Safe Dynamic Linking in an Extensible Operating System

Emin Gün Sirer Marc E. Fiucynski Przemysław Pardyak
Brian N. Bershad

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

February 1, 1996

1 Introduction

The protection of operating system code from user code in most systems is based on the separation provided by an architecturally enforced user/kernel boundary. The boundary isolates an application from the kernel and from other applications. Only through the system call interface can applications interact with kernel services or one another. The system call interface has worked well in the past because the number of services and service interfaces offered by the operating system has been relatively small and static, and the frequency of service interaction has been low. Consequently, trust relationships could be accurately and efficiently expressed through the user/kernel boundary.

Operating system requirements are changing, though, and thereby changing the structure of systems. Systems are now being used to support a variety of applications, such as multimedia [Rad93], multiprocessing [WW94], and distributed memory management [FMP+95] that were once considered "fringe" services. In response to these changing demands, systems are now being designed to support application-specific extensions that change the behavior, and commonly the interfaces, of the operating system.

In this paper, we describe the dynamic linking mechanisms used to define and access service interfaces in the *SPIN* operating system. *SPIN* is an extensible system that provides extensive support for executing *safe* code in the kernel's address space. Code is safe if it cannot violate the interfaces against which it has been compiled without causing a checked runtime error. Safety requires that interfaces can not be circumvented using unsafe memory operations such as pointer casting. For example, a program using unsafe memory operations could force a call to a system-private function (such as `HALT`) by forging a pointer to a function through a cast operation. Clearly, such casts must be disallowed if dynamically linked code is to execute safely. Several projects, including our own [BSP+95], are now exploring the use of kernel extension technologies which preclude unsafe pointer operations [WLAG93, Luc95, VGA94, EKJ95]. With *SPIN*, applications define system extensions using the well-defined safe subset of Modula-3 [Nel91].

Once an infrastructure for the safe execution of code is in place, it is necessary to consider the machinery for naming and linking that safe code into a running system. In this paper we describe a dynamic linker that provides for the safe presentation and occlusion of interface implementations for safe code executing within the kernel. Our linker defines the environment in which safe code executes, enabling code to name,

This research was sponsored by the Advanced Research Projects Agency, the National Science Foundation (Grants no. CDA-9123308 and CCR-9200832) and by an equipment grant from Digital Equipment Corporation. Bershad was partially supported by a National Science Foundation Presidential Faculty Fellowship. Sirer was supported by an IBM Graduate Student Fellowship. Fiuczynski was partially supported by a National Science Foundation GEE Fellowship.

combine, communicate, and authorize interfaces and collections of interfaces safely within the kernel. The key attributes of our linker are:

- Precision. The unit of linkage in our system is the *domain*, which is a collection of code, data, and exported symbols. Domains are constructed from interfaces and their underlying implementations. The symbols exported from a domain are those that are described in the domain's component interfaces.
- Security. Domains are named by typesafe references that act as capabilities. A domain can be safely passed to other code for use as a source of symbols in a subsequent linking operation. Domains can also be registered with an authorization function in an in-kernel nameserver. A potential interface client can interrogate the nameserver for a domain containing the interface's implementation. The nameserver provides access to the domain only if the registered authorization procedure allows it.
- Flexibility. Domains can be stacked or combined allowing services to be arbitrarily composed and presented to clients at runtime.
- Simplicity. Any code within the kernel can create a domain from an interface against which it has already been linked. This allows extension code to construct new domains that encapsulate existing and accessible service interfaces. We use automatically generated stubs to handle the import and export of interfaces through domains.
- Speed. Code in separate domains is linked by replacing symbol names with symbol values within the code as in traditional static linking. Once linked, cross-domain access can occur without system interaction.

SPIN's dynamic linker enables, but does not require, applications to execute within the kernel. *SPIN* also provides a conventional address space abstraction and a mechanism for crossing the user/kernel boundary. In this way, applications can execute entirely within the kernel, entirely in user-space within their own address space, or in some combination of both spaces. For example, our UNIX server is partitioned into a kernel component that provides a thread and virtual memory interface similar to the one found in the Mach kernel, and a user-level component with a structure similar to the CMU's UNIX server [GDFR90].

In the rest of this paper we describe the design and interface for our dynamic linker. We describe the interface first in terms of the linker's primitive operations used to manipulate domains at runtime. We then describe a higher level configuration interface for system and application programmers. The higher level interface is used to automatically generate code that invokes the appropriate lower level linking operations to achieve some desired configuration at runtime.

2 Interfaces, protection, and access

An operating system kernel relies on interfaces to both define *and* restrict access to system services. In a conventional kernel, the system call dispatch facility completely defines the set of services available to applications. In a system where untrusted code executes within the kernel, services must be protected through more restrictive linkage mechanisms. Simply because a potential client has been compiled against an interface does not imply that the client has the right to use that interface. For example, the *SPIN* kernel implements a set of functions that manipulate processor priority level in the `SPL` interface. Clients call functions such as `SPL.High()`, or `SPL.Restore()` to raise and restore the processor interrupt level. These functions can not be made available to arbitrary applications running in the kernel, yet application code can easily be compiled against the `SPL` interface, which is described by a file somewhere in the file system. Even if the file were protected, there are no mechanisms that prevent clients from providing their own version of the interface file and presenting it to the compiler. However, a client that intends to use a service described by an interface must be authorized prior to the use.

Authorization can occur either at link-time or at call-time. Link-time authorization is superior in that it need be done only once, when a client first contacts a service, and its overhead can be amortized over many calls. Traditional operating systems that export a system call interface effectively perform authorization at call time, since all exported procedures are accessible by clients regardless of their privilege status. The system calls that are restricted to a subset of users must check, themselves, if the call is being performed by an authorized user. This check can be error prone and costly when services export fine-grained operations.

Our linker enables services to perform authorization at the point where extension code is being linked into the kernel. For example, the kernel exports the specific `SPL` interface only to privileged applications. There are no restrictions to compile-time access to the interface description file for `SPL`, but at link time, non-privileged applications would fail to link.

3 Dynamic linking primitives

The kernel's dynamic linker manipulates code within specific domains, each of which is a collection of code and data with an associated symbol table that describes the symbols that are exported by the domain. The symbols exported by a domain are those that are intentionally *externalized* by the compiler and the assembler based on programmer directives. For example, a code module that implements an interface implicitly externalizes the symbols that define the entry points described in the interface. In terms of dynamic linking, all domains are created at runtime, either by operating on accessible interfaces, or by manipulating existing domains. Our build environment provides tools for specifying at compile time, any domains that should be created at runtime, or that are required at runtime, for a particular target being built.

Domains can be intersecting or disjoint, which enables applications to share services or define new ones. Figure 1 summarizes the major operations on domains. A domain is created using either `Create` or `CreateFromInterface`. The former define domains from object files or libraries, while the latter defines them from interfaces already accessible within a loaded module. Any symbols exported either by modules defined in the object file (`Create`) or the modules implementing the interface itself (`CreateFromInterface`) are exported from the domain. Any imported symbols and unresolved symbols are left unresolved. Unresolved symbols correspond to interfaces imported by code within the domain for which implementations have not yet been found.

The *Resolve* operation serves as the basis for dynamic linking. It takes a target and a source domain, and resolves any unresolved symbols in the target domain against symbols exported from the source. During resolution, text and data symbols are patched in the target domain, which ensures that, once resolved, domains are able to share resources without protection overhead or system interaction. Resolution only resolves the target domain's undefined symbols; it does not cause additional symbols to be exported. For example, suppose domain `A`, exports symbol `x` and imports symbol `y`. If `A` is resolved against domain `B`, which exports symbol `y`, then `A`'s export set has `x` in it, but not `y`, and `A`'s reference to symbol `y` has been patched to access the exported `y` from `B`. Cross-linking, a common idiom, occurs through a pair of *Resolve* operations.

The *Unresolve* operation is the inverse of a *Resolve* operation, in that it breaks a symbol to value association that was made in a previous *Resolve* operation. The symbols that were previously imported from the source domain are changed to be undefined, and can be relinked against a newer version of the interface.

The *Combine* operation creates linkable namespaces that are the union of existing domains, and is used to bind together collections of related interfaces. For example, if domains `A` and `B` were combined, the resulting domain would export both `x` and `y`. A combination domain is similar to an incrementally linked object file supported by many UNIX loaders.

We commonly combine sets of kernel interface implementations into a single domain that kernel extensions can resolve against, rather than a list of individual domains to simplify linking. At boot time, for example, the kernel defines a domain called `SpinPublic` which contains the interfaces available to all code that executes within the kernel. Any extension can be linked against the `SpinPublic` domain. Conceptually, the interfaces in `SpinPublic` correspond to those defined in a traditional kernel system call interface – simply making them available for use does not violate the system's integrity. A second domain, `SpinPrivate`, contains interfaces for privileged services. These interfaces, if misused, could result in system failure, so the `SpinPrivate` domain

```
INTERFACE Domain;

TYPE T <: REFANY;  (* Domain.T is opaque *)

PROCEDURE Create(coff:CoffFile.T):T;
(* Returns a domain created from the specified object
   file (``coff'' is a standard object file format). *)

PROCEDURE CreateFromInterface(interface: RTCode.InterfaceUnit):T;
(* Create a domain containing interfaces defined by the
   calling module.  This function allows modules to
   name and export themselves at runtime. An RTCode.InterfaceUnit
   is a runtime descriptor for a specific interface. *)

PROCEDURE Resolve(source,target: T);
(* Resolve any undefined symbols in the target domain
   against any exported symbols from the source.*)

PROCEDURE Unresolve(source,target: T);
(* Change all symbols in the target domain that were previously
   imported from the source domain to be unresolved. *)

PROCEDURE Combine(d1, d2: T):T;
(* Create a new aggregate domain that exports the
   interfaces of the given domains. *)

END Domain.
```

Figure 1: *The Domain interface.*

is only made available to a few, privileged clients. An unprivileged, but potentially malicious, extension, could not resolve its reference to an interface implemented within the `SpinPrivate` domain.

3.1 Creating domains

Modules can either create domains for interfaces which they themselves implement, or for which they have already been linked. In the first case, a module can create a domain for itself and pass it out to authorized clients. For example, the kernel's `SPL` module creates a domain that contains the implementation of the `SPL` interface using a code fragment that looks like this:

```
MODULE SPL EXPORTS SPL;
IMPORT Domain;
VAR d : Domain.T;
PROCEDURE High() : T ...
PROCEDURE Restore(s: T)...

BEGIN
   d := Domain.CreateFromInterface(INTERFACE_UNIT(SPL))); ...
END SPL.
```

The builtin function `INTERFACE_UNIT` describes the specific language-level interface named in the argument [HFC+96]. It returns a valid value only if the caller has already been linked against an implementation of the named interface.

Until a domain has been completely resolved (that is, all symbols have been assigned values by the linker), it is not eligible for execution. This restriction prevents code that is "language-safe" but not "system-safe" from executing. Language-safe code is any code that has passed through the compiler, and has been verified to not cause any unchecked runtime errors. More strongly, system-safe code is any code that is language-safe, and does not access interfaces for which it is unauthorized.

Domains created using the `CreateFromInterface` operation require a pre-existing linkage to the interface. The call to `INTERFACE_UNIT(SomeInterface)` forces a reference to symbols within `SomeInterface`, and can only execute after the calling module has been fully resolved. Simply put, the ability to call `INTERFACE_UNIT` on an interface is prima facie evidence that the caller has been given the right to access the specific interface. Once the caller has been linked, either statically or dynamically, the interface becomes legitimately nameable by the caller and can be encapsulated within a domain with which it can be passed to others.

3.2 Why not objects?

Object-based systems [WLH81, LCJS87, JLHB88, Bro94] offer a model in which a protected object instance acts as an unforgeable capability that governs access to the methods of that object. In effect, the object reference enables an implicit form of call-time authorization. For instance, in an object-based system, a privileged server would protect its interface by exporting an object type, e.g. `SPL.T`, that contains the entry points as its methods. Clients of such an interface would have to obtain an instance `spl` of an `SPL.T`, and invoke a method on it to request service, e.g. `spl.restore()` (the C++ equivalent would be `spl->restore()`).

We rejected relying on objects for protection because that would place all interfaces into a single namespace, causing the names of interfaces and types to become a global resource for which conflicts can occur. Moreover, correctly guessing the name of a type can allow a client to narrow a restricted object to a subtype with greater privileges. Consequently, some sort of type namespace management facilities are required to rely on objects for protection. Our dynamic linker in effect provides these facilities at the interface level, making it possible to use interfaces to control access to both implementation modules and object types.

3.3 Exporting and importing domains at runtime

A domain can either be passed directly to another module as an instance of a `Domain.T`, or to a nameserver that maps all domains into a namespace indexed by strings. In the first case, the exporting module can define an arbitrary protocol through which potential clients obtain a legitimate instance of the domain descriptor. For example, our kernel's console module exports two interfaces: `Console` and `ConsolePrivate`. All extensions are given access to the `Console` interface, but privileged code can use the `ConsolePrivate` interface. `Console` exports a procedure `GetPrivateInterface()` that returns a `Domain.T` for the implementation of `ConsolePrivate`. The code that implements this function is essentially as follows:

```
MODULE Console EXPORTS Console, ConsolePrivate;

PROCEDURE (*ConsolePrivate*)GetPrivateInterface() : Domain.T =
  BEGIN
    IF IsPrivileged(Identity.Self()) THEN
        RETURN Domain.CreateFromInterface(INTERFACE_UNIT(ConsolePrivate));
    ELSE
        RETURN NIL;
    END;
  END GetPrivateInterface();
```

Alternatively, domains can be exported through a nameserver that is part of the `SpinPublic` domain (all code that executes within the kernel can access the nameserver). The nameserver is not part of the linker itself; it is simply an auxiliary structure that modules can use to advertise and learn about domains. An exporter registers a `Domain.T` descriptor with the nameserver, together with an optional authorization procedure that is called each time the domain is requested from the nameserver. A module acquires a domain handle by calling `NameServer.Query` and passing in the name of the interface to be imported. The call either returns a `Domain.T` that holds the implementation of that interface, or raises an exception when the named domain has not been registered with the nameserver, or when access to the caller is denied.

Since it is an error to execute code within a domain that has not yet been fully resolved, domains are unable to resolve themselves. We use small, specialized binder procedures to link together a collection of domains into a single executable. The binder is automatically generated when the extension is itself built. The binder acquires a domain descriptor for each interface in the program that it is binding, and cross links and combines these domains into a new domain in order to create a fully resolved application.

3.4 Automatic construction of domains and binders

Programming the runtime import and export of modules and interfaces can be tedious and error prone since all components must be precisely described, both to the compiler at static compilation time, and to the domain system at link and run time. For the most part, though, usage is stylized and can be expressed within code templates that are automatically generated during program build time. From the standpoint of the provider or client of an interface, dynamically linked code looks and runs like statically linked code. Minimally, the only difference between static and dynamically linked code is found in the build directives that drive the system's configuration. Neither the dynamic linker nor the kernel implements the machinery for making resolution transparent. This occurs within code generated during the build process. Indeed, the import and export "stubs" used to make dynamic linking transparent are similar in spirit and structure to the import and export binding stubs found in every Remote Procedure Call system built in the last decade. The big difference, of course, is that dynamically linked code, once linked, is accessible directly, whereas RPC services require several levels of indirection.

Our build process relies on a program configuration language called *quake* to automatically construct domains and binders. Quake is distributed as part of DEC SRC's Modula-3 build environment, and serves the same purpose as *make*, in that it operates on files and dependency information.

We have defined several quake directives that make it easy to create, export, and import domains for dynamic linking. Quake is also used to compose modules statically. To illustrate the use of the linking commands, we briefly describe the specification of a code module using static linking, and then show how to define it to be dynamically linked.

Consider a kernel extension that sends and receives UDP packets. The extension is implemented in a module called UDP which exports an interface of the same name.

The quake commands that package the interface and implementation as a library against which clients statically link is described as:

```
Module("UDP") # compile the module UDP against the interface UDP.
Library("UDP") # package the result into a library called libUDP.a
```

A client called `SomeClient` that imports this library is described by

```
Import("UDP") # statically link against the UDP library.
Module("SomeClient") # Compile the module "SomeClient."
Program("test") # Generate the program called "test."
```

With dynamic linking, we compile the module "UDP," but also provide directives at buildtime that enable the interface to be exported at runtime:

```
# Create a new domain called UDP-extension.
DomainCreate("UDP-extension")
# Compile the module UDP against the interface UDP.
Module("UDP")
# Export the UDP interface as part of the UDP-extension domain.
```

```
DomainModule("UDP-extension", "UDP")
# Create a dynamically linkable object file called "UDP-extension."
Extension("UDP-extension")
```

The `DomainModule` command creates an auxiliary interface and module called `UDPInterface` that defines an `Export` function. The implementation of `UDP` should call the export function to register the UDP interface with the nameserver. The `Extension` command compiles the `UDPInterface` module as well as the `UDP` module, and statically links them together.

To illustrate the client side, consider the structure of a simple multicast extension called "UDPmcast" that takes packets in on one port and retransmits them out a number of others. This extension, which is to be dynamically linked against the "UDP-extension" domain, is described as follows:

```
# Generate a binder that imports the UDP-extension domain.
DomainImport("UDP-extension")
# Compile the module UDPmcast against its interface.
Module("UDPmcast")
# Create a dynamically linkable object file called "UDPmcast."
Extension("UDPmcast")
```

The `DomainImport` directive generates the binder function for `UDPmcast` that at runtime queries the nameserver for the domain associated with "UDP-extension", creates a domain for the module `UDPmcast`, and resolves it against the extension domain.

4 Related work

Operating systems now commonly provide an interface that allows applications to install new code into a running kernel. Typically, these interfaces are used to dynamically link device drivers, file systems, and network protocols into a running system. The advantage of this approach is that a kernel can be shipped minimally configured, and specialized at runtime to a particular installation with little runtime penalty. In the most straightforward and common implementations of kernel dynamic linkers, the download interface provides unconstrained access to system services, which enables any piece of code, once installed into the kernel, to interact with any other system resource. Consequently, the right to install code into a running kernel is either limited, or the system has poor stability in the presence of extensions as is the case with either the Macintosh operating system, or Windows.

The FLEX project at the University of Utah [OBLM93] provides a set of rich linking facilities based on primitives similar to those described in this paper (`Create, Resolve, Combine`). That system is concerned with the flexible specification of link-time constraints using a specialized linking language for composing object files. Although we use a different language for specifying the composition, our approaches are similar. We are not aware of any support in FLEX for interface authorization, so it is not clear how effective the service is in managing shared code and data resources in a potentially hostile environment. We believe that the simple mechanisms we describe for authorization could easily be provided within FLEX.

The Spring system [MGH+94] includes a versatile linker that supports sharing between applications through cross address space linking. The linker manipulates object files, rather than interfaces or collections of interfaces, making it more system-oriented rather than service-oriented.

5 Summary

Dynamic linking is a service that touches directly on compilers and operating systems. A compiler generates directives that manipulate interfaces and collections of interfaces, and the operating system provides the mechanisms to ensure that those directives are respected. In this paper, we have described a dynamic linking interface that allows operating system services to safely present themselves to clients as linkable code units which represent implementations of interfaces and collections of interfaces. Applications and system services can interact with the dynamic linker either through its low-level domain-oriented service interface, or through a higher level one specified at program build time.

References

[Bro94] Kraig Brockschmidt. *Inside OLE 2*. Microsoft Press, 1994.

[BSP+95] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin G n Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, December 1995.

[EKJ95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, December 1995.

[FMP+95] Michael J. Feeley, William E. Morgan, Frederic P. Pighin, Anna R. Karlin, Henry M. Levy, and Chandramohan A. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, December 1995.

[GDFR90] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. Unix as an Application Program. In *Proceedings of the 1990 Summer USENIX Conference*, pages 87–95, June 1990.

[HFC+96] W.C. Hsieh, M.E. Fiuczynsko, C.Garrett, S.Savage, D.Becker, and B.N. Bershad. Language Support for Extensible Systems. In *The First Workshop on Compiler Support for Systems Software*, February 1996.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[LCJS87] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. Implementation of argus. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 111–122, November 1987.

[Luc95] Steve Lucco. The Bridge Web Page. Technical Report http://www.cs.cmu.edu/afs/cs.cmu.edu/project/sfi/www/top.html, Carnegie Mellon University, 1995.

[MGH+94] J. Mitchell, J. Gibbons, G. Hamilton, P. Kessler, Y. Khalidi, P.Kougiouris, P. Madany, M. Nelson, M. Powell, and S. Radia. An overview of the spring system. In *Proceedings of Compcon Spring 1994*, February 1994.

[Nel91] Greg Nelson, editor. *System Programming in Modula-3*. Prentice Hall, 1991.

[OBLM93] Doug Orr, John Bonn, Jay Lepreau, and Robert Mecklenburg. Fast and Flexible Shared Libraries. In *Proceedings of the 1993 Winter USENIX Conference*, June 1993.

[Rad93] Steven Radecki. *Multimedia With Quicktime*. Academic Press, 1993. ISBN 0-12-574750-0.

[VGA94] Amin M. Vahdat, Douglas P. Ghormley, and Thomas E. Anderson. Efficient, Portable, and Robust Extension of Operating System Functionality. Technical Report CS-94-842, University of California at Berkeley, December 1994.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, December 1993.

[WLH81] William A. Wulf, Roy Levin, and Samuel P. Harbison. *Hydra/C.mmp: An Experimental Computer System*. McGraw–Hill, 1981.

[WW94] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–11, Monterey, CA, November 1994.