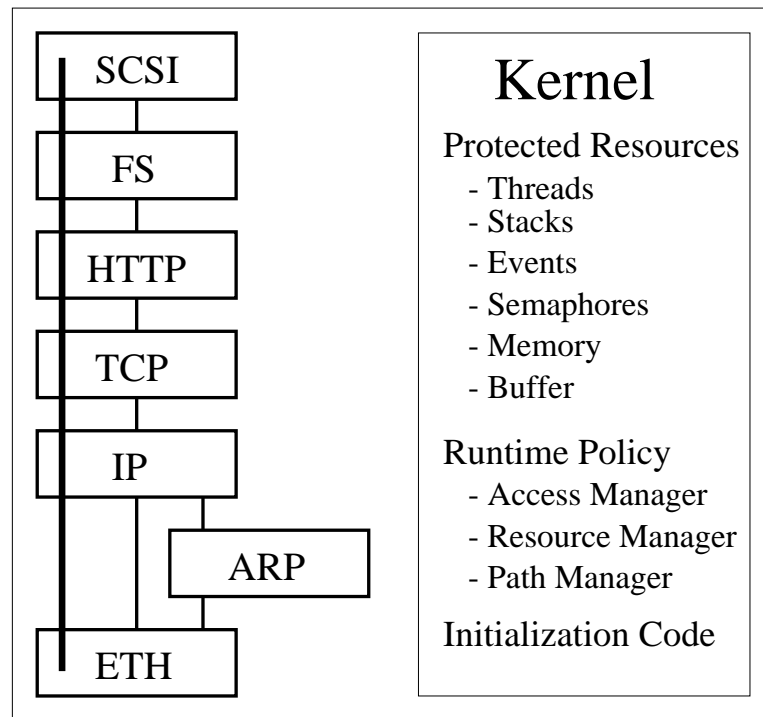


Figure 2.1: Escort Overview: A simple WWW server.

provides the most basic services and might be compared to other microkernels [1, 60, 78, 70, 39]. It provides resource management and access control to memory, the CPU, events, threads, stacks, and semaphores. Hardware enforced protection domains, shared memory and management for path creation and destruction are also provided by the kernel. Most functionalities implemented within more general kernels like WindowsNT or Linux have to be implemented explicitly in Escort using modules. In the example shown, both the networking stack and the file system are part of the configurable module stack, and not part of the base kernel.

The Escort kernel contains a resource manager, access manager and path man-

ager to enforce the configurable runtime policy. It also contains initialization code that executes when the Information Appliance boots. The functionality of this manager can be configured using configuration files described in Section 2.11. Their basic behavior is described below.

- *Initialization Code:* The initialization code is generated during the configuration process (as described in Section 2.11) and is trusted. It is used to initialize the protection domains, the resource manager, path manager and all modules in the system.

Modules are initialized with a callback, passing the configurable initialization information as arguments. The initialization of modules might trigger the creation of certain paths. In our WWW server, for example, an ARP path will be created during initialization. Since the creation of the ARP path is triggered during initialization, the order in which modules are initialized is important and has to be configured.

- *Path Manager:* The path manager determines the topology and the type of a path during path creation and is also responsible for limiting path creation according to a given policy.
- *Resource Manager:* The resource manager not only accounts all resources towards a principal (as described in the section 2.6), but it also is able to

revoke all resources held by a principal and enforce the resource policy as specified during configuration.

- *Access Manager*: Just as the resource manager limits access to resources based on a configurable policy, the access manager limits access to the kernel interface.

2.2 The Module Graph

Modules are the units of program development and configurability in Escort. Each Escort 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.

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 Escort. By configuring Escort with different collections of modules, we can build many specialized Information Appliances for different purposes, including network-attached devices, web and file servers, firewalls and routers, and multimedia displays.

The left side of Figure 2.1 introduced in the previous section shows such a module graph for a Escort 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 assembles the corresponding modules into an executable kernel.

The granularity of the modules is an important factor in minimizing assurance costs. As we will see in the following sections, other Escort mechanisms can be used to restrict communication between modules. This seems to suggest that fine grain modularization ought to be used, since it allows fine grain policy enforcement using those Escort mechanisms. On the other hand, too fine of granularity comes with a performance penalty, as well as with an increase in the complexity of assuring that the combination of modules enforces a given policy.

In principle, Escort does not impose granularity restrictions as long as each module exports well defined interfaces and provides well defined functionality. However, it has been shown in [46, 37, 54] that the granularity of one network protocol—e.g.,

IP or TCP—per module is a useful tradeoff, especially for Information Appliances that mainly contain network protocols. Network protocols can be easily reused and provide a well-defined interface and functionality. Compared to more traditional modular operating systems, the granularity of modularization supported by Escort is much finer. In Mach, for example, a file server is usually implemented as a single daemon. In Escort, the file server would be implemented from a set of modules.

2.2.1 Multiple Instantiation

To facilitate the separation of modules, Escort allows modules to be instantiated multiple times. This allows us to reuse the same modules with different security constraints, relying completely on the separation provided by the OS. This is in contrast to requiring the modules to separate data governed by different security constraints.

Escort implements multiple instantiation by passing a closure during each inter-module call. This closure explicitly specifies the execution environment (state) of the module. In addition, the module graph not the modules themselves restricts communication between instances of modules.

2.2.2 Filters

Economic constraints dictate that modules have to be reused in many Information Appliances. For example, it is not feasible to program a specialized TCP version

for each Information Appliance. Instead one module providing the functionality of the entire TCP protocol [56] is used in all Information Appliances requiring TCP's functionality.

This reuse of modules is not only desirable from an economic perspective, it is also desirable since highly used modules reach a higher level of assurance. The problem with this approach, however, is that modules provide functionality that is not strictly required by the Information Appliance. In many cases this additional functionality can be exploited for attacks against the Information Appliance.

For example, the TCP protocol in a simple WWW server only has to accept incoming connections on port 80, the designated WWW port number. However, a general TCP module is capable to accept connections on any port number. An attacker might use this additional functionality for a denial of service attack by generating state in the TCP module for non port 80 connections.

Escort provides *Filters* to prevent this kind of attack. Filters restrict the interfaces between modules. Since the interface between a pair of modules is module-dependent, filters are specialized to enforce an ACL between two specific modules. In our WWW example, one filter between TCP and IP might restrict the port numbers on which TCP can receive SYN requests to port 80. From Escort's perspective, filters are identical to regular modules. However, the Escort framework provides configurable filters for common modules.

The main advantage of filters is that they are small (more easily assured) and

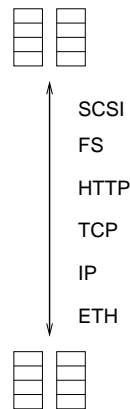


Figure 2.2: Example HTTP Path

can be used to enforce properties on an interface of a general reusable module.

2.3 Path

Escort uses Scout’s communication-oriented abstraction—the *path*. As previously introduced, a 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 and charged for resource usage.

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 2.2 schematically depicts a path that traverses the module graph shown in Figure 2.1. 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 allows Escort to combine the local state and resources of all modules involved in performing a single task, like the HTTP request in the example module graph. In addition, paths can be used to keep global state associated with a single task.

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 in `config.path`, as is described in Section 2.11.4.

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., a “HTTP path”). The possible path types and topologies are configured in `config.pathmanager` as is described in Section 2.11.4 . Next, the kernel consults `config.path` 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 [50]), the calling protection domain (see Section 2.4), and the calling owner (see Section 2.5). 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 the path to which 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;

```

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

```

Figure 2.3: Path Data Structure

(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.

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.

The path data structure, as shown in Figure 2.3, 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.

2.4 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 runs in a privileged protection domain. The protection domain in which each module executes is specified at configuration time in `config.pd`, as is described in Section 2.11.3. 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 2.4 shows the module graph for our example WWW server partitioned into separate protection domains, with 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.

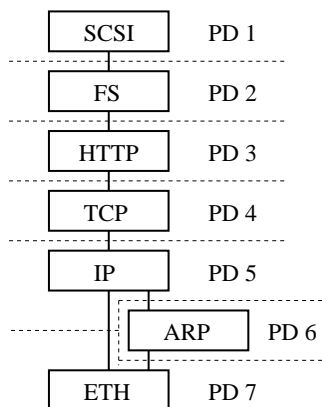


Figure 2.4: Modules Partitioned into Protection Domains

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 [81], type safe languages like Java [32], and proof carrying code [49] 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 Roscoe [63].

2.5 Owners

The Escort kernel provides only the most basic functionality. Therefore, it has no knowledge about users on the granularity of an Unix user or an user as defined by an application. Instead, it accounts resources and bases access control decisions on the owner of the current thread. Specifically, an *owner* is either a path or a protection domain. Paths are the preferred choice, since they most naturally correspond to the actual data flow of an application-level user. 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.

For many Information Appliances and security policies it is essential to keep track of an application-level user that might use more than one path, or a group of application-level users that have the same security constraints. To support this

important feature, Escort allows modules that create new owners to associate those owners with a subject id and subject class id. It is important to note here that both subject id and subject class id are interpreted only by the modules and the security policy. The Escort kernel only manages the ids; it does not interpret them.

For example, a TLS [24] module that provides transport layer security for our WWW server could assign a subject id based on the authenticated identity of the remote user. This subject id is available to the resource manager, for example, to restrict resource usage based on the identity of the remote user and the resource policy. The Escort kernel does not interpret the id.

2.6 Accounting

A key goal of Escort is to account for all resource usage. Towards this end, all resources are charged to an *owner* that corresponds to either a path or a protection domain.

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 transferred back to the protection domain. The destructor function usually frees all memory charged toward the path. However, the domain is ultimately responsible for freeing the memory by returning the page back to the kernel.

Another difference between paths and protection domains 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 to paths.

Figure 2.5 shows the **Owner** data structure; this structure is the first element of both the path and protection domain data structures. The **Owner** structure contains five sub structures. The first structure (**Accounting**) shown in Figure 2.6 keeps a count of the resources—kernel memory, memory pages, IOBuffer, threads,

```

struct Owner {
    OwnerType type; /* PATH or PD */
    /* Accounting */
    struct Accounting used;
    /* Tracking */
    struct Tracking used_list;
    /* Resource monitoring */
    struct Resource limits;
    /* Access Control */
    union ACLS access_limits;
    /* Scheduling */
    struct Scheduler scheduler;
};

```

Figure 2.5: Owner Data Structure

stacks, CPU cycles, events, and semaphores—used by this owner. The `Accounting` structure is 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 `Tracking` structure depicted in Figure 2.7 contains doubly-linked lists of the actual kernel objects associated with this owner; these objects are described in the following sections. These lists support the fast removal of the corresponding objects in the event that the owner must be destroyed. The `Tracking` structure is followed by the `Resource` structure containing the resource limits of the owner. The resource structure is described more detailed below. The `ACLS` union contains the access control lists for the owner and is described in Section 2.7. The last element of the owner structure is the `Scheduler` structure, which contains the information

```

struct 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;
};

```

Figure 2.6: Accounting Data Structure

```

struct Tracking {
    PageList pages;
    ThreadList threads;
    IoBufferLockList iobufferlock;
    EventList event;
    SemaphoreList semaphore;
};

```

Figure 2.7: Tracking Data Structure

necessary to schedule threads belonging to this owner. The exact contents of this data structure depends on the scheduler used.

Whenever a new resource is requested, the owner is explicitly passed as an argument to the kernel allocator. Owners are charged for resources they use, with any limits placed on this usage specified at system configuration time, and at path creation time. The resource limits for a particular owner are given by the Resource

```

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 2.8: Limit and Resource Data Structures.

structure of the Owner data structure. The action to be taken when a given limit is exceeded is specified in the Limit structure; possible actions include destroying the path, denying the request, or preventing further demultiplexing of incoming data to the path. Figure 2.8 shows both the Resource and Limit 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 that 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 multilevel 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.

2.7 Kernel ACL

The kernel interface is protected by a set of access control lists. Protection domains are restricted by a single access control list specified in `config.pd` (Section 2.11.3) that determines which kernel functions a protection domain can call. Paths are

```

union ACLS {
  ACL pd;
  ACL path[MAXPD];
};

```

Figure 2.9: ACL Data Structure

restricted by one access control list per protection domain as specified in `config.path` (Section 2.11.4). The current owner of the thread initiating the kernel call is used to determine the appropriate access control list.

Figure 2.9 shows the access control list (ACL) data structure contained in each `Owner` data structure. If the owner is a protection domain, a single kernel access control list is stored in the `pd` field. If the owner is a path, one access control list for each protection domain the path crosses is stored in the `path` array. In both cases access control lists are stored as bit vectors of type `ACL`, with one bit for each kernel call.

In addition to restricting access to kernel functions, it is also necessary to restrict the arguments. For example, the first argument of most kernel functions that allocate resources is the owner of the resource. Obviously it should not be possible to allocate resources in the name of an arbitrary owner.

This problem can be addressed in two ways. Either the ACL contains restrictions on arguments as implemented using separate argument restrictions for the `pathCreate` call specified in `config.path`, or there are default restrictions that limit

arguments to meaningful values. We decided to implement the second method, since it is faster and does not add configuration overhead. The drawback is that it offers less flexibility. The overall guideline is that the owner of the current thread has to own the data structure passed or referenced in the kernel call. However, there are two exceptions to this rule.

The first exception is that a PD that is crossed by a path can issue operations in the name of the path if the path could have executed the same operation in that particular PD. The rationale behind this exception is that it allows PD's to allocate resources in the name of a path during the creation of the path. A PD is trusted in any case, since it can cause the destruction of a path by charging memory to the path and exhausting cycles using the path's threads.

The second exception is that a path that crosses the same PD as another path can issue operations in the name of the other path if the other path could have executed the same operation in that particular PD. The rationale behind this exception is to allow a path to hand over data to another path without having to first hand the data to a thread owned by the PD.

The exceptions are limited to the kernel calls that are necessary to create paths (exception one) and move data between paths (exception two).

2.8 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 [10] 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, however, Escort provides a hand off 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 twice the protection domain containing IP.

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 2.9) 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 preemp-

tive 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.

2.9 IOBuffer

Since Escort promotes the use of protection domains between modules, rather than applications, the impact of copying data from one protection domain to another

during path traversal could impose a severe performance penalty on the system. To minimize this problem, Escort provides regions of shared memory called IOBuffers. These regions are used to transfer data without copying between protection domains. Escort also provides a library that exports an interface useful to process network packets, thereby hiding the IOBuffer details from the programmer.

IOBuffers are similar to FBufs [25], 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 and including the termination domain.

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 subsequently altered by the original writer.

Unlocking an IOBuffer decrements the reference count 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, `ioBufferAssociate`, 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.

A message library [45] is used to 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 transparently with the possibility that it might lose write permission to an IOBuffer. 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.

The following list summarizes the restrictions an IOBuffer places on its shared memory.

- There is always at most one writer, the one that created the IOBuffer.
- Readers can be limited to a consecutive set of modules along the path as defined in an ACL for this path.
- Each reader is allowed to lock the IOBuffer. The writer automatically holds a lock after creating the IOBuffer.

- If there is more than one lock on an IOBuffer, then the writer loses its write privileges.
- If the lock count reaches zero, the IOBuffer is removed.
- Each holder of a lock is charged for the IOBuffer.

2.10 Demultiplexing

Escort, like Scout, requires data received by a device to be enqueued in the path queue corresponding to that device. The process of finding this path is called demultiplexing. It is not only important to perform scheduling to meet quality of service quarantines, but it is also essential to prevent data of being delivered to unauthorized users and to enforce a resource policy necessary to prevent denial of service attacks.

Demultiplexing for non-network devices is usually trivial. Devices like a keyboard or a mouse, for example, only support one path that delivers keyboard or mouse events to a windowing system. Another example of simple demultiplexing is a SCSI disk device fulfilling SCSI requests. In this case, the incoming data can be matched trivially against the path from which the SCSI request was received.

For network devices, this matching is not as trivial. Currently, Escort supports Scout's distributed packet classifier in which the kernel identifies the path incrementally by invoking a `demux` operation on a sequence of modules, as described in

Section 2.3.

2.10.1 Privacy and Integrity

The base demux mechanism in Escort trusts the demux functions contributed by each module in two ways. First, Escort assumes that the contributed functions are memory safe. Second, Escort assumes that they do not leak information via the demultiplexing decision.

The first assumption can be removed if a typesafe language like Java is used to write the demux function, or if the Escort packet classifier uses protection domains.

The second assumption—that they not leak information via the demultiplexing decision—seems like an overly liberal trust assumption at first; however, this is not necessarily true. In most cases, a demultiplexing decision can be reached early in the networking stack. For example, SPIs in IPSEC and TCP port numbers using TLS are keys that can be used to make such an early demultiplexing decision. In these cases the trust placed in the demux function seems reasonable since only low level network modules which are usually trusted are involved in the demultiplexing decision.

If Escort’s demultiplexing mechanism seems too liberal for the planned Information Appliance, alternative mechanisms—e.g., pattern-based demultiplexers like PathFinder [5] and the current system augmented with Proof Carrying Code [49]—would be more appropriate, since they do not trust the demultiplexing code to

be correct and to not leak information via the demultiplexing decision. Such a mechanism can be integrated easily into Escort.

2.10.2 Denial of Service

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 to denial of service attacks from the time a packet arrives until it has been demultiplexed and authenticated. Escort 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 [3] can authenticate an IPv6 datagrams inexpensively using a secure hash function. This happens during demultiplexing, earlier than it would be possible using TLS [24]. 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 possess valid shared keys, thereby maintaining connectivity to the current set of clients while under attack. We will demonstrate this feature in a later chapter, showing how a web server might limit the cycles spent processing new connections (e.g., SYN packets) by giving preference to existing connections.

2.11 Configuring Escort

As described in previous sections, Escort can be configured to support a wide variety of Information Appliances. Information Appliances are configured using modules, protection domains and paths. Modules define the code that is executed, while protection domains and paths represent the principles that execute the code. The goal of the configuration process is to provide the functionality necessary for the planned Information Appliance as well as to restrict the actions of protection domains and paths.

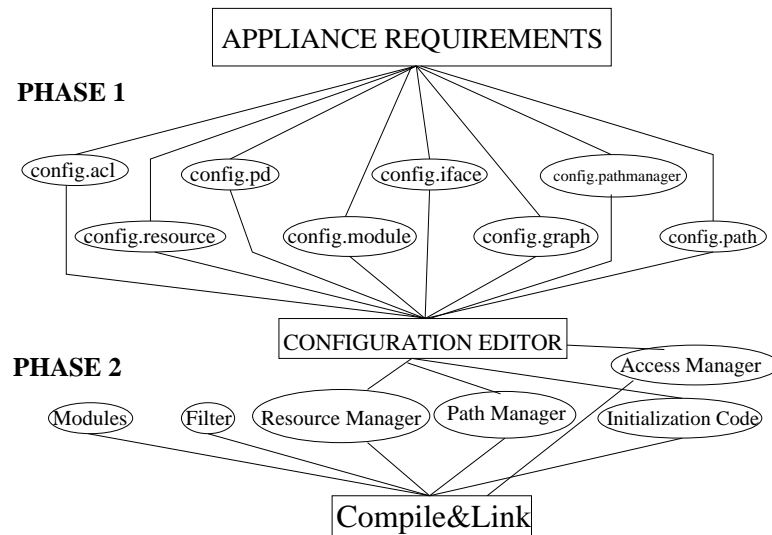


Figure 2.10: Escort Design Process

2.11.1 Overview

Figure 2.10 depicts the configuration process. In phase 1, the requirements of the appliance are decomposed into multiple configuration files. This phase is performed by a human, whom we call the Information Appliance designer. This part of the design process requires the Information Appliance designer to select the necessary modules from the module library, and add new modules if required. All the configuration files are described in detail in the following sections.

The configuration editor uses these configuration files in the second phase to automatically generate the necessary Initialization Code, Resource Manager, Access Manager and Path Manager. After generating the managers, the resulting code is compiled and linked with the modules to form an executable kernel.

The configuration process meets the two goals of configuration: to provide and restrict functionality. It allows the Information Appliance designer to add modules and initialization codes that provide functionality. Furthermore, it restricts functionality by controlling the access a protection domain or path has to resources or state.

Modules are black boxes from the perspective of the kernel. Therefore, restricting access is the more difficult goal to achieve. The kernel can limit module functionality only by limiting module interfaces (access), module access to the kernel and the resources that are available to each module. Resources are directly accounted to their owner, an ACL is used to restrict kernel access, and filters are used to limit the interface of modules.

The path abstraction provides additional handles to limit the functionality of modules in that it provides a kernel enforced mechanism to account resources towards an owner that crosses the module instead of to the module itself. It can also be used to provide additional knowledge about the caller on the module interfaces and, therefore, allows fine grain control of the modules.

In addition to providing and restricting functionality, we also have to deal with assurance. If all modules were to function as advertised, no protection domains would be necessary. However, this can not always be assumed. Modules might have unknown origin (mobile code) or might simply be too complex to trust. In such cases, it is desirable to limit the damages a module can do to the module and all

```

.....
HTTP_SERVER_ACL=>I_ID; I_MEM; I_SEMAPHORE; I_EVENTS; I_NETLISTEN;
  I_AIO; I_pathCreate
.....

```

Figure 2.11: Example config.acl.

paths that cross the module (since paths trust the modules they cross). Towards this end, Escort allows the configuration of modules into different protection domains, thereby isolating faults, and automatically destroying all paths that cross a module if the module violates its resource or access limits.

2.11.2 Definitions and Macros

This section contains a set of configuration files that define macros used in the configuration files subsequently described. It starts with macros defining access control lists (`config.acl`) followed by macros for resource limits (`config.resource`) and module interface and internal resource definitions (`config.iface`).

config.acl

During runtime, the kernel interface is protected by the Access Manager, which is configured using a set of access control lists. The access granted to a thread owned by a protection domain is restricted by a single access control list. Threads owned by paths are restricted by one access control list per protection domain.

All access control lists are defined as macros during configuration time in con-

`fig.acl`, as shown in Figure 2.11. The figure shows the ACL definition necessary for the HTTP module to function properly.

It defines an ACL named `HTTP_SERVER_ACL`, that allows access to the following kernel interfaces: configuration (`I_ID`), memory (`I_MEM`), semaphore (`I_SEMAPHORE`), event (`I_EVENT`), netlisten (`I_NETLISTEN`) and aio (`I_AIO`). It also provides access to the single `pathCreate` kernel interface function. All ACL's are defined in `config.acl` and are used in the subsequent configuration files.

config.resource

Resource limits are enforced during runtime by the Resource Manager. They are defined as macros in `config.resource` (similar to access rights in `config.ac`) during configuration time, and associated with PD's and paths in subsequent configuration files.

Figure 2.12 shows a set of resource limits named `HTTP_PATH_LIMIT`, which we will use later to limit a path used to retrieve HTTP documents. All resources described in Section 2.5 can be limited by this set of resource limits. The example shows limits on the following resources: `kmem`, `pages`, `IOBuffer`, `threads`, `stacks`, `events`, `semaphores`, `yield interval`, `total number of cycle` and the `scheduling priority`. The resource name is followed by the limit and the action to be taken if the limit is violated. By default, the limit is zero; a limit of `INF` represents no limit at all. The action taken during runtime by the Resource Manager when a given limit is

```

.....
HTTP_PATH_LIMIT=>
kmem= (INF,KILL),
pages= (INF,KILL),
IoBuffer= (1638401,NO_DEMUX),
threads= (10,KILL),
stacks= (10,KILL),
events= (2,KILL),
semaphores= (3,KILL),
yield= (300000,KILL),
cycle= (30000000,KILL),
priority= (1,RETURN_ERROR)
attr[TCP_SYN]= (100,NO_DEMUX),
;
.....

```

Figure 2.12: Example config.resource.

exceeded is specified by the second argument. Possible actions include destroying the owner (KILL), denying the request (RETURN_ERROR), or preventing further demultiplexing of incoming data to the owner (NO_DEMUX).

In addition to the limits on kernel resources, `config.resource` can also be used to specify limits on module-specific resources associated with a path. In the example shown, path attribute `TCP_SYN` (`attr[TCP_SYN]`) is restricted to 100. Since path attributes are managed in the kernel and monitored by the Resource Manager during runtime, this functionality can be used to trigger certain actions if resource limits are exceeded. In the example shown, attribute `TCP_SYN` keeps track of the half open TCP connections created by a path listening for incoming TCP connections. If the attribute limit is reached during a denial of service attack at runtime,

the Resource Manager can use this limit to discard (NO_DEMUX) incoming connection establishment requests as early as possible. Without this functionality, all incoming connection establishment requests would have to be processed all the way to TCP since TCP is the module with the module specific knowledge of the number of half open TCP connections. In addition to the performance benefits shown in the example, this functionality also provides a clean interface that separates policy and functionality for module-specific resources.

Attributes are used to specify limits on resources specific to a module. Therefore, the names of the attributes (TCP_SYN in the example) are specific to a certain module and defined in `config.iface`, as shown in the following section.

config.iface

Each module requires an entry in the a `config.iface` configuration file that defines the interfaces exported and the path attributes recognized by the module.

Figure 2.13 shows the part of the `config.iface` file that corresponds to the HTTP and TCP module; similar information is given for each module in the appliance. The HTTP module exports three interfaces in the interface section named `network` of type `aio`, `passive_network` of type `aio_listen` and `file_system` of type `fs`. The `aio` type is a simple asynchronous interface. The `aio_listen` type provides similar functionality to the UNIX socket `listen` call [21], and the `fs` type provides a simple file system interface. The interface names are used to define the module graph as shown in the

```
http=>
    (aio=network,
     aio_listen=passive_network,
     fs=file_system),
;
tcp=>
iface = (aio=up,
        aio=down,
        aio_listen=listen),
attribute = (TCP_SYN, PARTICIPANTS, PASSIVE, ACTIVE),
;
```

Figure 2.13: Example config.iface.

next section.

The TCP module contains both an interfaces section and it also contains an attribute section. The latter specifies that the TCP module uses path attributes called TCP_SYN, PARTICIPANTS, PASSIVE and ACTIVE. The semantic of these attributes is specific to the modules and therefore has to be documented in the module documentation.

2.11.3 Owner

As stated in Section 2.5, protection domains and paths are the granularity on which Escort mediates access and resources using the Access and Resource Manager. Therefore, we have to configure the resource limits and access restrictions a given path or protection domain has. This part of the configuration process is done in the following two configuration files.

```
....  
HTTP_PD=>  
acl=HTTP_SERVER_ACL,  
resource_limit=HTTP_SERVER_LIMIT,  
privileged_mode=FALSE,  
scheduler=FixedPriority,  
;  
....
```

Figure 2.14: Example config.pd.

config.pd

Figure 2.14 shows an excerpt of a protection domain configuration file. The protection domain that is to contain the HTTP module is defined as HTTP_PD with the previously defined access restrictions HTTP_SERVER_ACL and the resource limits HTTP_SERVER_LIMIT. (HTTP_SERVER_LIMIT is not shown, but is similar to HTTP_PATH_LIMIT given in the previous section.)

In addition to kernel access control and resources limits, a protection domain can also be configured in privileged mode. If a protection domain is configured in privileged mode, the code is executed in superuser mode. This mode is useful for highly trusted modules like low level device drivers. Correct accounting and access control is still maintained so the trust placed in the code is limited to the memory safety of the code.

The last parameter that can be configured is the scheduler. Scout currently supports a FixedPriority, earliest deadline first, and proportional share scheduler.

```

...
type=HTTP_PATH,
subject_id=1,
subject_class_id=1,
owner=HTTP_PD,
pd=HTTP_PD,
=>
resource_limit=HTTP_PATH_LIMIT,
scheduler=FixedPriority,
acl[TULIP_PD]=TULIP_PATH_ACL,
acl[ETH_PD]=ETH_PATH_ACL,
acl[IP_PD]=IP_PATH_ACL,
acl[TCP_PD]=TCP_PATH_ACL,
acl[HTTP_PD]=HTTP_PATH_ACL,
acl[FS_PD]=FS_PATH_ACL,
acl[SCSI_PD]=SCSI_PATH_ACL,
;
...

```

Figure 2.15: Example config.path.

config.path

config.path is used to configure the resource and access limits for paths, similar to config.pd. Figure 2.15 shows the part of config.path responsible for a HTTP_PATH, assuming each module is in its own protection domain. The first match is used if multiple entries match.

The arguments matched during runtime when the Path Manager performs a path creation are the path type, as determined by config.pathmanager described in Section 2.11.4; the subject_id, and subject_class_id, which are described in Section 2.5 and are provided as arguments to the pathCreate call; the owner, which is

determined by the owner of the current thread; and the protection domain from which the create call is made. This set of arguments allows us to restrict, during configuration time, the right of path creation to a certain path (or pd) executing in a certain protection domain. It also allows the utilization of the authentication mechanisms of the modules in the path or protection domain during runtime by considering the caller supplied subject and subject_class id. The reason for this design decision is that there are many ways to authenticate remote users within a network. These mechanisms usually require too heavy weight mechanisms to include all of them in a minimal kernel. Therefore, Escort assumes the presence of a privileged module for each type of path that is responsible to determine the user and classify correctly the user into a subject and subject_id.

The definition of resource limits and the scheduler are identical to `config.pd` and are also enforced during runtime by the Resource Manager. However, paths cannot be configured to execute in privileged mode and they have one ACL per protection domain. For example, `acl[HTTP]=HTTP_PATH_ACL` specifies that the `HTTP_PATH_ACL` kernel access control list should be used while a thread belonging to the `HTTP_PATH` executes in the `HTTP_PD` protection domain.

2.11.4 Path Management

Path creation is currently the only way a new owner can be introduced during runtime. Therefore, it is important to keep tight control over the path creation

```
...  
TCP.listen<>HTTP.passive_network;  
TCP.up<>HTTP.network;  
HTTP.file_system<>WEBFS.file_system;  
...
```

Figure 2.16: Example config.graph.

process to ensure the correct and secure operation of the Information Appliance. We already introduced `config.path`, which is used to configure the restrictions enforced by the Access and Resource Manager for a newly created path, and to limit which owner can create certain path types. The following configuration files are used to determine the path type and topology of those paths.

config.graph

The goal of the module graph is to provide communication between the modules and to restrict this communication to the module graph at the same time. It can be used, for example, to interpose filters between two modules, thereby restricting the exported interfaces.

Figure 2.16 shows the part of the `config.graph` file for the instance of the HTTP module defined using the interface definition of the HTTP code base as defined in `config.iface` (Figure 2.13). It connects the HTTP module to the networking stack via the `network` and `passive_network` interface and to the file system stack via the `fs` interface.

```

...
creation_instance=HTTP,
extend=FALSE,
attr[ACTIVE]=TRUE,
=>
path_type=HTTP_PATH,
topology=HTTP.network>TCP.up>TCP.down>IP.up>IP.down>ETH.up>ETH.down>
TULIP.up,
;
creation_instance=HTTP,
extend=TRUE,
=>
path_type=HTTP_PATH,
topology=HTTP.fs>FS.fs>FS.scsi>SCSI.scsi,
;
...

```

Figure 2.17: Example config.pathmanager.

config.pathmanager

config.pathmanager contains multiple of the entries shown in Figure 2.17. Whenever a path is created during runtime, the arguments provided by pathCreate are matched by the Path Manager against the first half of the entries configured in config.pathmanager. The first match is used to determine the topology of the path and the type of the path.

The first entry in config.pathmanager shown in Figure 2.17 configures the initial HTTP path that is responsible for serving HTTP requests before it gets extended to include the file system. The creating module instance has to be HTTP which, is trusted to authenticate the user. The match does not represent an extension

of an existing path (`extend=FALSE`) and the path creation attribute `ACTIVE`—specifying that a regular path to retrieve documents should be created—has to be `TRUE` (`attr[ACTIVE]=TRUE`). If all these conditions are fulfilled, the path type is determined to be `HTTP_PATH` and the topology is determined by specifying the module instances and interfaces involved in the path.

The second entry in the example `config.pathmanager` file configures the extension of the path serving the HTTP request into the file system of the WWW server. For this entry to be matched by the Path Manager during runtime, `pathExtend` has to be called instead of `pathCreate` (`extend=TRUE`) by the HTTP module instance, and an unextended path of type `HTTP_PATH` has to be extended. The topology of the path extension is determined by the topology entry.

At this point it should be noted, that we could determine automatically `config.graph` by parsing `config.pathmanager`. However, this is no longer true if a more complex dynamic system is used, as is anticipated in the future.

2.11.5 Initialization

In addition to the initialization code for the Resource, Access and Path Managers that is automatically created after all configuration files introduced above have been parsed, modules also have to be specialized for the particular Information Appliance. This is done using `config.module` described below.

```
....  
HTTP=>  
module=http,  
pd=HTTP_PD,  
option.port=80,  
;  
....
```

Figure 2.18: Example config.module.

config.module

Modules have to be specialized for a given Information Appliance into module instances. This specialization is configured in `config.module` and performed during boottime of the system. Figure 2.18 shows an example for a HTTP module instance. Since modules can be multiply instantiated we have to name the instance of the module. The `module` argument determines the code part of the module instance; the `pd` argument determines in which protection domain the module instance is instantiated in, and the `option.*` argument is used to specify options known to the module which are used to influence the runtime behavior as described in Section 2.11.1. In the example, `option.port` is used to specialize the port number on which the HTTP module will listen for incoming HTTP requests.

2.12 Related Work

Escort is the security architecture of Scout [47, 44, 46] and therefore shares many common mechanisms with Scout. As already described in more detail in the intro-

duction, however, Scout fails to provide an adequate set of mechanisms for policy driven security on those systems.

Like Scout, Nemesis [48, 64] 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 [57, 35, 18].

Whereas Escort and Nemesis extend the operating system by moving functionality from the kernel to user space, Spin [11] and Vino [26] extend the OS by moving functionality into the kernel. However, all four systems face similar challenges. For example, Seltzer [74] 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.

Capability based systems like Keykos with KeySafe [15], Eros [75], Mach [1] or one of its many derivatives like DTOS [73], DTMACH [29], TRIAD [80] and TMACH [78] have the same goal as Escort—to support the principle of least privilege. However, capability based systems have the drawback of not being able to control the migration and efficient revocation of capabilities. Additionally, in many

cases the management of dynamic capability lists within the kernel also lead to denial of service attacks since the memory used for those lists is not accurately accounted for. Escort uses resource limits and access control lists [67] to restrict a principal. Since paths are created and destroyed during runtime the creation and destruction of access rights becomes dynamic. Paths also contain only the minimal access rights necessary to perform the task at hand so unintended sharing does not occur. However, paths still maintain the advantage of access control lists, which are no migration of access rights, easy access revocation, and tight resource control of kernel data structure.

Another group of similar operating systems are modular systems that can be configured for specific appliances like, for example, the x-kernel [54], Inferno [66, 72], MMLite [36] or GEOS-SC [31] of which none provides a path abstraction and the associated benefits.

The only other project with a path abstraction is CORDS [79], which uses paths for resource allocation. CORDS is a real time networking architecture derived from the x-kernel. CORDS addresses some aspects of using paths for resource accounting, but lacks the comprehensive security architecture and access control provided by Escort.

Rushby [65] describes the advantages of modeling a secure system after a distributed system. He argues that organizing an operating system in isolated protection domains that can communicate only 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 [10] and migrating threads [30] 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.

CHAPTER 3

SECURITY IMPLICATIONS

This chapter validates the usefulness of the mechanisms introduced in the previous chapter from a security perspective. To achieve this goal, we first introduce the concept of trust. In Escort, as in most commercially used systems, different levels of trust are placed in different parts of the system to achieve the overall security goals. To make the mechanisms easier to analyse, we document the trust assumptions using a simple notation.

3.1 Managing Trust

We argued in the introduction that Information Appliances should not put all their trust in a single TCB. Instead, they should place certain trust in certain system components. Since this will lead to a complex set of trust assumptions, we introduce a process that can be used to optimize the design of a network appliance, while documenting all design decisions.

3.1.1 Requirements

Before we describe the trust relationships found on an Information Appliance in more detail, we first add the following security related requirements to the characterization of an Information Appliance.

- *Single Application:* We assume a single application—e.g., a WWW server—is running on the system. This allows us to analyze the system as a whole, which is essential for policies dealing with integrity and DOS attacks since integrity and prevention of DOS can only be guaranteed if at least parts of an application are trusted.
- *Untrusted Network:* We assume that Information Appliances are connected to the Internet, and as a consequence, we do not make any assumptions about the trust or reliability of the network. This makes the problem harder in the sense that we have to assure that all data is properly authenticated and, or encrypted before it is transmitted to, and after it is received from, the network. However, it also makes the problem easier since the network layer makes no reliability guarantees; this fact can be exploited to prevent certain DOS attacks.
- *Remote Authentication:* Since all users are remote, all users have to be authenticated using either source address information or some cryptographic protocol. The trust in source address information is rather weak and the

trust in authentication protocols varies [61]. Therefore, the trust that can be placed into user authentication varies with the trust placed into the authentication process.

- *Cost*: We assume that the price of failure is not infinite. This implies that we have to be very careful in trading off assurance costs against system costs.

The most important factor influencing the design process outlined below is the single application assumption. This assumption allows us to analyze the system as a whole, and does not require us to distinguish between application and system policy. As argued in Stern95 [76], an integrity policy can only be enforced if applications cooperate, and privacy can only be enforced if the semantics of all interfaces are known to the TCB enforcing the policy. In the case of a Information Appliance, the application and operating system policy can be taken into account during the design process, and it can be enforced at the most suitable place.

This restriction is not intended to prevent us from including a Java interpreter, for example, in a Information Appliance. However, if such an interpreter is present and the interpreted code is not trusted, the operating system and appliance code can be viewed as a single TCB from the perspective of the dynamically loaded Java code.

3.1.2 Trust

Regular TCB systems have a single trust assumption: the privacy policy is enforced by the TCB. The CAS approach splits this trust assumption into multiple assumptions. It trusts the CAS-TCB to provide tamper protection, non-bypassability, a trusted path, mediated access to CAS subjects and programs, functional correctness, and multiple CAS domains. The application is trusted to not exploit covert channels and to maintain integrity according to a given policy.

The approach we propose takes the decomposition of trust a step further, and allows the Information Appliance designer to break trust assumptions in arbitrary ways. To document this process, we introduce a trust relationship that captures the dependencies of trust between different parts of the system. We then show how to document and assure the decomposition with the overall goal of balancing assurance cost, system cost, and failure cost (risk). An obvious limitation of our approach is the complexity of assuring the decomposition, which has been shown to be difficult even for simple trust relationships like noninference or noninterference [42].

A trust relationship is defined by the following triple:

$$(X, Y, T) \tag{3.1}$$

where X and Y are system components (e.g., a system policy, user, module, or

thread), and T is a trust assumption (e.g., “does not use covert channels”). This triple says that component X trusts component Y to enforce trust assumption T . For example, a system that uses a TCB to enforce an MLS policy can be represented with the single trust relation:

(policy, TCB, enforces MLS policy)

The CAS approach tries to enforce such an MLS policy by finding an equivalent set of trust relations to the one stated above, but which does not require a perfect TCB. Our approach for Information Appliances also takes into account that the TCB might be the biggest component in the system, and therefore, identifies the fine-grain trust relationships between components traditionally contained in this TCB.

To transfer those sets of trust relationships, we need to define how one set of trust relationships *dominates* another. This is because during the design process, it has to be assured that a new set of trust relationships makes at least the same security guarantees as the original set. We define dominance as follows:

Dominance: One set of trust relationships dominates another if the system resulting from the dominating set enforces all trust relationships of the dominated set.

In general, it will be impossible to prove dominance formally, since the domain of components and relationships depends on the policy and mechanisms used by

a particular application. However, we believe that the explicit documentation of all such assumptions and arguing informally about their correctness is a significant improvement over not formally documenting these assumptions at all.

3.2 Security Implications of Escort's Mechanisms

To allow the Information Appliance designer to generate a good design he has to have a thorough understanding of the security implications of the Escort mechanisms. This section discusses the security implications and shows, by example, how trust relationships can be used to document them.

3.2.1 Modularity

Modularity can be used in different ways to reduce assurance cost of trust relationships. For example, an Information Appliance has to authenticate user data to be able to mediate access to applications and their state. This would be expressed in the following trust relationship if no modularity is present:

```
{(policy, application, authenticate user data)}
```

This single trust relationship is dominated by the following trust relationship if modularity is used to reduce assurance cost. The simple module graph is given in Figure 3.1.

```
{(policy, all modules, do not exploit covert channels),
```

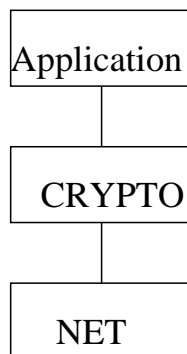


Figure 3.1: Module graph for user authentication.

```

(policy, crypto module, authenticate user data),
(policy, kernel, separate modules),
(policy, kernel, enforce communication graph)}
  
```

The second set of trust relations dominates the first one because authentication has been isolated in the crypto module, and we prohibit all communication not explicitly allowed in the module graph. The result is that all user data is passed through the crypto module, which is responsible for the authentication. Communication is restricted to the module graph by the kernel and the assumption that no covert channels are exploited, as stated in the remaining three trust relations.

Using this approach, assurance has been made more cost effective. The crypto module is most likely reused, and the set of trust relationships assumes only that modules below the crypto module do not exploit covert channels. In the original set of trust relationships, this code was part of the code base trusted to authenticate

the user.

In addition to separation, the kernel now has to enforce a module graph. In Escort, this module graph is explicitly defined. During startup, an ACL is used to restrict inter-module communication .

In contrast, some capability-based systems use capabilities to restrict this access. Verifying if capability systems are enforcing a given communication graph is not trivial. The use of an explicit module graph simplifies this verification. The drawback is that a fixed module graph is less flexible. Escort overcomes this problem by using the module graph for coarse-grain policy decisions, and the path abstraction for dynamic fine-grain decisions.

3.2.2 Multiple Instantiation

To show the benefits of multiple instantiation, we extend the simple example introduced above to contain two applications with different security constraints. If authentication is the goal, this would imply the following additional trust relationships for the crypto module:

```
{(policy, crypto, do not leak data between applications)}
```

Multiple instantiating the crypto module, as shown in Figure 3.2, can replace the amount of trust placed in the crypto module to trust placed into the kernel.

```
{(policy, all modules, do not exploit covert channel),
```

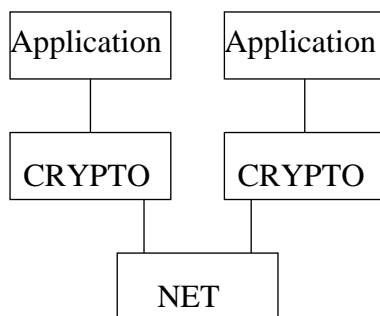


Figure 3.2: Module graph for user authentication with multiple applications..

```

(policy, crypto module, authenticate user data),
(policy, kernel, separate modules),
(policy, kernel, enforce communication graph)}
  
```

This set of trust relations is exactly the same as the one needed to enforce authentication for a single application in Section 3.2.1. Therefore, if multiple instantiation is used, multiple different applications with different security constraints can be implemented without increasing the assurance effort compared to a single set of security constrains.

The second set of trust relations dominates the first set since the relationship in the first trust relationship is concerned with the interference of state within the crypto module on behalf of different applications. Multiple instantiation removes this shared state completely, requiring only the general separation of modules.

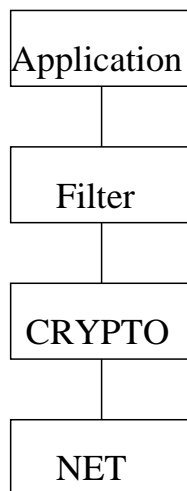


Figure 3.3: Module graph for user authentication with filter.

3.2.3 Filters

Continuing with our previous example, assume that only certain users are allowed to access a certain application. This would require one of the following two trust assumptions:

```
{(policy, crypto, admit only certain users)}
```

or

```
{(policy, application, admit only certain users)}
```

In both cases, either the application or the crypto module is responsible for making a policy-specific access control decision. The filter shown in Figure 3.3 can be used to reduce the trust relationship to:

```
{(policy, filter, admit only certain users),
 (policy, all modules, do not exploit covert channel),
 (policy, crypto module, authenticate user data),
 (policy, kernel, separate modules),
 (policy, kernel, enforce communication graph)}
```

The second set of trust relationships reduces the cost of assurance since both the application and crypto modules can be reused without modification; only the small filter is policy-specific. All other trust relationships contained in the set are already necessary to assure the user authentication previously described.

The second set dominates either of the first two trust relations since access is again restricted to the communication graph, and we trust the crypto module to authenticate the user and the filter to make the policy-based access control decision.

3.2.4 Protection Domains

Using the previously introduced mechanisms in our example, the resulting set of trust relationships always contains the following trust relationships, which were necessary to achieve an assurance advantage:

```
{(policy, kernel, separate modules),
 (policy, kernel, enforce communication graph)}
```

These trust relationships are directly enforced by the kernel using the protection domain mechanism. The kernel has to be verified to assure its correct operation.

An alternative dominant set of trust relationship is shown below:

```
{(policy, all modules, separate modules),
 (policy, all modules, enforce communication graph)}
```

In this case, the modules themselves are trusted to not violate memory constraints (separate modules) and to communicate only via the communication graph. The second set dominates the first one because if all modules are trusted to enforce a certain separation, then the kernel does not have to enforce it.

Since Escort allows the application designer to place modules in protection domains arbitrarily, the trust relationships can be further refined by using the first set between certain sets of modules and the second set between other modules, thereby trading assurance for performance.

Type safe languages can also be used to cheaply assure separation between modules. Thus, type safe languages allow the use of the second set of trust relationships and are an alternative to hardware protection domains.

3.2.5 Kernel Access Control

In our running example we always assume:

```
{(policy, all modules, do not exploit covert channel)}
```

This trust relationship can be transformed in the following dominant trust relationship by splitting the relationship “do not exploit covert channel” into two relationships: one concerned with the device interface, the other with the remaining system.

```
{(policy, all modules,
    do not exploit covert channel in device interface)
(policy, all modules,
    do not exploit covert channel in all interfaces except
    device interface)}
```

Since most modules do not access the device interface, we can use the ACL to restrict access to this interface. This gives us the following trust relationship:

```
{(policy, all modules not accessing device interface,
    do not exploit covert
    channel in all interfaces except device interface),
(policy, all modules accessing device interface,
    do not exploit covert channel),
(user, kernel, enforce device interface ACL restriction)}
```

At first glance, the last set of trust relationships seems more complicated. However, the last set does not require assurances against covert channels in the device

interface for all modules that do not access the device interface. Instead, we have to assure that the kernel ACL is enforced. Since the code enforcing the kernel ACL is smaller and heavily reused, this seems to be the cheaper alternative.

The last trust relationship dominates the previous since, if services are not accessible at all, no covert channels can be exploited using those services. Overall, using a kernel ACL follows the general security principle of least privilege.

3.2.6 Paths

The execution restrictions on threads owned by a path are a major benefit, and can be used to prevent modules from performing requests on behalf of an owner not allowed to perform the request, solving the problem of the confused deputy [34].

Another advantage of paths is the prevention of denial of service attacks. Paths keep detailed information about all resources provided by the kernel and the modules associated with them. These allow each module and the kernel to keep track of resources provided to one user or user class and to revoke those resources.

The example module graph shown in Figure 3.4 depicts an IP module connected to two networks. It also shows a path going from each network to the application. If access to the two networks is controlled by a policy, then the IP module has to be prevented from forwarding packets from NET1 to NET2. This provides the trust relationship:

```
{(policy, IP, no forwarding from NET1 to NET2)}
```

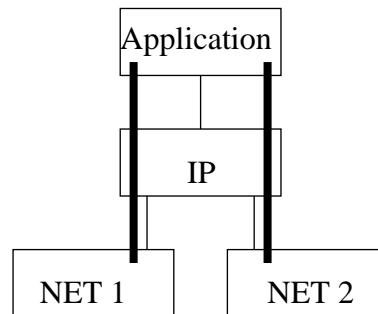


Figure 3.4: Module graph with two networks.

Since routing state has to be shared within IP, IP can not be multiply instantiated to simplify assurance. Instead, paths are used to prevent threads from forwarding data, transforming the relationship to:

```

{(policy, IP, start no threads),
 (policy, IP, do not use covert channels to forward),
 (policy, kernel, implement path correctly)}
  
```

This trust relationship dominates the previous one since now threads can only migrate along the path and we trust IP not to use its state to forward via a covert channel. The kernel ACL can simplify this trust assumption even further to:

```

{(policy, kernel, enforce kernel ACL),
 (policy, IP, do not use covert channels to forward),
 (policy, kernel, implement path correctly)}
  
```

This trust relationship dominates the previous one since the kernel enforces, using its ACL mechanism, that IP does not generate new threads.

Both transformations simplify the assurance since the policy of not forwarding is enforced outside of IP. Therefore, IP can be reused without changes, and kernel components that already have to be assured cover most of the trust.

Capabilities can be used to perform similar restrictions. However, the advantage of paths is that the access control decision can be made completely in advance, at path creation time. This allows us to prevent an attack in which a user changes state in some modules without having the privileges of completing its task. It also increases performance by removing capability checks from common operations, and it simplifies assurance by making the paths explicit. Another major advantage is fast revocation of privileges. Fast revocation of privileges in capability based systems has been shown to be rather complicated [75]. Since in Escort the path represents all access rights, the destruction of the path revokes instantly all those rights.

3.2.7 IOBuffer

Consider the negative impact the IOBuffer mechanism has on security. The writer of the IOBuffer can transfer arbitrary data to all modules in the read set of the IOBuffer. In addition, every reader can signal via the lock function to the writer and probably through timing channels to all other readers. Therefore, the following

trust relationships have to be assured if IOBuffers are to be used.

```
{(policy, IOBuffer writer,
  write only appropriate data for modules in path),
 (policy, all path modules, do not exploit covert channel)}
```

The restrictions placed on IOBuffer help assure these trust relationships. The assumption that IOBuffer mappings are limited to certain segments of the path allows us to transform the above trust relationships in the following dominant set:

```
{(policy, kernel, enforces IOBuffer restrictions),
 (policy, IOBuffer writer, write only appropriate data
  for specified path segment),
 (policy, all path segment modules,
  do not exploit covert channel)}
```

This set dominates the previous one since it requires us to assure that the kernel maps the shared memory only over specific segment, and the original trust relationships are true in this segment.

Another problem with shared data is that the reader cannot assume that two back-to-back read operations return the same value, since the writer could change the data inbetween. A reader, therefore, assumes the following additional trust relationship:

```
{(reader of IOBuffer, writer of IOBuffer,
  do not change data in IOBuffer reader depends on)}
```

Using the lock mechanisms of IOBuffer, this trust relationship can be replaced by the following one:

```
{(reader of IOBuffer, kernel, enforces IOBuffer restrictions),
  (reader of IOBuffer, reader of IOBuffer,
  hold lock on IOBuffer before depending on its data)}
```

The second trust relationship dominates the first one because the lock mechanisms remove all write privileges. Therefore, the kernel enforces the requirement that data does not change, assuming the reader holds a lock. Again, we replaced module specific assurance by assurance of the kernel.

Another advantage of IOBuffers is that only modules that make the assumption that the data does not change, have to hold locks. For example, modules that just add another IOBuffer and hand both of them to the next module are common in a networking stack; they do not have to hold a lock on the original IOBuffer.

3.3 Related Work

Similar arguments about the limitations of the TCB approach were made at a panel discussion [14] in 1997. They are also found in Procter[58] and Stern[76] which offer solutions to the problem. [58] tries to solve the problem of covert channels due to

resource allocation using a distributed approach. Stern[76] is closer to the approach presented here, in that it removes the absolute trust in the TCB. However, we take the approach even further and allow the unrestricted assignment of trust to different system components.

Trust relationships are similar to thread trees presented in Weiss[83]. Thread trees deal with the decomposition of system threads in a systematic fashion. Trust relationships use a more proactive approach in that they guide the design process to achieve a higher level security policy using an iterative process.

CHAPTER 4

WEB SERVER

This and the next chapter describe how Escort can be used to build two different Information Appliances: a web server and a TCP forwarder. Each chapter demonstrates different features of Escort.

This chapter evaluates the costs and benefits of accounting for resource usage across multiple protection domains. In the context of a web server the most important benefit shown is the containment of denial of service attacks, which is made possible by accurate accounting within Escort.

The system used for the experiments is the web server shown in Figure 4.1, which is identical to the one used in Chapter 2 to introduce Escort's mechanisms. It contains an Ethernet (ETH) and SCSI device drive to communicate with the network and disk; the regular TCP/IP protocol stack (ARP, IP, TCP); a simple HTTP 1.0 server (HTTP); and a simple file system (FS). During boot time, HTTP initializes one or more paths (depending on the configuration) on which incoming TCP establishment requests are received (SYN requests). After a SYN request is received, an active path is established from HTTP to ETH. After the TCP connection is established, an HTTP request is received. The request is processed

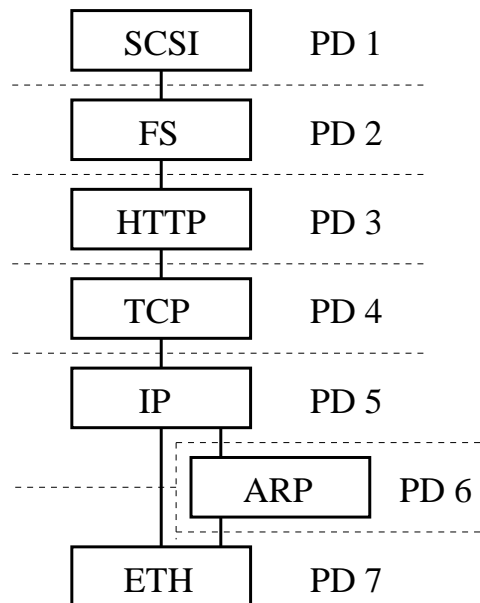


Figure 4.1: Router graph for WWW server with all routers configured in separate protection domains.

by HTTP, which then extends the path via FS to SCSI. After the path is extended, the requested document is served and path destruction is triggered, at which time the TCP connection is closed. Figure 4.1 shows a configuration which each module is in its own protection domain. This is one of the two configurations used; the other includes all modules within a single privileged protection domain.

4.1 Experimental Setup

All experiments were performed in four different setups and with the hardware described below.

4.1.1 Configuration

We tested four configurations of the web server. The first three run on Escort and implement the module graph shown in Figure 4.1. The fourth configuration runs on Linux. We denote the four configurations as follows:

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

Accounting: Like Base, 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 4.1.

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 infinite-loop thread that emulates a runaway CGI script. This experiment simulates the impact a single user that 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.

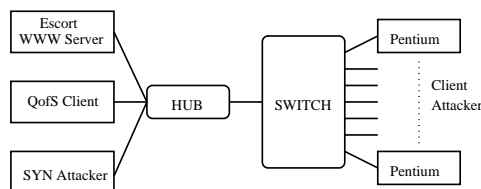


Figure 4.2: Experimental Setup

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 4.2. 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.

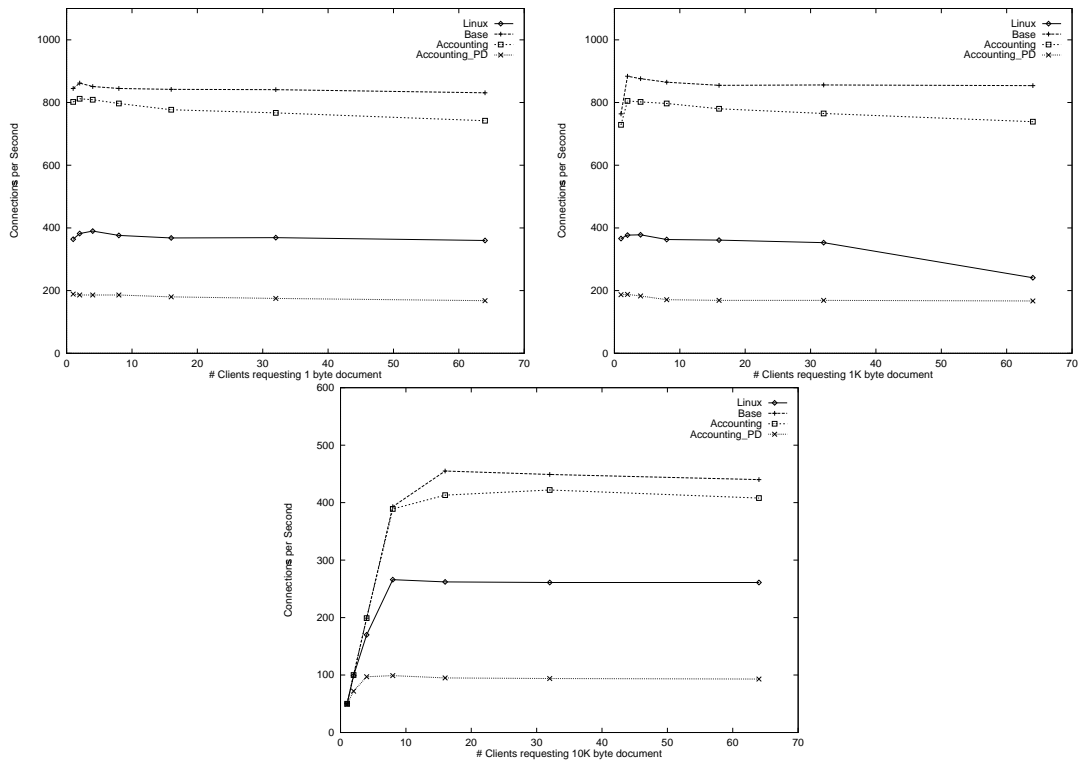


Figure 4.3: Basic performance of the different configurations in connection per second for a 1Byte document 1KByte document and 10KByte document.

4.2 Results

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

4.2.1 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 4.3 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 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 [48] have shown significant performance improvements by replacing the OSF1 PAL code with their own specialized PAL code. We expect these 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.

4.2.2 Accounting Accuracy

Table 4.1 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

¹Passive 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

Owner	Accounting	Accounting_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 4.1: Average number of cycles spent serving 100 serial requests of a one-byte web document.

by Escort; the last row (Total Accounted) corresponds to the sum of the preceding five.

We measured two configurations with the results shown in Table 4.1: 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.

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 than 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

(e.g., TCP SYN packets), while the latter correspond to open connections on which data messages are sent and received.

	Accounting	Accounting_PD	Linux
Cycle	17951	111568	11003

Table 4.2: Cycle needed to destroy non cooperative path.

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 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.

4.2.3 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.

Table 4.2 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 numbers 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.

4.3 Defending Against Attacks

This section evaluates 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.

4.3.1 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

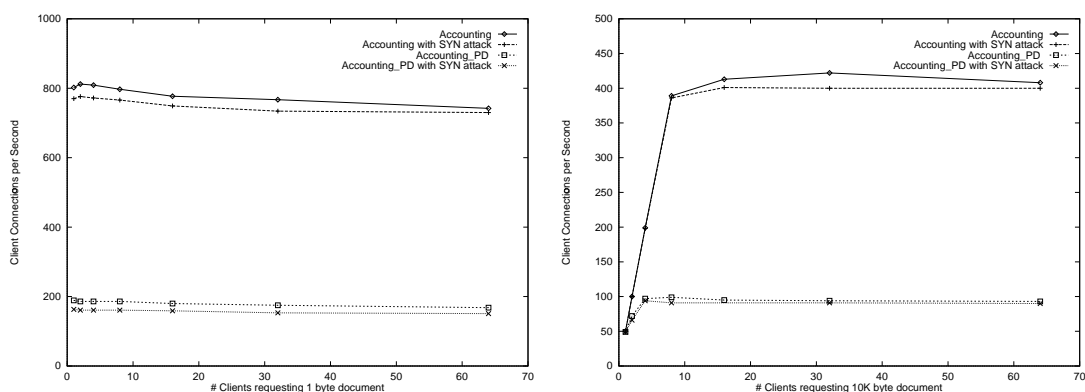


Figure 4.4: Performance for 1-Byte and 10K-Byte documents for Escort with and without protection domains, with one SYN Attacker generating 1000 SYN requests per second.

the untrusted subnet.

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 that 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 4.4 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

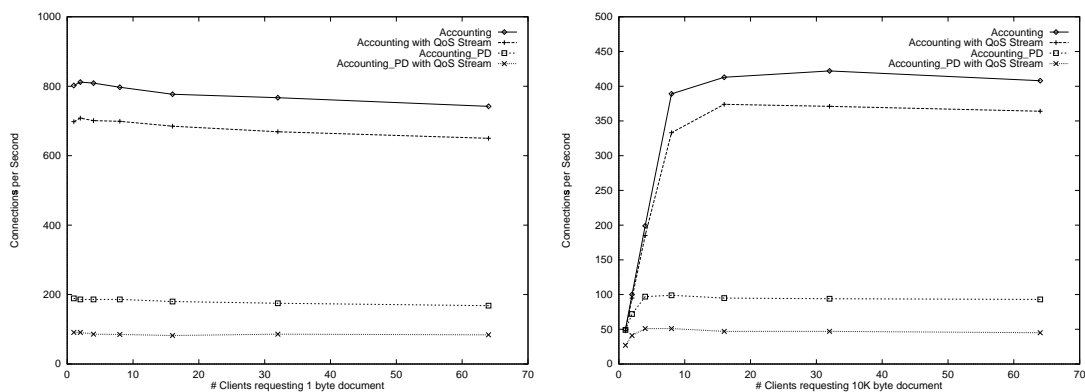


Figure 4.5: Performance of different configurations with and without a 1MByte/sec Qofs stream in connection per second.

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

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

4.3.2 QoS Stream

In the next experiment we add one 1MBps TCP stream to the base experiment described in Section 4.2.1. 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 4.5 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

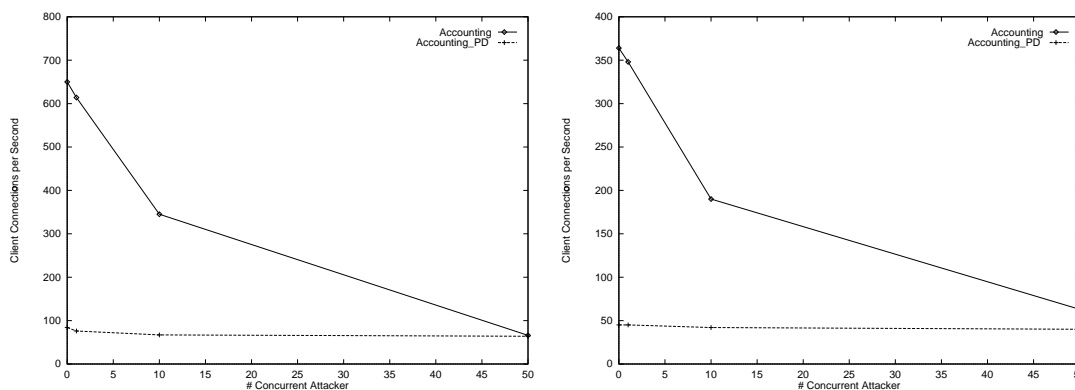


Figure 4.6: Performance for 1-Byte and 10K-Byte (top down) documents for Escort with and without protection domains, with one 1MBps QoS stream, 64 clients, and a variable number of attackers.

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.

4.3.3 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 offend-

ing 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 4.6 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.

4.3.4 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.

4.4 Related Work

Due to the increased importance of quality of service on WWW server there has been much work in accurate resource accounting for this purpose. In addition to the quality of service operating systems already discussed in Section 2.12, IOLite [53] and Resource Containers [7] are similar to the Escort mechanisms used in this chapter.

IOLite unifies buffering and caching within a system to increase the performance of a server and in particular a WWW server. Escort provides the same functionality with its IOBuffer. IOBuffer can be associated with more than one owner, a feature that can be used to cache documents. The cached documents are association with the caching module and with all paths serving those documents. IOBuffers have two additional benefits not achieved by IOLite. First, they provide accurate mandatory resource accounting via their lock mechanism. Second, IOBuffers can be automatically mapped along a path and therefore modules do not have to get involved in the decision as to which protection domain a buffer should be mapped into.

Resource Containers provide the ability to account resources over multiple pro-

tection domains in a server environment. This feature is similar to the accounting of resources towards a path crossing multiple protection domains. The advantage of Escort is again that this accounting is mandatory in Escort and does not involve the cooperation of the server process as it does with Resource Containers.

Most of the policies used to deal with denial of service attacks are ad hoc solutions fixing a single attack. Miller [43] describes a more comprehensive approach to dealing with denial of service attacks. The *Denial of service Protection Base* required by his approach can be easily implemented with the mechanisms provided by Escort. In particular, Escort provides the revocation capabilities on the granularity required by a *Denial of service Protection Base*. This granularity is not supported by more general operating systems.

CHAPTER 5

TCP FORWARDER

This chapter presents another example of an information appliance built with Escort. This example—a TCP forwarder—is used to demonstrate the performance gains possible if protection can be flexibly configured and global state can be identified by paths. The example also uses the additional invariants provided by a security policy to decide when it is safe to optimize. These additional invariants are not explicitly provided by the original Scout system. In the original Scout system the security policy is embedded into the modules.

This chapter highlights that Escort can not only be used to increase security, but it can also be used to increase performance.

5.1 TCP Forwarding

When two entities communicate indirectly through two separate TCP connections, an entity called a *proxy* mediates the communication, interposed between the two connections, and controls the flow of data between the communicating parties (Figure 5.1). The proxy runs the show—it decides if the parties can communicate, and

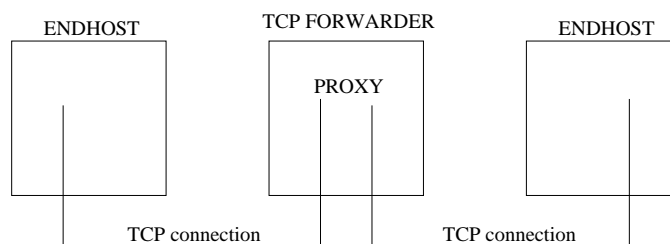


Figure 5.1: TCP forwarding via a proxy.

if so, what is communicated. A proxy can both restrict and enhance the communication. For example, a telnet proxy can restrict to which computers the outside world may connect, and perhaps which users may log in. On the other hand, a telnet proxy could also serve as a clearinghouse for a collection of servers by providing a single connection point for outside telnet accesses. The telnet requests are processed by the proxy and forwarded to the appropriate computer, shielding the outside world from the internal structure of the site.

We use the term *TCP forwarding* to refer to communication relayed over two TCP connections via a proxy. TCP forwarding is not as simple as copying bytes from one connection to the other, however. The proxy must control the communication as well as relay bytes, and therefore, a proxy has two modes: *control mode* and *forwarding mode*. In control mode, the proxy processes either out-of-band or in-band control information. Once the control functions have been completed, the proxy switches to forwarding mode to move data between the connections. After the data transfer, the proxy may switch back to control mode. For example, a

telnet proxy starts off in control mode, and processes a telnet request to determine if the connection should be allowed, based on the target machine, port, and perhaps user ID. Once the connection has been completed, the proxy switches into forwarding mode to transfer data between the two computers. Switching between these two modes of operation is the primary difficulty in developing an optimized TCP forwarding mechanism.

The processing that is done in control mode varies greatly between proxies, from very little during connection setup to continuous monitoring of the data stream while forwarding to extract control information. Proxies can be broadly classified into four categories, depending on the degree of control processing they do. Proxies that perform a minimum of control processing are those that only restrict and route based on IP addresses and port numbers. They are in control mode only during connection setup; after that they switch to forwarding mode for the duration of the connection. An FTP proxy is an example: it processes an FTP request in control mode on the control connection, sets up a data connection between the two computers, and switches to forwarding mode on the data connection until it is closed. The control connection remains in control mode to process subsequent FTP requests.

The second class of proxies performs more control processing because they authenticate the user or request and base routing decisions on either the result of the authentication or control information passed in the TCP connection. A telnet

proxy is a member of this class. Typically, a telnet proxy requests a user ID, password, and the destination of the telnet request. This information is received on the TCP connection by the proxy and is used to authenticate the user and establish a connection to the correct remote machine. At this point the proxy simply forwards data between the two connections.

The third class of proxies remains in control mode for all data transferred in one direction, but switch to forwarding mode for data transferred in the other. An example is an HTTP proxy that processes the HTTP requests (control information) sent by clients, but simply forwards the data returned by the HTTP server.

The fourth class remains in control mode and continuously monitors data passed in both directions. This might be the case for a proxy that allows users on a protected network to access HTTP servers on the Internet. The proxy could filter outgoing accesses to restrict the servers that can be reached, and filter incoming access responses to remove (untrusted) Java code.

TCP forwarding has many uses, including such diverse functions as a network firewall, an HTTP proxy, and a mobile computing system. These three examples illustrate the power of TCP forwarding, and the necessity for an efficient mechanism for doing so.

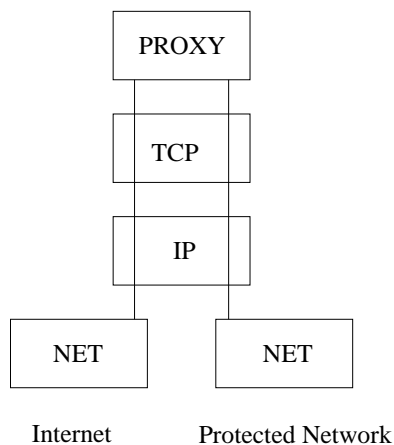


Figure 5.2: Overview of a application-level firewall. Data from one network passes through the proxy which forwards them to the other network if the desired security guarantees are not violated.

5.1.1 Firewall

A firewall provides limited connectivity between a protected network and the relative chaos of the Internet, as shown in Figure 5.2. The firewall contains different types of proxies, each handling a different type of communication between the two networks, such as telnet, FTP, etc. A typical proxy accepts connections on one network, authenticates the entity making the connection request, and forwards the data to the other network, perhaps after applying a filter. The firewall either uses its own IP address (*classical proxy*) or is completely transparent to the user (*transparent proxy*) [19]. A classical proxy must use the control information in the request to determine the connection's true destination.

5.1.2 HTTP Server Proxy

TCP forwarding can also be used to develop scalable servers such as HTTP servers. HTTP server names are embedded in the URL namespace, making it difficult to implement a single HTTP service from a collection of servers. Load-balancing across the collection is a problem; Web sites typically offer the users a selection of servers from which to choose, manipulate the DNS mappings to change dynamically the IP address associated with a site name, or use the HTTP redirection mechanism to redirect requests to unloaded servers. The first two offer coarse-grained load balancing, while the last requires two HTTP connections per URL accessed.

An HTTP server proxy that forwards TCP connections is a better solution. Clients connect to the proxy, which processes their requests and forwards them to the appropriate server. The proxy must continually monitor the data received from the clients, however, so that requests can be extracted and processed, and the connections re-forwarded as appropriate. The data returned from the servers, however, is simply forwarded to the clients.

Such an HTTP proxy might implement a variety of forwarding policies, in addition to load-balancing over a set of homogeneous servers. The proxy could forward connections to servers based on the URL requested, allowing a collection of servers, each of which serves a different collection of pages, to appear as a single site. It could also provide more complex functionalities as described by Brooks et al. [17].

5.1.3 Mobile Computing

Our final example involving proxies is from the area of mobile computing. Here proxies are used to improve the performance of mobile hosts operating across wireless links by separating TCP connections into two connections; one covering the wireless link and one covering the wired network. The performance enhancement can either be simply an improvement caused by the separation of flow control on the two different types of network, or it can rely on transformation or filtering of data, e.g., the proxy reduces the resolution on graphics sent to the mobile host over a low capacity link and removes all video clips from email. The situation is complicated by the fact that mobile hosts often use a mixture of wireless and wired networks, switching between them on the fly. When the mobile host is connected to a wired network, the proxy merely relays data in the forwarding mode, but cannot be removed from the path of communication due to the presence of the bipartite TCP connections.

Another use of proxies is to allow a mobile host to change its point of attachment to the network without jeopardizing any open connections. In this case the proxy would operate in the forwarding mode when the mobile host is connected, but would switch to control mode both when the mobile host connects and when it disconnects. This would allow the mobile host to terminate its TCP connections, move to a new location with a new IP address, and establish a new set of TCP

connections to the proxy without affecting the peer hosts on the other side of the proxy.

5.2 Connection Splicing

This section describes an optimization technique, called *connection splicing*, that improves TCP forwarding performance. It includes a discussion of the many complications that make connection splicing difficult in practice. To simplify the following discussion, we focus on the flow of data in a single direction; the same work must also be done for data going in the other direction.

5.2.1 Overview

The proxy involved in TCP forwarding operates in either control mode or forwarding mode. The basic idea of connection splicing is to detect when a proxy makes a transition from control mode to forwarding mode, and then splice the two TCP connections together into a single forwarding path through the system. The resulting spliced connection replaces the processing steps (and associated state) required by two TCP connections with a single reduced processing step (and associated state).

Figure 5.3 schematically depicts the optimization. The standard (unoptimized) forwarder on the left requires TCP segments to traverse TCP twice, with each instance of TCP maintaining the full state of the connection. In this case, the Proxy simply passes segments from one connection to the other when it is in forwarding

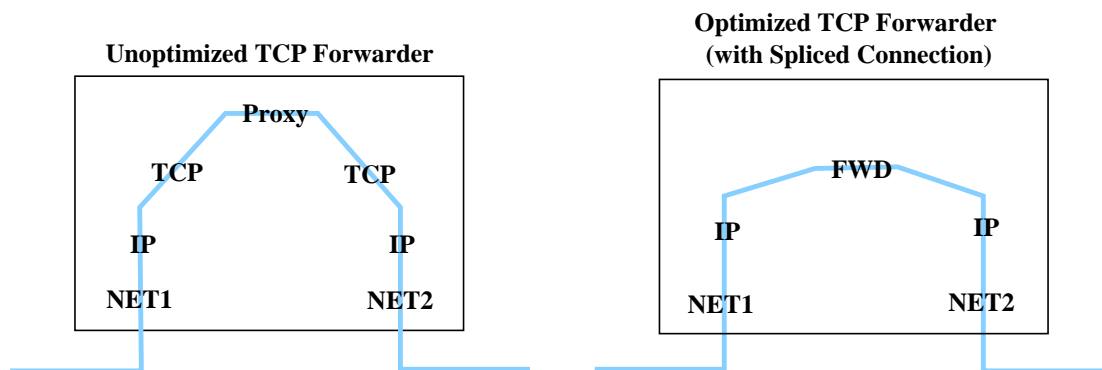


Figure 5.3: Optimizing two TCP connections into a single spliced connection.

mode. The optimized forwarder on the right replaces the Proxy and two TCP processing steps with a single FWD processing step. FWD maintains just enough state to forward TCP segments successfully from one network to another. The state FWD needs to maintain is described later in this section.

A single proxy might require both configurations, however. The configuration on the left *must* exist when the proxy is in control mode; the proxy must be in the loop because it needs to inspect the data flowing between the two TCP connections. The configuration on the right *may* exist while the proxy is in forwarding mode. Forwarding can also happen in the left configuration, but performance suffers. With this perspective in mind, there are three cases to consider: how the optimized configuration on the right works in the steady state (Section 5.2.2), how the system makes the transition from the left-hand configuration to the right-hand configuration (Section 5.2.3), and how the system makes the transition from the

right-hand configuration back to the left-hand configuration (Section 5.2.4).

Typically TCP forwarding starts in the unoptimized configuration, makes a transition to the optimized configuration when the proxy shifts from control to forwarding mode, and sometimes reverts back to the unoptimized configuration should TCP forwarding go back to control mode. Note that while the connection splicing optimization is in effect, the two independent TCP connections shown on the left no longer exist on the forwarder.

5.2.2 Forwarding

The primary task of the FWD processing step shown in Figure 5.3 is to change the header of incoming TCP segments to account for the differences in the two original TCP connections. Since the two TCP connections were established independently, their respective port numbers and sequence numbers are probably different. The IP addresses associated with the connections might also differ, resulting in changes that affect the IP pseudo header as well.

Figure 5.4 depicts the TCP segment header; the boldface fields are those that FWD modifies. The following outlines the transformations FWD applies to each segment it forwards from one connection (A) to another connection (B). For now, we ignore the problem of moving a TCP forwarder into the optimized state, and focus instead on the work involved in forwarding segments once FWD is in place. Also, we assume that the two TCP connections were established independently. If

SrcPort		DstPort	
SeqNum			
Ack			
Hlen	Resv	Flags	AdvWin
Cksum		UrgPtr	
Options		Padding	
Data			

Figure 5.4: TCP segment header with fields modified by FWD in bold.

their establishment was in fact interleaved—so that one connection knew what port and sequence numbers were being used by the other connection—then additional optimizations are possible, as described in section 5.2.6.

Port numbers: If the TCP forwarder operates as a classical proxy, the port numbers of both TCP connections will probably differ. Therefore, the source and destination port numbers of segments arriving on A have to be changed to the port numbers of connection B. If the TCP forwarder is a transparent proxy, this change is unnecessary because the proxy uses the same port numbers as the server.

Sequence Number: The sequence number used by segments received by FWD on A are probably different from those used for segments sent by FWD on B. This is because TCP initializes sequence numbers randomly for each independent

connection. The sequence number for an outgoing segment is computed by adding a fixed offset to the sequence number in the incoming segment.

Acknowledgment Number: The acknowledgment number acknowledges the sequence numbers forwarded *in the other direction*. Thus the acknowledgment number in an outgoing segment is computed by subtracting from the sequence number in the incoming segment the sequence number offset for segments flowing in the other direction.

Checksum: Modifying the other fields requires adjusting the TCP checksum. A constant checksum patch representing the “delta” in the checksum is used to do this efficiently. If the FWD acts as a classical proxy, the changes to the IP address fields in the IP pseudo-header are also reflected in this checksum patch.

The following pseudo code describes the changes to a segment transferred from A to B. All header fields marked **Input** represent the segment header values in the received segment. The header fields marked **Output** represent the segment header values used in the outgoing segment. Bold variables indicate constants that are part of FWD’s state. Subscripts indicate the direction for which these constants are used; e.g., **SeqNumOffset**_{A→B} represents the sequence number offset used to patch sequence numbers on segments received from A and sent to B.

Output.DstPort = RemotePort_B

$$\text{Output.SrcPort} = \text{LocalPort}_B$$

$$\text{Output.SeqNum} = \text{Input.SeqNum} + \text{SeqNumOffset}_{A \rightarrow B}$$

$$\text{Output.Ack} = \text{Input.Ack} - \text{SeqNumOffset}_{B \rightarrow A}$$

$$\text{Output.Cksum} = \text{Input.Cksum} + \text{CksumPatch}_{A \rightarrow B}$$

The checksum calculation shown in the pseudo code is more complicated than simple addition. To account for overflows or underflows during sequence number and acknowledgment number calculations it is necessary to add or subtract one from the checksum. This is because the checksum is the one's complement of the one's complement sum of the segment.

Splicing two TCP connections significantly changes the behavior of the forwarding proxy. In the unspliced case, segments sent to the proxy are acknowledged when they are processed by the incoming TCP stack. The proxy then takes responsibility for the data, resending them as necessary to ensure they reach their destination. Data are buffered in the outgoing TCP stack until they are acknowledged by the destination. When the two connections are spliced, the segments no longer traverse the two TCP protocol stacks. The proxy does not acknowledge data coming from the sender, nor does it resend data to the destination. Data and acknowledgements are forwarded without processing, requiring the two endpoints to handle retransmission and reordering.

Forwarding segments requires internal state in FWD. Some of this state is re-

quired to modify the header fields, such as the port numbers, sequence offsets, and checksum patch. FWD must also detect, reset or termination of the TCP connection. To do so it parses the flags in the header and keeps a simplified TCP state machine. FWD also keeps one timer that is used to timeout the connections; all other TCP timers are not used.

If it is possible that the optimized forwarder will revert back to a standard forwarder, FWD also needs to store the current advertised window, the highest sequence number sent, and the highest ACK seen that will fit in the advertised window. The process of converting a spliced connection back to an unoptimized TCP forwarder is discussed in Section 5.2.4.

5.2.3 Splicing

The header modifications required to forward a segment are relatively straightforward; the more difficult task is transitioning from the unspliced state to the spliced state. The difficulty is caused by acknowledged data buffered in the forwarder. This data might be buffered by the receiving TCP's receive buffer, within the proxy itself, and in the sending TCP's send buffer (Figure 5.5). The acknowledged data must be reliably forwarded to its destination. This data also influences the offsets calculations required by the spliced connection.

First, all data acknowledged by either connection on the unoptimized TCP forwarder must be reliably delivered to their destination. The important point

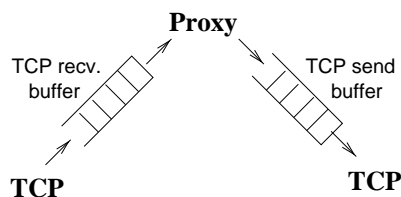


Figure 5.5: TCP buffers potentially containing acknowledged data.

is that these data have already been acknowledged by the forwarder, so it cannot depend on the source host to take responsibility for possibly retransmitting the data in the future. Thus, the forwarder must continue to run TCP—the only way it can reliably deliver data—until the currently buffered (acknowledged) data are reliably delivered to the destination. During the time the data are being drained, however, new segments may arrive. The forwarder obviously cannot let TCP acknowledge the new data, because doing so will just give it even more data to deliver reliably, and it is impractical to wait until the two connections go idle before completing the splice. Fortunately, there are two ways to handle newly arriving segments during this transition period.

The first option is to delay the activation of the spliced connection until after the buffers have been drained. During this time, a limited amount of new segments that arrives is delivered to TCP (for acknowledgment processing only) and held in a separate buffer for FWD; they are not acknowledged and they are not placed in the incoming connection's receiver buffer. If the buffer overflows while TCP is still processing acknowledgments, the segments are dropped after the acknowledgements

are processed. When the transition is complete, these buffered data are processed by FWD as though they had just arrived. Again, the TCP protocols are suspended as soon as all buffers are drained. This solution may drop data if the FWD buffers overflow while the TCP buffers are being drained. If the amount of data buffered in TCP is small, then the FWD buffers are unlikely to overflow.

The second option is to allow FWD to begin forwarding data concurrently with draining the buffers. Should any new data arrive during the transition, it is important that the original TCP protocols not acknowledge this new data; they are only allowed to process the acknowledgments contained in those segments so that the buffers drain. In other words, all newly arriving segments are delivered to both the original TCP protocol (for acknowledgment processing only) and to FWD (for forwarding to the receiver). This solution does not drop data, but may cause data to be delivered out-of-order. Segments processed by FWD may be delivered before segments traversing the original TCP connections. This will not affect correctness because the destination will reorder the segments.

During the time that FWD operates concurrently with the draining process, both forwarded segments and drained segments will arrive at the destination. This means it is possible that the TCP draining buffers on the forwarder might receive an acknowledgement for a sequence number that is larger than maximum sequence number in its send buffer. This acknowledgement is really meant for the source host, but since the forwarder is still processing acknowledgements in an attempt to

drain its buffers, it will receive this acknowledgement too. To allow for this possibility, TCP running on the forwarder during the transition must be able to accept acknowledgments up to one full window size larger than the maximum sequence number in its send buffer.

The second thing that must be done during splicing is to initialize the internal state of FWD. This requires computing the sequence number offsets (**SeqNumOffset**_{A→B} and **SeqNumOffset**_{B→A}) and the checksum patches (**CksumPatch**_{A→B} and **CksumPatch**_{B→A}) used by FWD. The sequence number offsets can be calculated as soon as all acknowledged data have been drained. If acknowledged data still exist in one of the forwarder's buffers, then it is necessary to subtract the length of this buffered data from the corresponding sequence number offset. This is because the sender of a segment that is directly forwarded assumes that the buffered data was delivered, and therefore, the sequence number of the source's TCP protocol has already been increased. The checksum patch can be calculated as soon as the other offsets are known since the changes in port number and IP address are already known.

5.2.4 Unsplicing

When the forwarding proxy switches from forwarding mode to control mode the connections must be unspliced. There are two complications; the first is to be able to detect that it is necessary to switch back to the unoptimized state; i.e.,

that the forwarder has moved from forwarding mode to control mode. The second is to correctly make the transition. The solutions to these two complications are intertwined.

It may be difficult to decide when the proxy should switch back to control mode. If the control information is sent over the spliced connection the proxy has to monitor the data being forwarded to detect the control information. This is difficult because the FWD protocol does not reorder the segments it receives, nor does it buffer segments. The proxy has to find the control information by looking at out-of-order segments, one at a time. This makes it unlikely that the proxy will be able to filter the data to find control information. However, it seems useful to trigger a switch back in unoptimized mode as soon as data are transmitted in a certain direction. An HTTP 1.1 proxy, for example, might allow the forwarding of HTTP replies but want to examine all, possibly pipelined, HTTP 1.1 requests.

Dealing with acknowledgements makes it difficult to unsplice a connection. When the forwarder reverts to two TCP connections and proxy it must take over handling acknowledgements. If there are no unacknowledged segments outstanding on the spliced connection, the transition back to unspliced is easy. The reconstructed TCP connections are initialized with the sequence numbers, acknowledgment numbers, and advertised window sizes stored as FWD state. The state-machine is progressed to the current state, the timers and the send window are initialized with their initial values, and a slow start is initiated. This makes it pos-

sible to stop instantly forwarding new segments, but will require the retransmission of lost segments.

If there are outstanding unacknowledged segments, however, the forwarder must either wait for them all to be acknowledged—dropping data if necessary and then switch as described above—or else continuously monitor the segment stream until it has copies of all unacknowledged segments. It then uses this information to initialize the TCP connections and buffers. This solution does not drop any segments, but up to two full window sizes might have to be buffered before the switch over can be completed.

5.2.5 Flow Control

During unoptimized operation flow control is handled by the two independent TCP protocols on the forwarder, and the TCP protocol on the end hosts. During optimized operation, flow control is handled by the end hosts only; the forwarder merely drops segments, just as a congested router drops IP datagrams.

There is a complication during the transition to a spliced connection, however. Shortly after the switch to the spliced connection, the advertised window might be either too big or too small. For example, the window advertised by the destination host to the forwarder might be smaller than the window advertised by the forwarder to the source host. In this case, the source host will suddenly see a smaller advertised window after the connection is spliced, possibly triggering unnecessary

retransmissions. Similarly, the send window of a host might also be bigger than the advertised window of its new peer. If so, it is likely that data will be transmitted unnecessarily. Note that RFC1122 strongly recommends that the advertised window not be reduced to eliminate this unnecessary data transmission. To minimize this problem, the TCP forwarder can restrict the size of the window it advertises to the source host to the window size advertised by the destination host, minus the size of the buffered data.

More subtly, the send window of both end hosts might not represent the bandwidth of the link. If the send window is too big, the host will send too many segments and generate unnecessary congestion. However, this can only happen if traffic is extremely bursty. Otherwise, the limited buffer space available on the TCP forwarder should synchronize the send window sizes of both TCP connections.

5.2.6 Additional Optimizations

The connection splicing optimization can be applied not only at the TCP level, but also to unfragmented IP datagrams. In addition, the optimization can be applied to the first IP fragment of an IP datagram if we allow the unfiltered forwarding of all remaining fragments, and if the MTU is large enough so that the first fragment will contain the TCP segment header.

In these two cases, the forwarder can process the IP datagrams similarly to an IP router, with the additional TCP segment header manipulation described in

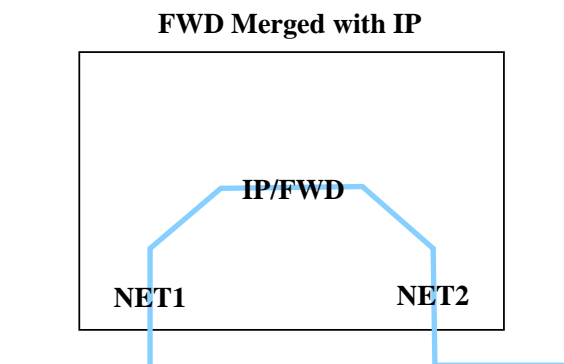


Figure 5.6: Further optimizing the spliced connection when there is no fragmentation.

the previous section. Figure 5.6 illustrates this scenario, which we denote as a combined IP/FWD processing step. The important consequence of being able to forward TCP segments at the IP level is that it makes it possible to apply any of the optimizations one might apply to an IP router. For example, if the forwarder is connected to two Ethernets, we can modify and forward the Ethernet packets directly.

Finally, under certain circumstances it is possible that the TCP forwarder can tolerate the unfiltered forwarding of all IP fragments, that is, FWD implements the identity transformation. This would happen if the unoptimized forwarder is configured as a gateway intercepting TCP connections and was careful in selecting port numbers and the starting sequence numbers when the original pair of TCP connections were opened. This being the case, FWD can be omitted and the TCP forwarder operates just like an IP router.

5.2.7 Other Issues

Additional IP-level filtering can also be done on a spliced connection. For example, to avoid attacks against the TCP stack the forwarder should limit the sequence and acknowledgment numbers of the current spliced connection to meaningful values, and drop all segments that have the SYN flag set. This is possible since all connections are established first with the proxy, not with the final destination.

As many routers already do, the forwarder can also perform network address translation (NAT). It is possible to perform NAT on spliced and unspliced connections. If, however, IP addresses are passed within the data stream, as in FTP for example, the connection has to either be spliced after the IP addresses have been altered by the proxy, or additional IP-level filters have to be added.

A final issue is TCP options. Our prototype currently supports only the MSS option, which is negotiated with the forwarder. If the MSS of both segments do not match, the ICMP Destination Unreachable message will be used to adjust the MSS after the connection is spliced. The other TCP options can be handled in much the same way as the prototype patches sequence numbers; some (e.g., SACK) don't require additional state, but others (e.g., TIMESTAMP) do.

5.3 Connection Splicing in Escort

Connection splicing can be implemented in any operating system; Section 5.6 discusses an implementation in Unix. This section describes an implementation in Escort. Escort provides three important benefits compared to general purpose OS.

First, the path abstraction combined with the application level security policy contained in a filter can be used to decide when it is safe to perform the optimization. On a firewall, for example, a filter used to restrict the internal network access to certain users has the knowledge when to optimize. The path provides the information about what to optimize.

Second, the ability to arbitrarily combine modules in protection domains and to multiply instantiate modules, allows the combination of the protocol modules that are part of the optimization to be within one protection domain. In the firewall example, the proxy, the filter, and instances of IP and TCP can all be combined in one protection domain. This allows secure access to module specific path state needed during the optimization.

The third benefit of Escort is that the global state associated with a path can be used to account accurately for data in transit allowing the optimization to be performed without delay.

The path abstraction lends itself to a natural implementation of TCP forwarding. Figure 5.7 schematically depicts a naive implementation of TCP forwarding

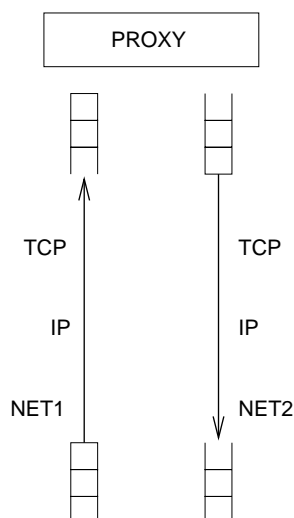


Figure 5.7: TCP forwarding implemented in two Escort paths.

(the unoptimized case) in Escort. It consists of two paths: one connecting the first network interface to the proxy and another connecting the proxy to a second network interface. In this figure, the path has a source and a sink queue, and is labeled with the sequence of software modules that define how the path “transforms” the data it carries.¹ To a first approximation, the configuration of Escort shown in Figure 5.7 represents the implementation one would expect in a traditional OS.

The two-path configuration shown in Figure 5.7 has suboptimal performance because it requires a handoff of each incoming segment from the first path to the proxy, and then from the proxy to the second path. In Escort, the entire device-to-device data flow can be encapsulated in a single path (Figure 5.8). This is the

¹As in Section 5.2, we focus on data flowing in one direction. In reality, Escort paths, like TCP, support bi-directional data flows.

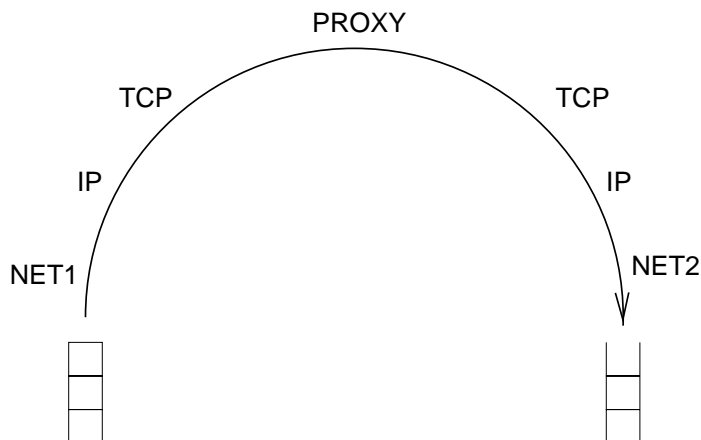


Figure 5.8: TCP forwarding implemented in a single Escort path.

implementation of choice for the unoptimized TCP forwarding case in Escort.

Connection splicing is then implemented within the same framework. Figure 5.9 illustrates the two optimized configurations discussed in Section 5.2: the path on the left corresponds to the right-hand case from Figure 5.3, while the path on the right corresponds to the case shown in Figure 5.6. Note that the right-hand path looks very much like an IP router would in Escort.

Looking at the implementation in a bit more detail, each path consists of a linked list of *stages*, where each module that the path traverses contributes a stage to the path during path creation. Abstractly, each stage contains the path-specific code and state for the corresponding module; e.g., the TCP control block is contained in the TCP stage of the paths shown in Figures 5.7 and 5.8. When the proxy in an unoptimized TCP forwarding path detects a transition to forwarding mode, it does five things:

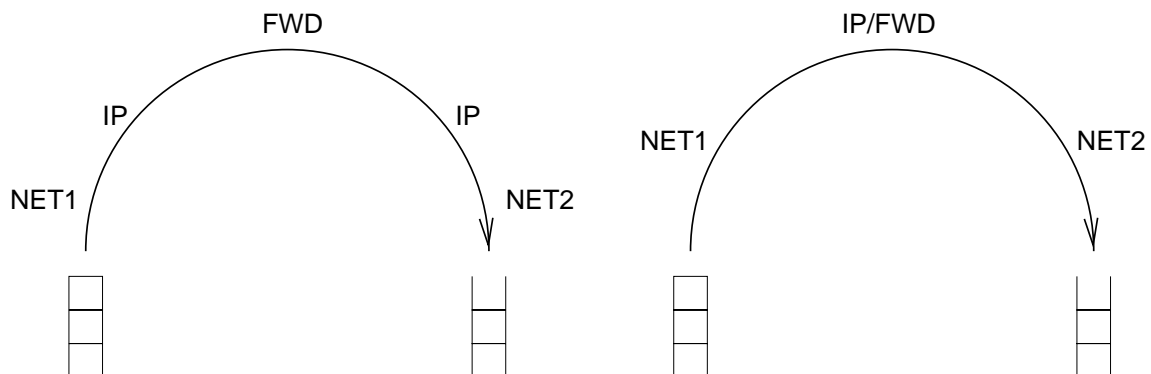


Figure 5.9: Connection spliced paths in Escort.

- Stops processing incoming segments and allows segments to accumulate in the path's input queue.
- Unlinks the two TCP stages and the proxy stage from the path and replaces them with a preliminary FWD stage.
- Continues processing incoming segments and data in the TCP buffers until the TCP buffers are drained.
- Unlinks the preliminary FWD stage and replaces it with the final FWD stage.
- Continues processing incoming segments.

The difference between the preliminary FWD stage and the final FWD stage is that the former forwards the segments and reliably drains the TCP buffers, whereas the latter only adjusts segment header fields.

One subtlety is that there are seldom any segments queued *within* the path that need to be drained; Escort is non-preemptable, so in practice once a segment is removed from the input queue it is processed completely and deposited in the output queue. The only time a segment gets buffered in the middle of a path is when the scheduler selects the path for execution, the segment makes it as far as the outgoing TCP stage, but the advertised window on the second connection is closed. It would be possible to take the outgoing window into account when making the scheduling decision—i.e., not schedule a TCP forwarding path until it was certain that the segment could make it all the way to the output queue—but the consequence is that the segment would remain in the input queue, and thus, not acknowledged on the incoming TCP connection.

5.4 Experimental Setup

This section provides measurements of the effect of connection splicing on TCP forwarding. To make the study concrete—and to give us an existing system against which we can compare our approach—we focus on a simple firewall configuration within one protection domain. The proxy in the firewall does not perform any processing in control mode; it is always in forwarding mode.

We measured the following configurations of Escort:

2-Path: This is a full blown TCP forwarder, as depicted in Figure 5.7. This

TCP forwarder uses two separate TCP paths meeting at the proxy—one to

each network device. As going from one path to another often will require a context switch, this configuration is the closest to the structure of a firewall in a regular operating system like Unix or NT.

1-Path: This is the configuration shown in Figure 5.8. This case is similar to the 2-path configuration, except the two network devices are connected by a single path. This is the natural way of expressing a TCP forwarder in Escort. Note that this configuration still involves two separate TCP connections but a single Escort path.

FWD: This is an optimized version of 1-Path. Here the TCP connections have been spliced into a single connection, and the forwarder is reduced to updating the TCP headers. This configuration still supports reassembly of IP packets. This case corresponds to the left-hand configuration in Figure 5.9.

IP/FWD: This is a further optimized version of **FWD**. The network level packets are modified directly and forwarded. As a consequence, this configuration does not support reassembly of IP packets. This is the case corresponds to the right-hand configuration in Figure 5.9.

IP Router: This is an IP router. It also modifies network packets directly in the same way as **IP/FWD**, but it does not update TCP headers. It is included to show the lowest possible overhead for an intermediate host in Escort.

To compare the Escort performance with a more general-purpose operating system—so as to demonstrate that the Escort numbers are in-line with a more conventional system—we also measured the performance of a firewall and IP routing on Linux. We compiled the Linux kernel to optimize for IP routing. We consider three configurations:

TIS Firewall: The TIS firewall toolkit offers full filter functionality[59]. We have configured it to use a null filter (plug-gw).

Filtering IP Router: The in-kernel Linux IP forwarding has support for filtering on IP addresses, protocol numbers and port numbers. This is the closest thing in Linux to the IP/FWD case in Escort, except Linux neither permits starting with a proxy and later dynamically switching to the spliced connection, nor updating TCP headers.

IP Router: This is the basic in-kernel Linux IP forwarding with no filtering. This shows the lowest possible overhead of the Linux configuration.

Note that while there is a myth that Linux networking performance is not very good, we have not found this to be the case with recent releases. For example, the Linux IP forwarding numbers given below are better than comparable numbers reported on BSD Unix [41]. While researchers have reported dramatic improvements in BSD forwarding performance as a result of aggressive optimizations [84], there is

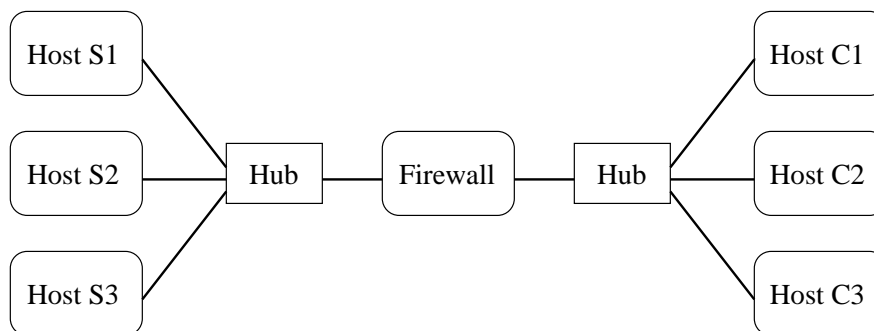


Figure 5.10: Test Setup

no reason to believe the same optimizations could not be applied to Linux. In any case, we use the Linux numbers to calibrate the baseline case; the important results are in the incremental costs of each mechanism layered on top of this baseline.

Finally, we measure the performance of two machines connected back-to-back to evaluate the overhead of injecting a third host on the network path.

All hosts used in our experiment are 200 MHz PentiumPro workstations with 256KB cache, 128MB ram, and Digital Fast EtherWORKS PCI 10/100 (DE500) 32-bit PCI 10/100 Mb/s adapters. The Linux version used was 2.0.30. The physical configuration of our test setup is shown in Figure 5.10. To saturate the network during throughput tests, we connected three hosts on each side of the firewall. All tests are performed between a server (hosts S1 to S3) and a client (hosts C1 to C3). In the back-to-back case, the setup was modified by connecting the two hubs to each other. All servers and clients were running Escort, as the lower complexity of Escort resulted in less variation in the measurements than Linux.

Latency		1B TCP Segment	1460B TCP segment
Back-to-Back		77.9 μ secs	243.2 μ secs
Network Interfaces	Transmission	5.2 μ secs	121.1 μ secs
	Other	9.8 μ secs	11.7 μ secs
Total		92.9 μ secs	376.0 μ secs

Table 5.1: Non-processing related overhead removed from latency measurements.

5.5 Results

For all configurations, we measure the per-packet processing time for small (1-byte) and large (1460-byte) segments, and the aggregate throughput achieved with multiple connections. For Escort, we also measure the time it takes to switch from unoptimized to optimized.

5.5.1 Processing Overhead

To measure the per-packet processing overhead, we measured the packet round-trip times for 10,000 packets, and subtracted the back-to-back latency and network interface latency. The subtracted components are summarized in Table 5.1. The network interface latency was obtained by measuring the processing time of a packet in the IP router configuration—that is, the time from when the packet is removed from the network interface by the interrupt handler to the time it inserted into the transmit queue of the other network interface—and subtracting this time from the total latency added by the router.

Table 5.2 summarizes the processing of a single packet in the firewalls and

routers for both Escort and Linux. The 1-byte numbers reveal that connection splicing achieves a considerable speedup. Most notably, the IP/FWD case is almost a factor of three faster than application-level forwarding. In terms of packets-per-second that can be processed by the firewall, this is an increase from 14,600 to 41,600. For large packets, the speedup is even greater—a factor of four. Eliminating the extra message copy and the checksum calculation required when transferring the message from one TCP connection to another accounts for the speedup.

Also note that in both the small and large message cases, the performance of the spliced connection is very close to the performance of the IP router configuration; the TCP header transformations amount to an extra 1.6 μ secs of processing. This suggests that any improvement made to IP router performance (such as those reported by Wroclawski [84]) will be propagated to TCP forwarding. For example, the use of polling instead of interrupts and employing a highly optimized classification algorithm have the potential to improve IP routing performance (and hence TCP forwarding) by close to a factor of two. On a similar note, it would be interesting to wed connection splicing with hardware supported tag switching.

Comparing the Escort and the Linux numbers, we see that the 2-path case in Escort is slightly faster than the TIS firewall on Linux. IP router performance is approximately the same for the two systems. This indicates that other types of operating systems would also benefit from connection splicing. In a Linux implementation, the IP/FWD should perform close to that of the filtering IP forwarding—the

Configuration		1B TCP segments		1460B TCP segments	
		Processing time (μ secs)	Speedup	Processing time (μ secs)	Speedup
Escort	2-path	68.5	–	101.1	–
	1-path	66.1	1.04	98.6	1.03
	FWD	39.0	1.76	39.5	2.56
	IP/FWD	24.0	2.85	24.0	4.21
	IP router	22.4	3.06	22.4	4.51
Linux	TIS Firewall	83.9	–	113.0	–
	Filtering IP router	27.5	3.05	29.0	3.90
	IP router	25.5	3.29	25.4	4.45

Table 5.2: Firewall and router processing per TCP segment.

updating of the TCP and IP headers would make it slightly slower. Keep in mind, however, that simple IP filtering does not permit a proxy that can sometimes operate in control mode.

5.5.2 Aggregate Throughput

The sustained throughput of a TCP forwarder is also a measure of its performance. The expectation is that the improved processing overhead of the optimized forwarders should allow them to support more concurrent TCP connections.

We measured the aggregate throughput of one, two, and three concurrent TCP connections over each configuration. Each TCP connection is between a client and a server from our test setup, such that each host supports only one TCP connection. The data unit transmitted by the client process was 1460 bytes. The aggregate throughput was obtained by adding the average throughput over the

last 10 seconds of the individual connections. This was done when the throughput had reached a stable state. Not surprisingly, these measurements turned out to be bounded by the bandwidth of the 100 Mbit Ethernet, i.e., regardless of the number of TCP connections the aggregate throughput was close to 10 MB/s.

The more interesting question is how TCP forwarding behaves in the limit, that is, what bandwidth it can sustain. We can derive these numbers from the per-packet processing times presented in the previous section. For the 2-path and the IP/FWD configurations, we calculated the maximum throughput for different TCP acknowledgement patterns—either an acknowledgement is sent for every third, second, or single segment. For example, if an acknowledgement is sent for every third segment, the processing requirements for the data in the three segments would be three times the processing of a 1460-byte segment, plus the processing of a single empty segment. In our case, we have approximated the empty segment with a 1-byte segment.

The results are shown in Table 5.3. The 2-path TCP forwarder in our measurements is operating at almost maximum bandwidth of a 100 Mbit Ethernet, whereas the IP/FWD configuration is capable of supporting up to four times the bandwidth, corresponding to two full duplex OC-3 ATM connections.

Configuration		Message to acknowledgement ratio		
		3:1	2:1	1:1
Escort	2-path	11.7	10.8	8.6
	IP/FWD	45.6	40.6	30.4

Table 5.3: Estimated maximum throughput of firewall in MB/s

5.5.3 Cost of Splicing

The next question is how long it takes to splice (or unsplice) a forwarding path. As we have not yet implemented unsplicing, we focus on the cost of splicing two TCP connections. Our analysis has two parts. First, we establish the base processing overhead of splicing two TCP connections together. Second, we examine the end-to-end behavior of a TCP connection sending at maximum speed when the splicing is done.

To get the basic cost, we measured the time taken to splice two idle TCP connections. In this case, the measurements are free of any processing that might occur due to the draining of TCP buffers. In the test we continuously opened a TCP connection, waited 15 seconds and closed it again. The null proxy in the firewall optimizes the path 10 seconds after it is established. The numbers are the average time over 1000 such optimizations. Optimizing from TCP forwarding to FWD takes 25 μ secs on average. Adding the IP/FWD forwarding takes 94 μ secs on average. The higher cost of switching to IP/FWD is due to the fact that Escort requires a new path creation, whereas the FWD optimization is applied to the same

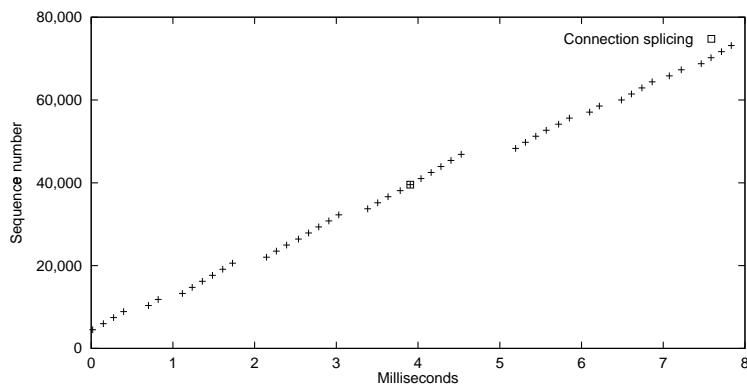


Figure 5.11: TCP Sequence Number Trace showing the effects of splicing

path by doing code substitution.

As we concurrently forward new TCP segments and empty the buffers of the old segments, the cost of performing the optimization should be small even during high load. The more important question is whether or not the switch affects TCP's flow or congestion control algorithms. To see the effects of the switchover on a busy TCP connection, we performed the optimization 15 seconds into a throughput test. By tracing the sequence numbers of segments received at the server, we were unable to see any negative effects (Figure 5.11).

As moving to FWD forwarding reduces the processing by an average of 27.1 μ secs for small messages and 59.1 μ secs for large messages, it is always a good idea to switch to the FWD optimization, independent of how much data will flow over the spliced connection. Moving from FWD to IP/FWD reduces the processing by an extra 15 μ secs per packet, and thus, it will take six subsequent packets to make

this optimization worthwhile.

5.5.4 Buffer Requirements

Buffer size is an issue for large-scale TCP forwarders. First, just having enough memory to accommodate thousands of TCP connections can be a problem, as each connection can easily require up to 256KB of buffering—two send buffers and two receive buffers of 64KB each. This translates to buffer requirements of 256MB per 1,000 TCP connections. As the use of persistent TCP connections is becoming more widespread, thousands of connections per TCP forwarder is not uncommon. Splicing TCP connections together reduces the memory requirements of a TCP forwarder, since the forwarder is operating like an IP router and does not buffer segments.

Dynamic buffer allocation is another solution to this problem, but it requires processing to determine how much buffer to provide each connection. In this scenario, the TCP connections used for large data transfers are the most important. These TCP connections are the mostly likely candidates for splicing, thereby removing the buffer requirements all together. In other words, splicing can also make the administration of a TCP forwarder easier.

5.6 Related Work

The idea of TCP splicing was developed independently by researchers at IBM [41], and its utility shown in supporting mobility [40]. Their work was done in the context of the Unix kernel, and so involves extensions to the socket interface. A more fundamental difference, however, is that the IBM approach is more restrictive than the one described in this chapter. First, it supports splicing only at connection-setup time. Second, it allows only certain interactions among the client, proxy, and server. In particular:

- Before the connection is spliced, only the client and the proxy can exchange data; the server is not allowed to send or receive data before the splice is complete.
- The proxy waits for an ACK of all data it has sent the client before engaging the splice.
- Once the splice is in place, the client is allowed to send data to the server. It is the arrival of these data at the server that notify the server that the splice is complete; the server is not allowed to send until this time.

This interaction is enforced by the SOCKS library package that must be linked with both the client and the proxy [38].

In contrast, our approach allows the splicing optimization to be transparently engaged at any point in the lifetime of the two TCP connections, including after the client and server have exchanged data. This is accomplished by having the proxy simultaneously process buffered data and forward newly arriving data, as described in Section 5.2.3. The important consequence of this difference is that our approach allows the proxy to filter arbitrarily the data passed between the client and server before it initiates the splice and removes itself from the path. This means, for example, that a proxy is able to parse a URL in an HTTP stream. The IBM approach does not support such general filtering.

More broadly, TCP forwarders are used to separate the TCP connection on a wireless link from that of a wired network [6]. This increases performance as the characteristics of the two types of networks are very different. As a mobile host moves around, it might sometimes connect directly to a wired network, in which case the TCP forwarder becomes superfluous and can be removed. This is done in the TACO system [33], where mobile hosts can—depending on what is required from their current type of network attachment—switch between having a TCP forwarder and not, without destroying their TCP connections. The system differs from the one presented in this paper in two ways; it does not support filtering, and it uses interleaved connection establishment. This allows the TCP forwarder to be removed completely from the network path in the optimized case as no translation is necessary, but it at the same time limits the applicability of the solution. The

lack of filtering makes it unsuitable for more advanced proxies such as firewalls.

Another research topic related to this paper is that of efficiently classifying packets [5, 27]. Of particular note are new algorithms to do fast routing table lookups based on variable length IP address prefixes [16, 82]. It is easy to imagine such techniques being extended to support fast IP filtering. Such an advance would be complementary to connection splicing, which can also exploit improved algorithms to determine to which path a particular packet belongs. Connection splicing is more general than IP filtering, however, since the proxy permits complex control operations.

CHAPTER 6

CONCLUSION

This dissertation defines a set of mechanisms that can be used to build secure, high performance Information Appliances. The Escort architecture provides these mechanisms in addition to a configuration interface. The security analysis presented in Chapter 3 allows the designer of an Information Appliance to more thoroughly understand the issues involved, and can be used as guideline to building secure Information Appliances.

6.1 Contribution

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). Escort supports these features while maintaining high performance. To achieve these goals, several novel ideas in the fields of thread migration, buffer management, resource accounting, resource revocation, and single address space operating system were introduced.

To validate the contributions, we have used Escort to build a secure web server and a TCP forwarder. Experiments with the web 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.

We also 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 work, we believe Escort provides the necessary mechanisms for implementing such policies.

A performance study shows that an optimized TCP forwarder requires between one-half and one-quarter of the processing requirements of an unoptimized forwarder. In addition to the novel idea of connection splicing, this experiment shows how paths can be used to increase performance without compromising security.

6.2 Future Work

Escort has two major limitations which suggested directions for future work. The first is the complexity of the configuration process. The translation from an in-

formal functional specification and security policy to a set of configuration files is not straightforward and can, in general, not be automated. In addition, all data flows between modules have to be identified in advance (as paths). This limits Escort to Information Appliances with a limited scope, as demonstrated within this dissertation.

One future work item suggested by this limitation is to provide tools for appliance specific environments. For example, it seems feasible to develop a tool that takes high level policies for a web server and generates automatically the necessary configuration files. A different approach to ease the configuration process is the use of graphical tools.

Another future work item inspired by this problem is: to find out what kind of appliances can be built with Escort. This also involves the question of which decisions should be made dynamically (during runtime) and which should be contained in configuration files. Currently, Escort makes most policy decisions during build time in configuration files. Many of these decisions can be delayed to runtime, which would allow the use of additional information not available during build time. The current static approach seems to work well with the appliances discussed. However, the dynamic approach extends clearly the number of appliances which can be implemented by Escort.

The second limitation is that accurate accounting requires authentication, and authentication requires resources. This general problem allows datagrams arriving

from the network to consume resources before they are authenticated. As shown in the WWW server example, this problem can be reduced, but not eliminated.

This problem inspired interest in how to define denial of service policies. These policies show certain similarities to quality of service policies, and will be extremely important on an increasingly hostile Internet.

REFERENCES

- [1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, June 1986.
- [2] Edward Amoroso. *Fundamentals of Computer Security Technology*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1994.
- [3] Ran Atkinson. *RFC 1825 - Security Architecture for the Internet Protocol*. IETF, August 1995.
- [4] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila 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.
- [5] Mary L. Bailey, Burra Gopal, Michael A. Pagels, Larry L. Peterson, and Prasenjit Sarkar. PathFinder: A pattern-based packet classifier. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 115-123, Monterey, CA, 1994. ACM/USENIX.
- [6] Ajay Bakre and B.R. Badrinath. Implementation and performance evaluation of indirect TCP. *IEEE Transactions on Computers*, 46(3), March 1997.
- [7] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server system. In *3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, 1999.
- [8] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, The MITRE Corp., Bedford MA, May 1973.
- [9] S. M. Bellovin. Security problems in the TCP/IP protocol suite. *CCR*, 19(2):32-48, April 1989.

- [10] 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, February 1990.
- [11] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan 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.
- [12] E. Bertino, E. Ferrari, and V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information and System Security*, 2(1):65–104, February 1999.
- [13] K. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, Mitre, Bedford, MA, April 1977.
- [14] B. Blakley and D. M. Kienzle. Some weaknesses of the TCB Modell. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy, May 4-7, 1997. Oakland, California*, pages 3–5, Los Alamitos, CA, USA, May 1997. IEEE Computer Society Press.
- [15] Allen C. Bomberger, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. The KeyKOS(R) nanokernel architecture. In USENIX Association, editor, *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures: 27–28 April, 1992, Seattle, WA, USA*, pages 95–112, Berkeley, CA, USA, April 1992. USENIX.
- [16] Andrej Brodnik, Svante Carlsson, Mikael Degermark, and Stephen Pink. Small forwarding tables for fast routing lookups. In *Proceedings of SIGCOMM '97 Symposium*, pages 3–14, Cannes, France, September 1997. ACM.
- [17] C. Brooks, M. Mazer, S. Meeks, and J. Miller. Application-specific proxy servers as HTTP stream transducers. In *Electronic Proc. 4th Int. World Wide Web Conference "The Web Revolution"*, Boston, MA, December 1995.
- [18] Jeff Chase, Hank Levy, Miche Baker-Harvey, and Ed 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.
- [19] M. Chatel. RFC 1919: Classical versus transparent IP proxies, March 1996.

- [20] D. Clark and D. Wilson. A comparison of commercial and military computer security policies. In *IEEE Symposium on Security and Privacy*, 1987.
- [21] D. E. Comer and D. L. Stevens. *Internetworking with TCP/IP, Volume III: Client-Server Programming and Applications, BSD Socket Version*. Prentice Hall, Engelwood Cliffs, NJ, 1993.
- [22] S. Deering and R. Hinden. RFC 2460: Internet Protocol, Version 6 (IPv6) specification, December 1998.
- [23] Department of Defense. *DoD 5200.28-STD: Department of Defense (DoD) Trusted Computer System Evaluation Criteria (TCSEC)*, 1985.
- [24] Tim Dierks and Christopher Allen. *Internet Draft: The TLS Protocol Version 1.0*. IETF, November 1997.
- [25] Peter Druschel and Larry 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, December 1993. ACM Press.
- [26] 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.
- [27] Dawson Engler and M. Frans Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proceedings of SIGCOMM '96 Symposium*, pages 53–59, Stanford, CA, August 1996. ACM.
- [28] D. Ferraiolo, J. Barkley, and D. Kuhn. A role-based access control model and reference implementation within a corporate intranet. *ACM Transactions on Information and System Security*, 2(1):34–64, February 1999.
- [29] Todd Fine and Spencer E. Minear. Assuring distributed trusted Mach. Technical report, Secure Computing Corporation, 1210 West Country Road E, Suite 100, Arden Hills, Minnesota 55112.
- [30] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the Winter 1994 USENIX Technical Conference and Exhibition*, pages 97–114, January 1994.
- [31] Geos-sc architecture. Technical Report PRSK-CM01-1297, Geoworks, 1997.
- [32] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

- [33] Jørgen S. Hansen, Torben Reich, Birger Andersen, and Eric Jul. Dynamic adaptation of network connections in mobile environments. *IEEE Internet Computing*, 2(1), January/February 1998.
- [34] N. Hardy. The confused deputy (or why capabilities might have been invented). *ACM Operating Systems Review, SIGOPS*, 22(4):36–38, October 1988.
- [35] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Jerry Vochtelloo. Mungi: A distributed single address-space operating system. Technical Report SCS&E Report 9314, University of New South Wales, Australia, November 1993.
- [36] Johannes Helander and Alessandro Forin. Mmlite: A highly componentized system architecture. Technical report, Microsoft Research.
- [37] Norman C. Hutchinson and Larry L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *TSE*, 17(1):64–76, January 1991.
- [38] M. Leech and et. al. SOCKS Protocol Version 5. RFC 1928, March 1996.
- [39] Jochen Liedtke, U. Bartling, U. Beyer, D. Heinrichs, R. Ruland, and G. Szalay. Two years of experience with a mu -kernel based OS. *ACM Operating Systems Review*, 25(2):57–62, April 1991.
- [40] David Maltz and Pravin Bhagwat. MSOCKS: An architecture for transport layer mobility. In *Proceedings of IEEE INFOCOM*, pages 1037–1045, April 1998. <ftp://ftp.monarch.cs.cmu.edu/pub/dmaltz/msocks-infocom98.ps.gz>.
- [41] David Maltz and Pravin Bhagwat. TCP splicing for application layer proxy performance. Technical report, IBM, March 1998. <ftp://ftp.cs.cmu.edu/user/dmaltz/Doc/splice-perf-tr.ps>.
- [42] John McLean. A general theory of composition for a class of “possibilistic” properties. *IEEE Transactions on Software Engineering*, 22(1):53–67, January 1996.
- [43] J. K. Millen. A resource allocation model for denial of service. In *Proceedings of the 1992 IEEE Computer Society Symposium on Security and Privacy (SSP '92)*, pages 137–147, Washington - Brussels - Tokyo, May 1992. IEEE.
- [44] A. Montz, D. Mosberger, S.W. O'Malley, L. Peterson, and T. Proebsting. Scout: A communications oriented operating system. In *Proceedings of the Fifth HotOS Workshop*, May 1995.

- [45] David Mosberger. Message library design notes. Technical Report TR97-19, The Department of Computer Science, University of Arizona, Tuesday, November 25 1997.
- [46] David Mosberger and Larry 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, October 1996. USENIX.
- [47] David Mosberger-Tang. Scout: Path-based operating system. Technical Report TR97-06, The Department of Computer Science, University of Arizona, May 13 1997.
- [48] Sape J. Mullender, Ian M. Leslie, and Derek 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.
- [49] George 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 January 1997.
- [50] Peter 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.
- [51] NIST. *Research and Development for the National Information Infrastructure Technical Challenges*. NIST, March 1994.
- [52] M. Nyanchama and S. Osborn. The role graph model and conflict of interest. *ACM Transactions on Information and System Security*, 2(1):3–33, February 1999.
- [53] V. Pai and W. Zwaenepoel P. Druschel. IO-Lite: A unified I/O buffering and caching system. In *3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, 1999.
- [54] Larry L. Peterson. The X-kernel: a platform for accessing Internet resources. Technical report TR 89-23, University of Arizona, Dept. of Computer Science, Tucson, AZ, USA, October 1989.
- [55] J. Postel. Internet Protocol. *Network Information Center RFC 791*, pages 1–45, September 1981.

- [56] J. Postel. Transmission Control Protocol. *Network Information Center RFC 793*, pages 1–85, September 1981.
- [57] Dave Probert and John 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.
- [58] N. E. Proctor and P. G. Neumann. Architectural implications of covert channels. In *Proceedings of the Fifteenth National Computer Security Conference*, pages 28–43, October 1992.
- [59] Marcus K. Ranum and Frederick M. Avolio. A toolkit and methods for Internet firewalls. In USENIX Association, editor, *Proceedings of the Summer 1994 USENIX Conference: June 6–10, 1994, Boston, Massachusetts, USA*, pages 37–44, Berkeley, CA, USA, Summer 1994. USENIX.
- [60] R. F. Rashid. MACH kernel interface manual. *CMU Technical Report*, December 1987.
- [61] Michael K. Reiter and Stuart G. Stubblebine. Toward acceptable metrics of authentication. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy, May 4-7, 1997. Oakland, California*, pages 10–20, Los Alamitos, CA, USA, May 1997. IEEE Computer Society Press.
- [62] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.
- [63] T. Roscoe. Linkage in the Nemesis single address space operating system. *Operating Systems Review*, 28(4):48–55, October 1994.
- [64] Timothy Roscoe. *The Structure of a Multi-Service Operating System*. PhD thesis, University of Cambridge Computer Laboratory, April 1995.
- [65] 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.
- [66] P. Winterbottom S. Dorward, D. Presotto, H. Trickey, R. Pike, D. Ritchie. Inferno. 2(1):5–18, 1997.
- [67] Saltzer and Schroeder. The protection of information in computer systems. *Proc. of the IEEE*, 63(9), September 1975.

- [68] R. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and System Security*, 2(1):105–135, February 1999.
- [69] Manfred Schlett. Trends in embedded microprocessor design. *IEEE Computer*, 31(8):44–49, August 1998.
- [70] S. Schönberg. L4 on Alpha, design and implementation. Technical Report CS-TR-407, University of Cambridge, 1996.
- [71] C. L. Schuba, I.V. Krsul, M.G. Kuhn, E.H. 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.
- [72] Sean Dorward, Rob Pike, David Leo Presotto, Dennis Ritchie, Howard Trickey and Phil Winterbottom. The Inferno operating system. *Bell Labs Technical Journal*, 2(1):5–18, Winter 1997.
- [73] Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536. *DTOS User Manual*, 86-0902042 edition, July 1995.
- [74] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith 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, October 1996. USENIX.
- [75] J. S. Shapiro, J. M. Smith, and D. J. Faber. Eros: A capability system. Technical Report MS-CIS-97-03, Distributed System Laboratory University of Pennsylvania, Philadelphia, PA 19104-6389, June 1997.
- [76] Daniel F. Stern and Glenn S. Benson. The controlled application set paradigm for trusted systems. In *Proceedings of the 1995 National Information System Security Conference*, 1995.
- [77] D. F. Sterne. On the buzzword "security policy". In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy (SSP '91)*, pages 219–231, Washington - Brussels - Tokyo, May 1991. IEEE.
- [78] Trusted Mach philosophy of protection, May 1993. NIST Document No: TMACH 93-014.

- [79] F. Travostino, E. Menze, and F. Reynolds. Paths: Programming with system resources in support of real-time distributed applications. In *Proceedings of the 1996 IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, pages 36–45, Laguna Beach, Ca, February 1996.
- [80] Philosophy of protection for the Triad system. Technical Report TIS Report 505, Trusted Information System, 11340 West Olympic Boulevard, Suite 265, LA, CA 90064, August 1996.
- [81] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of 14th ACM SOSOP*, pages 175–188, Asheville, NC, December 1993.
- [82] Marcel Waldvogel, George Varghese, Jon Turner, and Bernard Plattner. Scalable high speed IP routing lookups. In *Proceedings of SIGCOMM '97 Symposium*, pages 25–38, Cannes, France, September 1997. ACM.
- [83] J. Weiss. A system security engineering process. In *Proceedings of the 14th National Computer Security Conference*, 1991.
- [84] John. T. Wroclawski. Fast PC routers. URL: <http://ana-www.lcs.mit.edu/PC-Routers/pcrouter.html>.
- [85] Zakinthinos and Lee. A general theory of security properties. In *RSP: 18th IEEE Computer Society Symposium on Research in Security and Privacy*, 1997.