A Secure, Language Independent, High Performance Component Interface

Daniel Potts, Charles Gray, Ben Leslie and Gernot Heiser

School of Computer Science and Engineering & National ICT Australia
University of NSW, Sydney 2052, Australia
{danielp,cgray,benjl,gernot}@cse.unsw.edu.au

Abstract

In this paper we examine three interfaces for secure method invocation in single-address-space operating systems. We examine the advantages and drawbacks of each model, and how these models relate to linking and loading in the single address space. A model is chosen based on its ability to securely interface multiple languages with low overhead.

1. Introduction

Single-address-space operating systems (SASOS) simplify sharing of data between programs running on a system. Furthermore, instances of a program are able to easily share program text and read-only data in place.

Sharing text in-place with data located at different addresses requires a different model of linking and loading to multiple-address-space operating systems (MASOS).

Our approach is somewhat different to that of the Nemesis [Ros94] system because we present a more traditional programming interface in which you are free to have global state within a program.

The issues surrounding linking and loading programs in Mungi has been addressed in earlier work [DH99]. This paper presents a new linking and loading model in Mungi that allows transparent cross-module calls.

We evaluate several protected procedure call models that allow components to share program text while enforcing encapsulation of data in separate data segments using hardware memory protection. Furthermore, this new model provides a language independent, high performance component interface for executing protected procedure calls.

2. Mungi

Mungi [HEV+98] is a research operating system developed at the University of New South Wales. Mungi provides a single global address space in which all data can be accessed using only its virtual address.

The virtual address does not lose its meaning when passed between programs or nodes, eliminating the need for explicit file or network operations within a distributed Mungi system. Instead, programs communicate via shared memory.

A traditional operating system uses address spaces as the unit of protection, and all threads in an address space have equal permissions to each part of that address space. The Mungi address space instead is divided into regions called *objects*, the permissions on which are defined by capabilities. Threads execute inside a *protection domain* — a collection of capabilities that defines what objects can be accessed and with what permissions.

Because all object references are the same format they can be used as opaque handles and stored in components such as the Mungi naming service, used by most programs in the system to look up services and data. Mungi is designed to take advantage of processors with virtually indexed caches because all data is accessed through the same virtual address.

2.1 Protection domain extension

To be secure, a system needs a controlled mechanism to call procedures across protection boundaries. In a traditional operating system this is done through mechanisms such as pipes, sockets, signals and System V IPC to separate address spaces. All of these mechanisms involve sending a message from one thread of execution to another, possibly via a queue.

For cross domain calls Mungi has a *protected procedure call* called PDX. A PDX call logically migrates a thread from its current protection domain to a kernel registered entry point in another protection domain. For the duration of the call the thread may carry with it permissions to access parameters or shared data structures.

2.2 A component system on Mungi

Mungi lends itself naturally to a component software model [EH01]. Protection domains and PDX offer a simple yet powerful method of encapsulation. Rather than utilising a marshal and message paradigm like typical systems, threads flow naturally through the execution of the program across protection boundaries.

In a typical operating system, serial execution of RPCs is the norm. A thread takes messages out of a message queue and processes them serially, or pushes them off onto other threads. Mungi instead offers parallel execution as the default. Mungi handles any thread management operations itself. Threads only need include serialising operations such as locks when they are required.

```
prettyPrintAST( parse_node* root );
```

Figure 1. An abstract syntax tree pretty-printer interface

The single address space (SAS) helps simplify remote invocation even further. When a thread migrates from one protection domain to another, all pointers maintain their meaning. For example, the method shown in Figure 1 can use a pointer to denote the root of the object and follow the pointers inside the objects without any need for modification. In the example, a parser can share read-only references to a data structure and the module can securely operate on the data in-place.

Traditionally this type of communication would require marshaling the data in the form of serialisation to a textual or binary form and then re-parsing it back into the original structure on the callee side. A MASOS system with shared memory need not marshal the data, however it must still translate pointers between address spaces.

The PDX model simplifies component model features such as aggregation. Multiple namespaces that exist when using pipes or IPC, implies that a client must decide through which pipe or queue to send requests based on methods and interfaces. In Mungi all entry points are in the same namespace, so invocations may go to arbitrary component implementations transparent to the caller.

2.3 Language-based security

Recent language-based security [SMH00] developments in systems such as Java [GJS96] attempt to develop approaches for ensuring the integrity and trustfulness of third-party programs.

While systems such as J-Kernel [HCC+98] implement multiple protection domains for Java-based code, ensuring safety cannot always be guaranteed as applications frequently need to utilise the functionality of existing modules or other applications.

Providing a method for secure, cross-domain invocation allows applications to call other modules or applications while enforcing encapsulation, is discussed in Section 4.

3. Linking and loading

Linking and Loading in a SASOS is very different to that of a typical MASOS [CLFL94]. In a MASOS you have the concept of both *statically linked* and *dynamically linked* executables.

A statically linked binary has to be loaded at a fixed location in its address space because it contains absolute references to code and data.

A dynamically linked (shared) library is somewhat different. Shared libraries are modules loaded into another process' address space and linked at runtime. Because shared libraries must work in varying address space layouts with other shared libraries, they cannot have absolute references to code or data. Shared libraries must be position-independent.

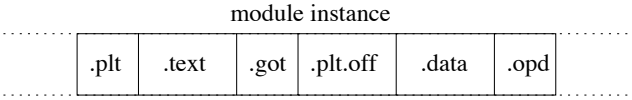

Figure 2. Layout of program sections in a traditional operating system.

Figure 2 shows a typical layout of a loaded module in a MASOS. All sections are loaded in a fixed layout at an arbitrary (but aligned) base address. The fixed layout means that code can always locate pointers in the global offset table (GOT) through PC-relative addressing. It is the job of the dynamic linker to resolve these indirect pointers at module instantiation time.

Linking in a SASOS is a combination of both the static and dynamic linking of a MASOS. This is because all instances of a module in a SASOS can share the same text segment, but require separate data segments. Code can contain absolute references within its own text segment or read-only data segment. These absolute addresses can be resolved either at install time or by linking the objects in-place if compiling from source.

In a SASOS, as shown in Figure 3, the data segment of a module instance is not at a fixed offset from the text. Otherwise, all module instances would incorrectly refer to the same data segment. To allow for multiple module instances

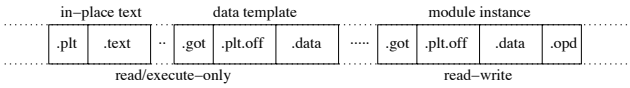

Figure 3. Layout of program data in single-address-space.

we can either make a separate copy of the text segment for each instance of the library, which is inefficient, or not rely on a fixed offset between text and data.

To instantiate a module, the template data is copied to a free region of the SAS. The dynamic linker then fixes relocation entries in the data segment for the new module instance. All indirect pointers are maintained in a *global offset table* (GOT) which is pointed to by the *global pointer* (GP) register. The GP register removes the need for text and data to be loaded in a fixed layout. Shared libraries in a MASOS typically use a GP register to increase the speed with which global data can be referenced.

Depending on the architecture, the GP is set up either before or after a function call is made. A caller loaded system such as Itanium [Int00] sets up the GP prior to executing a function. Callee loaded systems compute the GP as a fixed offset from the instruction pointer (IP) upon entry into the function.

The need for a caller loaded GP register implies that a function is no longer denoted just by its entry point. A function entry point is ambiguous as to the data segment context in which the code should run. A function is therefore an IP and GP pair, denoted by (ip, gp). This is expressed on architectures such as Itanium using a *function descriptor*. A function descriptor is a small (ip, gp) structure allocated by a linker.

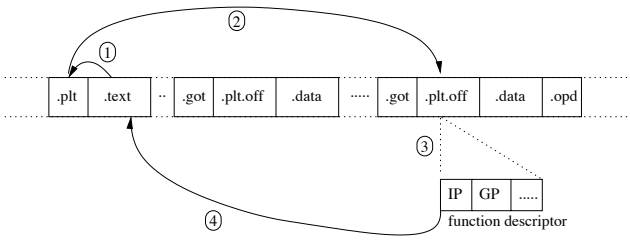

Figure 4. Execution path of a component function call.

A module instance contains two sets of function descriptors in its data segment: `.plt.off` and `.opd`. This is necessary to accommodate both function calls and function pointers.

Figure 4 shows a call to an exported function in a caller-loaded architecture like Itanium. To preserve standard linking semantics at runtime, it is necessary that a call to an exported function is resolved by performing a lookup in the procedure linkage table (PLT).

A function call involves a relative jump from the text segment to the PLT. The code stub in the PLT locates the function descriptor in the `.plt.off` section by a fixed link-time offset from the GP. The old GP is stacked and the new (ip, gp) pair is loaded from the function descriptor. The code then jumps to the new IP. On return the original GP is reloaded from the stack.

Function pointers are used in C/C++ to provide late binding. In a caller loaded ABI the function pointer is actually the address of the *official procedure descriptor* (OPD). The OPD is a function pointer that resides in the data segment of the module instance exporting that function. It is necessary to have a specific OPD to allow comparison of function pointers.

Invoking a function by its pointer causes the code to load the new GP and jump to the IP. This allows transparent cross-module calls at the cost of loading via an indirect pointer. It is an optimisation for the linker to cache a copy of an OPD in the `.plt.off` section allowing a GP-relative load.

It is the role of the dynamic linker to ensure that the function descriptors point to the appropriate code.

4. Protected procedure call models

To execute a procedure call of a component instance, the ability to specify not only the entry point of the procedure, but also a data segment pointer that refers to the appropriate component instance is required.

To enforce encapsulation, the kernel is required to authenticate the procedure call before it is executed. This requires the kernel to be aware of the format of these calls.

The model of protected procedure call is closely related to the linking model and calling convention of languages. Below we discuss three implementations of protected procedure calls.

4.1 Model 1: Kernel holds a code and data pointer

```
ObjCrePdx( pd, ip,  gp, passwd );
PdxCall( ip, gp, parameter );
```

The naive solution is simply to place the (ip, gp) pair into the kernel for each protected procedure call. The (ip, gp) pair uniquely identifies a procedure call within a particular module instance.

A PDX entry point is registered with `ObjCrePdx` by explicitly specifying both the IP and GP to be used on an invocation. The kernel stores both these internally and uses the GP to discriminate between different instances of a

module. This ensures that permission for a client to call an instance of a module in no way implies permission to call any other instances of that module.

The PDX entry point is invoked by calling `PdxCall` with the (ip, gp) pair and an optional parameter.

This model requires applications to handle a pair of pointers that do not easily map to a unique SAS address. This makes the process of looking up a PDX entry point more complex, or requires modifications to applications to handle multi-word addresses. Practically this complicates user code in many ways.

Furthermore, this model exposes low-level implementation details of the architecture's ABI to the user.

Any modification of the system ABI would require changing any code that exports or calls a PDX entry point. This would limit the portability of a system using this model. A change to the ABI would result in changes to the API. For example, PowerOpen ABI [TL03] uses 3-word function descriptors. A PDX call would therefore need to specify (ip, gp) on Itanium and $(ip, gp, environment)$ on Power. This provides no transparency for RPC invocations across different architectures. While it would be possible to implement PowerOpen ABI using a pseudo-gp that pointed to the real procedure descriptor, it is messy and a performance problem to emulate at the ABI level.

4.2 Model 2: Kernel holds a function descriptor pointer

```
ObjCrePdx( pd, fd*, passwd );
PdxCall( fd*, parameter );
```

In an attempt to solve some of the problems in Model 1, an alternative model similar to the function descriptors used in caller loaded ABIs is presented.

The function prototypes are shown above. Instead of passing an explicit (ip, gp) function descriptor pair, the caller passes a pointer to this pair (shown by `fd*` in the function prototype).

For lookup and validation, the kernel need only store this pointer in its internal structures. On a `PdxCall` the caller supplies this pointer for the kernel to look up. It is important to note that the client need not actually have any permissions on the pointer, except the right to invoke it. On the callee side the kernel executes a short piece of stub code in the callee's protection domain to load the (ip, gp) pair from the function descriptor pointer and then executes it.

As for Model 1, we can still differentiate between distinct instances of a module because the function descriptor data pair exists in the instances' private data segment, making it unique to each instance.

However, because a single pointer references a PDX entry point, the code to handle PDX invocation is far simpler in both the kernel and application programs. The kernel only needs to work with a single pointer for comparison and storage. User code can safely handle and store an entry point in the same way it does any other pointer.

Importantly, this also allows cross-platform transparency because all architectures use the same size pointer. It is up to the callee's kernel's stub code to interpret the data pointer.

This model has a potential problem of aliased function descriptors. It is possible to register two distinct function descriptors that have the same (ip, gp) pair. Practically, however, this has not proven to be a problem.

A minor drawback of this module is that during module creation, the loader must create a 'fake' function descriptor in the new program's domain for the kernel code to load. While this is somewhat unclean, in practice this code is infrequent and for highly specialised uses, so it is not considered a problem.

While this solution alleviates most of the problems with Model 1, it still has problems. Of particular concern, the structure of the function descriptor that stub code loads is specific to the C ABI for the callee platform. This is an inconvenience for languages such as Java or Python [Fou03] which, while their virtual machines use the C ABI, code written in these languages themselves has no concept of a GP. This limits cross-language compatibility.

For a non-C language to implement a PDX service it needs to generate short C-ABI function code stubs at runtime for each non-C function to be called as a protected procedure entry point. While it would possible to add another opaque pointer registered with the kernel to provide a method number selector, this would suffer from all the problems of Model 1, only partially alleviating the original problems.

4.3 Model 3: Kernel holds a handle and a code pointer

```
ObjCrePdx( pd, handler, handle*, passwd );
PdxCall( handle*, parameter );
```

Removing the dependency on the C-ABI also removes the cross-language limitations of Model 2. This yields the function prototypes above. Essentially we introduce a model shift. Instead of "calling to a specific instruction pointer" you "invoke a method handle". While the actual changes to application code are minimal, it allows for a far more flexible implementation of the kernel.

`ObjCrePdx` now takes a *handler* parameter, the IP (not function descriptor) of code to be run on invocation, with the opaque *handle** as the parameter. The `PdxCall` however, takes only the corresponding *handle** as the argument. This means the kernel needs to store extra state over Model 2 — the *handler* as well as the *handle**. However, on a

`PdxCall`, the *handle** alone describes the actual entry point.

Essentially this is a generalised version of Model 2. Standard C code registers a function descriptor as the handle and the equivalent of the kernel stub to load the (ip, gp) pair and jump to it. Other architectures register similarly and cross-platform calls work the same as Model 2.

The advantage of this model comes from being able to register anything as the handle and define your own ABI. Another scripting language, for example, may register a pointer to a string which is the script to execute as the handle, and the interpreter code as the handler.

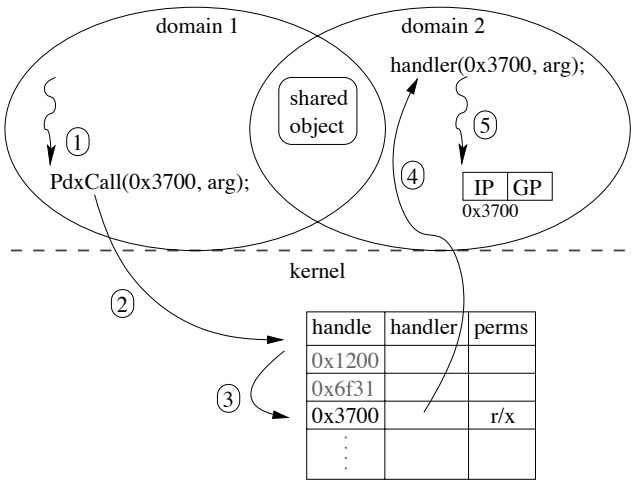

Figure 5. Protected procedure call in Model 3.

Figure 5 demonstrates the steps to execute a full PDX call in Model 3. Firstly the calling thread executes the `PdxCall` system call, specifying the handle and an argument. The kernel receives control and searches for an entry point with a matching handle. A permission check is made to ensure that caller has the right to invoke the callee. The kernel then migrates the thread into the callee domain and executes `handler` with the passed handle and argument. When the callee returns, the kernel migrates the thread back to the caller domain with the return value.

```
struct python_method_desc
{
       void *interpreter; /* gp */
       PyObject *method;  /* bound method */
};
```

Figure 6. An example python method descriptor.

Figure 6 is an example method descriptor that would be used in Python. Essentially it defines a Python ABI. This allows the same (ip, gp) pair to be re-used (that of the interpreter) but because each python method in an instance has a separate descriptor, each method can be registered with the kernel. There is no need for runtime code generation.

This model cleans up module startup code. There is no need to create false function descriptors because the start of a program can be specified with the entry point as the handler and a NULL handle.

This model also has the advantage that it can make the alias issue of Model 2 a potential feature. Applications are free to define their own ABI, independent of the caller and the operating system. Programs can annotate entry points with whatever data they need, such as permissions. This allows multiple references to the same code and data segment but with varying application defined parameters.

This could be utilised in interfaces where a method invocation can perform actions at different privilege levels. One such system would be an SQL command evaluation method in a database. One handle may restrict execution to only SELECT queries while an administrative handle for the same entry point would allow all operations.

Without these annotations it is necessary to implement access control measures in the application protocol. This may involve either adding an extra authentication mechanism or changing the interface so that no method can perform an operation at more than one permission level. We believe this is an unacceptable limitation for the interface designer.

The cost for this added flexibility is an increased number of registered entry points in the kernel. It is yet to be seen whether this has an impact on kernel performance and caching.

5. Performance

A cross-domain method invocation system is useless if the performance of the system is prohibitive in using it. We ran benchmarks on Mungi and Linux to demonstrate the cost of a PDX call in reference to traditional function and cross module calls.

Call type	Linux	Mungi
Local function	5–15	5–15
Cross module	7–15	7–15
Cross protection domain	4600	624

Table 1. Performance of procedure calls measured in cycles.

Table 1 shows the cost in cycles of local and cross module function calls. These results were obtained using a 900Mhz Itanium-II. In Linux, the cross protection domain mechanism is implemented using a 2.6-series kernel primitive called *futexes*. These provide locking and semaphore building blocks that are significantly faster than using System V IPC or pipes.

The *local function* call results show the cost of executing a function compiled directly into a program. The *cross module* call results show the cost of executing a function that is in a different module from the caller, such as a library. As expected, these results are the same for both platforms as the compiler generates the same code.

For *local function* and *cross module* calls we obtained a range of results. The best results correspond directly to a cycle count of the instructions executed for each call type. Different results were obtained due to function alignment at compile time as well as cache alignment and data position issues at runtime that are not always under the control of the programmer or compiler.

The results show that the cost of the extra steps involved in a cross module call are approximately 2 additional cycles with hot caches.

The *cross protection domain* call results show the potential of the PDX mechanism for a fast, light-weight module invocation.

The additional overhead of a PDX call is mostly due to context switch overhead when switching from the caller's address space to the callee's address space and back when the call returns.

For Mungi, we believe the added cost of execution is well within a reasonable range due to the fast context switch and low PDX overheads Mungi provides.

The primitives used in Linux are not designed with a fast cross protection domain crossing in mind. While context switch time is the dominating cost, Linux is optimised more for interactive work. This adds scheduling overhead in the path of execution for a PDX call.

6. Conclusions

Our work has shown that PDX offers an attractive model for cross module calls in a component system providing hardware protection. The thread migration model works naturally with the flow of program execution. The light-weight nature of PDX allows protection boundaries to be introduced in programs with lower overhead than in more traditional operating systems.

The new semantics of PDX, based on the linking and loading model of the SAS simplify a number of features including application defined ABI, transparent cross-language invocations and heterogeneous distribution.

The single-address-space model of Mungi promotes sharing between applications at the memory level allowing data to be shared across protection boundaries and code to be shared between instances of modules in-place.

References

[CLFL94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *Trans. Comp. Syst.*, 12:271–307, 1994.

[DH99] Luke Deller and Gernot Heiser. Linking programs in a single address space. In *Proc. 1999 Techn. Conf.*, pages 283–294, Monterey, Ca, USA, Jun 1999.

[EH01] Antony Edwards and Gernot Heiser. Components + Security = OS Extensibility. In *Proc. 6th ACSAC*, pages 27–34, Gold Coast, Australia, Jan 2001. IEEE CS Press.

[Fou03] Python Software Foundation. Python reference manual. Internet Web Page http://www.python.org, 2003.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[HCC+98] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in Java. In *Proc. 1998 Techn. Conf.*, pages 259–270, New Orleans, USA, Jun 1998.

[HEV+98] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Softw.: Pract. & Exp.*, 28(9):901–928, Jul 1998.

[Int00] Intel Corp. *Itanium Architecture Software Developer's Manual*, Feb 2000. URL http://developer.intel.com/design/itanium/family.

[Ros94] Timothy Roscoe. Linkage in the Nemesis single address space operating system. *Operat. Syst. Rev.*, 28(4):48–55, 1994.

[SMH00] Fred B. Schneider, Greg Morrisett, and Robert Harper. A language-based approach to security. In *Informatics: 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 86–101, 2000.

[TL03] Ian Lance Taylor and Zembu Labs. *64-bit PowerPC ELF ABI Supplement*. IBM Corporation, January 2003.