SLK: A Capability System Based on Safe Language Technology

Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski,
Deyu Hu, and Thorsten von Eicken
Department of Computer Science
Cornell University

— DRAFT —

ABSTRACT

Safe language technology allows multiple protection domains to coexist within a single address space. The protection is enforced by the language system, in particular the type system, which provides unforgeable object references. This paper develops a new capability model (called the type-capability model) which relates the protection offered by safe languages to that of traditional capability systems. This model is used to show that the power of capabilities can be obtained in a safe language at low cost through a combination of link-time and run-time protection checks.

The Safe Language Kernel (SLK) leverages the type-capability model to implement multiple protection domains with low-overhead fine-grain sharing in a single address space. A Java-based prototype implementation of SLK is used to describe the mechanisms available for sharing data and code. A set of micro-benchmarks demonstrates the low overhead of crossing protection domain boundaries and three server-based application studies (an extensible web server, a shared annotation system, and Active Networks) illustrate the use of SLK.

1 Introduction

Several operating systems projects [3,7,23] have explored the use of safe languages to allow untrusted extensions to be loaded into the address space of the kernel. The language system is used to enforce protection boundaries so that loaded extensions can only access data structures explicitly made visible by the kernel. With the advent of Java-enabled web browsers, protection based on language safety has come into widespread use. The main advantages of language-based protection are the fine granularity of access control on a per-object basis, and the low cost because the compiler and linker perform the vast majority of checks.

Both extensible operating systems and web browsers have focused on sharing between the central core and extensions, but not necessarily among extensions. In fact, there is relatively little work to show how a multitude of protection domains can share code and data structures safely among each other at a fine-grained level. This is most obvious when noting the absence of well-defined inter-applet communication in web browsers.[1]

Arguably, capability systems provide the best understood protection model for the flexible sharing of resources among a multitude of protection domains. A capability is an unforgeable handle onto a resource and consists of a reference to the resource and the access rights of the capability holder. Capabilities allow programs to implement sharing policies themselves by controlling how capabilities to their resources are passed around. A major problem with capabilities lies in the run-time access rights check required on every use. For this reason, most systems have resorted to either providing hardware support for capabilities or using capabilities only for “large objects.”

This paper presents a model and associated mechanisms that almost provide the best of both worlds, e.g., capabilities at the cost of regular object references. The key is to identify a large class of access rights checks that can be performed at link-time, and that thus do not incur any run-time overhead. Only the access checks that cannot be performed at link time require run-time checks. In order to explain the relationship between linking and capability access rights checks Section 2 introduces a new protection model, called the type-capability model, which

[1] The need for such sharing was demonstrated by the developer uproar, which occurred when the loopholes used for inter-applet communication in beta versions of browsers were plugged. The current state of affairs is that browsers provide ill-defined back-door communication channels.

relates capabilities with object references (pointers) in safe languages. Unlike previously claimed [3], object references are not in themselves capabilities. Rather, within a linkage module (the unit of code manipulated by the linker), all uses of object references of a given type share the same access rights, and these access rights are determined at link time.

In summary, the type-capability model serves a number of purposes:

- It relates the protection model offered by safe languages to capability systems, showing that unforgeable object references in safe languages provide a subset of the protection features of capabilities,
- It demonstrates that the type system is not only central to protection in safe languages, but that giving programs control over the dynamic loading of their own types leads to useful protection paradigms while preserving safety.
- It subsumes the protection models offered by Java and Modula-3, which can be described and implemented in terms of the model. In addition, mapping Java onto the model clarifies a number of "obscure" areas in the language definition that have led to a series of bugs in Java class loaders.

The Safe Language Kernel (SLK) is an implementation of the type-capability model that uses a type safe language and provides mechanisms for multiple programs to share data and code safely. The prototype implementation discussed in this paper is based on standard Java virtual machines. It takes the form of a library that offers tools to manipulate type-capabilities that are checked at link-time as well general capabilities that require run-time checks. The primary motivation for the implementation is to provide a substrate for developing server systems into which untrusted component software can be loaded.[2]

Due to the fact the current implementation uses standard Java virtual machines, it is limited to the subset of the type-capability model that can be exposed through the built-in features of the Java API. Future version of SLK will replace parts of the JVM (in particular the class loader) to provide the full flexibility of the model. Even the current implementation, however, uses the available protection mechanisms in new ways beyond the goals of their designers.

Applications that can immediately benefit from SLK are Internet-related systems such as web servers. These servers contain most of the features expected in operating systems (thread scheduling, memory allocation, protocol stacks, etc.) and can benefit from extensibility in the same way. Indeed, all current web servers support the CGI interface, which is inefficient as it requires the creation of a new process for each HTTP request. While newer proposals, such as FastCGI, reduce the cost to an LRPC per request, leading servers now allow extensions to be loaded directly into their address space. Microsoft IIS' ISAPI or Jeeves' servlets are examples. The major problem is that neither of these servers is fault isolated from loaded extensions, that is, faults (or malicious code) in the extensions can affect the server's operation. SLK provides mechanisms for extensions to execute in the server address space, reducing the cost of processing requests, while still protecting the server from the extensions, and the extensions from one another.

To illustrate this web server example further, the paper describes the implementation of a dynamic web server that allows users to customize and service a URL sub-space by uploading untrusted programs. A shared annotation system for collaborative learning is built on top of this server and illustrates the sharing between server extensions.

The remainder of this paper is organized as follows. Section 2 presents the SLK protection model. Section 3 describes static protection mechanisms in SLK. Dynamic protection mechanisms are introduced in Section 4. The performance of the mechanisms is evaluated through a number of microbenchmarks in Section 5. The design of three larger applications is described in Section 6. Section 7 discusses related work and relevant operating system techniques. Finally, Section 8 summarizes the paper.

2 Capabilities in a Safe Language Environment

In traditional operating systems, protection is based on address spaces and privileged execution modes. In particular, threads can attempt to access data at arbitrary addresses, but the system, running in a privileged mode, controls the mapping from addresses to data. Protection in SLK, in contrast, is based on controlling which addresses a

[2] The security models described could also be used to build a multi-user Java based operating system running directly on bare hardware in the style of JavaOS [19]. However, although interesting, this setting is not further discussed in this paper.

computation can form. The fundamental notion is that object references cannot be forged, so that a computation can access an object only if it is explicitly given a reference to it.

Basing the protection model on unforgeable references can be traced back to the beginnings of capability systems [9] (see [17] for a summary). A capability consists of an object reference and a set of access rights. Every operation performed on the object checks the access rights to determine whether the operation is permitted. The machine also ensures that capabilities cannot be forged. In one form of traditional capability systems, a thread of execution has an associated capability list (often called C-list) and uses indexes into the C-list as object references.[3] Each capability in the C-list consists of the address of an object and a description of the thread's access rights to that object. One of the powers of capability systems derives from controlled run-time modifications of the C-list. In particular, a cross-domain call not only transfers control and passes arguments but also modifies the C-list. For example, the caller can remove capabilities or access rights to restrict the permissions of the callee. Similarly, the called procedure can have an associated template to cause the C-list to be augmented such that, for the duration of the call, the callee gets privileges that the caller didn't have.

Because of their flexibility in specifying access rights, capabilities are an attractive mechanism to address the needs of applications requiring fine-grained sharing in a safe language environment. However, while the flexibility of capabilities is appealing, the run-time overhead associated with capability access rights checks is not. This section introduces the type-capability model which partitions capabilities into two categories: those with access rights usage that can be modeled by the type system of a safe language, and those where this is not possible. The advantage of this partitioning is that the cost of dynamic checks is incurred only for those capabilities that need them. Capabilities whose access rights can be expressed with a language's type system require virtually no run-time checks because the type-checking can be done at link-time. It is important that the type-capability model retains the tools necessary to form capabilities with dynamic checks when the extra functionality is required.

The observation underlying the type-capability model is that types can represent access rights in that a code module can only access the elements and methods of objects for which it "knows" the type. For example, if a module receives a reference to an object for which it does not know the type, it cannot access any part of the object. Similarly, sophisticated type systems allow a module to export objects with a restricted type that does not describe all elements and methods of the object, consequently other modules receiving a reference to these objects do not have access rights to the hidden elements and methods.

The major restrictions imposed by using types to encode access rights are that all objects of the same type share the same access rights and that these access rights cannot be changed at run time (unless the type of an object can be changed at run-time). This makes sense, because …

2.1 The Type-Capability Protection Model

The type-capability model augments object references with access rights derived through the type of the object reference. The combination of the object reference and these access rights forms a capability for which the access rights checks can generally be performed at link time.

Each type is represented in the model by a *type descriptor*, which is an object that describes all attributes (including fields and methods) of the type. Access to a type descriptor is controlled by *type capabilities*, which contain the name of the type descriptor and a bit vector of access rights. Each bit in the access rights vector enables the holder of the type capability to perform certain operations on objects of the type specified by the type descriptor. (Operations might be accessing an object's field or invoking a method on the object).

[3] Another major class of capability systems eliminates the C-list and uses the capabilities directly as object references. This allows capabilities to be embedded into data structures but requires tagging to protect the capabilities from tampering. A third class uses a large sparse namespace to prevent tampering with capabilities. In general, the systems are equivalent, but the C-list style is more easily compared to the language-based approach.

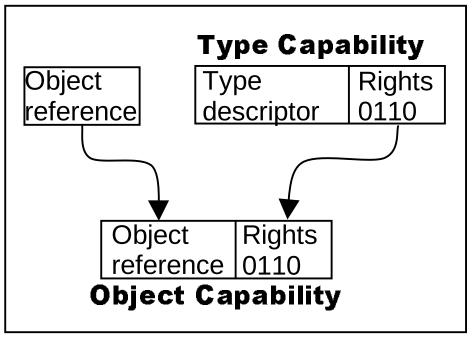
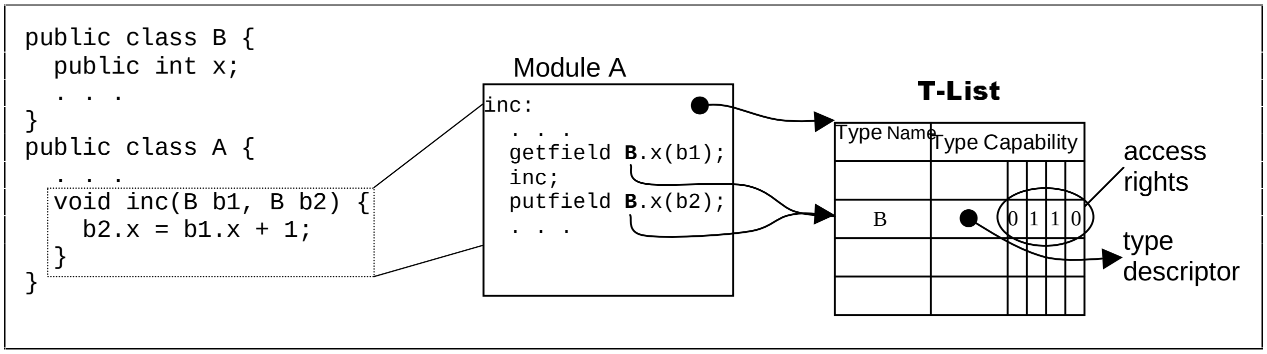


Figure 1: Combining an object reference with a type capability to form a capability for the object.

In order to allow for link-time checking of access rights, type capabilities are associated with object references by the linker. For each code module, the linker builds a list of type capabilities, called the T-list. An entry in the T-list associates a type name used in the module with an appropriate type descriptor and the corresponding access rights. The linker can thus form a capability by combining an object reference with the access rights of the type capability in the T-list whose type name matches the compile-time type of the object reference.[4] Figure 2 demonstrates how the compile-time type of a reference is used to index into a module's T-list.



```
public class B {
  public int x;
  . . .
}
public class A {
  . . .
  void inc(B b1, B b2) {
    b2.x = b1.x + 1;
  }
}
```

```
inc:
  . . .
  getfield B.x(b1);
  inc;
  putfield B.x(b2);
  . . .
```

Figure 2: The method `inc` in class `A` performs read and write accesses to objects `b1` and `b2` of the type `B`. A type descriptor is obtained by looking up the type capability of `B` and checking the access rights for field `x` in `A`'s T-List.

Due to its tight integration into the protection model, linking is not simply an add-on to the language model but is instead an integral part of the language itself. By manipulating type capabilities which, like object references, can be passed around freely, a program can control the name space and the access rights of newly loaded modules. It can similarly

Two key observations can be made about this protection mechanism:

- Because a module's T-list is constructed at link time and never modified thereafter, and because the compile-time type of an object reference is fixed, the access checks for all operations can be performed at link-time.
- The combination of an object reference and access rights from the matching type capability forms a capability for the object. However, the type-capability model is strictly less powerful than traditional capability models because, within a module, all objects of the same type share the same access rights, and those access rights cannot be changed at run-time.

[4] The compile time type of an object reference is the type of the variable holding the reference. In an object-oriented language, the type of the object being referred-to may be a sub-type of the variable's type.

Java's class system (a Java class is a type) can be described in terms of type capabilities as follows. Each type capability holds a reference to a Java class descriptor and four access bits. These access bits determine whether the holder of the capability can access the private, package (default scope), protected[5], and public members of an object of that type. Java programs manipulate *class descriptors* rather than type capabilities: at link time, a program uses a Java class loader to link new code, and the program supplies a class descriptor for each unresolved class name. The Java class loader enforces the language's scoping rules by expanding the class descriptor into a type capability with the appropriate set of access rights[6]. This type capability is then added to the new module's T-list.

Most treatments of Java have described the scoping rules associated with the four access qualifiers (private, package, protected, public) without any reference to class descriptors. These treatments miss an important dimension of the Java type system: access to members of an object is determined first by which class descriptors a module can gain access to, because this determines which type capabilities go in a module's T-list. The four access qualifiers then determine which access bits are set in each type capability, but these only matter if a module can gain access to the type capability in the first place.

The model underlying SPIN's dynamic Modula-3 linker corresponds closely to the type-capability model. Modules must gain access to a domain object exporting a set of interfaces in order to link against these interfaces. In Modula-3, a single type can be exported differently by multiple interfaces. Each interface reveals a subset of the members of a type and hides the other members. Linking against an interface gives a module access to all of the members of a type revealed by that interface. A type capability is then a type plus one access bit per interface that reveals members of the type.

In summary, unforgeable object references are less powerful than traditional capabilities, in that the access rights associated with an object reference are shared among all objects of that type and are fixed on a per-module basis at link time. The major advantage of language based protection is efficiency: enforcement of regular capabilities requires expensive run-time checks, while enforcement of type-safety can be done largely at compile and link time. The next section shows how the type-capability model can be used to share classes among protection domains and the following section shows how dynamic checks can be added to realize the full power of capabilities.

3 Static protection mechanisms

This section describes how the Safe Language Kernel implements the type-capability model and allows programs to manipulate the type capabilities that newly loaded code modules have access to.

The type-capability model provides a foundation for reasoning about protection at the type level or, from the Java perspective, at the class level. This allows programmers to determine whether a module can access fields or methods of objects based on the type capabilities provided at link time. However, for writing programs consisting of many classes, a higher-level view is desirable. The SLK program loader provides the abstraction of a Java program, which is defined as a collection of classes that are linked with each other under a single set of rules. Specifically, each program provides its own class loader, which is a class that is invoked by the run-time system to resolve class names to class descriptors.

Any program can start a new program by invoking the SLK program loader and by passing to it the code of the new program's class loader as well as a directory of class descriptors. The program loader instantiates the new class loader and asks it to load the class `main`. The directory is used by the program loader to resolve external references in the class loader itself. The class loader has full control over the namespace of the program it is loading: it can load classes from the file system or the network, or it can acquire class descriptors from other running programs. Most class loaders will search for class files along a simple path, but when programs share classes, the ability to rename them is often required and simple to provide. The class loader is only limited by the class files and class descriptors it can acquire, as well as by the type-check that occurs after it resolves a reference.

[5] Java's semantics for protected are complex and require some extensions to the type-capability model which are beyond the scope of this paper.

[6] These scoping rules are more limited than the model in ways that are sometimes frustrating—for instance, Java does not allow a class to belong to more than one Java package, although the type capability model would handle this.

The current SLK program loader is implemented using the Java class loader mechanism, which introduces a number of unfortunate limitations. The Java Virtual Machine (JVM) asks a given class loader to resolve a given class name only once, making it impossible to map two references to a class name to two different classes. The JVM also forbids the renaming of classes, i.e., the class name in the class descriptor provided by the class loader must match the name requested by the JVM. The type capability model shows that these restrictions are not required for safety, as long as the correct type checks are performed.

3.1 Private and shared classes

A program can completely insulate itself from others by loading its own private copy of every class it needs, with the exception of some system classes in the `java.lang` package. By loading its own copy, a program creates a new class descriptor for each class, thereby obtaining its own set of static fields (i.e., global variables).

Programs can share classes by sharing their class descriptors. Typically one program loads the class from a file, places the resulting class descriptor into a repository, from which the second program fetches it for linking. Sharing classes in this manner has two implications. First, both programs share the same static fields and second, both programs can examine fields and invoke methods on objects of that class.

When sharing classes, trust must be considered. In particular, a program retrieving a class descriptor from a repository cannot be exactly sure of the behavior of the shared class, since its code was defined by the first program. In some cases, “blind trust” or some form of authentication may be appropriate. (The next section introduces mechanisms that allow class descriptors in repositories to be signed.) However, in other situations programs may need to have guarantees over behavior of a shared class. One example of this is when a shared wrapper class is used (introduced in Section 4). In this case, the SLK program loader validates that a shared wrapper code satisfies certain requirements.

This explicit sharing of class descriptors gives programs a tool to restrict the interfaces of shared objects. Instead of sharing a class outright, a program can share a more restrictive super-class. Figure 3 shows an example where elements of the class `Point` can be selectively exported through the `Point_if` interface.[7] It is important to note that as long as a program does not share the class descriptor of `Point`, it is impossible for programs importing `Points` as `Point_if`’s to cast the references to `Point`. This holds even if they “know” that the true type of the object is `Point` and even if they have loaded their own copy of `Point`.

```
// Class of objects to be exported
public class Point implements Point_if {
  private int x, y, color;
  // constructor
  public Point(int x, int y, int c) {…}
  // draw point on screen
  public void draw() {…}
  // change point attributes
  public void setColor(int newC) {…}
  public void setPos(int x, int y) {…}
}

// Limited interface
public interface Point_if {
  public void draw();
  public void setColor(int newColor);
}
```

Figure 3: Sharing classes and interfaces.

For this hiding of classes behind interfaces to work in current Java virtual machines, some minor restrictions on the Java API have to be introduced. In particular, access to the methods `getClass` of `java.lang.Object` and

[7] In Java, an interface is a class that only specifies the interface of methods. The bodies of the methods are then provided by the class(es) that “implement” the interface.

getClassLoader of java.lang.Class must be restricted, because they provide backdoors for obtaining class descriptors.

4 Dynamic Protection Mechanisms

SLK provides a toolbox of dynamic protection mechanisms that overcome the limitations of the underlying type capabilities. These dynamic mechanisms emulate traditional capabilities and can provide some forms of authentication as well. All the mechanisms shown create new objects to "wrap" some resource and provide a controlled interface to it.

4.1 Wrappers

A wrapper is an object controlling access to another individual object on a per-method basis. In essence, a wrapper object contains a private pointer to the real object and provides public methods that can be invoked on the real object. Each of these methods can apply arbitrary protection checks and then indirect to the method of the real object. Figure 4 shows how objects of type Point can be hidden under wrappers of type Point_d that provide only limited access.

```
// Wrapper for exporting Points that can only be drawn
public class Point_d {
   // handle onto real object
   private Point p;
   // constructor for creating a wrapper
   public Point_d (Point p) { this.p=p; }
   // exported draw method
   public void draw() { p.draw(); }
}
```

Figure 4: Simple wrapper objects.

The language feature required for implementing wrappers is the hiding of fields from the exported interface of an object, here using Java's private qualifier.

Wrappers allow the same object to be exported to different domains with different access rights by using multiple wrappers as shown in Figure 5: the second wrapper Point_dc also exports the setColor method. While powerful, the two distinct wrappers introduce a typing problem in that Points with different access rights have different types. This can be solved in Java by introducing a Point_if interface shown in Figure 5 that is implemented by Point_dc as well as a revised Point_d.

```
public interface Point_if {
  public void draw();
  public void setColor(int newColor);
}

// wrapper for points that can be drawn & changed
public class Point_dc
    implements Point_if {
  private Point p;
  public Point_dc(Point p) {this.p=p;}
  // exported methods
  public void draw() { p.draw(); }
  public void setColor(int c)
  { p.setColor(c); }
}

// revised Point_d wrapper to implement Point_if
public class Point_d
    implements Point_if {
  private Point p;
  public Point_d (Point p) {this.p=p;}
  public void draw() { p.draw(); }
  // must export a setColor to comply with Point_if
  public void setColor(int c)
  { throw new AccessViolation (); }
}
```

Figure 5: Wrappers with interfaces.

4.2 Capabilities

The basic wrappers can be extended to check rights dynamically on each access. This turns them, in essence, into capabilities. Figure 6 shows the implementation of a simple capability wrapper. It contains private fields describing the access rights and these are checked in the wrapper methods.

```
// Capability for accessing Points
public class Point_Capa {
  private Point p;
  // cookie identifying creator
  private Object cookie;
  // access rights, initially everything is allowed
  private boolean dOK = true, sOK = true;
  // constructor: init the wrapper
  public Point_Capa(Point p, Object cookie) {
    this.p = p; this.cookie = cookie;
  }
  // exported methods checking access rights
  public void draw() {
    if (dOK) p.draw();
    else throw new AccessViolation( … );
  }
  public void setColor(int c) { if (sOK) … }
  // rights revocation
  public void revokeDraw(Object k) {
    if (k == cookie) dOK = false;
    else throw new AccessViolation( … );
  }
  public void revokeSetColor(Object k) { … }
}
```

Figure 6: Simple capabilities

The capabilities shown support rights revocation. That is, the creator of a capability can revoke access rights arbitrarily after having handed it out. To allow only the creator to revoke rights a cookie consisting of an object reference known only to the creator is stored in the capability: this cookie is set when the capability is created and must be passed to the revocation methods as a form of authentication.[8,9]

4.3 Keys, Safeboxes, Signatures, and Turnstiles

The next set of abstractions introduces several forms of authentication using keys, which are modeled on public/private key schemes [28]. The language safety allows keys local to a host to be implemented without resorting to cryptographic support. A private/public key pair consists of two objects, one of which can be publicized (e.g. in a directory) while the other should remain private. The only operation on a key is `match()` which checks whether its argument is a reference to its "other half". Figure 7 shows a Java implementation of keys.

[8] Note that the cookie is an object reference, which cannot be forged, thus it is impossible for a third party to guess the cookie.

[9] The revocation of these simple capabilities may be enhanced to allow "bulk revocation" by adding a level of indirection into the rights check. Instead of maintaining the rights in the capability itself, it is possible to keep a pointer to a bit vector and to share that vector among multiple capabilities. Clearing bits in that vector removes the corresponding rights from all capabilities using the vector.

```
package SLK;

// A key points to its sibling, forming private/public key pairs
public final class Key {
  Key sibling; // accessible within this package only
  public boolean match(Key k)
  { return sibling == k; }
}

// helper class to create key pairs
public final class KeyPair {
  public Key publicKey, privateKey;
  public KeyPair() {
    publicKey = new Key();
    privateKey = new Key();
    privateKey.sibling = publicKey;
    publicKey.sibling = privateKey;
  }
}
```

Figure 7: Keys

Keys alone are not of any use, but they can be embedded in other objects to implement a number of abstractions. The first is a "safebox" that allows an arbitrary object reference to be "encrypted" with a public key such that only the holder of the corresponding private key can gain access to the reference. A typical use of safeboxes is to protect an object from access by intermediaries. Once an object is placed into a safebox, the latter can be passed around arbitrarily with the security that only the holder of the correct private key can extract the object.

The implementation of `safebox` is shown in Figure 8 and it is part of the same SLK package as `key`, which must be trusted by all parties using `keys` and `safeboxes`.

```
package SLK;

// A safebox holds an object which can only be retreived if the
// correct "private key" is presented
public final class SafeBox {
  private Object o; // locked-up object
  private Key pubKey; // key used when locking
  // constructor locks an object up
  public SafeBox(Object o, Key publicKey) {
    this.o = o; pubKey = publicKey;
  }
  // return the object if the right key is presented
  public Object unlock(Key privKey) {
    return pubKey.match(privKey) ? o : null;
  }
}
```

Figure 8: Safeboxes

The origin of an object can be authenticated using `signatures`. The implementation is almost identical to `safebox`, except that the creator "signs" an object with the private key and anyone can use `verify(Key publicKey)` to check authenticity. A separate `get()` method returns the object without key verification. More sophisticated protection mechanisms can be constructed from `safeboxes` and `signatures`. For example, by combining them one can obtain a signed and encrypted message exchange service.

Keys can also be used to authenticate a caller to a callee using a "turnstile" shown in Figure 9. The idea is that the caller presents its private key to the turnstile (via `cross()`), which then invokes the callee (implemented in `method())` with the corresponding public key as argument. In this way, the callee knows the identity of the caller while the latter does not reveal its private key.[10]

```
package SLK;

public abstract class Turnstile {
  protected abstract Object
      method(Key callerPubKey, Object arg);
  public final Object
      cross(Key callerPrivKey, Object arg)
  { return method(callerPrivKey.sibling,arg); }
}
```

Figure 9: Turnstiles authenticate the caller to the callee

4.4 Benefits and limitations of wrappers

The wrappers shown above allow a precise definition of which operations are allowed on a given object. The model provides protection at the method level at the expense of extra storage for the wrapper as well as a level of indirection and a check at every use. Wrappers can implement all aspects of capabilities, including rights revocation.

The wrappers shown here are similar in spirit to proxy objects, stubs, and surrogate objects used in distributed systems. The idea of using safeboxes and signatures in the form shown here can be traced back to early work on type safe languages [24]. However, the functional framework used at the time did not allow the implementation of these concepts in the language itself.

Compared to traditional capability systems with hardware support for dynamic access rights checking, the type-capability model in conjunction with the wrappers provides a different tradeoff. Statically checked accesses to objects are expected to be frequent and can be mapped efficiently onto conventional instruction set architectures, while dynamic protection checks are expected to be infrequent and cost more.

4.5 Threads

In general, a program will have its own set of threads, and these threads need to be protected from interference by other programs (for instance, another program should not be able to kill, suspend, or change the priority of these threads). Java provides some protection of threads in the form of hierarchical thread groups. The thread groups provide two features: they isolate the threads of each program, and they allow the creator of a program to terminate it by killing all threads in the thread group. Cross-domain calls should not break this protection model and therefore should meet the following requirements:

1. if the callee never returns, the caller should be able to back out of the call and recover its pre-call state,
2. the callee should not acquire the right to affect the caller's thread after the call, and
3. the caller should not be able to affect the callee's execution during the call (for instance, the callee would not want to be suspended by the caller while it holds a lock on sensitive data structures).

These requirements are most easily satisfied by using two threads for a cross-domain call. A call suspends the caller’s thread and creates a new thread for the callee. The corresponding return terminates the callee’s thread, and resumes the caller. This ensures that the callee can only affect itself, not the caller. The caller can back out of the call by unblocking its thread. If the callee eventually returns, any returned data is discarded. SLK provides this fully protected cross-domain call through a threaded wrapper, which is based on the `turnstile`. The implementation uses stand-by threads to avoid thread creation, which has a high overhead on current Java implementations (measurements are shown in the next section).

[10]The implementation shown here uses some of the more subtle features of the Java scoping rules. A turnstile is created by extending the abstract Turnstile class and overriding `method()` to implement the desired function. The `protected` qualifier for `method()` restricts its visibility to sub-classes of Turnstile, thereby ensuring that it cannot be called directly. The `final` qualifier on `cross()` prevents it from being overridden in subclasses where it could steal the private keys of its callers.

For situations where full protection is not required, SLK provides single-thread cross-domain calls, where the callee executes in the same thread as the caller. The Java thread interface makes protection difficult in this case. Both the caller and callee can gain access to the `Thread` object controlling the thread, and each domain can use this object to affect the other's execution, violating both requirements 2 and 3 listed above. To deal with this problem, SLK provides a new type of thread, called `SlkThread`, which requires a special thread control object to modify the state of the thread. This allows the caller to protect itself from the callee by simply not handing out the thread control object. A thread can re-bind to a new thread control object to allow the callee to protect itself by binding the thread to a control object it owns. SLK provides wrappers implementing both the callee-protected and caller-protected single-thread cross-domain calls.

4.6 Automatic wrapper generation

SLK includes a wrapper generator to eliminate the manual writing task and to provide certified wrappers. The input to the wrapper generator consists of a class file, a list of methods that the wrapper should expose, and the kind of wrappers to be generated (ordinary, threaded). The wrapper generator examines the class file and generates a new class with the wrapper methods. The wrapper generator then loads the wrapper class and places the resulting class descriptor together with a wrapper descriptor into a `signature` object. The signature allows all recipients of the wrapper to verify its origin and properties (for example, that it indeed implements full cross-domain call protection).

5 Microbenchmarks

This section evaluates the basic costs of the protection mechanisms in SLK using three off-the-shelf Java virtual machines, all of which use just-in-time (JIT) compilers: Symantec Café 1.5 and Microsoft Visual J++ 1.0 running on a 200Mhz Pentium Pro with Windows/NT 4.0, and Sun JIT compiler 1.0.2 running on a 166Mhz UltraSparc-2 with Solaris 2.5. In addition, Toba [27], a Java to C compiler and runtime environment developed at the University of Arizona, is used on both platforms. Each mechanism is measured in a tight loop and thus the results are optimistic with respect to cache behavior.

Table 1 shows the cost of creating and using keys and safeboxes. For comparison purposes the first two rows indicate the cost of calling the standard C `malloc()` and of allocating an empty Java object with a null constructor. The cost of creating wrappers, capabilities, safeboxes, and signatures can be attributed mainly to memory allocation for an object and to executing a constructor. The most expensive operation is constructing a `KeyPair`, which involves creating three objects: two `Keys` and one `KeyPair` (see Figure 7).

	P6			Ultra-2	
Operation	**Café**	**VJ++**	**Toba**	**Sun**	**Toba**
C malloc()		0.58		2.87	
Java null object allocation	1.80	2.60	0.76	2.08	2.38
SLK key pair creation	6.00	10.90	2.78	8.10	8.67
SLK key match	0.11	0.06	0.05	0.26	0.17
SLK capability/safebox creation	2.00	3.00	0.99	2.48	3.04
SLK safebox unlock	0.26	0.14	0.16	0.47	0.31

Table 1: The cost of operations on keys, in µs.

The second group of benchmarks, shown in Table 2, measures the cost of invoking a null method through wrappers, capabilities, and turnstiles. The baseline for performance comparisons is set by a null C function call and a null Java method invocation (NMI) on an object. NMIs through wrapper, capability, and turnstile classes cost about 2-2.5 times more than a plain Java NMI, which is expected given the level of indirection involved. The cost of a NMI through interfaces is much higher, though, because interfaces introduce multiple inheritance which prevents a direct dispatch from being performed by the Java implementations used.

Table 2: The cost of method invocations, in µs.

Operation	P6 Café	P6 VJ++	P6 Toba	Ultra-2 Sun	Ultra-2 Toba
C null function call		0.03		0.05	
Java null method invocation (NMI)	0.06	0.04	0.04	0.15	0.15
SLK NMI via a wrapper	0.17	0.10	0.08	0.40	0.28
SLK NMI via a wrapper (interface)	0.24	0.57	0.18	1.04	0.48
SLK NMI via a capability	0.17	0.12	0.08	0.45	0.27
SLK NMI via a capability (interface)	0.25	0.58	0.19	1.07	0.53
SLK NMI via a turnstile	0.19	0.21	0.10	0.43	0.32

The last set of tests, shown in Table 3, measures the cost of an NMI via threaded wrappers, and via single-thread protected invocations. The main cost in the case of a caller-callee protected call is that of creating a control object. The cost of an NMI via thread switching includes, in addition to building a control object, two context switches (which are at least as costly as a double context switch with native NT or Solaris threads) and several lock acquisitions and releases. Both of these inter-domain method invocation mechanisms are cheaper than inter-process communication (LRPC under NT, RPC under Solaris).

Operation	P6 Café	P6 VJ++	Ultra-2 Sun
double kernel thread switch	12.00		21.00
null native OS LRPC	89.00		274.00
SLK caller-callee protected NMI	11.00	8.10	15.40
SLK NMI via thread switching	80.00	56.00	86.60

Table 3: Method invocations with thread protection, in µs.

Several observations can be made about the microbenchmark results. Java just-in-time compilers do not use all optimizations performed in C compilers, primarily because compilation time is favored over compiled code quality. In many cases, longer compilation is acceptable and may result in much better code, as demonstrated by comparing Toba performance with Café and VJ++. The experience with Toba suggests that there is still room for improvement in Java compilation technology. In the case of interface method calls, in particular, a 2x improvement appears possible. Another observation is that SLK should use a virtual machine that has user-level threads, since relying on kernel threads is a big performance disadvantage for cross-domain calls. Finally, wrappers and capabilities are attractive as protected inter-domain control transfer mechanisms, compared to the cost of RPCs in traditional operating systems.

6 Applications

The rising importance of the client-server model of computing and the "commoditization" of processing cycles is breaking the traditional server model in which only a limited set of carefully installed services is available to the user. The success of web-space sold by Internet service providers has shown that there is considerable demand for service organizations providing reliable and well-networked server platforms. Current technology restricts this form of web hosting to static web pages because there is no security model that would allow users to upload code that handles requests dynamically. The trend towards dynamic web hosting is clear in recent web servers. Most include an API that allows extensions to be loaded into the server process. The best known are probably Jeeves' servlet API [34] and IIS' ISAPI [20]. However, all these APIs offer only the most rudimentary form of protection, largely because their primary goal is to provide an efficient alternative to CGI scripts.

This section describes a number of prototype applications that demonstrate how SLK can be used to develop server environments that permit end users to upload code to customize the services offered. While the first two examples describe web-related applications, the SLK model can be used in other servers as well. This is similar to Jeeves which is often perceived as a web server, yet really offers a general environment for writing Java based servers.

6.1 An HTTP server for dynamic web hosting

The World Wide Web Consortium recently released a Java-based web server called Jigsaw [39]. Jigsaw is designed to be extensible and its core provides a rich set of features to facilitate the writing of new extensions. Jigsaw represents the URL namespace it serves as a tree of internal objects called *resources*. These resource objects form the core of Jigsaw and provide methods for traversing the namespace and servicing HTTP requests. Resources fall into three broad categories: containers, filters, and leaves. Container resources correspond to directories in the URL namespace and provide, among others, a *lookup* method for walking down the namespace tree. Filter resources can be attached to other resources in the URL tree to filter requests and responses. For example, filter resources can be hooked onto the top node of a sub-tree to authenticate requests, to cache responses, or to encrypt or compress responses. Finally, leaf resources correspond to documents and compose the HTTP response.

Jigsaw can be extended by writing new resource classes that conform to the existing interfaces but provide new functionality. For example, a new set of container and leaf resource classes can map a database into a sub-tree of the URL namespace. However, Jigsaw does not provide any security or fault isolation when new resources are loaded.

Using SLK, a new version of Jigsaw (Jigsaw/SLK) was developed to support secure extensions of two forms. New resources and filters can be inserted into the URL namespace in order to customize individual nodes or entire sub-trees, and new threads can be started to perform background computations or to provide asynchronous services. The API used to load untrusted code is an extension of Jeeves' servlet API. Each servlet is loaded as a separate SLK program with its own class loader and running in its own thread group. The servlet is authenticated, assigned a pair of public/private keys, and hooked into the URL namespace using a Jigsaw resource, and runs in its own protection domain with its own thread group. Jigsaw/SLK performs safe downcalls into the servlets by wrapping them with automatically generated threaded wrappers. Likewise, servlets are protected from Jigsaw/SLK by performing upcalls through a wrapped context object that implements the servlet API upcalls.

Jigsaw/SLK provides an object repository for servlets to share objects with other servlets as well as the core. These shared objects can be protected using SLK's safeboxes, capabilities, etc. For additional flexibility, Jigsaw/SLK shares a set of interfaces with servlets that allows them to augment parts of the resource tree. In this manner, servlets can add filters to resources, can add new resources, or existing resources to the tree.

	P6	**Ultra-2**
Cross-Domain Call	**Café**	**Sun**
Unprotected	754	1250
Caller-Protected	754	1250
Caller-Callee-Protected	764	1265
Thread-Switching	867	1390
Null C CGI Program	25600	11900
Null Perl CGI Script	56000	12400

Table **4** shows the time taken by Jigsaw/SLK to process a null 2-level URL HTTP GET request to a servlet running on two of the platforms described in Section 5. When the request arrives, Jigsaw/SLK traverses the resource tree and hands the request to the servlet. Measurements were taken for different levels of thread protection during this hand-off: a direct call with no protection at all (*Unprotected*), a single-threaded call with Jigsaw's thread protected (*Caller-Protected*), another in which neither Jigsaw nor the servlet has control over the thread (*Caller-Callee-Protected*), and a call with thread switching (*Thread-Switching*) to fully protect both Jigsaw and the servlet. These times are compared to the processing time of the same HTTP GET request to null C and Perl CGI scripts. Results show that the difference between the *Thread-Switching* and *Unprotected* versions is roughly the overhead of two context switches and synchronization (`wait`/`notify`). While the overheads in both *Caller-Protected* and *Caller-Callee-Protected* are small compared to *Thread-Switching*, the costs of all protected cross-domain calls are between one and two orders of magnitude faster than CGI scripts.

Cross-Domain Call	P6 Café	Ultra-2 Sun
Unprotected	754	1250
Caller-Protected	754	1250
Caller-Callee-Protected	764	1265
Thread-Switching	867	1390
Null C CGI Program	25600	11900
Null Perl CGI Script	56000	12400

Table 4. Request processing time with varying degrees of protection in Jigsaw/SLK, in µs.

Low-overhead cross-domain calls are not the only motivation for extending Jigsaw/SLK by loading servlets into the same address space. Jigsaw contains a number of sophisticated resource management mechanisms that the servlets can benefit from seamlessly. For example, each resource in Jigsaw is designed such that it can be saved to secondary storage if memory usage becomes too high and then demand-loaded when needed. The resources installed by servlets automatically inherit this mechanism. If the servlets were external to Jigsaw, they would have to re-implement this feature and thereby create a distributed resource management problem.

6.2 The CoNote Annotation System

CoNote [8] is a cooperative work system providing shared annotations on a set of web documents. It is used in undergraduate courses at Cornell where it replaces newsgroups and mailing lists. The instructor places course documents on the web and inserts annotation points (using a special HTML tag) in appropriate places. Students can add annotations that appear similar to a threaded newsgroup right at the annotation point.

The implementation of CoNote presented here is interesting in that it uses sharing among servlets to build an extension to Jigsaw/SLK that is itself extensible. The previous CoNote implementation used a proxy server that fetches the underlying documents from a regular HTTP server and adds the annotations. By re-implementing CoNote within Jigsaw/SLK using servlets, the performance is improved and the implementation is simplified by leveraging Jigsaw's built-in mechanisms.

CoNote/SLK consists of a CoNote *maintainer* servlet, which manages a sub-tree of Jigsaw's URL namespace. The maintainer servlet adds an *authorization* and an *annotation* filter to the root of every document tree in CoNote's URL sub-tree. The authorization filter maintains a student database to ensure that only registered students can see the annotated documents. The annotation filter acts on responses, removing the annotation tags in documents and inserting information about the annotations that have been made at that point. Each course instructor can provide a CoNote *client* servlet that adds additional filters to transform the annotation information into HTML appropriate for the style of the document being annotated (a *formatting* filter), to collect access statistics, or to restrict access of certain annotated documents to the course staff.

The CoNote design stresses the importance of low-overhead cross-domain calls provided by SLK. Each CoNote filter introduces two cross-domain calls in the critical path of request processing: one during the lookup of the annotated document and another during the processing of the HTTP response. For example, the processing of an HTTP GET request issued to `hw.html` in

Figure **10** would perform a total of six cross-domain calls.

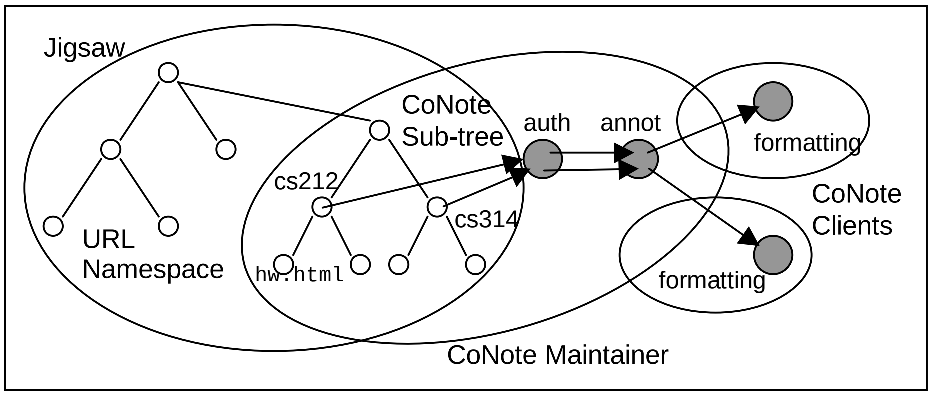


Figure 10: The CoNote System: the maintainer servlet shares a URL sub-tree with Jigsaw and installs the authentication and annotation filters on the root of the sub-tree. Client servlets interface with the maintainer servlet to install additional filters.

6.3 Routers for Active Networks

Traditional networks expose a fixed set of services and interfaces, which tend to be complex because of the large range of application demands. A recent initiative proposes Active Networks (AN) [36] as an extensible environment in which applications can load code into routers or gateways in order to obtain customized services. Application specific proxies and firewalls can similarly offer new services if applications can upload custom functionality. SLK is a good match for these types of applications. Simple wrappers are useful for exposing selected parts of communication data structures so that different users can have different access rights to them, while more specialized wrappers make billing and bookkeeping transparent. Revoking, restricting or changing rights to network resources are also accomplished using wrappers. Threaded wrappers allow keeping track of a router's CPU usage by user. Note that SLK is used differently here than in applications described above – instead of threads in the core calling extensions, the threads in the extensions operate on protected data structures provided by the core.

7 Related Work

This section first discusses similarities and differences between SLK and closely related work, and then provides a broader perspective on related operating systems techniques for extensibility.

7.1 Safe language based systems

The SPIN operating system [3] uses a safe subset of Modula-3 [26] to provide an infrastructure for running user-level code in the kernel. Although the motivation for SPIN, which is to support extensible operating system kernels, is quite different from that of SLK, the two systems use a similar set of mechanisms such as relying on unforgeable pointers and providing link-time access control. The important differences are: (i) while SPIN rejects an object-based model because it would create a single name space due to its linkage model [31], SLK shows how multiple name spaces can be constructed within such a model; (ii) SLK integrates linking into the programming model, while SPIN uses a separate meta-language for it; and (iii) SLK explores the differences between object references and capabilities and shows how true capabilities can be implemented in a safe language.

Systems that use Java as the primary language for writing operating systems and extensible environments are just emerging. Sun's JavaOS [19] focuses on small-footprint embedded single-user systems, where protection is a secondary concern. The concepts advocated by SLK could probably be adapted to JavaOS easily. Sun's Jeeves [34] shares many goals with SLK in providing a framework for building extensible network servers. Jeeves defines an API for uploading servlets dynamically. However, the protection mechanisms in Jeeves are very limited and focus on complete isolation of the extension code, not on enabling sharing. As described in Section 6, SLK implements Jeeves' servlet API but provides more flexible protection and sharing across servlets.

7.2 Object-based systems

Traditionally, protection mechanisms in object-based systems [17] have centered on the capability concept. Capability-based systems, especially Hydra, provided much inspiration for SLK. Capability implementations have tended to rely on hardware support, while SLK's protection mechanism is exclusively software-based. In addition, access rights encoded in capabilities are enforced at run-time, whereas part of the SLK protection can be enforced statically at compile/link time. Despite these differences, SLK provides equally flexible and powerful protection as capability-based systems do. To briefly illustrate this, the following paragraph describes how two interesting examples presented in the Hydra paper [15] can be mapped into SLK.

The first example from Hydra revolves around an untrusted courier that stores and delivers a classified document from one protection domain to another without being able to examine its contents. Objects in SLK can be handed to an untrusted courier using several mechanisms. Control over the namespace can be used to prevent the courier from obtaining the type of the object, in which case it can only access it as an instance of `Object`. If this is impractical, the object can be wrapped into a `safebox` for which only the recipient holds the key. The second example uses rights amplification to sort the contents of an object for which the caller has only read access. There are several ways to implement rights amplification in SLK as well. It can be achieved statically if *sort* is defined as a method in the class of the object or in another class in the same package. For a dynamic solution, a `Capability` can have an amplify method which takes another `Capability` or a `Key` as argument and returns a new `Capability` with more rights. Another solution is to provide an uninitialized wrapper to the sorter (i.e. a wrapper with a null object pointer) which can only be initialized given another wrapper.

Legion and Opal are related to SLK in that they use object-based models. Legion's [40] focus on a worldwide distributed system and on multi-language interoperability is quite different from the goals of SLK and, consequently, the two systems are very different from each other. The one common concept, however, is that objects can specify their own protection policies using run-time checks on a per-method invocation basis. Opal [5] supports object-based sharing of data and services; however, it emphasizes support for persistent objects. Sharing and access control are at the granularity of a VM page. Opal protects certain objects with *protected pointers,* which are cryptographically protected capabilities (the design of which was borrowed from Amoeba system [35]).

7.3 Other operating system extension techniques

In general, operating system extensibility can be realized either by moving modules out of the kernel (system call interception, microkernel personalities) or going in the opposite direction and providing means of extending the kernel (software fault isolation, proof carrying code, safe languages).

With system call interception [12], an extension module can be loaded into a process in a standard OS and a set of security wrappers intercepts all system calls to prevent compromising the server's integrity. The advantage of this approach is that no modifications to either the OS or the application are required. The security model is relatively simple and orthogonal to the rest of the system. However, it does not enable fine-grained sharing between untrusted modules. Communication among extensions and the OS is limited to the traditional system call and RPC interfaces and shared virtual memory pages, which incur high overhead and force coarse granularity of sharing.

The microkernel approach to extensibility leaves most of the OS functionality to user-level servers and libraries. Exokernel [10] takes this approach to an extreme by pushing almost all OS abstractions out of the kernel and only efficiently and securely multiplexes hardware resources among applications. This enables most OS functionality and policies to be easily extended or replaced [16] by writing application-specific user-level OS libraries. In addition, small pieces of untrusted code (ASH – application specific handler [38]) can be downloaded into the kernel to improve application performance. Although SLK shares similar goal as Exokernel in providing system extensibility, the emphasis of SLK is to allow not only extensions to be installed safely, but also the safe and fine-grained sharing among extensions. This is not directly addressed by Exokernel, which does not specify how ASHs or user-level specialization of OS libraries can be shared by multiple applications.

Software fault isolation (SFI) [37] is used in a number of extensible systems [29,10]. SFI sandboxed extension code runs in the same address space as the kernel after binary rewriting is performed to enforce protection boundaries. Sharing of exported procedures among untrusted modules is done through a safe control transfer mechanism. The protocol for exporting and naming procedures among fault domains is left unspecified. Sharing of data is achieved by mapping the shared region into multiple fault domains. In general, SFI focuses on protecting large segments of the address space and is not well suited for sharing small objects.

The main idea of proof carrying code [25] is that an extension carries with it a proof that the code obeys safety policies defined by the kernel. The kernel can then check the validity of the proof and load the extension. Once loaded, the extension code module requires no run-time checks. The current state of the art has a number of problems: it is difficult to create proofs for an arbitrary piece of code and the lower bound on the proof size is very high. If these issues are solved, PCC could be incorporated into SLK to enable multiple language interoperability.

7.4 Cross domain communication

Capability-based object systems and monolithic kernels have used different models for cross protection domain control transfer and communication. Capability-based systems use protected procedure calls [17], which use a single thread of control across domains. This implies that the thread of control itself is not protected during the call. RPC is used in most other systems; it requires a thread switch and provides protection by separating the caller's and callee's threads. A large body of work has improved the performance of RPC, in particular local RPC [4]. [6,14,11] further explored a model of migrating threads where a single thread of control is used in an RPC. However, the emphasis of this work is on improving performance at the expense of fault isolation.

The transaction model proposed by VINO [29] offers another solution for protected control transfer between domains. Each extension code invocation is wrapped in a transaction, which can be aborted such that the system can be restored to a consistent state. However, experiments to date report consistently high overhead for the system, which limits its applicability.

8 Summary

This paper presents a novel type-capability model that relates the protection offered by safe languages to that of traditional capability systems. The model is based on unforgeable object references and introduces the notion of type capabilities, which link object references with the corresponding access rights via the type system. Type capabilities are fixed at link time on a per-module basis with the effect that, within a module, all object references of the same type have the same access rights and that these cannot be changed at run-time. Full capabilities with modifiable per-object access rights can be layered on top of the type-capability model at the expense of run-time checks. This provides a tradeoff between flexibility and run-time cost.

The Safe Language Kernel builds on the type capability model to provide mechanisms for multiple programs to share data and code. The prototype implementation discussed in this paper is based on standard Java virtual machines. It takes the form of a library that offers tools to manipulate type-capabilities that are checked at link-time as well general capabilities that require run-time checks. A set of SLK micro-benchmarks demonstrates the low overhead of crossing protection boundaries. The primary motivation for the implementation is to provide a substrate for developing server systems into which untrusted component software can be loaded. Internet-related application studies, in particular a dynamic web server, demonstrate that SLK offers a good substrate for flexible and efficient extensibility.

9 References

1. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. *SuperWeb: Towards a Global Web-Based Parallel Computing Infrastructure*. In Proceedings of the 11th IEEE International Parallel Processing Symposium (IPPS), Geneva, April 1997.

2. B. N. Bershad, S. Savage, P. Pardyak, D. Becker, M. Fiuczynski, and E. G. Sirer. *Protection is a Software Issue*. In Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS-V), pages 62-65, Orcas Island, WA.

3. B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. *Extensibility, Safety and Performance in the SPIN Operating System*. In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), Copper Mountain, CO, December 1995.

4. B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. *Lightweight Remote Procedure Call*. In Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP), pages 102-113, Arizona, December, 1989.

5. J. S. Chase, H. M. Levy, E. D. Lazowska, and M. Baker-Harvey. *Lightweight Shared Objects in a 64-Bit Operating System*. In Proceedings of the ACM Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), October 1992.

6. R. K. Clark, E. D. Jensen, and F. D. Reynolds. *An Architectural Overview of the Alpha Real-time Distributed Kernel*. USENIX Workshop on Micro-kernels and Other Kernel Architectures, Seattle, WA, April 1992.

7. E. Cooper, R. Harper and P. Lee. *The Fox project: Advanced Development of Systems Software*. Technical Report CMU-CS-91-178, Carnegie Mellon University, August 1991.

8. J. Davis and D. Hutttenlocher. *Shared Annotation for Cooperative Learning*. In Proceedings of Computer Support for Cooperative Learning Conference, 1995.

9. J. Dennis and E. Van Horn. *Programming Semantics for Multiprogramming Systems.* Communications of the ACM 9(3), December 1966.

10. D. R. Engler, M. F. Kaashoek, and J. James O'Toole. *Exokernel: An Operating System Architecture for Application-Level Resource Management*. In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), Copper Mountain, CO, December 1995.

11. B. Ford and J. Lepreau. *Evolving Mach 3.0 to a Migrating Thread Model*. In 1994 Winter USENIX Conference, January 1994.

12. I. Goldberg, D. Wagner, R. Thomas, E. A. Brewer. A Secure Environment for Untrusted Helper Applications – Confining the Wily Hacker. In Proceedings of the 1996 USENIX Security Symposium.

13. J. Gosling, B. Joy, G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

14. G. Hamilton and P. Kougiouris. *The Spring Nucleus: A Microkernel for Objects*. In Proceedings of the Summer 1993 USENIX, June 1993.

15. A. K. Jones and W. A. Wulf. *Towards the Design of Secure Systems*. Software Practice and Experience, Vol. 5, No. 4.

16. M. F. Kaashoek, D. R. Engler, G. R. Ganger, and D. A. Wallach. *Server Operating Systems*. SIGOPS European Workshop, September , 1996.

17. H. M. Levy. *Capability-Based Computer Systems*. Digital Press, Bedford, Massachusetts, 1984.

18. J. Liedtke. *On μ-kernel Construction.* In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), Copper Mountain, CO, December 1995.

19. P. Madany, et. al. *JavaOS™ : A Standalone Java™ Environment*. White Paper, Sun Microsystems.

20. Microsoft Corp. *Internet Service Support*. http://www.microsoft.com/infoserversup.

21. J. G. Mitchell, et. al. *An Overview of the Spring System.* COMPCON, Spring 1994, February 1994.

22. P. Mosberger and L. Peterson. *Making Paths Explicit in the Scout Operating System*. In Proceedings of the 2nd Operating Systems Design and Implementation (OSDI), Seattle, WA, October, 1996.

23. H. Mossenbock. *Extensibility in the Oberon System.* Nordic Journal of Computing 1(1), February 1994.

24. J. H. Morris Jr. *Protection in Programming Languages.* Communications of the ACM 16(1), January 1973.

25. G. Necula and P. Lee. *Safe Kernel Extensions Without Run-Time Checking.* In Proceedings of the 2nd Operating Systems Design and Implementation (OSDI), Seattle, WA, October 1996.

26. G. Nelson, ed. *System Programming in Modula-3.* Prentice Hall, 1991.

27. T. Proebsting, G. Townsend, P. Bridges, J. Hartman, T. Newsham, S. Watterson. *Toba: Java for Applications: A Way Ahead of Time Compiler.* Technical Report 97-01, Department of Computer Science, University of Arizona.

28. R. L. Rivest, A. Shamir, L. Adelman. *A Method for Obtaining Digital Signatures and Public Key Cryptosystems.* Communications of the ACM 22:4, December 1978.

29. M. Seltzer, Y. Endo, C. Small, and K. Smith. *Dealing with Disaster: Surviving Misbehaved Kernel Extensions*. In Proceedings of the 2nd Operating Systems Design and Implementation (OSDI) , Seattle, WA, October, 1996.

30. R. Sharma, S. Keshav, M. Wu, L. Wu, *Environments for Active Networks.* Submitted to NOSSDAV'97.

31. E. G. Sirer, M. Fiuczynski, P. Pardyak, and B. Bershad. *Safe Dynamic Linking in an Extensible Operating System.* 1st Workshop on Compiler Support for Systems Software, February 1996.

32. E. G. Sirer, S. Savage, P. Pardyak, G. DeFouw, M. A. Alapat, and B. Bershad. *Writing an Operating System Using Modula-3*. 1st Workshop on Compiler Support for System Software, February 1996.

33. C. Small, and M. Seltzer. *A Comparison of OS Extension Technologies*. USENIX Conference, San Diego, CA, January 1996.

34. Sun Microsystems. *Java Servlet Application Programming Interface White Paper*. Draft version 2. http://www.javasoft.com

35. A. S. Tanenbaum, M. F. Kaashoek, R. van Renesse and H. Bal. *The Amoeba Distributed Operating System--A Status Report.* Computer Communications, July/August 1991.

36. D. L. Tennenhouse, D. J. Wetherall. *Towards an Active Network Architecture.* Telemedia, Networks and Systems Group, MIT.

37. R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. *Efficient Software-Based Fault Isolation*. In Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP), Asheville, NC, December 1993.

38. D. A. Wallach, D. T. Engler, and M. F. Kaashoek. *ASHs: Application-specific handlers for high-performance messaging.* In Proceedings of ACM SIGCOMM Conference, 1996.

39. World Wide Web Consortium, *Jigsaw 1.0 Alpha 3*, http://www.w3.org/pub/WWW.

40. W. A. Wulf, C. Wang, D. Kienzle. *A New Model of Security for Distributed Systems*. University of Virginia, Department of Computer Science, Technical Report CS-95-34, August 1995.