


198989

OUR REF.	DATE	INVOICE NO.	DESCRIPTION	AMOUNT
	2-24-89		Technical Report CS-88-43	\$2.00
COMSHARE, INC. ANN ARBOR, MICHIGAN				
DETACH BEFORE DEPOSITING CHECK 				TOTAL \$2.00

# COMSHARE

INCORPORATED

March 1, 1989

University of Waterloo  
Dept of Computer Science  
Attn: Research Report Secretary  
Waterloo, ON, CANADA  
N2L 3G1

Dear Research Report Secretary:

Please send a copy of technical report, CS-88-43, to Meredith Anderson, Comshare, 3001 S. State Street, Ann Arbor, MI 48108, USA. Enclosed is check #198989 for \$2.00. Should you have any questions, please call 313-994-4800.

Sincerely,

*Deborah Edwards-Onoro*  
Deborah Edwards-Onoro

*Enclosure*

*sent  
report  
Mar. 22/89*

# Printing Requisition / Graphic Services

15103

1. Please complete unshaded areas on form as applicable.
2. Distribute copies as follows: White and Yellow to Graphic Services. Retain Pink Copies for your records.
3. On completion of order the Yellow copy will be returned with the printed material.
4. Please direct enquiries, quoting requisition number and account number, to extension 3451.

TITLE OR DESCRIPTION

**A Protection Mechanism for Message-Passing Systems**

**CS-88-43**

DATE REQUISITIONED

DATE REQUIRED

ACCOUNT NO.

**Oct. 31/88**

**ASAP**

**1 2 6 6 0 1 2 4 1**

REQUISITIONER - PRINT

PHONE

SIGNING AUTHORITY

**J. Black**

**4459**

*J. Black*

MAILING INFO -

NAME

DEPT.

BLDG & ROOM NO.

☒ DELIVER

**Sue DeAngelis**

**C.S.**

**DC 2314**

☐ PICK-UP

Copyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

NUMBER OF PAGES **22** NUMBER OF COPIES **50**

TYPE OF PAPER STOCK

**Alpac Ivory**

☐ BOND ☐ NCR ☐ PT. ☐ COVER ☐ BRISTOL ☐ SUPPLIED ☐ **140M**

PAPER SIZE

**10x14 Glosscoat**

☐ 8 1/2 x 11 ☐ 8 1/2 x 14 ☐ 11 x 17 ☐ **10 pt Rolland Tint**

PAPER COLOUR

INK

☐ WHITE ☒ ☐ BLACK ☐

PRINTING

NUMBERING

☐ 1 SIDE PGS. ☒ 2 SIDES PGS. FROM TO

BINDING/FINISHING

☒ COLLATING ☐ STAPLING ☐ HOLE PUNCHED ☐ PLASTIC RING

FOLDING/PADDING

**7x10 saddle stitched**

Special Instructions

**Beaver Cover**

**Both cover and inside in black ink please**

**Thanks.**

COPY CENTRE

OPER. NO. BLDG. NO. MACH. NO.

DESIGN & PASTE-UP

OPER. NO. TIME LABOUR CODE

TYPESETTING

QUANTITY

**P A P 0 0 0 0 0 0** **T 0 1**

**P A P 0 0 0 0 0 0** **T 0 1**

**P A P 0 0 0 0 0 0** **T 0 1**

PROOF

**P R F**

**P R F**

**P R F**

NEGATIVES

QUANTITY

OPER. NO.

TIME

LABOUR CODE

**F L M** **C 0 1**

**F L M** **C 0 1**

**F L M** **C 0 1**

**F L M** **C 0 1**

**F L M** **C 0 1**

PMT

**P M T** **C 0 1**

**P M T** **C 0 1**

**P M T** **C 0 1**

PLATES

**P L T** **P 0 1**

**P L T** **P 0 1**

**P L T** **P 0 1**

STOCK

**0 0 1**

**0 0 1**

**0 0 1**

**0 0 1**

BINDERY

**R N G** **B 0 1**

**R N G** **B 0 1**

**R N G** **B 0 1**

**M I S 0 0 0 0 0 0** **B 0 1**

OUTSIDE SERVICES

\$ COST

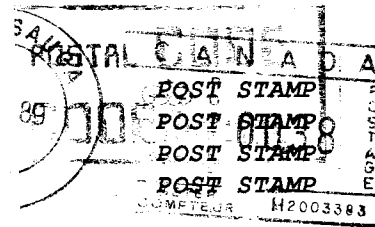
TAXES - PROVINCIAL ☐ FEDERAL ☐ GRAPHIC SERV. OCT. 85 482-2



For the attention of :

DR. A. GOSCIN'

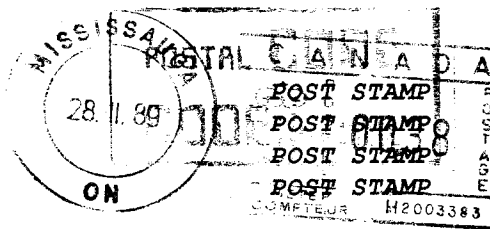
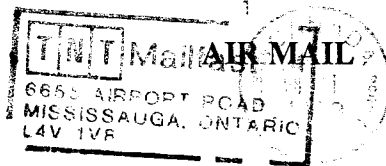
Sue -  
Please handle.  
Thanks.  
Jay.



Address:

Department of Computer Science  
The University of New South Wales  
University College  
Australian Defence Forces Academy  
Northcott Drive  
Campbell, ACT 2600  
AUSTRALIA

DR. J. P. BLACK  
DEPT. OF COMPUTER SCI.  
UNIV. OF WATERLOO  
WATERLOO  
ONTARIO N2L 3G1  
CANADA



For the attention of :

DR. A. GOSCINSKI

Address:

Department of Computer Science  
The University of New South Wales  
University College  
Australian Defence Forces Academy  
Northcott Drive  
Campbell, ACT 2600  
AUSTRALIA

DR. J. P. BLACK  
DEPT. OF COMPUTER SCI.  
UNIV. OF WATERLOO  
WATERLOO  
ONTARIO N2L 3G1  
CANADA

Date: FEB. 22, 1989

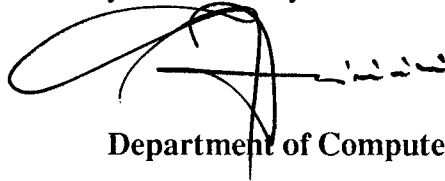
Dear Sir/Madam,

If available, I would greatly appreciate a reprint of your paper

A PROTECTION MECHANISM FOR  
MESSAGE-PASSING SYSTEMS

RESEARCH REPORT CS-88-43

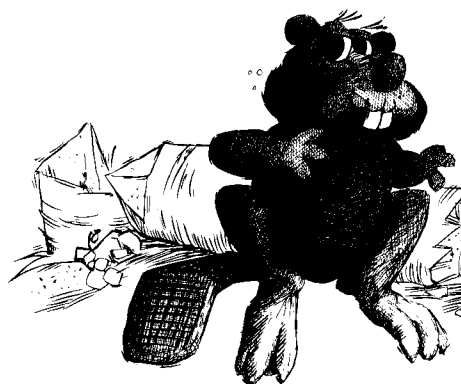
Thanking you in advance,  
yours faithfully

A handwritten signature in black ink, appearing to be "A. J. ...", written over a horizontal line.

Department of Computer Science

*mailed  
March 6/89*

UNIVERSITY OF WATERLOO  
UNIVERSITY OF WATERLOO  
UNIVERSITY OF WATERLOO  
UNIVERSITY OF WATERLOO  
COMPUTER SCIENCE DEPARTMENT  
COMPUTER SCIENCE DEPARTMENT  
COMPUTER SCIENCE DEPARTMENT  
COMPUTER SCIENCE DEPARTMENT



*A Protection Mechanism for  
Message-Passing Systems*

*J.P. Black*

*Research Report  
CS-88-43*

*October, 1988*

# A Protection Mechanism for Message-Passing Systems

J. P. Black

Department of Computer Science  
University of Waterloo

October 25, 1988

## Abstract

In message-passing systems, client and server processes interact with messages to perform computations on behalf of a number of human users of the system. Most existing protection mechanisms for such systems are based on capabilities. We show that it is also possible to use the other well-known protection model based on access control lists in these systems. We propose a mechanism, based on *signatures*, which is simple to use from a programming viewpoint, can be implemented efficiently, and does not require the use of cryptographic techniques. Through the use of signatures accompanying messages, access of clients to services and information can be controlled and changed on a per-message basis, as clients of one service in turn implement higher-level services for their own clients. We also discuss some duality properties of signature and capability mechanisms.

## 1 Introduction

There is a trend in many modern operating systems toward the use of lightweight processes communicating through message passing. This has resulted in the development of new programming styles involving, for example, client-server models and remote procedure call. Protection mechanisms for these systems tend to be based on the use of capabilities. However, we

feel that it is also possible to base protection policies on the use of more conventional access control lists with the assistance of a mechanism which is practical, easy to understand, and relatively cheap to implement and use.

In this paper, we propose a protection mechanism for message-based distributed operating systems which we feel has these properties. It involves little additional programming complexity, adds very little overhead to the operating system kernel, is easy to understand, and does not require the use of cryptographic techniques. In Section 2, we present our view of a distributed system based on processes communicating by messages. In Section 3, we recall some of the background material on protection models, and discuss the relationships between these models and the class of distributed systems which we describe in the previous section. Section 4 then gives our signature mechanism for protection, which is justified in greater detail by the examples of Section 5. Section 6 presents our conclusions.

## 2 The Message Passing Model of Distributed Systems

We consider systems in which a number of *nodes* are connected by some communication medium. Each node, which may be implemented by one or perhaps many processors, supports a number of *processes*. Processes communicate only by sending and receiving messages; they have disjoint virtual address spaces. (In some cases, this is only a convention which is not enforced by the system.) A process is identified by a fairly long bit string called a process identifier or *PID*. The PID is allocated by the system when the process is created, and is guaranteed not to be reused for a reasonably long period of time (*i.e.*, PIDs can be assumed to be unique in time). Processes are transient, in that they survive neither system crashes nor orderly system shutdowns. Some small amount of stable storage is used to ensure the PID uniqueness across shutdowns and crashes.

Examples of systems which fit this model include several with strong links to Waterloo [2,3,4,8], and others such as Accent [14] and Amoeba [12]. Several systems based on the use of remote procedure call (RPC) [5,7,10] also fit the model, although the client-server type of interaction in the former is slightly more general than the interactions typically found in the

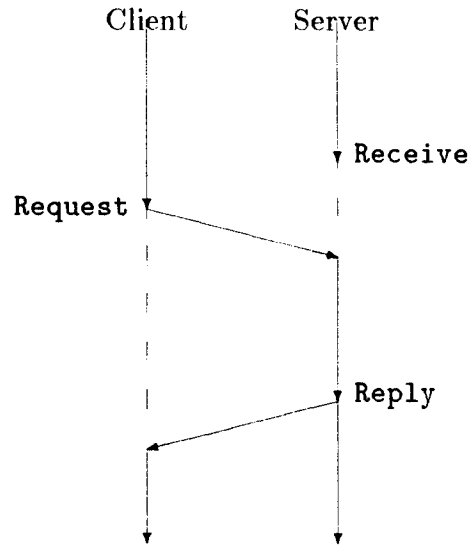


Figure 1: A client-server exchange with **Request-Receive-Reply**

latter. Commercial systems such as UNIX 4.3BSD are also developing large numbers of client-server applications. However, they must also continue to support the more classical process and operating system interactions originally developed for centralised time-sharing systems.

For the sake of exposition, we will assume the style of communication primitive adopted by the Waterloo systems. In these systems, there are three primitives for message passing, called **Request**, **Receive**, and **Reply**. They are used as shown in Figure 1. A client wishing to request service from a server uses the **Request** primitive, supplying the identifier of the target server process and buffers for the request message and the reply message from the server. This client is then blocked until the server receives the message, processes it, and replies to it. From the client's point of view, this is very much like a remote procedure call, ignoring details such as argument type-checking and parameter marshalling.

Whenever the server wishes to service requests, it makes a blocking call to **Receive**, which either blocks the server if no messages are queued, or removes a client message from the server's queue. Some time later, the server completes the client's request and calls **Reply**, supplying the

client's identifier and the reply message. However, since the server is free to make a number of calls to `Receive` without issuing any `Reply` messages, and also to reply to multiple clients without intervening calls to `Receive`, the client's illusion of remote procedure call is intentionally missing at the server. This freedom allows the server to serve multiple clients simultaneously if it chooses and removes from the system any need to create or allocate processes in response to individual RPCs from clients.

One of the common ways in which this client-server model is used to construct more complex software systems is called the administrator model [9]. In this model, all requests for a particular service or class of service are first interpreted by an *administrator* process, which has a main loop containing a call to `Receive`. In order to avoid becoming a bottleneck, an administrator never calls `Request`, as it might block indefinitely. Instead, it manages a pool of worker processes, each of which calls the administrator with `Request` when it finishes a task; the administrator's `Reply` message to the blocked worker contains the next task for it to perform. Thus the administrator typically expects two classes of `Request` messages (and manages one or more associated queues of outstanding requests): one class from clients requesting service and one class from workers requesting new tasks to be performed. In effect, client requests are propagated through an administrator to a worker and perhaps beyond, while the client remains blocked awaiting the `Reply` from the administrator. The administrator does some amount of interpretation of client requests, but most of the work is performed by its worker processes. See Figure 2.

Up to now, we have viewed the client-server paradigm in terms of dynamic control structure. A complementary view in terms of specification of behaviour is that the server presents some abstraction (or abstract data type) to its clients. Clients request service (or invoke abstract operations) through some message protocol published as part of the specification of the server. The details of how the server implements the service and of how many other processes are involved are hidden from the client, and not specified by the message protocol. In the most general view of this client-server relationship, the server is free to implement whatever protection, authentication, and accounting policies it wishes; in practice, these policies are often severely limited by the trusted information available to the server and by its ability to exploit this information with acceptable performance. Examples

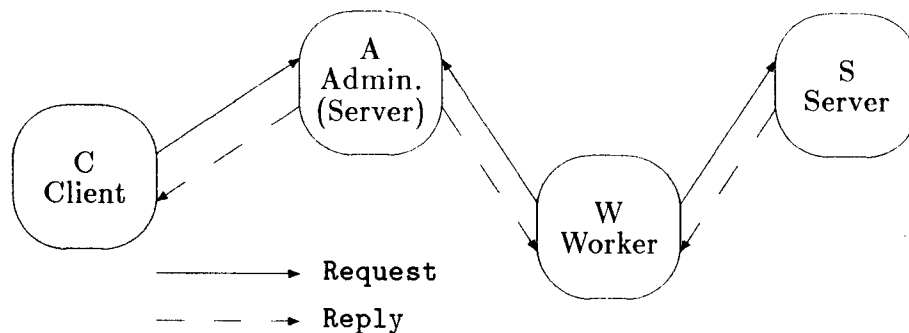


Figure 2: A Client, an Administrator, a Worker, and another Server

of common services are file and directory service, printing, name service, communication with other systems, electronic mail, window management, compilation, and document preparation.

In many operating systems, processes request system services by making supervisor or system call requests of the underlying kernel. In message-passing systems, the tendency is to provide many system services in server processes outside the kernel, keeping the kernel as small, efficient, and understandable as possible. Thus, the only system calls in message-passing systems are those to send and receive messages, and perhaps a very few others to perform process operations such as creation and termination, for which the ultimate responsibility rests with the kernel. Typically, normal system services are requested through message exchanges with server processes, leading to extensive and frequent use of message passing for fundamental services such as those described above. From the point of view of the kernel, there is not necessarily any difference between a user process and a process providing a critical and trusted system function. While the kernel may have some notion of trusted and untrusted users, it is no longer the case that the hardware protection boundary between user and supervisor states can be used to separate untrusted user software from trusted system software.

Compared with monolithic centralised operating systems, there has been a very significant degree of function migration out of the user process and out of the kernel. To a large extent, the kernel no longer performs

any system services beyond message passing and process creation: all the functionality normally associated with operating system supervisor calls has been migrated out of the kernel, into system server processes. Even more importantly, many functions previously performed by library modules within the user process are now performed in disjoint server processes: common examples are file services and window management. Even when there is no obvious benefit to be obtained from extra parallelism, functions tend to migrate into disjoint processes due to the pervasiveness of the client-server model. In a sense, the two-layer view of user process and kernel has evolved into one with multiple layers of process abstractions.

This is a very different environment for user-level protection from that of the centralised operating system.

### 3 Protection Mechanisms and Trust

Protection within a computer system is required if the system is to be shared by a number of mutually suspicious users. Sharing in this sense may involve arbitrarily long timescales, since protection is typically required both among a number of simultaneous users of the system, and for information stored on external storage by active and currently inactive users.

The general literature on protection is based on the access matrix model, which involves subjects, objects, and rights [11, Chapter 8]. Subjects are active entities such as human users and processes. Objects are the protected entities in the system, such as files, processes themselves, devices, memory segments, or procedures, to name only a few. The granularity of objects in a particular model can be low (memory segments, procedures) or high (files, users, devices, the system itself). Each subject possesses a set of rights to perform operations upon each object, and the protection state of the system can be encoded into a (sparse) *access control matrix*. Each row corresponds to a subject, each column to an object, and each entry in the matrix encodes the particular subset of operations which can be performed by a subject upon an object. In the case of a client-server system, what is required is some mechanism allowing individual servers to implement arbitrary objects and access rights for subjects such as users and client processes. The two

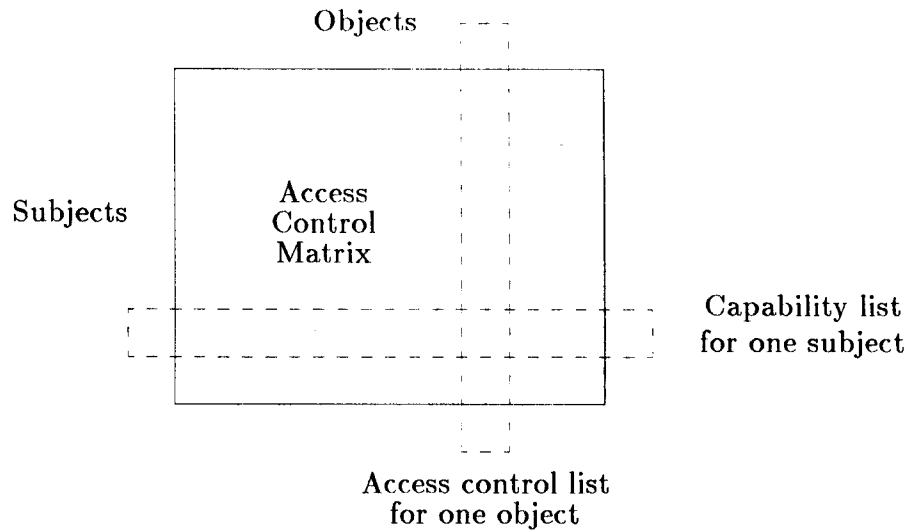


Figure 3: Protection Models

main approaches to implementing the access control matrix involve the use of either access control lists or capabilities. See Figure 3.

An access control list encodes one column of the matrix: for a particular object, it lists the rights of each subject with access to the object. Typically, it is stored with the object. In the common case of files and directories, the subjects are human users or pseudo-users which implement system services, and the file system keeps some information with each file or directory on who can access it in which fashion (read only, read-write, execute only, search, *etc.*). Groups of users may also be subjects in this sense. The advantages of access control lists are that it is easy to change individual entries in a column of the matrix, as each entry is stored with the object, and it is easy to enumerate those subjects with access to the object. One disadvantage is that on each access by a subject, the operation requested must be checked against those allowed for that particular subject. For objects implemented by general server processes, the server must have available information about the subject who is requesting an operation, in order to verify the rights of the subject.

Historically, the subjects in the access control list approach have been

processes working on behalf of a particular human user, and many operating systems associate some user identifier with each active process. This is perfectly adequate when essentially all of the computation can be attributed to the user process, and the synchronous nature of most system calls makes well-defined the notion of which user is performing each operation. In client-server systems, system processes are not necessarily associated with any single user. Each request serviced by a system process may be on behalf of a different user. As each request is executed, the server or its workers may make further requests of other servers, and each of these is then also on behalf of the originating user, not on behalf of the user or system process which created the server or worker. This leads us to conclude that conventional protection schemes based on user identifiers are not particularly applicable to operating systems based on the client-server paradigm.

In protection models which use capabilities, the access control matrix is stored by rows, so to speak. That is, each subject (a client process) possesses a list of capabilities which completely defines the set of objects and operations on them which is available to the subject. Each capability is encoded and/or protected in such a fashion that simple possession of a capability is sufficient to guarantee to a server that the client has the rights to perform each operation. Capabilities are appealing because the cost of storing them is pushed back onto the client and because they may also contain low-level object identifiers which can be used by the server to make access more efficient. However, it is difficult to revoke them or to perform garbage collection on them, because the system does not keep track of all clients with capabilities to a particular object. Also, capabilities have fixed formats in most systems, which make it difficult or impossible to implement new objects with operations and capabilities which cannot be encoded into the existing system-wide standard. Finally, the system must be able to guarantee the integrity and validity of capabilities presented by clients: thus, the system must explicitly control and store all capabilities, or some relatively strong form of cryptographic sealing of the capabilities is required to prevent forgery by malicious or buggy processes.

In both models, some minimal mechanism must be provided to ensure that a server can trust the information upon which some authorisation decision is made. For access control lists, the server must convince itself of

the identity of a client subject. For capabilities, the server must be able to trust the authenticity of the capability presented by the client. This trust may be established based upon the presumed security of the kernel which transports subject identities or capabilities, or upon cryptographic properties of the capability encoding, the subject identity, or the communication channel between the client and the server.

Capability-based protection mechanisms for distributed systems have been quite well-represented in the recent literature, in such systems as Amoeba [12] and Accent [14]. They seem well-suited to client-server interactions, since the simple presentation of the capability is evidence of authority to access the object it represents. Access control lists have been less visible in the research literature, but perhaps more common in extensions of centralised systems such as UNIX into distributed environments. In such systems, the two key problems are how a server may trust the origin of a message and how the server may appear to operate on behalf of various clients when he makes requests to third parties.

We first make the obvious assumption that users and processes may trust the kernel of a machine on which they run to implement the protection mechanism correctly. In particular, they trust the kernel to identify them to other kernels and to identify other kernels correctly in incoming messages. That is, each kernel is trusted to transport each message to its ultimate destination host, and to identify the source host of each incoming message correctly. However, we suggest that the problem of secure and trusted kernel-to-kernel communication is separate. If inter-kernel communication is not secure and trusted, the problem can be solved by standard encryption techniques and/or specialised protocols, such as that suggested by [1]. If remote kernels cannot be trusted to identify their users correctly, then the local kernel can collapse all subjects on the untrusted remote machine into one in a fashion which is transparent to the local processes. With the increasing use of personal workstations, this becomes more and more realistic, since the complete software of a workstation must be assumed to be under the control of a single user. In very large distributed systems which tend to be more loosely-coupled, an effective protection granularity of the human user is quite acceptable.

The second problem of how a server can authenticate himself to third parties on behalf of various clients is addressed in the next section.

## 4 A Protection Mechanism Based on Signatures

The protection mechanism we discuss in this section is intended to be implemented by the trusted kernel of a single computer. In the general case, the computer is shared among a number of human users and communicates with other computers which implement their own versions of the mechanism. The individual users of the distributed system are assumed to be autonomous and mutually suspicious. As mentioned above, we assume that the inter-kernel transmission medium is secure and trusted.

In the model, a process consists of an unchanging load image and a changeable data segment. A *signature*  $(w, u)$  is associated with every process. Each component  $w$  or  $u$  identifies a subject, in the terminology of the previous section. With the exception of a distinguished subject called *root*, the signatures and subjects are never of interest to the kernel.

The first component,  $w$ , is a subject called the *owner* of the process. Roughly,  $w$  identifies the source of the unchanging data and executable text in the load image of the process. If a second process chooses to trust the owner with certain operations on its behalf, the kernel guarantees that the process has been vested with the authority of the owner. Owners may be human subjects, or artificial subjects like trusted file systems, mail servers, or printer servers.

The second component,  $u$ , is called the *user* of the process. It identifies the source of the data part of the process, or equivalently, of a particular instance of the process image. Each time the load module is used to create a new process, the contents of the changeable data segment will in general depend on the input data to the process. The distinction here between owner and user is very similar to that found in UNIX between the real and effective user identifiers of a process, and can be used in much the same way.

Note that, in particular, our mechanism does not specify any structure of signatures beyond the existence of the owner and user components: system designers may wish to encode or structure signatures to contain group identifiers or other more complex attributes of a user. For example, the file system may choose to implement arbitrarily complex notions of protec-

tion and permissions based on signatures. One possible scheme based on a lattice structure is given in [15].

Under the Request-Receive-Reply model of interaction, a process may choose to transfer its signature to a server when it makes a blocking **Request**. The kernel will guarantee that only one other process possesses the signature at any point in time, and that the signature is returned to the issuer when it is unblocked by the reception of a **Reply** message. The server may use the signature to authenticate the client and may forward it to other processes in the course of responding to the request from the client. To avoid forgery, process identifiers are used to refer to signatures in kernel primitives, and processes may ask the kernel to provide an explicit signature in the form  $(w, u)$ , given any process identifier.

More formally, the following primitives are available for manipulating signatures. The reader may wish to refer to Figure 2.

- **Request(topid, msg, msg\_sigpid, rslt, rslt\_sigpid)**

The first, second, and fourth parameters are the server process identifier, and buffers for the request and result messages. If the two signature process identifiers are zero, no signature is sent or returned, and **Request** behaves according to its normal semantics. Otherwise, the signature of process `msg_sigpid` is transferred to the server.

If `msg_sigpid` is that of the client, the client is temporarily supplying its own signature to the server while it is blocked awaiting the reply. In this case, `rslt_sigpid` is guaranteed to be equal to `msg_sigpid` on return. In Figure 2, this is the case when C sends a request to A.

If `msg_sigpid` is not that of the client, that process must be blocked on this client or some other process, and this client must have received its signature previously. On return from **Request**, `rslt_sigpid` may contain a signature transferred by the server; this would be the normal means of transferring a signature from an administrator to a worker. In Figure 2, W receives the signature of C as `rslt_sigpid` when `rslt` is returned to him in response to his previous “Give me new work” message to A.

An exception is signalled if the client does not possess `msg_sigpid`.

- `Receive(frompid, msg, msg_sigpid)`

If `msg_sigpid` is non-zero, it is the PID of a process currently blocked on a `Request`. If `frompid = msg_sigpid`, the signature has been transferred directly from that process. If not, process `frompid` may be sending the signature away from its originator (as when S receives it in Figure 2), or back towards the originator (as when A receives a work completed or “Give me new work” message and signature from W).

- `Reply(topid, rslt, rslt_sigpid)`

If `rslt_sigpid` is non-zero and is possessed by the caller, the signature is transferred to `topid`. This is used by A to transmit the signature to W, and by A and S to transmit it back towards C and W respectively.

An exception is signalled to both process `topid` and the caller if `rslt_sigpid` is that of the caller, or if the caller does not possess `rslt_sigpid`, or if process `topid` sent its own signature but the caller does not return it.

- `Get_sig(sigpid, sig, status)`

This primitive discloses signature information and process status to the caller. Possible values for `status` are `Signature`, `No_signature`, and `No_such_process`. The first two cases indicate whether the caller currently possesses the signature for process `sigpid`. In both these cases, `sig` contains the signature  $(w, u)$  of process `sigpid` when the call returns.

- `Set_uid(v)`

If the caller initially has signature  $(w, u)$ , then the caller’s signature is set to  $(w, v)$  if and only if  $v = w$  or  $v = u$  or  $w = \text{root}$ . An exception is signalled otherwise.

When a process is created, it must also be given a signature. Roughly, we want the creator to have some control over the signature of the child. However, being more precise depends on the exact semantics of process creation, which we do not wish to specify. Nevertheless, we give two examples

of reasonable semantics. In both UNIX and Shoshin [2], the kernel maintains an abstraction of a loadable process image. In the case of UNIX, this is simple because the kernel also implements the file system, and so a load module is simply a file in the file system. In the case of Shoshin, process images are either loaded with the kernel or installed dynamically. In either system, we assume that the owner component of the newly-created process is taken from the owner component of the load module, while the creator can influence the user component by supplying a `sigpid`. For example,

- `pid := Create(image, sigpid)`

Let  $(x, u)$  be the signature of process `sigpid`, or of the caller if `sigpid` is zero. Then if the image has owner  $w$ , the child process is created with signature  $(w, u)$ . In order for a process with owner *root* to be able to create a child with a different owner, we assume that such a process is able to set the owner of the image to an arbitrary value.

In another form of process creation, such as used by the V Kernel [3], an “empty” process is created by the kernel, and the creator is given direct write access to the code segment of the new process. We would then propose a primitive like

- `Create(sigpid, owner flag, user flag)`

The two flags indicate whether the two components of the signature of the child process are to be obtained from those of the caller, or from those of process `sigpid`. In order for a process with owner *root* to be able to create a child with a different owner, we also require a variant form of `Create` which uses a signature as the first parameter, and whose use is restricted to *root* processes.

The semantics given above require that the kernel validate each signature transfer, and maintain some notion of the current location of the signature. In addition, we require that any exception which abnormally terminates the client’s `Request` result in the return of the signature to the client. As a special case, this includes failure of the process or host possessing the signature. (This is very similar to the problem known as “orphan-killing” in the remote procedure call literature [13].)

- Since each process has exactly one signature, the process descriptor provides a logical repository for any information required by the implementation of the mechanism.
- In the case of a local transfer, only local information is required.
- In the case of a transfer to a second machine, the signature can be transmitted with the message, and kept on that remote machine as part of the state information associated with the IPC exchange. Some care is needed to ensure that signatures are reclaimed whenever some exception causes resumption of the client. Examples of such an exception include failure or disconnection of the remote machine.
- In the case of a signature transfer to a process on a third host, more complicated “signature tracking” is required. If this tracking information is maintained at the site of the owner process, an interkernel message exchange may be required to update the location of the signature in the owner’s process control block. We are currently developing and implementing a distributed signature tracking algorithm which avoids these extra interkernel messages in the normal case of exception-free interaction [6].
- Because process identifiers are unique in time, a signature authenticating a client need only be transferred on an initial interaction with a server, not on every request issued by the client to the same server.
- A server and other processes who trust each other can retain signatures as data indefinitely, and transmit them in the clear among themselves.
- As `Set_uid` changes the signature of a process, the frequent use of it by a client may require more frequent signature transmission between client and server if the semantics of the interactions depend on such signature changes.

The examples in the next section suggest that the cost of signature use need not be unreasonable.

## 5 Examples

The previous section discusses a kernel mechanism which can be used to protect users from each other. However, it does not discuss how this mechanism might be used to implement one or more protection policies at a higher level. It is not our purpose to propose such policies, but rather to give some indication of the range of policies which can be constructed on top of the mechanism described above. Thus, the set of examples which follows is simply intended to illustrate the use of the mechanism, and to convince the reader of its utility in implementing more conventional protection policies as are currently prevalent in centralised and distributed systems.

### 5.1 System Initialization

When a system is bootstrapped, some kernel software is loaded and executed, and at least one distinguished user process must be started by the system. Ultimately, all trust in the system software must stem from trust in the source of the kernel and initial process image, which may be physically secure on a disk in a machine room or perhaps transmitted in a secure fashion across some communication medium. We need to show that our protection mechanism allows human users to trust the identity of various system servers, based on their trust in the source of the kernel and initial process image.

The initial process must be created with signature (*root*, *root*), for otherwise, no non-trivial executions of `Set_uid` will succeed, and no non-trivial set of processes without *root* signatures can be created. The initial process normally creates some system service processes, which it can arrange to give arbitrary signatures. Examples of standard system services might include a name server, file system servers, printer servers, and login servers. Many of them would be created with well-known or *root* signatures. If desired, the initial process could create specialised or trusted processes of various types on demand from users or other system services. Thus, at the end of system initialization, a number of standard and trusted system services have been created. In general, they run with known signatures, so that suspicious users can confirm process identities with `Get_sig` before

trusting these processes to perform as claimed.

## 5.2 File System

It is tempting to consider the file system as just another set of user processes outside the kernel and as no different in principle from other user processes. However, the reliance placed upon the file system by users necessitates careful consideration of how it can effectively safeguard information for users and guarantee that this information is only used in ways permitted by the protection policies of the human user and system administrator.

In the simple case of file and directory access, the user process uses some name server to find the process identifier of the file system server, and then uses `Get sig` to convince itself of the identity of that process. On individual file system requests, it sends its own signature, which the file system uses according to its published specifications to permit or deny access and, possibly, to store with the data committed to it for safekeeping. The file system makes requests of other lower-level server processes, such as disk storage subsystems, to whom it may pass the signatures of its clients.

## 5.3 Print Servers

We suppose that print spooling with accounting for physical pages printed is required. If requests to print have copy semantics, a user process can forward its signature to the print server when it requests printing, and can remain blocked while the server forwards the user signature to the file system to request a copy of or access to the file for printing.

If spooled requests are serviced without making copies at the time of the request, the print server must somehow arrange to have access through the file system to the stored information. This is only a problem because, at the time of printing, we assume that the requesting user process is no longer blocked on the print server, and indeed, may no longer even exist. Efficient solutions to this appear to involve file system policies allowing some form of controlled access to some processes not owning the data. In many existing systems, similar services run with *root* privileges to ensure they have access to the data.

In addition, the print server can use the signature of the requesting process to account and charge for pages printed.

## 5.4 Abstract Data Types

The intent of the separation between the *owner* and *user* components of a signature is in some sense to capture the difference between the unchanging code of a process and the changeable state of the process. Many processes can be instantiated from a single process image, but they will all obey the same interface and protocol for message exchange. This is very similar to the difference in object-oriented systems between the class of an object and an instance of the object. In our signature model, one could think of the *owner* component as indicating the class of the process, and the *user* component as indicating the human user on whose behalf state is being stored in the process.

As first presented above, signatures encode ownership of resources and control over the use of them. Here, the owner component encodes some equivalence class of behaviours. However, this view is, in a sense, “overloading” the use of the signature mechanism. Whether these two views are mutually exclusive or compatible is a subject requiring further investigation.

We believe that both these viewpoints may be useful to servers implementing protection policies for mediating among mutually suspicious users.

## 5.5 Combined Signature and Capability

Servers may choose to implement capabilities on a per-user basis, by considering a capability to be a pair consisting of a signature and an integer chosen from a large sparse space to make forgery unlikely. When an object is created, the signature of the client is stored with it, and a record is kept of the integer. By communicating the integer to other servers along with the signature, the client can grant them explicit access to the object. Because the servers cannot forge the client’s signature, they can only access the object in ways implied by the capability. In this way, the kernel is relieved of any need to maintain the correspondence between a capability and a server for the object for an indefinite period of time.

Such a scheme might be attractive for at least two reasons. For example, the client could avoid having to obtain the signature of a third party in order to contact the server to modify its access control list. In addition, the server could replace what might be a relatively large access control list for the object with a potentially shorter list of capabilities.

## 6 Conclusions

The model presented above provides a simple and cheap protection mechanism which can be added to client-server distributed systems. It permits clients to identify themselves to servers in a fashion guaranteed by the underlying kernel. The servers are then free to implement arbitrary objects and operations upon them, and to maintain access control lists of arbitrary complexity for each object. It is more symmetric than a pure capability system, as both client and server can examine the other's signature. The use of a model with a granularity of human users helps to bring down the cost of using the model. We need to show that in practice, this level of granularity is sufficient. Furthermore, signatures need not be passed or checked on each IPC interaction, which further reduces the per-message cost of their use.

There seems to be an interesting duality relationship between the use of capabilities and the use of signatures. (Some of this is suggested by the geometry of Figure 3.) One can consider a capability to be an authority plus some sort of name for a server, and the authority to act on an object can be transferred easily from one client to another by capability transmission. On the other hand, one can consider a signature to be an authority plus some sort of name for the client, and this authority can be transferred from one server to another by signature transmission. Whereas a server may have no control over transmission of capabilities to the objects it implements, a client has no control over transmission of its signature by servers. Corresponding to the problem of garbage collection with capabilities is one of signature tracking. Whether this duality is in fact an equivalence between the two mechanisms, or whether they are in fact complementary requires more investigation.

In this paper, we have presented the mechanism, but done little more

than sketch some possible policies which can be implemented on top of it. In our view, one of the main advantages of the model is that it provides a simple mechanism in the kernel, without requiring that the policies be implemented there for all possible objects and object types for which protection is to be provided. Relative to capability-based systems, no fixed set of capabilities is provided for all objects. The particular protection facilities made available for an object depend only on the server implementing the object. This, of course, means that in any system using such a model, a number of higher-level, extra-kernel conventions need to be established and published as part of the specifications for the various objects created and supported by the system. However, this is no different from the analogous publication of policies which must accompany any protection system.

We are currently implementing a distributed signature tracking mechanism to support a full implementation of signatures on Shoshin. We also plan to experiment with the use of signatures, and to explore the duality between signatures and capabilities further.

## Acknowledgements

Thanks to Jeff Frontz and David Taylor for careful reading of a previous draft, as well as for the design and implementation of the distributed signature-tracking mechanism. Thanks to John Dobson and Gord Cormack for thoughtful comments on a previous draft.

## References

- [1] D. P. Anderson, D. Ferrari, P. V. Rangan, and B. Sartirana. A protocol for secure communication in large distributed systems. Report UCB/CSDE 87/342, Computer Science Division (EECS), Univ. of California, Berkeley, February 1987.
- [2] J. P. Black, W. H. Cheung, E. C. Lam, F. C.-M. Lau, and E. G. Manning. Shoshin: Developing and understanding distributed system software. Technical Report UW/ICR 87-04, Institute for Computer Research, Univ. of Waterloo, May 1987.

- [3] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.
- [4] David R. Cheriton, M. A. Malcolm, L. S. Melen, and G. R. Sager. Thoth, a portable real-time operating system. *Communications of the ACM*, 22(2):105–115, February 1979.
- [5] Robert Cooper. Pilgrim: A debugger for distributed systems. In *7th Conference on DCS*, pages 458–465. IEEE, September 1987.
- [6] J. H. Frontz. Signature-based protection for Shoshin. M. Math. essay, Dept. of Computer Science, University of Waterloo, August 1988.
- [7] N. Gammage and L. Casey. XMS: A rendezvous-based distributed system architecture. *IEEE Software*, 2(3):9–20, May 1985.
- [8] W. M. Gentleman. Using the Harmony operating system. Technical Report 24685, National Research Council of Canada, May 1985.
- [9] W. Morven Gentleman. Message passing between sequential processes: The reply primitive and the administrator concept. *Software—Practice and Experience*, 11:435–466, 1981.
- [10] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–313, March 1988.
- [11] M. Maekawa, A. E. Oldehoeft, and R. R. Oldehoeft. *Operating Systems—Advanced Concepts*. Benjamin/Cummings, 1987.
- [12] S. J. Mullender and A. S. Tanenbaum. The design of a capability-based distributed operating system. *The Computer Journal*, 29(4):289–299, 1986.
- [13] F. Panzieri and S. K. Shrivastava. Rajdoot: A remote procedure call mechanism supporting orphan detection and killing. *IEEE Trans. on Software Engineering*, 14(1):30–37, 1988.
- [14] R. F. Rashid and G. G. Robertson. Accent: A communication oriented network operating system kernel. In *Proc. 8th Symp. on Operating Systems Principles*, pages 64–75, December 1981.

- [15] R. S. Sandhu. The NTree: A two dimension partial order for protection groups. *ACM Transactions on Computer Systems*, 6(2):197–222, May 1988.