Components + Security = OS Extensibility

Antony Edwards Gernot Heiser

School of Computer Science and Engineering
University of NSW, Sydney 2052, Australia
{antonye,gernot}@cse.unsw.edu.au

Abstract

Component-based programming systems have shown themselves to be a natural way of constructing extensible software. Well-defined interfaces, encapsulation, late binding and polymorphism promote extensibility, yet despite this synergy, components have not been widely employed at the systems level. This is primarily due to the failure of existing component technologies to provide the protection and performance required of systems software. In this paper we identify the requirements for a component system to support secure extensions, and describe the design of such a system on the Mungi OS.

1. Introduction

Extensibility is revolutionising system construction. Traditional monolithic operating systems have evolved in an ad hoc manner, making them large, complex, unreliable and slow. Extensible operating systems replace this chaos with a model for controlled evolution, resulting in smaller, faster and more reliable systems. Current research into extensibility focuses on two approaches [20]:

- Allowing user-developed modules to be dynamically added to the kernel. Modules then export an interface that can be called by users. Such systems rely on 'safe' languages (e.g. Modula 3 [19]), compile-time analysis and dynamic reference checks for safety. Prominent examples include SPIN [3] and VINO [22].

- Providing a *trusted path* [24, 17] mechanism, such as a protected procedure call [8] or IPC. Extensions execute as user tasks, using the standard system protection mechanisms for safety. Clients invoke extensions via the trusted path. For example, Amoeba [18] used a client/server model with an IPC based trusted path.

An extensibility mechanism requires flexibility, safety and performance. It is now widely accepted that flexibility and safety can be provided in user space. Kernel-module based systems are therefore motivated solely by performance, which has been acknowledged by the designers of such systems [3]. Performance oriented research on μ-kernel construction, however, has resulted in trusted path mechanisms with overheads of 100-200 cycles [16, 9]. Given the disappointing performance of kernel-modules [3, 22], such systems are obsolete. Kernel-module based systems also require programmers to use a specific 'safe' language, and rely on a complex compiler for protection [5], resulting in a large *trusted computing base (TCB)*.

Trusted path based extensibility mechanisms are more suitable for the current generation of μ-kernel based operating systems. These systems provide flexibility, safety and performance at user level, however, as yet there has been limited investigation into the development of an *extension model* using these mechanisms. Such a model is essential for the interoperability of extensions. Without such interoperability, these systems simply translate the chaos of traditional systems to user level.

Components [23, 6, 7] are a natural model for extensibility, and can be securely implemented using a system supplied trusted path. Due to the lack of protection and poor performance of existing component architectures however, components have not been widely embraced as a suitable model for *system* extension. In particular, existing component models do not support the protection requirements introduced by system extensions, i.e. the ability to:

1. Execute their methods in an amplified protection domain. This allows extensions to provide secure access to privileged resources.

2. Store privileged per-instance data (e.g. a spooler file). Extensions must provide *data-encapsulation*.

3. Perform access control. A user must only be able to invoke methods they have access to, on extensions they have access to.

4. Build upon, and extend, existing services.

5. Add code to the base system which is then invoked through an existing interface.

This paper describes how these requirements have been addressed in the design of a component model for system extension, currently being developed on the Mungi operating system. Section 2 provides an overview of Mungi, Section 3 describes the protection features of the component model and Section 4 presents performance measurements of a prototype implementation.

2. Mungi

Mungi [13] is a single-address-space operating system (SASOS), i.e. all processes on all computing nodes in the system share the same virtual address space. The single address space contains all data, transient as well as persistent. Within this *single-level store*, data are identified through their (64-bit) addresses.

Virtual memory is allocated in contiguous, page-aligned segments called *objects*, which are also the unit of protection. A process is granted certain rights to all or none of an object. Access is controlled via password capabilities [2]; when an object is created, the system returns a capability to the user which contains the object's base address and a password. Such a capability grants full (read, write, execute and destroy) rights to the object and is called an *owner capability*. A process holding an owner capability can register less powerful capabilities for an object.

Capabilities can freely be stored or passed around without system intervention. They are protected from forgery by their password, which is registered in a global, distributed data structure called the *object table* (OT). When validating a capability the system compares the capability's password with the list of valid passwords stored in the OT, and grants access if the requested operation is compatible with the access mode stored with the password in the OT. Validations in Mungi are cached for performance.

Extensibility is supported by a protected procedure call mechanism called *protection domain extension* (PDX) [26]. Each thread in Mungi executes within a protection domain, which is the set of objects accessible to the thread. Protection domains are implemented as a set of pointers to *capability lists* (C-lists). Contrary to classical capability systems, these C-lists are not system objects but are user maintained. PDX allows a thread's protection domain to be extended, in a controlled manner, for the duration of a procedure call using the system call:

```
int PdxCall( void *entry_pt, void *param, cap_t *ret, void *pd );
```

This invokes the method at `entry_pt` in a protection domain that is the union of the passed protection domain `pd`, and a clist registered with the object. The entry point address must be registered with the system, and the caller must have a capability to this entry point. Figure 1 illustrates how the protection domain (shaded) associated with a thread changes during a `PdxCall()`. The Mungi protection model is described in detail in [27].

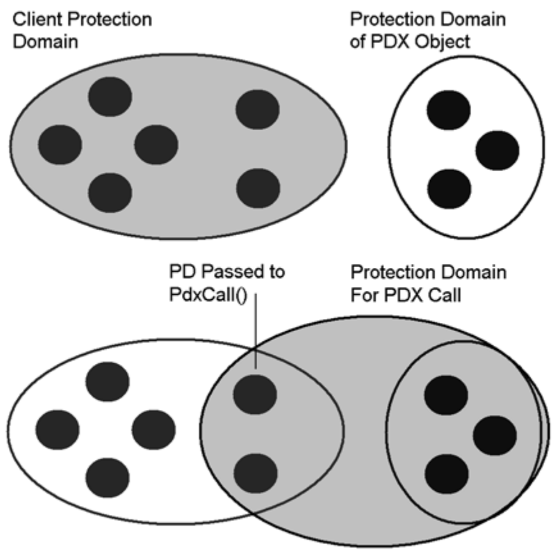
Client Protection Domain
Protection Domain of PDX Object
PD Passed to PdxCall()
Protection Domain For PDX Call

Figure 1. PDX Protection Domains

3. Protected Components

This section describes the protection features of a component model being developed on the Mungi SASOS.

3.1. Method Execution in an Amplified Domain

Figure 2 shows the basic architecture of the component system, in which component implementations are contained within Mungi PDX objects. As described in Section 2, clients can only invoke such code using the `PdxCall()` system call. The argument list of a procedure call contains the component reference to be operated on, similar to the `this` pointer implicitly passed to C++ methods. To execute in the amplified protection domain the client *must* specify one of the registered entry points and they *must* posses a capability to that entry point. By registering only the methods of the component as entry points to the PDX object, a component's interface is enforced, and interface methods execute within an amplified protection domain, which would contain the service's privileged resources.

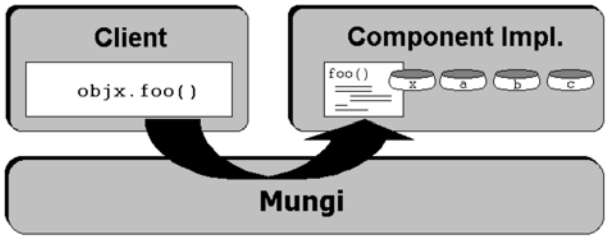

Figure 2. System Architecture

3.2. Encapsulation and Access Control

Component instances provide a service implementation with a location to store possibly privileged data, e.g. a print queue or the contents of a directory. If this location is not protected *by the system* then the amplified protection domain cannot store privileged data in the instances. This results in the component implementation having to manually maintain per-user state.

Capability systems provide natural support for encapsulation, however the relationship between encapsulation and access control contains subtleties that are often neglected.

Essentially, a client invokes a constructor to create a new instance of a component. This constructor is an entry point into the component implementation (a Mungi PDX object) and executes in its associated amplified protection domain. Constructors create a new instance of the component inside a Mungi object that exists only within the component's protection domain. It then passes back the *address* of the component (which uniquely identifies an object in a SASOS) to the client, but does not return a capability to the instance. The component implementation is therefore the only domain holding a capability to the new instance, and as such is the only domain that can access its internal data. As it only performs such an access in response to a request to one of the registered method entry points, instances are fully encapsulated. Furthermore, because the encapsulation is achieved simply by using the system provided mechanisms appropriately, it does not impose any performance penalty.

Although the above scheme does indeed encapsulate the component instances, it does not allow for instance-granularity access control. To invoke a method, the client requires a capability to the entry point in the PDX object, a reference to the instance, but no *capability* to the instance. Therefore there is no protection against a client with access to the component type from invoking operations on arbitrary references in the hope of revealing some private data, or performing a privileged action. The problem is that the system capabilities are providing protection at type granularity, whereas we require protection of instances.

As we are dealing with a capability system, the natural solution is to create a new capability that confers upon the holder the right to invoke methods on the instance. Unfortunately this scheme is also inadequate. Capabilities provide protection at page granularity, therefore to protect instances a component implementation would have to place each instance in its own page. The majority of components contain only a few bytes of data, and so this leads to extremely inefficient use of memory and dismal translation lookaside buffer (TLB) performance.

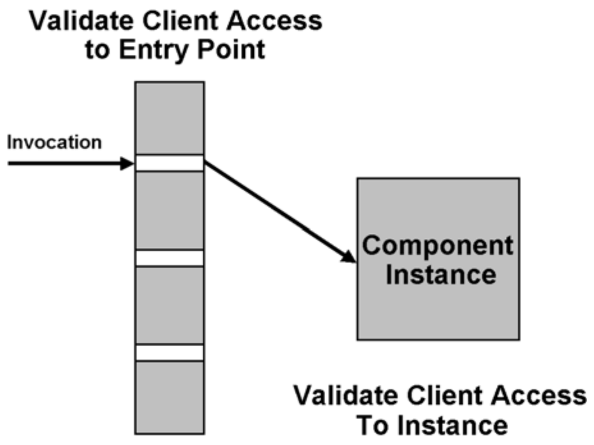
Validate Client Access to Entry Point
Invocation
Component Instance
Validate Client Access To Instance

Figure 3. Access Control Granularity

Mungi objects must be page aligned since their protection derives from the memory management unit (MMU), which deals with memory at page granularity. Components do not have such a constraint, because their protection is based on every method invocation having to pass through the system's trusted path mechanism, i.e. `PdxCall()`. A reference to the instance being used is explicitly presented to the component implementation on every call, allowing it to perform access control at arbitrary granularity. Thus, component implementations carry out their own instance capability validations. Constructors create a random 64-bit password which is returned to the client along with the reference, as well as being stored in the encapsulated component instance. This (reference, password) tuple is called a *component instance capability* (CICAP) and must be passed and validated on each method invocation. Passed CICAPs are validated by a simple integer comparison with the CICAP stored in the component instance, imposing little overhead (see Section 4). As CICAPs provide protection at the granularity of component instances, instances for different clients may be safely placed into the same Mungi object. Access control now consists of a validation that the client has access to the interface entry point, and to the component instance, as illustrated in Figure 3. It is important to note that these validations are independent. This means that if a client holds CICAPs for two instances of a given component, it cannot be allowed access to a certain interface for one of the instances, but not the other.

3.3. Extending Components

Rights amplification, encapsulation and user access control allow components to provide applications with additional services. This is one-level of system extensibility. True system extensibility allows new extensions to build upon existing services, and to be invoked via existing interfaces. It is the combination of these two mechanisms that gives extensible systems their flexibility and power [11]. Sections 3.4 and 3.5 describe how these operations are provided by the Mungi component model, and identify their associated protection issues.

3.4. Forwarding and Aggregation

Forwarding (also known as *component composition*) is a simple model for reusing existing components, that avoids the semantic problems caused by implementation inheritance [23]. To extend an existing service (C_B), a new component (C_E) is developed that provides a super-set of C_B's interfaces. An instance of C_E contains a reference to an instance of C_B (either created in the constructor or supplied by the client), which it uses as any normal client would, i.e. there is no concept of a specialisation interface [15]. Since the extending component provides a super-set of the interfaces provided by the base component, polymorphism allows instances of the extending component to be substituted for instances of the base component. Forwarding relies on the natural reusability of components, rather than introducing a new model, such as inheritance, when a client happens to be another component.

Since forwarding is essentially just components acting as clients, it does not directly introduce any new protection issues. It does however introduce the concept of a dynamic system interface, which creates a number of security issues. When a new component (C_E) extends an existing service (C_B), C_B is no longer part of the system interface and should become inaccessible to applications. An applications view of the system interface is a system-level policy, that should therefore be enforced by a mandatory protection mechanism, i.e. one where security attributes are controlled by an administrator rather than users. Mandatory protection in extensible systems is briefly discussed in Section 3.6.

Aggregation is an optimised form of forwarding, in which interfaces of a base component are directly exported by the extending component. This avoids the overhead of the extending component having to relay the request to the base component, when it does not add any extra processing. Figure 4 illustrates forwarding and aggregation.

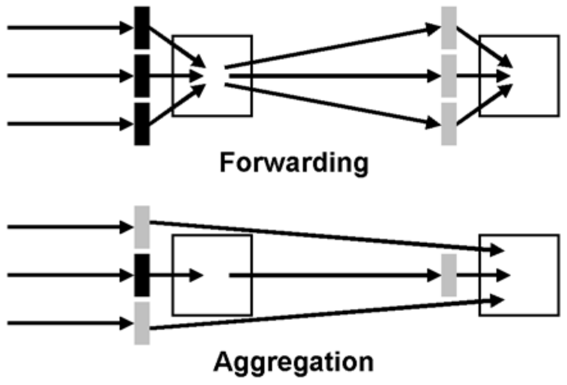

Figure 4. Forwarding vs Aggregation

Normally when a component (C_E) extends another component (C_B), it does not distribute C_B's CICAP. This means that no clients may directly access C_B, which is therefore *contained* in C_E. As the extending component usually assumes that it is the only entity invoking methods on the base component, this containment is very important. Under aggregation, clients directly invoke methods on the base component. As described in Section 3.2, a client cannot be granted access to a given interface of a component for some instances, but not others. To allow clients to directly invoke methods on the base component for aggregated interfaces, C_E must provide clients with a valid CICAP for the base component. As CICAPs allow access to all interfaces, in general there is no protection against clients also invoking methods on non-aggregated interfaces, and thereby violating containment.

This problem is addressed by introducing the concept of an instance *owner*, who may create new CICAPs with access to a subset of interfaces. When an instance is created, an *owner* CICAP allowing access to all interfaces is generated and returned to the client. Each component provides an entry point that allows a client holding an owner CICAP to request the creation of a new CICAP with access to a specified set of interfaces. A new password is generated and recorded in the instance, along with the set of interfaces this CICAP permits access to. The new CICAP is then returned to the client. When a method is invoked, the component compares the passed CICAP with the owner CICAP, and any CICAP generated with access to the invoked interface. If a match is found, the invocation is allowed to proceed. Therefore, when a component aggregates interfaces from a base component, it requests the creation of a new CICAP with access only to the aggregated interfaces. Clients are then only provided with this CICAP, ensuring that they cannot call non-aggregated interfaces of the base component.

3.5. Delegation

Delegation allows new components to be invoked via existing interfaces, i.e. it is the component-oriented equivalent of virtual inheritance. It allows a new component-instance C_D to register itself with an interface of an existing component-instance C_B, so that if a client invokes a method on that interface of C_B, the request is redirected to C_D. Figure 5 shows the flow of control when a delegated method A is invoked on a base component. Requests are shown as solid lines, replies as dotted lines.

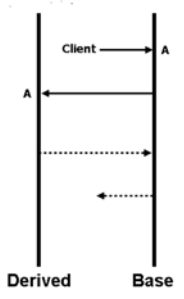

Figure 5. Delegation

Delegation allows authorised entities to customise system services to improve performance, correctness and simplicity [21]. For example, a new component could delegate an existing printer instance to provide fair scheduling, or a directory set to provide a customised file cache. Delegation allows these new services to be added in a manner that is transparent to clients. Without delegation, all clients must manually migrate to the new service.

Delegation is essentially a form of dynamic aggregation, and so protection is mostly provided using the mechanisms presented in the previous section. One difference however, is that delegation requires an entry point for delegating instances to register themselves. Obviously, delegation is a sensitive operation, e.g. delegating write requests to the password file. For protection, the entry point to register a delegating instance is placed on its own interface. Again however, this is insufficient due to the independence of entry point and instance validations discussed in Section 3.2. For example, in a component-based file system a user would be able to extend their own directories. This requires that they possess a capability to the delegation entry point for directory components. The user should also be able to read certain directories (e.g. `/home`), requiring that they possess a CICAP to these directories. A mechanism is needed to prevent users from delegating directories they do not own. This can be achieved by the owner creating a new CICAP with access to all interfaces except the delegation interfaces. This CICAP can then be safely distributed.

3.6. Mandatory Security

"A given system is 'secure' only with respect to some specific policy" [1]

So far, discussion has been limited to *discretionary* protection, i.e. protection mechanisms in which security attributes are controlled by users. Mungi capabilities and CICAPs can both be be freely distributed without system intervention, and are therefore discretionary protection mechanisms. Such protection is incapable of enforcing system-wide security policies, because it cannot defend against careless or malicious users.

A mechanism for enforcing system-wide security policies, i.e. *mandatory security*, is being increasingly recognised as a requirement for a general-purpose operating system [14, 10]. As extensible systems are inherently dynamic and fine-grained (Section 3.4) such mechanisms are vital.

Currently, mandatory access control mechanisms capable of effectively supporting extensible environments do not exist [14]. For example, existing mandatory mechanisms do not provide for controlled amplification of a client's protection domain, which was identified in Section 1 as a requirement for extensible systems. A protection mechanism, based on *type enforcement (TE)*, suitable for use in an extensible environment is currently being developed on Mungi.

3.7. Summary

This section described the protection features of a component model being developed on the Mungi SASOS. Specifically, it described how:

- A Mungi PDX trusted path, allows components to execute in an amplified protection domain.
- Mungi capabilities combine with PDX to provide natural data encapsulation.
- Effective discretionary access control can be performed using CICAPs, avoiding inefficient use of memory and dismal TLB performance.
- Extension can be safely performed using forwarding and aggregation.
- Existing services can be safely delegated.

4. Performance

This section presents initial performance results for a secure component-based programming system being implemented on the Mungi OS. Section 4.1 examines the overhead of the component model by comparing method invocation and component creation costs with the raw system costs. Section 4.2 presents results from a subset of the OO1 benchmark [4] to show that the micro-benchmark results lead to end-to-end performance gains.

Mungi benchmarks were run on a 100MHz MIPS R4600-based computer with 64MB of RAM developed at UNSW. Irix and Linux figures were obtained by [25] on a 100MHz R4600-base SGI Indy workstation.

4.1. Micro-benchmarks

As trusted path mechanisms become faster [16, 9], the overhead of the software construction model, e.g. Mungi components, becomes more significant [12]. This section examines the overhead of the described component model for method invocation and component creation.

Table 1 presents the performance of a method invocation, with the added overhead of CICAPs shown separately. Results were obtained by invoking a method 500 times and measuring the total elapsed time. The method invoked simply returns a 64-bit integer stored in the instance.

Operation	Arg. Size (b)	Time (μs)
Mungi PdxCall	8	30
No CICAPs	0	31
No CICAPs	1k	31
No CICAPs	16k	31
Standard	0	32
Aggregated (1st call)	1k	63
Aggregated (repeated)	1k	32
Delegated (1st call)	1k	64
Delegated (repeated)	1k	32

Table 1. Method invocation

A basic method invocation costs only 1 μs (around 100 machine cycles) more than a raw `PdxCall()`. The extra cycles are consumed handling the component reference, which is passed on each method invocation, handling parameters, and error checking. As all threads execute within the single address space, marshalling of parameters is not required and hence there is no cost associated with passing larger amounts of data. CICAPs add a further microsecond to the method invocation cost. This is the time taken to retrieve the stored CICAP from the component instance and perform an integer comparison with the passed CICAP. Invoking an aggregated or delegated method incurs twice the overhead of a standard method invocation for the first call, but is the same for further calls to the same interface. This is because the first request is sent to the aggregating, or delegating, component, who redirects it to the appropriate component, thus requiring two method invocations. Further requests are sent directly to the appropriate component.

Table 2 presents the performance of component creation. Results were obtained by invoking a single constructor 500 times and measuring the total elapsed time.

Operation	Time (μs)
Basic Creation	92
+ CICAPs	93

Table 2. Component creation

Component creation costs 93μs. This involves a request to the component domain to create a new instance (32μs), the creation of a Mungi object to be used as a parameter buffer (45μs), and the creation and initialisation of a local proxy object to be used by the client. CICAPs are also generated by the constructor. A simple hashing of the system clock was used by the measured component, resulting in a 1μs overhead. Obviously more sophisticated random number generators may result in higher overheads.

4.2. The OO1 Benchmark

The OO1 benchmark [4] is designed to simulate typical operations on an object-oriented database. The data stored in the database should be accessible to the user only via the defined interface functions, and so is a natural situation for components. A database component is constructed with appropriate access methods, and an instance is created to hold the database information. The database used consisted of 20,000 parts. Four operations are performed, lookup, forward traversal, reverse traversal and insert.

Table 3 shows the results from the OO1 benchmark running on Irix, Linux and Mungi. Irix and Linux use an RPC trusted path, with a client/server software construction model. Mungi uses the component model described in Section 3. Irix and Linux results are taken from [25]. All times are in milliseconds. *L.* is lookup, *F.T.* is forward traversal, *R.T.* is reverse traversal and *I.* is insert.

Mungi components outperform Linux by a factor of eleven and Irix by a factor of twenty-one. Total execution time for OO1 can be separated in three categories:

- **Executing application code**, which includes the client logic and the database operations. As application code is the same for all three systems, and contains no system calls, this cost should be constant across systems.

System	L.	F.T.	R.T.	I.	Time (ms)
Irix	949	1,409	1,411	203	3,972
Linux	344	467	461	842	2,114
Mungi	88.8	27.1	33.0	38.9	187.8

Table 3. OO1 benchmark results (in ms)

By placing the database in the same protection domain as the client, and re-executing the benchmarks, it was confirmed that the cost was constant, at 28μs. As this is over three orders of magnitude less than the total execution time, application code overhead is irrelevant.

- **Cross-domain call overhead**. Each operation results in (at least one) cross-domain call. Table 4 compares the cost of a cross-domain call on each system.

Mechanism	Time (μs)
Mungi PdxCall	30
Linux RPC	160
Irix RPC	450

Table 4. Cross-domain call performance

As 5869 operations are performed by the benchmark, the total cross-domain call overhead (X-Dom) can be calculated. These values are presented in Table 5.

System	X-Dom. (ms)	Other (ms)	Total (ms)
Irix	2641	1331	3972
Linux	939	1175	2114
Mungi	176	11.8	187.8

Table 5. Division of overhead

- **Model overhead** is the remaining difference between the three systems. Irix and Linux both use a client/server model in a multiple-address-space environment, while Mungi uses a component model in a single-address-space environment. Model overhead is primarily parameter marshalling and message dispatch, e.g. a message loop. As Irix and Linux use the same software construction model, which does not involve system intervention, it is expected that both should incur a similar model overhead.

Section 4.1 shows that, for Mungi, the model overhead is 6.7% ($\frac{2}{30}$) of the cross-domain cost. For the $176ms$ cross-domain cost reported in Table 5, this corresponds to $11.8ms$, which is exactly the value in the **Other** column. This confirms that cross-domain call latency and model overhead are indeed the differentiating factors for the OO1 benchmark. Therefore, the **Other** column of Table 5 can justifiably be used as reporting the model overhead. Irix and Linux have a similar model overhead as expected, though the 11.7% ($\frac{1331-1175}{1331}$) difference is greater than expected. Further analysis is required to fully explain this result.

5. Conclusion

Component-based programming is a natural way of constructing extensible software, but has yet to be employed at the system level due to issues of protection and performance. In Section 1 we identified five protection-oriented requirements for an extension model, and described the design of a component-system satisfying these requirements in Section 3. Initial performance results presented in Section 4 indicate that components *can* provide both the security and performance required for building extensible systems. Performance measurements with more macro-benchmarks and real workloads will provide further evidence.

References

[1] S. R. Ames, M. Gasser, and R. R. Schell. Security kernel design and implementation: An introduction. *IEEE Computer*, 16(7):14–22, July 1983.

[2] M. Anderson, R. Pose, and C. S. Wallace. A password-capability system. *The Computer Journal*, 29:1–8, 1986.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on OS Principles (SOSP)*, pages 267–284, Copper Mountain, CO, USA, Dec. 1995.

[4] R. G. G. Cattell and J. Skeen. Object operations benchmark. *ACM Transactions on Database Systems*, 17:1–31, 1992.

[5] D. Cheriton. Low and high risk operating system architectures, (panel). In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Monterey, CA, USA, Nov. 1994.

[6] The component object model specification. Technical report, Microsoft Corporation and Digital Equipment Corporation, 1995. http://www.microsoft.com.

[7] Corba components. TC Document orbos/99-02-05, Object Management Group, Mar. 1999. ftp://ftp.omg.org/pub/docs/orbos/99-02-05.pdf.

[8] J. Dennis and E. Van Horn. Programming semantics for multiprogrammed computers. *Communications of the ACM*, 9:143–55, 1966.

[9] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble component-based operating system. In *Proceedings of the 1999 USENIX Technical Conference*, pages 267–282, Monterey, CA, USA, June 1999.

[10] R. Grimm and B. Bershad. Access control in extensible systems. Technical Report UW-CSE-97-11-01, Dept of Computer Science & Engineering, University of Washington, Seattle, WA, USA, 1997.

[11] R. Grimm and B. Bershad. Security for extensible systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 62–66, Cape Cod, MA, USA, May 1997.

[12] A. Haeberlen, J. Liedtke, Y. Park, L. Reuther, and V. Uhlig. Stub-code performance is becoming important. In *1st USENIX Workshop on Industrial Experiences with Systems Software (WEISS)*, San Diego, CA, USA, Oct. 2000.

[13] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke. The Mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928, July 1998.

[14] T. Jaeger. Access control in configurable systems. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*. Sept. 1999. Available from: http://www.research.ibm.com/sawmill/.

[15] G. Kiczales and J. Lamping. Issues in the design and specification of class libraries. In *Proceedings of the ACM Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 435–451, 1992.

[16] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, and T. Jaeger. Achieved IPC performance (still the foundation for extensibility). In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 28–31, Cape Cod, MA, USA, May 1997.

[17] P. Loscocco, S. Smalley, P. Muckelbauer, R. Taylor, J. Turner, and J. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. Technical report, United Stated National Security Agency (NSA), 1995.

[18] S. J. Mullender and A. S. Tanenbaum. The design of a capability-based distributed operating system. *The Computer Journal*, 29:289–299, 1986.

[19] G. Nelson. *Programming in Modula-3*. Prentice Hall, 1991.

[20] Radical operating system structures for extensibility: A panel session. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 195–199, November 1994.

[21] P. Pardyak and B. Bershad. Dynamic binding for an extensible system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, USA, Oct. 1996.

[22] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 213–228, Nov. 1996.

[23] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley/ACM Press, Essex, England, 1997.

[24] US Department of Defence. *Trusted Computer System Evaluation Criteria*, 1986. DoD 5200.28-STD.

[25] J. Vochteloo. *Design, Implementation and Performance of Protection in the Mungi Single-Address-Space Operating System*. Phd thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, July 1998. Available from http://www.cse.unsw.edu.au/~disy/papers/.

[26] J. Vochteloo, K. Elphinstone, S. Russell, and G. Heiser. Protection domain extensions in Mungi. In *Proceedings of the 5th IEEE International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 161–165, Seattle, WA, USA, Oct. 1996. IEEE.

[27] J. Vochteloo, S. Russell, and G. Heiser. Capability-based protection in the Mungi operating system. In *Proceedings of the 3rd IEEE International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 108–15, Asheville, NC, USA, Dec. 1993. IEEE.