Encapsulating Mobile Objects*

Hermann Härtig Lars Reuther

Dresden University of Technology
Department of Computer Science
Dresden, Germany
{haertig,reuther}@os.inf.tu-dresden.de

Abstract

This paper describes a technique to effectively isolate mobile objects or processes that execute downloaded, potentially suspicious programs. It relies on wish lists, trust lists and capability lists. *Wish lists are carried along with programs or mobile objects and denote the resources requested by the program to do what it claims to do. Wish lists are transformed into capability lists when downloaded programs are started. Trust lists reside on stations and are used to determine which members of wish lists are taken over into capability lists. The capability lists are enforced during the execution of programs. All lists are symbolic to enable their interpretation in heterogeneous environments. The paper describes the technique, its integration in a Linux environment and first experiences.*

1 Introduction

Recent applications in distributed systems often use various kinds of mobile objects (e.g. Mobile Agents [1] or Java Applets). Since they are started on or migrated to local computers, mobile objects increasingly pose problems for the integrity of workstations and the confidentiality of their data. To protect a local workstation from malicious downloaded programs, effective techniques are required to isolate or encapsulate active entities within systems.

This situation is not new. Trojan horses, worms and viruses are a long term research area. However, due to recent technological developments the problem has achieved a new dimension, which renders current techniques less satisfactory as will be discussed in some detail in section 2 of this paper.

The aim of this paper is to define an environment for the secure execution of downloaded, possibly malicious programs. It makes it possible to determine a set of resources a program is allowed to access. The environment ensures that no other resources than those included in this set are accessed by a program. The paper describes an adaptation of a technique already present in the BirliX Security Architecture [3, 4]. Essentially, programs carry a list of names denoting the resources (i.e. files, connections, processes ...) they are allowed to use once the program has been started somewhere on a machine. In contrast to the BirliX security architecture, the list contains symbolic names. Programs, together with the appended list, are signed by the program's vendor. Before the program is started, the signature is checked. Whenever a resource is accessed which is not on the list, an explicit permission by a supervisor component is requested.

The remainder of the paper is organized as follows: in the next section the effectiveness of current protection techniques are discussed, from which our requirements for a safe treatment of mobile objects are outlined. In section 3 wish lists, trust lists and capability lists are introduced in general. An implementation for downloadable programs (applets) is demonstrated in a Linux environment in section 4. Some time measurements and experiences are presented.

2 Current Techniques

The most well-known approaches to deal with malicious entities are rule based security policies, most notably the Multics interpretation of the policy formalized in the Bell LaPadula-model [8]. It "confines" the flow of information using a set of rules that must be enforced by the underlying system. However, experience showed that systems operated under this policy are extremely hard to administrate. In addition, only few such policies are known, and their practical usefulness is doubted, in particular in an environment where mobile objects roam global networks.

Firewalls filter transmission of certain classes of traffic on various protocol levels. Relay services try

*In proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS'97), Baltimore, IEEE Computer Society Press, pages 355-362, May 1997

to compensate the effect of the filter. Firewalls are very useful to protect a bank or a chain of locally distributed banks; however, they prevent per definition the use of downloadable programs or general mobile objects in these systems. Firewalls isolate systems and not malicious entities within systems. What we need are firewalls that operate within a system to protect it from malicious *visitors*.

The most effective practical approach is based on a careful administration of discretionary access control, e.g. in Unix systems using Access Control Lists (ACL). A new user or group is introduced for imported programs, and the ACLs are carefully set with respect to the new user. In principle, the approach can be applied to mobile objects. However, it probably will not work in practice. Since the number of visitors is unlimited, it is not possible to invent new users or groups in every case. A feasible approach is to identify classes of visitors and remap these classes to users or groups with varying access rights. But then, with the limited abilities of Unix ACLs, safe administration is not simple if possible at all.

A further approach, e.g. used in some of the various Java-engines, is to rely on interpretation and to restrict the accessible resources severely (sandboxing). The interpreter ensures that only a few standard resources are accessed, which is overrestrictive for many interesting applications. If arbitrary libraries can be linked to applets, an additional danger occurs. We will shortly discuss the interpretation approach to implement our technique in a later section.

Another proposed technique is that applets contain signatures that can be used to authenticate the creator of the objects and to ensure that they have not been tampered with. The rationale for this approach is that if something happens, the vendor of the program can be held responsible. But even if the originator of an applet is known, this does not imply that a *blanco check* should be given to that applet. Hence, mechanisms are still required to express various degrees of trust to certain applets or software-vendors.

Some operating systems architectures — pioneered by Hydra [6], CAP [7] and Amoeba [5] — relied on capabilities to encapsulate processes. The BirliX Security Architecture (BSA) provided mechanisms to encapsulate suspicious, e.g. imported, programs or processes. Programs as well as (active) objects carry their *subject restrictions* wherever they move to. However, the fundamental limitation of the BSA and similar research operating systems is its requirement for homogeneity. Only on systems running BirliX, the various mechanisms of the BSA are enforced. This paper describes an adaptation of BirliX subject restrictions so that they can be enforced in operating systems as long as they provide separate address spaces for user processes. As proof of concept, an implementation in a Linux environment is described.

An interesting technique for access control are message redirection schemes. A prime example is the *Chiefs and Clans* concept invented in the L3 microkernel [13]. In a later section of this paper, an implementation of our technique in such an environment is outlined. Again, the major drawback of these mechanisms is that they work in homogeneous systems only.

In this paper we will not discuss various efforts to deal with other threats in World Wide Web scenarios. E.g., we will not comment on the S-HTTP [12] proposals that try to solve authentication problems and to enable safe (commercial) transactions in the Web. However, we assume that safe ways are available to verify signatures as are proposed by these protocols. We claim that these protocols need to be complemented by safe techniques to encapsulate mobile objects or processes running downloaded programs.

Here is a summary of the requirements for the safe execution of applets or mobile objects:

- mobile objects must be effectively isolated after they enter a system
- in spite of the isolation, the mobile object must be given access to the resources it needs (e.g. files, network connections, ...)
- the owner/administrator should be able to express a fine degree to which extent he trusts a mobile object and its vendor
- administration must be simple
- it must scale, i.e. work with large numbers of visitors in a system
- it should also work in a heterogeneous network of systems.

3 The Principle of the Technique

The main idea of this paper is introduced by describing the sequence of actions undertaken by the participating vendors and system administrators.

First, a vendor of a program adds a list of symbolic names to the executable. This list contains names of all resources (files, connections, processes,...) that the vendor requires to be accessible to a process executing the program to perform its documented function. In addition, the program is given a name by the vendor. The program, together with the list and the name, are signed by the vendor. It is assumed that reliable mechanisms are available to safely verify the signature, e.g. to obtain the public key of the vendor. Hence, the result of this step is an executable program extended with a list of names denoting the requested resources. This list is called a *wish list* in the following.

The operator (owner) of a system decides, to what extent he trusts specific programs or their vendors. He may decide that all programs bought from foo-soft are trusted with regards to all directories in the subtree `~/windows`. On the other hand he may allow a shareware program to access just `~/tmp` objects and a virtual terminal. Hence, the result of this step is a database mapping vendors and programs to resources. This data structure is called a *trust list* of a program or a vendor in the following. If a vendor has no entry in the trust list, this trust list is the empty set. The intended use of trust lists for programs is to enable an operator to express a larger degree of trust into certain programs of a vendor. Hence, specific programs may have more rights than they get by default from their vendors.

When a program is started, the vendor of the program is determined and the signature is verified. Then, the wish list is checked to see whether or not it is a subset of the union of the program's or the vendor's trust lists. If it is a subset, the program can be started immediately. Otherwise a supervising component designated by the owner of the station is queried to add the names which are not in the trust lists. As result of this step, a working set of usable objects for a process is created which is used as a *capability list* (see figure 1).

In the following, when we refer to wish lists, trust lists, and capability lists, we will use $*$-lists.

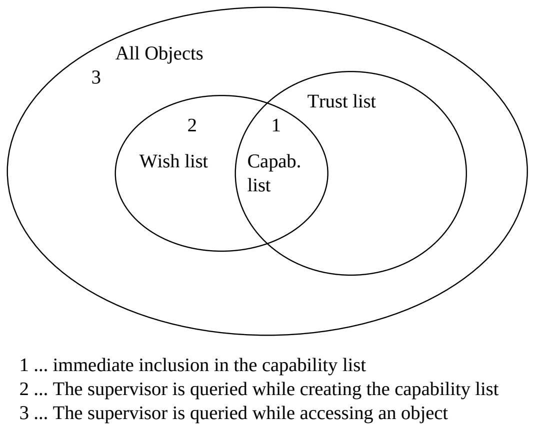

Figure 1: $*$-lists

Names in trust lists are given in simplified regular expressions[1]. Figure 2 shows an example of the use of $*$-lists. Foo-soft's trust list permits read access to all resources whose names are prefixed with `/pub/docs`. The wish list of the applet requests access to `/pub/docs/reports+`. Hence, the resulting capability list contains read access rights to `/pub/docs/reports+`. For many practical purposes, aliasing has been added. In the shown example DISPLAY is contained in the wish list and the trust list. The trust list aliases it to enable access to a connection named `georg:0.0`. Another useful application of aliases are negative access rights.

During the execution of the program, the capability list is checked each time a resource is accessed. If the accessed object is a member of the capability list, the access is granted, otherwise the supervising component mentioned above is queried. It either permits the access, denies it or substitutes another resource.

If a process starts executing another program, again the wish list is verified and inspected. The capability list of the new process is the intersection of the capability list of the active process, the trust list of the new program and the wish list of the new program. Therefore the new program can not have more rights than the old; this will be motivated in the description of the Linux integration.

The principle technique can be applied at several points in a system's architecture. The obvious and simplest place are interpreters or compilers for languages or abstract machines that are designed to support downloadable applets and mobile objects (e.g. Java engines). While this approach has the obvious advantage of easy portability, it has two major drawbacks:

- It limits the range of applications to programs and mobile objects for these abstract machines and
- it requires that the technique is reimplemented for all languages.

These drawbacks can be avoided by incorporating $*$-lists into the system call interface of operating systems. In other words, the enforcement of $*$-lists is based on hardware (address space protection) and a small piece of software (the added procedure at the system call interface). This has been done successfully by the authors for a prototype on a Linux kernel. In addition, we have studied the use of modern micro-kernel technology to incorporate $*$-lists as user level programs.

[1] Currently we use two different wildcards. '+' allows accesses to all files and subdirectories of the given directory, '$*$' only to files of the given directory.

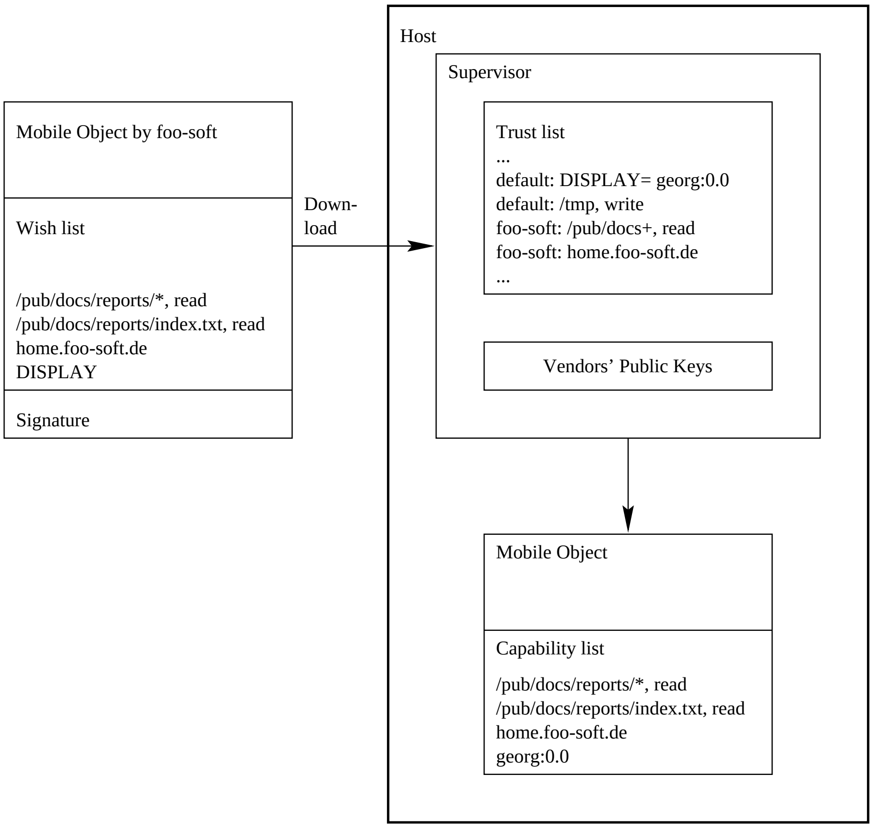

Figure 2: An example

4 An Implementation on Linux

To incorporate ∗-lists in an operating system, we require that the system is capable of the isolation of processes and and provides a possibility to intercept accesses to resources. In the following, we discuss how ∗-lists are incorporated into a Linux[2] environment.

4.1 Principles

Figure 3 shows the process structure of a system handling downloadable suspicious programs. First a supervisor process is created that maintains the trust lists. It causes the Linux kernel to intercept the relevant system calls of all processes created (forked) by the supervisor and to check if the accessed resources are included in the capability list of the process. The first process created (via fork/exec calls) by the supervisor is a handler for downloadable programs, e.g. a network browser. The exec operation to start it is intercepted and the capability list is created. The handler may then start to download and execute arbitrary programs. This situation is shown in figure 3.

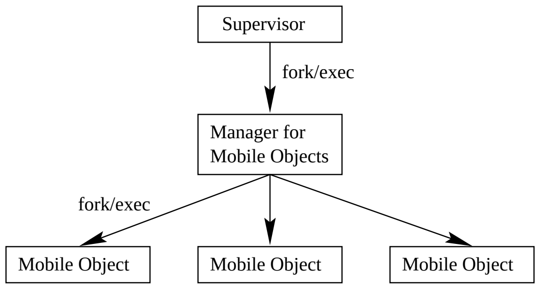

Figure 3: Process structure for the execution of Mobile Objects

Some programs, e.g. an editor, need very liberal wish lists. Hence, if a suspicious process creates a new process running such a program, the capability list of the new process must not have more rights than present in the current capability list (see figure 4(a)).

[2] We used a version of the Linux system that has been developed in our group, where the Linux kernel runs as a user level program on top of the L4 microkernel [15]. We have however made no use of the particular properties of that version. The changes we made can be repeated in the original version of the Linux kernel. The sole reason for using this system is that it is the current workbench for research within our group.

Otherwise, the suspicious process would be able to access resources via IPC with the new process. Hence, the policy implemented by the supervisor does not allow for the amplification of access rights based on liberal wish lists. If by contrast a program is needed with more access rights, the suspicious process must negotiate with another process (i.e. the Manager for Mobile Objects in our example) to create this process (see figure 4(b)).

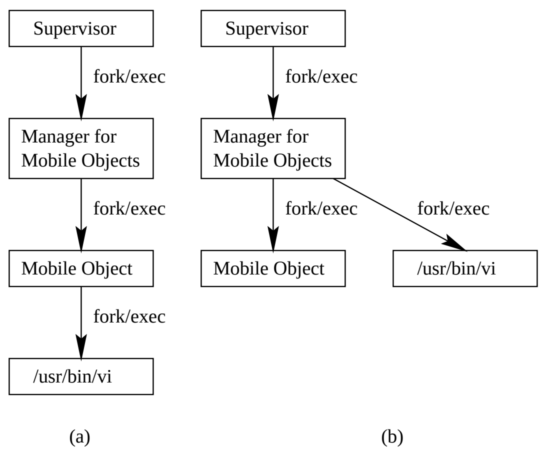

Figure 4: Passing on capability lists

4.2 The general structure

To adapt the described technique to a specific environment, these questions have to be addressed:

- which resources are controlled
- how are these resources named
- how are operations intercepted
- how is the supervising component integrated
- which is the standard environment required by all processes

The answers to these questions are fairly straightforward for Linux.

In this paper, we distinguish between three kinds of logical resources: files, network connections (through sockets) and processes. Files are named using symbolic pathnames building a hierarchical name-space. Network connections are established using protocol dependent names (e.g. IP-addresses). Processes are named using `/proc` pathnames or `PIDs`. Currently we do not regard physical resources such as CPU-time and main memory (see section 7).

Processes are isolated by the Linux kernel using the hardware address space protection mechanism. All

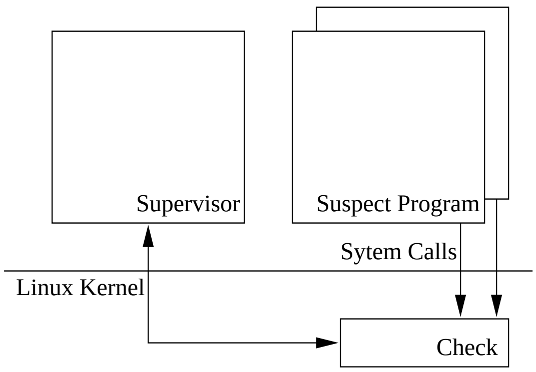

Figure 5: General structure

accesses to files or sockets are made via system calls which are treated in a single dispatcher routine in the Linux kernel. This routine is used to intercept system calls.

The capability lists are carried by the Linux kernel, the supervisor can replace an existing capability list using IPC.

In our prototype implementation, the supervisor has these functions:

- When an `exec` system call is intercepted, the Linux kernel sends the capability list of the executing process and the program to be started to the supervisor. The supervisor verifies the signature of the new program, builds the capability list and returns it to the kernel.
- If a program tries to access an object which is not listed in its capability list, the supervisor grants or denies the permission to access this resource. It makes the decision using arbitrary algorithms (e.g. it queries the owner).

The supervisor is implemented as a user-level process (see figure 5).

4.3 Relevant system calls

Obviously, `open` system calls have to be intercepted, the pathname checked against the capability list and the supervisor queried for permission if the name is not in the capability list. Currently, we return an error (permission denied) to the calling process if the permission is not granted.

In the capability list, only absolute pathnames are used. When a relative pathname is used as an argument we append it to the name of the current working directory. To this purpose, the current working directory is also maintained symbolically. The `fchdir` call changes the CWD of the process using a file descriptor as parameter. Hence, we had to intercept it and to maintain the symbolic name for each file descriptor in a task structure.

Symbolic links in conjunction with the Unix `'..'` convention create problems. They prohibit simple pattern matching checks in system calls using pathnames. Hence, in the current prototype we do not allow symbolic links.

With regard to network connections, we chose a relatively simplistic approach. We allow as names in $*$-lists just symbolic IP-addresses (e.g. 141.76.20.100) and UNIX domain names. More elaborate, user-friendly variants would allow higher level names, e.g. DNS names. The names are checked in intercepted `connect, bind, sendto, recvfrom` and `accept` system calls.

The `exec` system call also uses symbolic names to specify the program to execute. The given name must be included in the capability list and additionally the `exec` operation to the file must be permitted. Before the new program is executed, the signature and the wish list are checked as mentioned above.

If a new process is created by a `fork` system call, the capability list of the old process is copied to the new process. This ensures that the new process can not get more rights than the old one.

4.4 Interaction with UNIX mechanisms

An interesting interaction of $*$-lists with UNIX mechanisms can be observed. If a trust list for a *setuid* program contains aliases, accesses by the setuid program are redirected too. E.g. a user can create a trust list including an alias for `/etc/passwd` and can than execute the `su` program to gain root privileges. Therefore the use of aliases for setuid programs needs to be forbidden.

4.5 Time Measurements

We measured a few system calls in our very crude implementation on a Pentium/133. Capability lists add approx. 8250 cycles (62 s) overhead to an `open` system call and 24 s to a `connect` call. A large portion (approx. 25 s) of the `open` overhead is due to string handling which is done using unoptimized standard routines. Hence, we do expect drastic improvements. But even given the present times, the overhead is not prohibitive to use for external downloaded programs and mobile objects.

However, the large overhead due to checking of signatures in the `exec` system call means that the usage of our technique for all programs in a system (including standard services such as a mail handler) cannot be expected unless caching techniques are found to reduce the amount of checking for local, trusted programs.

System Call	Using $*$-lists	Without $*$-lists
open	18500	10250
connect	45500	42250

Figure 6: Cycles needed for executing a system call

Hence we expect a usage of our technique in subsystems dealing with suspicious programs, e.g. in network browsers.

4.6 A concluding remark

The described integration of our technique in the Linux/L4 environment still allows a process to send messages to arbitrary processes. E.g., if a filesystem is present in a system besides the Linux kernel server, accesses to its files are not intercepted. The following section describes the usage of microkernel message redirection mechanisms to integrate our technique into a system. This would also solve the mentioned problem.

5 Integration in Microkernel-based Environments

Microkernel-based systems do not necessarily provide one central point for the interception of critical operations. E.g., in a multi-server emulation of a UNIX-like interface, user processes may send messages to any server. In that situation, all servers providing such critical operations need to be analyzed and adapted or a single interceptor task must be inserted between mobile objects and servers.

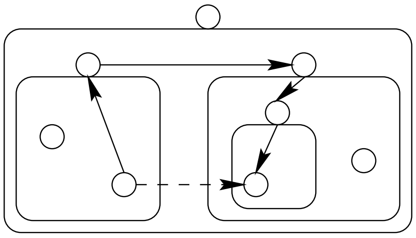

Figure 7: Chiefs and Clans

A specifically elegant technique to intercept messages is provided in the L4 kernel [14]. Tasks can be grouped into *Clans*. Each Clan has a *Chief*. IPC between Tasks of different Clans are redirected through the respective Chiefs (see figure 7), which can impose an arbitrary security policy on the Clan.

The incorporation of our technique in such an environment is straightforward. Mobile objects run as members of a Clan and each message to OS servers is intercepted by the Chief which acts as supervisor.

6 Related Work

The Janus project at the University of Berkeley [9] defines a secure environment for the execution of untrusted applications. A suspect program is executed under the control of a *framework* which monitors and restricts the performed system calls using the Solaris process tracing facility. System calls are inspected by policy modules which are initialized at the start of the framework. Therefore changing the security policy at runtime is difficult. However, the use of Mobile Objects does require the ability to dynamically change this policy for many practical reasons.

Independent of us and initial unknown to us, IBM developed the *FlexxGuard* system. It uses a technique similar to the $*$-lists to allow the secure execution of Java applets. FlexxGuard is a part of an implementation of the Java Virtual Machine, which limits the range of applications to Java applets, as already discussed in section 3. If an applets tries to execute a locally installed program, the execution needs to be forbidden because the program can not be controlled.

Access Control Programs (ACP) [11] are mainly used for the delegation of access rights to other processes or hosts in a distributed system. A server (e.g a filesystem) checks accesses of processes to objects using ACPs which are created by the owner of an object. Hence, Access Control Programs can be used to integrate $*$-lists in a distributed system which does not have a central point for the interception of critical operations.

7 Conclusions and Future Work

This paper presented an effective technique to isolate mobile objects within a computer system. Essentially, wish lists are attached to programs such that they are downloaded together with the applet or the mobile object and interpreted by the underlying system. The entries of the $*$-lists are symbolic, hence independently interpretable of the underlying system. The lists are protected using signatures.

$*$-lists are used to isolate processes which are running downloaded programs, e.g. applets. However, additional problems arise with truly mobile objects. Current techniques, e.g. BirliX subject restrictions, rely on network transparent mechanisms that are attached to migrating objects, hence are not portable. It is not yet clear to us, whether trust must also be present for previous hosts of mobile objects. The adaptation of $*$-lists to migrating processes requires a migration transparent interpretation and usage of $*$-lists.

The feasibility of the approach has been shown through an implementation in a Linux environment. However, the feasibility remains to be shown in systems with different naming conventions. To make $*$-lists a widely useful technique, naming conventions for different OS architectures must be agreed upon.

Another interesting area for research which has - to our knowledge - not yet been investigated in practical systems is the usage of capability techniques to limit physical resources for processes. On the basis of explicit scheduling techniques and external pagers of modern microkernels, the usage of physical resources may be controlled.

8 Acknowledgements

Wish lists are inspired by a talk given by Sape Mullender in St. Augustin. He proposed to add the names of used files to programs and transform them to Amoeba Capabilities [2].

The authors would like to thank Jean Wolter and Michael Hohmuth for their support in implementing $*$-lists in Linux/L4.

References

[1] D. S. Milojicic, M. Condict, F. Reynolds, D. Bolinger and P. Dale
Mobile Objects and Agents
Advanced Topics Workshop at the Second USENIX Conference on Object Oriented Technologies and Systems 1996

[2] Sape Mullender
The Amoeba Security Architecture
Joint INRIA, CWI, Workshop on Operating Systems
St. Augustin, July 1990

[3] Hermann Härtig and Oliver C. Kowalski and Winfried E. Kühnhauser
The BirliX Security Architecture
Journal of Computer Security,Vol. 2, 1993

[4] Oliver Kowalski and Hermann Härtig
Protection in the BirliX Operating System
Proceedings of the 10th International Conference on Distributed Computing Systems, 1990
IEEE Computer Society Press

[5] A.S. Tanenbaum, S. Mullender, R. van Renesse
Using sparse capabilities in a Distributed Operating System
Proceedings of the 6th International Conference

on Distributed Computing Systems, 1986
IEEE Computer Society Press

[6] W.A. Wulf, R. Levin, C. Pierson
Overview of the Hydra operating system development
Proceedings of the 5th ACM Symposium on Operating System Principles, 1975
Operating Systems Review

[7] M.V. Wilkes, R.M. Needham
The Cambridge CAP computer and its Operating System
Computer Science Library, 1979

[8] D. Elliot Bell, Leonard LaPadula
Secure Operating Systems: Unified exposition and Multics interpretation
Technical Report 2997, MITRE, March 1976

[9] I. Goldberg, D. Wagner, R. Thomas and E. A. Brewer
A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker)
Proceedings of the Sixth USENIX UNIX Security Symposium, 1996

[10] FlexxGuard, http://www.alphaworks.ibm.com

[11] M. M. Theimer, D. A. Nichols and D. B. Terry
Delegation through Access Control Programs
Proceedings of the 12^{th} International Conference on Distributed Computing Systems, 1992

[12] E. Rescorla, A. Schiffman
The Secure HyperText Transfer Protocol
Web Transaction Security Working Group
INTERNET-DRAFT, July 95

[13] J. Liedtke
On -kernel construction
15th ACM Symposium on Operating System Principles

[14] J. Liedtke
Clans & Chiefs
In 12. GI/ITC Fachtagung Architektur von Rechnersystemen, Kiel 1992, pp. 294-305

[15] M. Hohmuth, J. Wolter
Prinzessin auf der Erbse - Linux-Portierung auf den Mikrokern L4
iX Magazine 1/97