Operating System Protection for Fine-Grained Programs

Trent Jaeger, Jochen Liedtke, and Nayeem Islam
IBM T.J. Watson Research Center

Operating System Protection for Fine-Grained Programs

Trent Jaeger Jochen Liedtke Nayeem Islam
IBM T. J. Watson Research Center
Hawthorne, NY 10532
Emails:{jaegert|jochen|nayeem@watson.ibm.com}

Abstract

We present an operating system-level security model for controlling fine-grained programs, such as downloaded executable content, and compare this security model's implementation to that of language-based security models. Language-based security has well-known limitations, such as the lack of complete mediation (e.g., for compiled programs or race condition attacks) and faulty self-protection (effective security is unproven). Operating system-level models are capable of complete mediation and self-protection, but some researchers argue that operating system-level security models are unlikely to supplant such language-based models because they lack portability and performance. In this paper, we detail an operating system-level security model built on the Lava Nucleus, a minimal, fast μ-kernel operating system. We show how it can enforce security requirements for fine-grained programs and show that its performance overhead (with the additional security) can be virtually negligible when compared to language-based models. Given the sufficient performance and security, the portability issue should become moot because other vendors will have to meet the higher security and performance expectations of their customers.

1 Introduction

We demonstrate how operating system protection can be used to control fine-grained programs flexibly and efficiently. Operating systems use hardware-based protection to isolate processes from one another. However, the way that current operating systems implement this protection has caused researchers to deem them too slow and inflexible for controlling fine-grained programs. Fine-grained programs have different protection domains and may interact often in the course of a computation. The effect of a large number of protection domain crossings must be handled securely (i.e., correctly with respect to the security requirements) to prevent attacks and efficiently to minimize performance degradation. In this paper, we show that a security model implemented on a fast and flexible IPC mechanism can enforce security requirements that language-based systems cannot with little performance impact.

Several operating systems use hardware-based protection to prevent processes from inadvertently and/or maliciously modifying one another. Each process has an address space that defines a set of memory segments and the process's access rights to those segments. A process can only access memory in its own address space. In addition, the operating system has a security model that associates processes with their access rights to system resources. Using address spaces of suitable granularity and process access rights to controlled resources, an operating system can control a process's operations as desired.

However, operating system security models have been deemed to lack the performance and flexibility necessary to control fine-grained programs. While some systems have been built that efficiently control processes in dynamically-defined protection domains [3, 8, 14], these systems have been applied only to more traditional applications (e.g., PostScript interpreters). In an application composed of fine-grained programs, programs with different protection domains interact often (perhaps as much as on each method invocation). In a recent paper, Wallach *et al.* [25] discard address space-based protection from consideration for applications with fine-grained programs by noting that IPC between two COM objects on Windows NT takes 1000 times longer than a procedure call (230 μs to 0.2 μs). We claim that this discrepancy can be virtually eliminated while gaining security and maintaining flexibility.

We have developed a prototype implementation of a flexible security model for controlling downloaded content. This model is implemented on the Lava Nucleus. The Lava Nucleus provides address spaces, threads, fast IPC, flexible paging, and IPC interception that enable efficient and flexible control of processes. In this paper, we primarily concentrate on the effectiveness of the Lava nucleus for implementing a flexible security model and its resultant performance. We show that fast IPC and IPC interception enable the implementation of *dynamically authorized* IPC that can be performed in as little as 9.5 μs. We believe further room for optimziation is possible, given an ideal estimate for dynamically authorized IPC is about 4 μs. Also, flexible page mapping in the Lava Nucleus enables objects of size greater than the hardware page size to be shared among processes, so coarse-grained sharing of memory between processes is possible.

The structure of this paper is as follows. In Section 2, we compare language-based and operating system-based security models. In Section 3, we describe an operating system security model for downloaded executable content. In Section 4, we describe the implementation of this model on the Lava Nucleus. In Section 5, we examine the performance of the prototype implementation and compare its performance to language-based models for fine-grained programs of the sort discussed by Wallach *et al.* In Section 6, we conclude and present future work.

2 Language vs. O/S Protection

The basic problem is to implement a security kernel that effectively and efficiently enforces the security requirements of downloaded executable content. The basic requirements of any security infrastructure are that it can[1] (adapted from [2]):

- assign permissions dynamically to individual content,
- mediate all controlled operations using such permissions (i.e., an operation that a process is not unconditionally trusted to invoke),
- manage the evolution of permissions as content is executed,
- protect itself from tampering

For a system that uses dynamically downloaded executable content, permissions must be assignable to individual content. The security kernel must be able to mediate all controlled operations. To enforce least privilege on content permissions may be based on application state and evolve as application state evolves. The security kernel must be able to control this evolution within reasonable limits. Lastly, the kernel must be able to protect itself from modifications that may result in tampering with its behavior.

A question that has re-appeared recently is whether language-based or operating system-based protection is better suited for effectively and efficiently controlling such fine-grained programs. Or otherwise stated: to what extent can operating system protection *efficiently* provide *effective* security and to what extent can language protection *effectively* provide *efficient* security? As described below, operating system protection has several advantages over language protection from a security perspective, but the cost of domain crossings make it questionable whether efficient operating system protection for fine-grained processes is possible. On the other hand, language protection can be implemented efficiently, but some key security safeguards are weakened such that effective security may be lost.

Traditionally, operating systems have enforced system security requirements because hardware-based protection provides significant advantages in the key areas of economy of mechanism, fail-safe defaults, and complete mediation [23]. The operating system's TCB can protect processes by restricting them to their own address spaces which can be enforced by a simple mechanism (at least compared to a compiler). Since only the program requested is placed in the address space, other, independent programs are not affected by its failure (assuming the operating system adequately protects itself from such failures). Also, since the operating system can intercept any interprocess communication (IPC) between processes, controlled operations by a program (including compiled ones) can be completely mediated by the operating system.

Language-based protection has gained favor in recent years, however. We attribute this popularity to three factors: (1) improvements in the development of "safe" languages; (2) the perception that programs will become increasingly fine-grained, and fine-grained domains are prohibitively expensive to enforce; and (3) lack of flexibility in operating system security models. "Type-safe" languages are strongly-typed (i.e., all data is typed and casting is restricted or prohibited) and do not permit direct addressing of the system's memory (i.e., no pointers). Therefore, all data is accessed according to its interface, so complete mediation of controlled operations in pro-

[1] There is an additional requirement that any "security kernel" be simple enough that independent evaluators can assess whether it will operate properly. While we will not prove such a feature about our system here we do attempt to keep the security model as simple as is feasible.

grams written in such languages is possible. Other "safe" languages, such as Safe-Tcl [4, 21], Penguin [7], and Grail [24] depend on removal of operations that would enable the language security infrastructure to be circumvented. Also, with the increased popularity of component-based system infrastructure, such as Java Beans, and the ability to downloaded code dynamically, are leading people to predict that programs with become more fine-grained. Mediation of IPC between processes has significant performance implications for operating systems because they may need to perform expensive operations, such as handling TLB misses, upon domain changes. Also, the security models of current commercial operating systems, such as UNIX and DOS/Windows, lack the flexibility to dynamically assign permissions to user processes or permit controlled sharing of memory among processes.

Language-based protection has some significant weaknesses, however. First, the TCB of a system depending on language protection is larger because now we must trust the compilers and code verifiers as well as the "system" TCB objects provided by the system. While compilers are much better understood than they once were, a minimal TCB is still preferred. Second, since all programs run within a single address space, fail-safety is not what it is in operating systems. This means that a security flaw will result in the attacker gaining all privileges that the virtual machine has. Given that single domain operating systems based on language protection are being built (e.g., Java OS), this means that compromise of the virtual machine can result in significant losses. Third, it is not clear what the actual size and number of components that can share a protection domains will be. In Wallach *et al.*, they measured 30,000 domain crossings per second. It is not explicit in their paper, but we assume that many of these domain crossings involved trusted classes whose use have few security implications. Therefore, we believe that many of these claimed domain crossings can be avoided. Lastly, and most obviously, language-based protection is language-specific and does not apply to compiled code. Therefore, complete mediation depends on a homogeneous system which we believe is unlikely.

Also, language-based protection has its own performance problems and the optimizations to improve performance introduce subtle security flaws. For example, in the JDK 1.2 specification [10], excess authorizations (on every method invocation since there may be many real domains within a single protection domain) are prevented by using the call stack to determine the current authorization context. However, the current call stack may not represent the actual context since classes that are no longer on the stack may influence the execution. In addition, security depends on the proper placement of calls to the authorization object. For applications, this means that code must be changed to control a previously uncontrolled object.

The question is whether operating system protection can be made flexible enough and efficient enough for fine-grained programs. We believe that the flexibility question has been answered for a long time, but the systems for which it was answered are no longer in wide use. Several research operating systems were developed that used capabilities to attach flexible protection domains to processes in a secure manner [16, 20, 28]. For example, Hydra [28] obtained flexibility by attaching capabilities to procedures. SCAP [16] and PSOS [20] demonstrated that a number of security properties could be enforced by these systems. However, the applications of the time had much simpler security requirements, so much less secure systems were adopted as de facto standards.

We, therefore, believe that the key problem to using operating system protection for fine-grained programs is performance. At IBM, we are developing the Lava Nucleus which is a minimal, fast, and flexible μ-kernel. It provides the basic system constructs, such as processes, address spaces, and threads, and provides general mechanisms for using these primitives, such as flexible memory mapping primitives, fast IPC, and IPC redirection. In this paper, we show that a flexible operating system protection model for downloaded executable content can be designed and that it can be implemented efficiently using the Lava Nucleus. From this, we measure the currently achievable performance of operating system protection and evaluate how this affects the overall system performance. In the future, we expect that language-based and operating system protection to be jointly used (depending on the security requirements) with operating system protection providing a simple, fail-safe security mechanism that may completely mediate controlled domains.

3 Security Model

An effective security model for downloaded executable content requires significant flexibility in the creation of principals, assignment of their permissions, and management of their permissions throughout the content's execution. We use Lampson's protection matrix [17] to help describe these requirements. It shows the relationships among subjects, objects, and the operations that subjects can perform on objects (permissions). Now consider the security requirements for controlling downloaded executable content described below.

- **Subjects**: There may be one subject per downloaded content, although some subjects may be reused within a session.

- **Objects**: The objects may refer to logical objects that must be mapped to the individual downloading principals system at runtime.

- **Permissions**: The set of permissions that a subject has at any time to objects may evolve as system's content (subjects) executes.

In traditional systems, the set of subjects is generally mapped to the set of users and a set of well-known services. This granularity is too coarse for downloaded content. Each content may be associated with a different principal that is an aggregate of basic principals (e.g., downloading principal executing content from a provider within a specific instance of an application). Also, in traditional systems, the set of objects is well-known. This may not be the case in downloaded content systems because the content providers may have limited knowledge about the systems upon which their content is run. In addition, the amount of policy specification can be reduced if the logical objects that are mapped to a specific principal's domain at runtime can be used. Lastly, enforcement of least privilege on content is more important than for traditional programs because these programs are transient and from only partially trusted sources (i.e., may have bugs). In traditional systems, permissions are assigned to principals and must cover every execution of that principal, or else some application executions would fail. For content, the state of an application may also be used to limit the permissions to those necessary for the task at hand.

To solve these problems, we propose an security model that provides basic security mechanisms and policy representations to enable the control of downloaded executable content. The model is shown in Figure 1. In this security model, a *secure booting mechanism* loads a nucleus of an operating system. The *nucleus* provides the basic functionality to create and execute processes and enables them to intercommunicate (via IPC). The nucleus initiates a *process load server* that creates subsequent processes and can dynamically load libraries and code components into existing processes. The process load server uses an *authentication server* to authenticate content and determine the content principal. A *derivation server* derives the permissions for this content principal. The process load server assigns a monitor to enforce a process's permissions. *Monitors* intercept and authorize all IPCs emanating from or directed to the controlled process. Monitors maintain a representation of the controlled processes access rights and can both add to and revoke access rights from the controlled process.

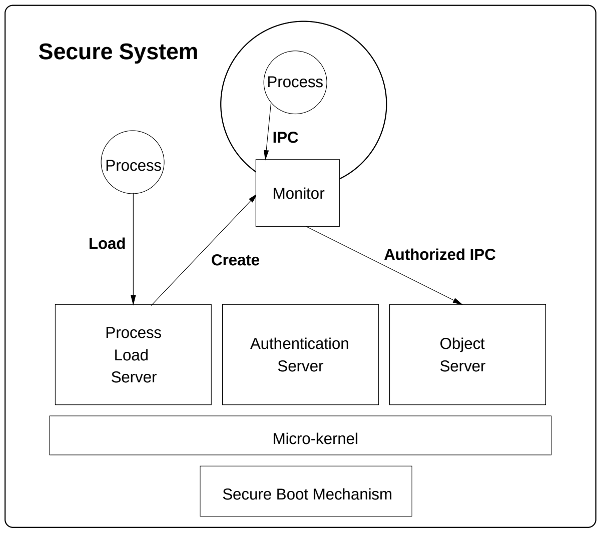

Figure 1: Security Model Architecture

In the design of our security model, we make the following assumptions. System administrators are completely trusted to set system policy. A set of certification authorities are trusted to vouch for the public keys of principals. A secure booting mechanism is trusted to initialize the operating system properly. We assume the existence of such a mechanism as proposed for the Taos operating system [27]. The kernel itself is trusted to create processes and threads properly, separate process address spaces, identify the source of IPC, and redirect the IPC of controlled processes to monitors. The authentication server is trusted to perform cryptographic operations correctly. The process load server is trusted to setup the processes and monitors properly, so the security requirements as specified can be enforced. Monitors are trusted within a domain limited by the rights that they can delegate to processes. Monitors may also be trusted to read and not leak server data to other processes. The system has limited trust in users, applications, and downloaded content.

The following three subsections describe how the first three of the security requirements of the system are enforced. The fourth security requirement's must be enforced throughout each of these operations.

3.1 Process Loading

The process load server receives load requests for objects and retrieves, validates, and loads the object. Some objects may be executable and some may not, but our discussion focuses on the loading of executable objects.

The process load server solves the following problems:

- retrieve requested executable content,
- uses the authentication server to verify the authenticity of content and derive the content principal,
- uses the derivation server to derive the content's permissions,
- find the process in which to load the content,
- load the content into that process

Our effort here is concentrated on a mechanism for loading executable content. We propose mechanisms and policy representations for authentication and permission derivation elsewhere [12, 13]. These mechanism enable permissions to be derived dynamically for downloaded content given limited input from multiple principals.

While IPC can be faster than 230 μs, it is still well understood that procedure calls are faster. IPC in the Lava Nucleus takes 4 times longer on a Pentium than a procedure call than Wallach *et al.* [26] measure for a PC. In addition, there are indirect costs that are a result of the context switch, such as the handling TLB and cache misses. While the Lava Nucleus is designed to enable these costs to be reduced, the fewer context switches the better.

To reduce these overheads, we want the ability to link supporting content in the requesting process. However, only links that ensure that both the requesting process and the downloaded content do not obtain any unauthorized access rights can be permitted. This is only possible if: (1) neither the downloaded content nor the requesting process gain any permissions as a result of the co-location; (2) the downloaded content is permitted access to the requesting process's data and vice versa; and (3) the downloaded content can run properly with the rights of the requesting process. For the first condition to hold, the permissions of the joint process must be the intersection of the permissions of the content loaded into it. Thus, neither process can use a permission unless both had it previously. Also, neither content may have data in its address space that it must keep secret from the other. In addition, the content and process must also be able to effectively perform their jobs with the resultant rights for the co-location to be feasible.

While these restrictions limit the content that can be loaded into the requesting process, a variety of useful content can still be downloaded and co-located. Trusted libraries can be co-located with the requesting process in many instances. For example, many Java classes in java.lang package (although not the Java ClassLoader whose functionality we are superseding) can be loaded into a requesting process. For example, the String class does not provide the user's process any additional rights (although it may be used to circumvent language-based security), so it can be loaded into the requestor's address space. Also, we think that all the classes in the java.io can be loaded into a requesting process, because restricted access to the file server can be enforced by the monitor. Of the 30,000 domain crossings per second measured by Wallach *et al.*, we expect that many of those do not really require a change in domain for the requesting process. Our experience with the FlexxGuard prototype system (a controlled Java interpreter) was that restricting the permissions of the Java system classes to that of the current applet being run still permitted many useful applications to be implemented [1].

3.2 Permission Management

A process's monitor also manages the evolution of its permissions throughout its execution. In general, permission changes occur for two reasons: (1) an application state change may warrant a change in the controlled process's permissions and (2) one content using another may result in a restriction of permissions to prevent information leakage. In the first case, a user may perform an operation that results in the delegation of rights to downloaded content. For example, the loading of a file into an application may result in the delegation of the permissions to read and write the file to content used in the application. These permission changes must be limited to prevent a process from obtaining an unauthorized right. Also, the interaction of two untrusted content processes may require that their permissions be intersected to prevent the callee from performing unauthorized operations on behalf of the caller.

Permission management requires:

- the ability to add rights to the current permissions,
- the ability to revoke rights from the current permissions,

- a mapping of events to permission transformations,
- a limit to the rights that can be delegated to content,

In this security model, a monitor can add a right to a process's permissions if: (1) the delegating process has the permission; (2) the delegating process is permitted to delegate that permission to the delegatee; and (3) the permission is within the *maximal permissions* for the process. Each monitor maintains a mapping of its processes to its delegating principals and permissions (called *assignment limits*) that that principal can delegate to this process. Any delegated right must be within the process's maximal permissions (both the initial current permissions and maximal permissions for a process are derived by the derivation service). This limits the rights available to a process in general.

Revocation of rights is more difficult because capabilities can be copied. To prevent a process from using a revoked capability, the monitor does not pass capabilities to its controlled processes. Instead, a descriptor is returned to the process which enables it to refer to the capability. Revocation of the capability invalidates the descriptor, so capabilities can be immediately revoked. Unlike Redell's capability indirection [22], no special capabilities are seen by the servers and the Java and UNIX APIs can be supported transparently. To revoke capabilities forwarded to other processes (actually their monitors), the monitors must maintain a mapping of capabilities to the processes/monitors that they were forwarded to. Servers must maintain a similar mapping.

Mapping events to permission modifications is done by *transforms* [13, 15]. Transforms map operations to permission transformations. Three types of transforms are defined that implement different methods of transformation (transformation by permission, transformation by membership to an object group, and transformation by combining permission sets). See Jaeger *et al.* for more details [13].

3.3 Process Mediation

Complete process mediation requires that each IPC be authorized by a monitor. We prefer that monitoring be triggered automatically. Otherwise, it may be possible to a programmer to forget to call the monitor (and lose complete mediation). Therefore, there must be a mechanism for redirecting IPCs to the monitor. Then, the monitor must be able determine what permissions are to be authorized. These permissions are authorized by the monitor, and if successful, the requested is forwarded to the destination.

Process mediation requires that the following tasks be accomplished:

- configure the system such that monitors can authorize all controlled operations,
- describe the authorization semantics of operations well enough to enable their authorization,
- provide monitors with the process's current permissions to authorization

Each IPC must be intercepted for complete mediation. Therefore, it is necessary to place a monitor on each IPC path. Typically, monitoring is associated either with the server (which enforces access control on its clients) or on the client (limits access of the client). There are limitations to both approaches. In client monitoring, monitors can, in theory, enforce arbitrary security policy, but in practice they have limited knowledge about the servers to which they are controlling access. On the other hand, servers are typically trusted to enforce security requirements on clients, but they may not understand the security requirements that the monitor is trying to enforce.

We choose a security model that enables both kinds of control, but erlies more heavily on client-side monitoring. Each process may have a monitor that can enforce both its incoming and outgoing IPCs. Therefore, a process that is both a client and a server in different interactions can have its operations to servers authorized and can restrict the operations that it perform for its clients. We do emphasize the client-side control of the monitor by enabling it to place tighter restrictions on the operations it can perform that server monitors may. An additional benefit that results from this model is that monitors themselves can be restricted to different domains because they are different processes. We overcome the problem of server security requirements for processes, by enabling the servers to delegate permissions to content and providing authorization semantics of server operations to the monitors (see below and Section 4.3).

A result of this decision is that an IPC from one controlled task to another requires an additional IPC between the two monitors. However, Lava Nucleus IPC and operation authorization should be fast ($<$ 1.5 μs for small IPCs), so the benefit can be gained at low cost. It is unclear yet whether the cost is worth the added security, however.

The authorization semantics of a server's interface is defined by *operation authorization* objects (see Section 4.3). These objects are used to transform oper-

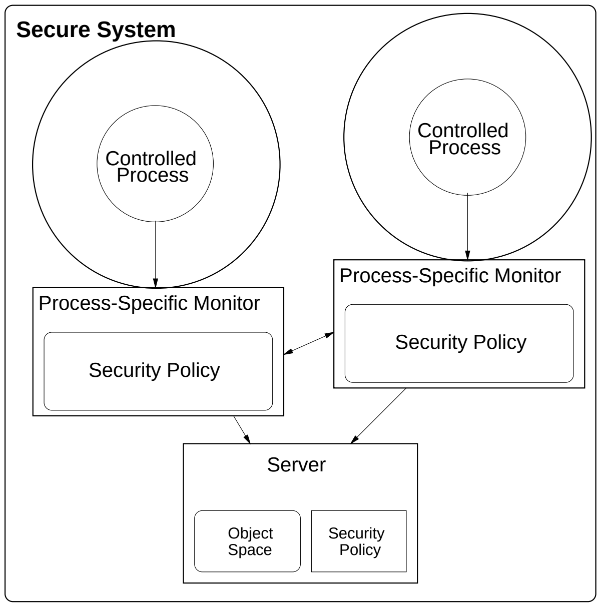

Figure 2: Monitor Architecture

ations into the set of permissions that must be authorized before the operation can be run. These are useful for enforcing least privilege with good performance (we have found the cost of processing these is low).

The monitor obtains authorization operation from the server definition (e.g., via an IDL extension). When a server is loaded, its operation authorization objects are stored in a place accessible to all monitors. The semantics of these operations must be well-understood. Therefore, monitors control client operations (based on the current and maximal permissions) to any server that they are permitted to access.

Our architecture for using monitors and servers to control processes is shown in Figure 2. In this model, a monitor is assigned to each controlled process. When a process makes an IPC, its monitor intercepts the IPC automatically via a kernel-provided mechanism. After the operation authorization semantics have determined the operations that need to be authorized, the monitor uses its process's permissions to authorize the operation. The operation must be within the content's current permissions and maximal permissions to be authorized. Since the content's current permissions may expand and the content's maximal permissions may be restricted, an operation at a certain time may need to be checked against both.

4 Implementation

The security model is implemented upon the Lava Nucleus. The Lava Nucleus provides minimal, general, and efficient functionality for building operating systems. It enables the creation of tasks (i.e., processes) with potentially-overlapping address spaces that may contain multiple threads of execution. An optimized IPC mechanism is provided for intertask communication. The Lava Nucleus also provides a mechanism for IPC redirection which we use to redirect controlled operations to our monitors.

The prototype implementation of this security model is as follows. The Grub boot loader loads the Lava Nucleus and the root Lava task. This task bootstraps the memory system and provides some basic system functionality (e.g., page-fault handling and task-id creation). This task initiates the process load server and the core system services (e.g., a network server to download components and the downloading principal's task). In general, these tasks may also have a monitor assigned to them, but do not at present. The downloading principal's task may request that a new executable task be downloaded by asking the process load server to retrieve the task's content. The process load server has the code authenticated and derives its permissions and transforms. Depending on the load option, the process load server assigns a monitor task to control the new executable. The monitor starts the new executable (or restarts the requestor if loaded into the same address space). Monitors perform both the permission management and authorization services for their tasks using transforms and permissions, respectively.

In this paper, we focus primarily on the implementation of the architecture's monitors. The monitors store the current and maximal permissions of its content, implement its content's permission transformations, intercepts IPCs that are sent by or destined for the its content, determines the authorization requirements of the operation encapsulated in the IPC, and authorizes the operation using the content's permissions. We first describe the monitor's permission representation and how it enables flexible and efficient authorization. Next, we detail the Lava Nucleus's IPC redirection mechanism. Then, we outline how an IPC is converted to the set of operations to be authorized. Lastly, we detail the authorization mechanisms used by the monitor. The monitor uses two mechanisms: a slow one for "binding" to an actual object and a fast one for subsequent calls to the same object.

4.1 Principals and Permissions

Each process in our implementation is associated with a principal data structure. A principal contains references to its complex identity, current and maximal permissions, composition mode, and transforms. The *complex identity* is the composition of principals used to form this principal (e.g., the downloading principal, content provider, application, application role, and session). Current and maximal permissions are as described above. A *composition mode* defines how the permissions of the caller transforms the permissions of this content. For example, a trusted principal may union the current permissions of the caller with their own. However, content not trusted to leak permissions may maintain its current permissions and intersect its maximal permissions. The composition mode is designed to implement permission set transforms (intersection or union of the permissions of two or more principals) efficiently, but other transforms are implemented in a traditional manner because they are not highly performance critical.

To implement both permanent and transient permission set transforms, both the current and maximal permissions consist of static and lexical permissions. Static permissions are the content permissions that apply each time the content is called. Lexical permissions modify the content's permissions based on its current call context. Both enable permissions of other principals to be unioned or intersected with those of this principal (either permanently using the static permissions or temporarily using the lexical permissions). Given that a composition may result in a union, intersection, or no change to the maximal and current permissions either statically or lexically, 24 composition modes are used. Compositions are always performed using the calling principal's static permissions to prevent huge concurrency control overheads that would be likely if lexical permissions were used.

The current permissions are divided into two categories: authorize and active capabilities (maximal permissions are only authorize permissions). Capabilities are software-managed objects stored by the monitors that are unforgeable, revocable, strongly-typed mappings from object identifiers to operations. As we will discuss below, bind operations establish a capability for the content to perform a set of operations on an object. These operations are authorized using *authorize capabilities* that may grant access to more rights than are required for the operation. The result of a bind operation is the generation of an *active capability* that specifies exactly the rights authorized by the bind. Active capabilities enable fast authorization.

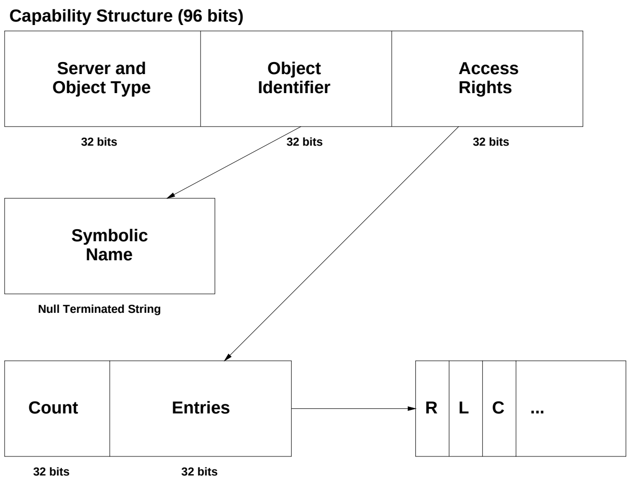

Figure 3: Capability format

Both types of capabilities use the same representation (shown in Figure 3): a server, a type, an object identifier, and the capability's rights. The server and type are captured in one 32-bit field. The server refers to the Lava task identifier for the server (12 bits is the Lava-specific limit). The remaining 20 bits are used to indicate the object type. An *object identifier* specifies the name of the object (reference to a string) or an object reference. References to names are word aligned, so the least significant bit is used to differentiate between the name pointers and OIDs.

The *capability's rights* indicates the operations that the capability grants. Again, a 32-bit field is used. There are 3 status bits, so 29 operations can be granted by default. One status bit signifies that an extended capability rights field is used. Using this field, access to an arbitrary number of rights entries (specified by *count* and *entries* reference) is possible. Each rights entry specifies the operations it applies to (the **R** field) and any limits on the use of these operations (the **L** field with the current count in the **C** field). Another status bit indicates whether the capability indicates a positive or negative access right. The third status bit indicates whether the capability is verified or not. For example, a change in the process's access rights may require a re-authorization of a capability.

Authorize capabilities are unforgeable because monitors will only accept them from process load servers or other monitors (i.e., processes trusted to provide them). They are revocable because the controlled processes never have access to them. Active capabilities derived from them can be revoked, as described below. They are clearly strongly-typed since they have a dedicated type field. However, the servers must enforce this

Active capabilities are formed from the results of the bind operations. For example, the OID is obtained from the object server for fast direct access. The capability's rights are set to those authorized in the bind operation. In addition, active capabilities may have an additional field that stores a unique identifier for the controlled process (called *principal identity*). This field could include a cryptographically secure number that the server can use to identify the calling process in a distributed environment [9].

Active capabilities are unforgeable on a single machine because the monitor is trusted to send them only to the proper server and the kernel is trusted to implement this delivery. In a distributed environment, the capabilities are unforgeable if a cryptographically secure principal identity is sent via a secure channel (i.e., that authenticates the sender). Active capabilities are revocable because they are never given to the controlled process. Instead, they are stored in the monitor, and the only index of the capability (e.g., a descriptor) in the active capability table of the principal is returned to the controlled process. We will see that fast IPC makes this indirection tolerable. Revocation of active capabilities as the result of a revocation of an authorize capability is also possible. Basically, all active capabilities can be marked unverified, so they must be re-authorized before they can be used. Therefore, the monitor must be able to retrieve the original object name (e.g., by caching it).

4.2 IPC Redirection

The Lava nucleus provides a general mechanism that can be used for implementing security policies based on IPC redirection. A monitor can be assigned to multiple processes. Any IPC to a process that is administered by another monitor is automatically redirected to this process's monitor which can inspect and handle the message. This can be used as a basis for the implementation of mandatory access control policies or isolation of suspicious processes. For security reasons, redirection must be enforced by the kernel.

A *clan* is a set of processes (denoted as circles) headed by a monitor. q Inside the clan all messages are transferred freely and the kernel guarantees message integrity. But whenever a message tries to cross a clan's borderline, regardless of whether it is outgoing or incoming, it is redirected to the clan's monitor. The monitor may inspect and/or modify the message. Clans may be nested.

Figure 5 shows a monitor which is used to enforce

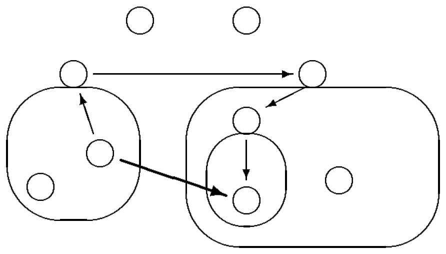

Figure 4: IPC Redirection ("Original" IPC is denoted by thick lines, redirected IPC by thin lines)

the security policy. All server requests from the en-

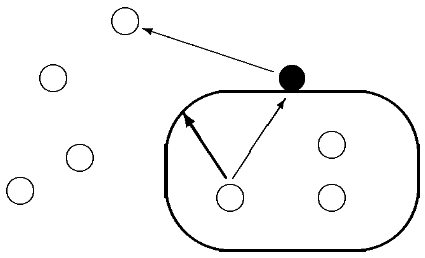

Figure 5: Security-Policy Monitor

capsulated tasks are inspected by the monitor (filled circle). The monitor drops any request which would violate the security policy. In particular, it uses accounting mechanisms to restrict denial-of-service attacks. Note that all page-faults and mappings are also handled by IPC. Therefore, the according resources are also under the monitor's control.

Instead of enwalling suspicious subjects, monitors can also be used to protect a system from suspicious subjects outside the own clan. In figure 6 the monitor

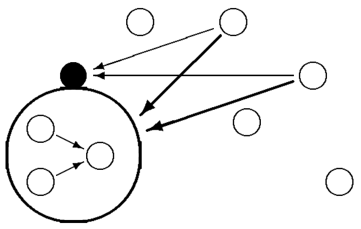

Figure 6: Attack-Blocking Monitor

(filled circle) inspects all messages coming from the outside and drops messages that cannot be authenticated or do not come from trusted partners. Furthermore, the monitor could encipher sensitive messages automatically (i.e., implement secure channels for its clan members).

4.3 Operations

Monitors need to be able to determine how to authorize an operation intercepted in an IPC. Given that new processes may be loaded with new interfaces, the monitors need a standard mechanism for deriving authorization requirements from interface information. This mechanism must be able to determine whether the operation is a bind or active operation, the type of object to which the operation applies, and which operands need to be authorized and the operations for each. The latter is particularly complex for bind operations because the real authorization requirements for the operation are placed in an operand.

We define a object for representing the authorization semantics of an operation below.

- **Definition 1**: An *operation authorization* defines the authorization information for an operation using the following fields:
  - **A Type**: Flag that determines the mechanism used to authorize the operation
  - **Number of Operands**: The number of operands in the operation
  - **Operand requirements vector**: A vector of authorization requirements for each operand consisting of the following fields:
    * **O Type**: Object type for the operation
    * **Ops operand**: Index of the operand that defines the operations to be authorized
    * **Op vector**: A vector of operations to be authorized indexed by the *ops operand* value above or the default value

The actual operation authorization entry used is retrieved from an operations table accessible to all monitors. This table is updated by the process load server (via add only, so no concurrency problems exist). The server and operation number are used to retrieve the entry. The *a type* field indicates the operation type (bind or active) and whether the zero operands, the first operand only, or another authorization is required. This enables fastest path code to be used for authorization. The *operands requirements vector* lists the authorization requirements for each operand. The *o type* specifies the type of the operand. The *ops operand* specifies the operand that determines the operations that are to be authorize. In a file *open*, the second argument determines whether the open is for read, write, and/or append. The *ops operand* may identify that either: (1) the operation requested is to be authorized; (2) a new set of operations are to be authorized; or (3) the operations to be authorized are determined by the value of an operand. For the third case, the *op vector* maps the *ops operand*'s value to the operations to be authorized.

This mechanism should suffice for many UNIX system calls and method invocations. For example, in file *open* and socket *connect* system calls only one operand needs to be authorized. In an object-oriented system, the first operand refers to the only object being operated upon, so only the operations on that object need to be checked. Other objects are passed as OIDs and cannot be accessed unless one of their methods is invoked (and authorized, if necessary).

4.4 Monitors

All IPCs (e.g., page faults, system calls, and RPCs) from the controlled process are automatically redirected to its monitor. The monitor uses its operation authorizations to determine the actual operations to be authorized on the operands. Operations authorizations and principals are stored in two tables shared by all monitors, so they can access any operation's authorization semantics and authorize operations against any principal (which is necessary if a principal's permissions can be lexically modified). Authorization is done using authorize capabilities for bind operations and active capabilities for operations subsequent to a bind.

When a monitor intercepts an IPC, the monitor retrieves the operation authorization to determine the operation's authorization type. Bind operations, such as file system *open*, require that the operation be authorized using authorize capabilities. Any one of the authorize capabilities for that object type may apply, so multiple capabilities may need to be checked. On the other hand, active operations, such as file system *read*, are authorized using the active capability whose index is specified in the operation. Upon a response to a bind operation, an active capability is created which is stored in the principal's active capabilities table. An index to this entry is returned to the controlled process for subsequent use (i.e., a descriptor/OID).

In either case, the operands must be copied into the monitor's address space. Lava supports fast and secure copying of data in IPCs, so such copying can be done automatically [18]. On a 166Mhz Pentium with a 256K L2 cache, up to 8 bytes can be sent in 0.95

μs. 128 and 512 byte messages can be transmitted in as fast as 1.79 μs and 2.60 μs, respectively.

The redirected IPC is then forwarded to the destination where it may be intercepted by that process's monitor. Currently, monitors only authorize outbound operations, but this monitor could restrict the IPCs that can be forwarded from specific sources. For example, if a process's IPCs result in an excessive number of errors (specified by a limit), then the monitor may retract the principal's permissions (e.g., via a transform executed upon an authorization failure).

Once an operation is executed, its results are returned to the controlled process via an IPC (through the monitors). The monitor also intercepts this IPC and may authorize its return. For example, limits on response "operations" can be specified. As yet, we have not exploited this functionality. Currently, the monitor creates active capabilities from the return values of bind operations and returns the active capability descriptor to the caller. For active operations, the return value is simply set in the return value register.

To further improve performance monitors are implemented using special Lava nucleus tasks, called small-address-space tasks. These tasks do not require a TLB flush upon a context switch, so TLB miss costs on a context switch between two small-address-space tasks are reduced (only one TLB miss for the IPC path). We expect that all monitors will be implemented as small-address-space tasks. However, the current implementation of Lava would not support using small-address-space tasks for many small content processes as well, because the number of such tasks is limited to a cumulative address space size of 512 MB in this Lava version.

5 Performance Results

In this section, we measure the performance of the security mechanism as implemented by the monitors. We first make micro-benchmarks of the individual steps in the monitoring mechanism. We then estimate the ideal performance (ignoring effects of TLB and cache misses) of authorized IPC from these benchmarks. We estimate that an authorized IPC (one-way) using active capabilities could be as fast as 4 μs (and typically less than 9 μs ideally). In our current implementation, our fastest one-way IPC is 9.5 μs (authorized using active capabilities). If we have 30,000 average authorized IPCs per second and 10% of these are bind operations (which is a very conservative estimate), then the ideal performance cost is about 20%. We have measured the cost of 30,000 actual active IPCs per second to be 30-40% (9.5 μs per IPC). These numbers are well below the 600% to 800% performance cost for IPC alone for COM objects on Windows NT. Since compiled code may be executed in the remaining 60-70% of the time, the performance impact of IPC interception relative to language-based models may be negligible. In addition, we believe that the number of IPCs can be reduced significantly by judicious code placement (taking security requirements into account) and precreation of active capabilities.

For our performance analysis, we have measured the costs of monitoring operations on a 166 MHz Pentium PC with 256K L2 cache. In our measured scenario, we have two controlled tasks, and each has a monitor that authorizes its IPC. The controlled tasks ping-pong requests back and forth, and we measure the time it takes for the authorized IPCs.

When a controlled process calls an operation the following actions are taken to authorize and forward the operation to its destination task.

1. Prepare the IPC to the destination with the operation data.
2. Send an IPC to the destination that is redirected to its monitor.
3. Determine the authorization requirements for the requested operation.
4. Authorize these requirements for operation and operands.
5. Source's monitor forwards IPC with the operation to the destination that is redirected to the destination's monitor.
6. The destination monitor forwards the IPC to the destination (no control on operation requests is enforced yet).
7. Create active capability descriptor (optional, for responses).
8. The destination receives the IPC.

Operations 1 and 8 are trivial and simply prepare to send an IPC or receive the IPC. Operations 2, 5, and 6 are all basically IPC operations (perhaps with data copying). Operations 3, 4, and 7 implement our authorization mechanism. The costs of all operations except 3 and 4 are fixed for messages of the same size. Therefore, we first list the performance costs of the fixed operations. These values are shown in Table 5. As shown, the fixed costs for monitoring in this configuration vary from 3.03 to 7.98 μs depending on the amount of data to be copied.

Operation	8-byte IPC	12-byte IPC	128-byte IPC	512-byte IPC
1. Prepare IPC	0.12	0.12	0.12	0.12
2. source-monitor IPC	0.95	1.37	1.80	2.60
5. monitor-monitor IPC	0.95	1.37	1.80	2.60
6. monitor-dest IPC	0.95	1.37	1.80	2.60
8. Receive data	0.06	0.06	0.06	0.06
Fixed intercept cost	3.03	4.29	6.58	7.98

Table 1: Performance of fixed interception actions (all times in μs)

A response to an operation goes through the same path, except that a new active capability descriptor may be created for a bind operation (e.g., file *open*). We measured the cost of active capability descriptor creation for a UNIX file descriptor to be 0.69 μs. Cost is kept low by pre-allocating (and reusing) the memory for these descriptors. Of course, active capabilities may be created at content load time to avoid bind operations.

The cost of deriving the authorization requirements is based on the costs of retrieving the operation authorization, determining the operands to authorize, and determining the operations to authorize upon the operands. We compare the costs for evaluating the operation authorizations for two operations: UNIX-style file *open* and file *write*. The *write* operation is an active operation in which only the first operand is authorized for write permission. Therefore, an operation authorization's *ops operand* indicates that the *write* operation is to be authorized and the *a type* indicates that active capabilities are used to authorize only the first operand (0.41 μs). The *open* operation is a bind operation in which the third operand indicates the actual operations that need to be authorized upon the first operand. Therefore, the operation authorization's *ops operand* indicates that the third operand's value determines the operations to authorize (using the *op vector*) and the *a type* indicates a bind operation in which only the first operand is authorized (0.48 μs).

Both the Lava Nucleus and language-based security systems must authorize bind operations (e.g., file *open* and socket *connect*). Language-based systems do not authorize further use of the resultant descriptors (i.e., access operations), so they cannot revoke them. In Table 5, we show the costs of authorizing using both authorization (for a *bind* operation) and access capabilities (for an *access* operation). As will be the case in language-based systems, the cost of verifying operations using authorization capabilities varies based on the size of the object name string (e.g., file path) and the number of authorization capabilities that are examined. In this example, the authorization capability verification does not include additional actions, such as checking inode information. We would expect similar performance for authorization in language-based system given that both systems are optimized.

Scenario	*bind* μs	*access* μs
1 cap-2 byte name	1.70	0.44
10 caps-2 byte name	12.84	0.44
50 caps-2 byte name	56.22	0.44
1 cap-21 byte name	3.50	0.44
10 caps-21 byte name	30.62	0.44
50 caps-21 byte name	138.84	0.44

Table 2: Performance of authorization

Authorization using active capabilities also includes the time for retrieving the active capability from the descriptor (approximately 0.20 μs).

Table 5 summarizes the performance of the Lava security model using address space protection. For access operations, IPC costs range from about 4 to 9 μs depending on the amount of data to be copied. Given 30,000 4 μs IPCs per second, a 12% overhead on processing is incurred. This percentage can be reduced by the percentage of IPCs that can be eliminated by linking content in the same address space. Bind operations can incur a much greater cost (6 to 150+ μs). However, language-based implementations also need to perform the same bind operations (either at load time for a new class or access time for system objects), so the operating system implementation should be faster. In general, since active capability creation is a reasonable cost operation, where possible, it should be used to eliminate unnecessary bind operations.

The actual performance of the entire IPC path for an active operation using 8-byte IPCs is 9.5 μs if the monitors and controlled processes are small-address-space tasks and 14 μs if only the monitors are small-address-space tasks. In the small-address-space case, the additional costs are incurred by the 22 cache misses on data and an slightly higher IPC cost for redirected IPC than the basic IPC benchmarked

Operation (data size)	*Fixed Costs*	*Auth Reqs*	*Auth*	*Cap Create*	*Total*
Active (8 bytes)	3.03	0.41	0.44	0	3.88
Active (128 bytes)	6.58	0.41	0.44	0	7.43
Active (512 bytes)	7.98	0.41	0.44	0	8.83
Short Bind (8 bytes)	3.03	0.48	1.70	0.69	5.90
Medium Bind (128 bytes)	6.58	0.48	30.62	0.69	38.37
Long Bind (512 bytes)	7.98	0.48	138.84	0.69	147.99

Table 3: Performance for different types of authorizations and data sizes (all times in μs)

above. In the large-address-space case, TLB misses become a factor. Despite the performance degradation from ideal, the overall performance for 30,000 IPCs per second is about 30% for small-address-space content processes and 40% for large-address-space content processes.

Unfortunately, we do not yet have performance numbers for handling large data transfers. However, the use of shared memory between monitors and a monitor and its controlled process can reduce the performance impact (if security requirements allow its use). For example, on a *write* operation, only the reference to the data and the data size need to be copied to the monitor for integrity reasons. If the controlled task modifies the data, it has no effect on the security of the operation as long as the operation does not complete before the data is copied to the destination. Therefore, only a single copy of the write data from the destination's monitor to destination is really required. Therefore, 8-byte IPCs can be used on two of the three IPCs in the authorized path. Also, Lava enables flexible memory mapping, so a bind operation that enables a large amount of data transfer may prepare shared memory for implementing such transfers efficiently.

Therefore, we believe that the performance of an operating system-level mechanism for controlling fine-grained is comparable to that of a language-based mechanism, particularly if references are typically passed between processes rather than object data. Since compiled code can execute significantly faster than JIT Java code, it is not unreasonable to estimate that the compiled code can do more processing in the remaining 60-70% of the time than the Java code can do in 100% of the time. Given the additional security benefits of operating system and address space protection (execute compiled code with complete mediation), these security models are worthy of strong consideration.

6 Conclusions

We presented an operating system security model and an analysis of its performance that shows that greater security than that of language-based security models can be achieved with minimal additional overhead for fine-grained programs. This security model enables complete mediation of all content using monitors that automatically intercept IPCs from controlled processes and can enforce security policy upon them. The cost of this interception is perceived to be high, but we have shown that using fast IPC and an efficient authorization mechanism we can perform authorized interception with a reasonable overhead. With little application data available at present, it is hard to estimate the exact overheads, but using micro-benchmarks, we predict an ideal overhead of 12% for 30,000 IPC/s and measure an overhead of 30-40%.

We are not the only researchers working in the area of operating system-level security models for extensible systems. Researchers in the area of extensible kernel architectures have embarked on the development of flexible security services [19, 11]. These systems currently focus on extending server functionality to gain more flexibility in control of client processes. Also, other researchers are focusing on operating system extensions to control downloaded executable content [5, 6]. These system describe how the operating system can enable principals to restrict the rights of their processes. We expect that these researchers will all have to deal with the issues regarding control of multiple fine-grained extensions in the future.

We expect that much interesting research in the future will examine the synergy between operating system and language security models. If a lot of data is to be shared between processes, it is yet to be determined if the best trade-off between security and performance is language-based protection or flexible memory mapping of shared data. Lava's flexible memory mapping enables two processes to share memory in a manner that is still revocable by their monitors. However, language-based protection offers safety (at a cost as well) for data structures within the address space. We predict that the future direction of system and application security will be strongly influenced by the answers to such questions.

Acknowledgements

We'd like to thank Asit Dan, Li Gong, Paul Karger, Ajay Mohindra, and Dan Wallach for their valuable insights. Also, we thank the anonymous referees for their comments which have led to significant improvements in this paper.

References

[1] R. Anand, N. Islam, T. Jaeger, and J. R. Rao. A flexible security model for using Internet content. *IEEE Software*, 1997.

[2] J. P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, James P. Anderson and Co., Fort Washington, PA, USA, 1972.

[3] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-specific file protection for the UNIX operating system. In *Proceedings of the 1995 USENIX Winter Technical Conference*, pages 165–175, 1995.

[4] N. S. Borenstein. Email with a mind of its own: The Safe-Tcl language for enabled mail. In *ULPAA '94*, pages 389–402, 1994.

[5] A. Dan, A. Mohindra, R. Ramaswami, and D. Sitaram. ChakraVyuha: A sandbox operating system for the controlled execution of alien code. Technical Report 20742, IBM T. J. Watson Research Center, 1997.

[6] C. Friberg and A. Held. Support for discretionary role-based access control in acl-oriented operating systems. In *Proceedings of the Second ACM Role-Based Access Control Workshop*, November 1997.

[7] F. S. Gallo. Penguin: Java done right. *The Perl Journal*, 1(2):10–12, 1996.

[8] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th USENIX Security Symposium*, pages 1–14, July 1996.

[9] L. Gong. A secure identity-based capability system. In *IEEE Symposium on Security and Privacy*, pages 56–63, 1989.

[10] L. Gong. New security architectural directions for Java. In *IEEE COMPCON '97*, February 1997.

[11] R. Grimm and B. Bershad. Security for extensible systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 62–66, May 1997.

[12] T. Jaeger, F. Giraud, N. Islam, and J. Liedtke. A role-based access control model for protection domain derivation and management. In *Proceedings of the Second ACM Role-Based Access Control Workshop*, November 1997.

[13] T. Jaeger, N. Islam, R. Anand, A. Prakash, and J. Liedtke. Flexible control of downloaded executable content. Technical Report RC 20886, IBM T. J. Watson Research Center, 1997.

[14] T. Jaeger and A. Prakash. Support for the file system security requirements of computational e-mail systems. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 1–9, 1994.

[15] T. Jaeger, A. Rubin, and A. Prakash. Building systems that flexibly control downloaded executable content. In *Proceedings of the 6th USENIX Security Symposium*, pages 131–148, July 1996.

[16] P. A. Karger. *Improving Security and Performance for Capability Systems.* PhD thesis, University of Cambridge, 1988.

[17] B. Lampson. Protection. *ACM Operating Systems Review*, 8(1):18–24, January 1974.

[18] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 175–187, 1993.

[19] D. Mazieres and M. F. Kaashoek. Secure applications need flexible operating systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 56–61, May 1997.

[20] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Technical Report CSL-116, SRI International, 1990.

[21] J. Ousterhout, J. Levy, and B. Welch. The Safe-Tcl security model, 1997. Available at http://www.sunlabs.com/ ouster/safeTcl.html.

[22] D. D. Redell. *Naming and Protection in Extensible Operating Systems.* PhD thesis, University of California, Berkeley, 1974. Published as Project MAC TR-140, Massachusetts Institute of Technology.

[23] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[24] G. van Rossum. Grail – The browser for the rest of us (draft), 1996. Available at http://monty.cnri.reston.va.us/grail/.

[25] D. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for java. Technical Report 546-97, Princeton University, 1997.

[26] D. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for java. In *Proceedings of the 16th Symposium on Operating Systems Principles*, 1997.

[27] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, February 1994.

[28] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.