A Case for Language-Based Protection

Chris Hawblitzel and Thorsten von Eicken

Technical Report 98-1670
Department of Computer Science
Cornell University
Ithaca, NY

Abstract

The use of language mechanisms to enforce protection boundaries around software modules has become increasingly attractive. This paper examines the advantages and disadvantages of language-based protection over more traditional protection mechanisms, such as standard virtual memory protection hardware, software fault isolation, and capability systems. Arguably, state-of-the-art language-based protection is more flexible and as safe as these other mechanisms. Two major remaining issues are the performance of language-based protection, and the management of resources. Regarding the latter, techniques to build an operating system kernel capable of managing resources and revoking rights are presented.

1 Introduction

The use of language mechanisms to enforce protection boundaries around software modules has increased over the past few years. The most notable example is Java [GJS96, GS98, WBD+97] where checks applied at the bytecode level enforce protection boundaries, which, for example, prevent access to arbitrary objects in memory. The motivation for using language-based protection in Java is portability (the famous "write once, run anywhere"). Extensible operating systems use language-based protection as well. In particular, the SPIN OS [BSP+95] uses a trusted Modula-3 compiler to protect the core of the operating system from extensions that are loaded into the OS by users. In this case, the motivation for using language-based protection is performance: calls across protection boundaries (i.e., in and out of extensions) are just regular function calls.

Recent work on micro-kernels, most prominently L4 [Här97, Lie95] and the Exokernel [EKO95], counter-argues that protection boundaries enforced by standard hardware mechanisms can be crossed with very low overheads. In essence, a well-implemented micro-kernel can expose the hardware mechanisms in a very flexible way and can provide very low-overhead inter-process communication. In addition, the hardware protection mechanisms are argued to be more secure than software-ones, to work with applications written in any language, and to not slow-down the execution of "normal code" through interspersed security checks or constraints on code generation [Che94, JLI98].

A major difficulty in evaluating these arguments stems from the fact that many levels of abstractions and mechanisms are involved: the protection mechanisms provided by the hardware, the abstractions provided by the operating system, and the features offered by the high-level language. The comparisons often hinge on the coupling between the layers of abstraction. For example, while it is possible to build a capability system with fine-grain protection on top of conventional page-level memory protection hardware, it has proven to be inefficient [JW75, WLH81]. In the end, what matters is the power of the protection abstractions available to the application programmer and the efficiency with which these can be mapped to the hardware at hand.

The argument made in this paper is that with language-based protection primitives, more powerful protection abstractions can be made available to the programmer and mapped efficiently to conventional hardware. The argument is made by dissecting the advantages and disadvantages of language-based protection to arrive at a coherent overall picture. The discussion takes the form of a case for language-based protection: we discuss each of the stereotypical advantages of language based protection in turn and then examine the disadvantages in turn. Following this discussion, we focus on the role of an operating system kernel when using language-based protection mechanisms. In particular, we argue that current such systems lack an OS kernel layer and we describe how a micro-kernel can be built on top of the language-based protection mechanisms to provide higher-level abstractions equivalent to processes, address spaces, threads, and inter-process communication. We argue that such a layer provides functionality that is necessary to build robust systems.

Throughout the paper, the discussion focuses on enforcing access control policies on executing programs. Other security policies, such as information flow and availability are only briefly touched. We also assume that higher-level security policies, e.g., governing access to files, are built on top of the primitives we discuss.

2 Background

Protection between executing programs, henceforth referred to as *tasks*, can be implemented using a variety of techniques, ranging from run-time checks in hardware and/or software to compiler analysis. This section describes the protection mechanisms discussed in this paper and organizes then into a framework.

2.1 Address-based protection mechanisms

Current operating systems rely on protection mechanisms provided in hardware, specifically, memory protection, user/supervisor execution levels, and traps to switch levels. Memory protection is based on checking the validity of addresses issued by every instruction accessing memory. The user/supervisor privilege levels provide one execution mode where the access privileges are enforced and one where they can be changed. The traps (or call gates) provide a controlled transfer from user to supervisor level.

The key aspect of address-based protection is that it is based on the values of addresses issued by a program—these are checked and possibly translated—and not on how the program uses the data stored at those addresses.

In current microprocessors, the address-based protection is implemented as part of the virtual memory hardware. In CISC microprocessors, the set of accessible addresses (i.e., the address space) is represented by the page tables, while in RISC microprocessors with a software-managed TLB it is represented by the current set of valid TLB entries. The common characteristic of all these implementations is that the granularity of the protection mechanisms is a virtual memory page frame on the order of a few kilobytes, 4KB being the most typical. While it is possible to use smaller page frames in some implementations, this is generally inefficient.

Address-based protection can also be implemented in software using a technique called software fault isolation (SFI) [WLA+93]. In SFI, an executable is analyzed and modified to enforce that all memory references lie within the address space. Part of the enforcement is by analysis and part is by the insertion of explicit checks. The explicit checks compare the value of an address in a register with bounds held in other registers that are protected from program modification. A privileged execution level is implemented by instruction sequences that are inserted without being subject to SFI. Using SFI, the granularity of address space is generally larger than with hardware-based mechanisms (e.g., an SFI address space is typically larger than a hardware page) because the cost of enforcement increases with the number of regions.

2.2 Language-based protection mechanisms

Language-based protection rests on the safety of a language's type system. The language provides a set of types (integers, functions, and records, for instance), and operations that can manipulate instances of different types. Some operations make sense for some types but not others. For instance, a Java program can invoke a method of an object, but it cannot perform a method invocation on an integer. Type safety means that a program can only perform operations on instances of a type that the language deems sensible for that type.

In contrast to address-based protection, the address at which a piece of data resides is irrelevant in language-based protection. Only the type of the data and the operation to be performed are relevant.

Type safety alone is not sufficient for language-based protection. For instance, it is always type-safe to read characters from a Java String object, but this does not mean that one task should be able to read characters from another task's strings. On top of type-safety, the language must also provide some form of access control for the objects manipulated by the tasks [Mor73]. Most safe languages provide two simple forms of access control:

- Dynamic access control mechanisms to determine to which objects an executing task has access. Usually a task only has access to objects that it created itself, and to objects that are explicitly passed to it by another task. In a language like Java where objects are referred to by pointers, dynamic access control is summarized by saying that “pointers to objects cannot be forged”.
- Static access control mechanisms determine what operations a piece of code can perform on objects of a particular type, once it has been granted access to them through the dynamic access control mechanisms. For example, a method defined inside a Java class has access to the private fields of objects of that class, while methods defined outside the class cannot access these private fields.

One can design safe languages that have more sophisticated access control features than the features above. For instance, Jones and Liskov [JL78] extended a language’s type system to talk about access rights directly, to provide a similar functionality to advanced capability systems such as Hydra [WLH81]. Recently Myers and Liskov [ML97] have extended this idea to cover information flow control. Another possible feature would be direct support for revocation; the features above allow programs to grant access to other programs, but not to later revoke access. This is explored in more detail in Section 4.

A key issue of language based protection is ensuring that type safety and access control are enforced. Type enforcement may be done dynamically or statically. In practice, most languages use some static enforcement and some dynamic enforcement. For instance, Java can statically verify that a variable of type String will always point to a String object or contain the null pointer. On the other hand, Java must dynamically perform bounds checks when an array is accessed. Some languages, such as Scheme, perform almost all type enforcement at run-time, which tends to slow down program execution.

In Java the protection is defined at the bytecode level and enforced by the so-called bytecode verifier [LY96]. Bytecode is structured such that the verifier can trace all control flows and can verify that the bytecode obeys by the restrictions of the Java type system. In SPIN, the protection is defined at the Modula-3 level and is enforced by the compiler, which generates code that is presumed to be safe “by construction”.

In a recent development, language-based protection mechanisms are being moved to lower and lower levels of abstraction in order to reduce the complexity of the code that transforms the verified code into executable machine instructions. Typed Assembly Language (TAL) [MWC+98] consists of a regular instruction set (currently an x86 subset) augmented with a memory allocation instruction and with type annotations. The type annotations allow a type-checker to verify that code is type-safe. The two most significant contributions of TAL are that (i) it is largely source language independent and (ii) the transformations necessary after verification are minimal. In particular, TAL allows code generation optimizations before verification. The following example shows TAL code for a function that finds a value in an array of integers.

```
static int indexof(int val, int a[]) {
    for(int i=0; i<a.length; i++)
        if(a[i] == val) return i;
    return –1; }

;; indexof takes the val and a[] arguments in EAX and EBX,
;; and returns the result in ECX.

indexof: [a1]{EAX: int, EBX: int[], ESP: sptr {ECX: int, ESP: sptr a1}::a1}
       alen 4, EDX, EBX      ;; load the length of a[] into EDX
       mov ECX, 0            ;; load i with 0
       jmp loopcheck[a1]

loopbody: [a1]{EAX: int, EBX: int[], ECX: int, EDX: int,
               ESP: sptr {ECX: int, ESP: sptr a1}::a1}
      asub 4, ESI, EBX, ECX ;; ESI = a[i]
      cmp ESI, EAX          ;; is ESI == val ?
      jne next[a1]          ;; if no, go to next iteration
      ret                   ;; if yes, return i
```

```
next: [a1]{EAX: int, EBX: int[], ECX: int, EDX: int,
           ESP: sptr {ECX: int, ESP: sptr a1}::a1}
      inc ECX              ;; i++
      fallthru[a1]         ;; not an instruction, needed for type annotation

loopcheck: [a1]{EAX: int, EBX: int[], ECX: int, EDX: int,
                ESP: sptr {ECX: int, ESP: sptr a1}::a1}
      cmp ECX, EDX         ;; check i < a.length
      jl loopbody[a1]      ;; if so, execute loop body
      fallthru[a1]         ;; not an instruction, needed for type annotation

notfound: [a1]{ESP: sptr {ECX: int, ESP: sptr a1}::a1}
      mov ECX, -1          ;; not found, so return –1
      ret
```

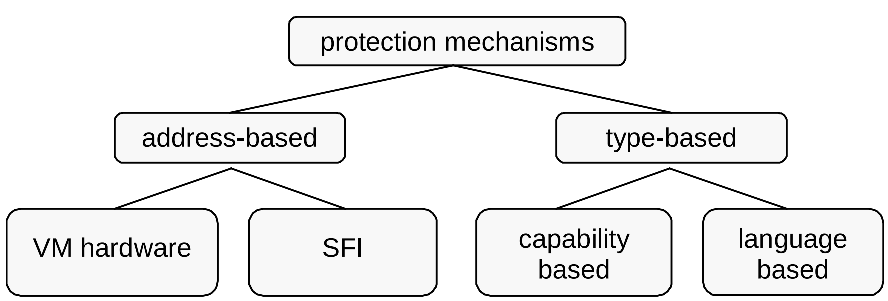
Almost all instructions in this example are real x86 instructions, and are fed directly to an x86 assembler. Exceptions are: "`alen edx,ebx`", which is a macro that expands to "`mov edx,[eax+0]`", and "`asub esi,ebx,ecx`" which expands to "`cmp ecx,[ebx+0]; jae array_bounds_error; mov esi, [ebx+4*ecx+4]`". Calling `indexof` is done by loading EAX and EBX with the correct arguments, and then using the standard x86 `call` instruction. The type annotations at each label define a pre-condition that must be satisfied when reaching the label. For example, the annotation at the entry of the function (label `indexof`) specifies that register EAX must hold an integer (`val`), register EBX an array of integers (`a[]`), and register ESP a stack pointer pointing to a return address and the "rest of the stack" denoted by the type variable `a1`. The return address is a pointer to code that expects an integer in register ECX (the return value of the function) and a stack pointer in register ESP pointing to the same "rest of the stack" (`a1`) as at entry.

Proof carrying code (PCC) [Nec97] generalizes many different approaches to software protection – arbitrary binary code can be executed as long as it comes with a proof that it is safe (or satisfies some other policy). PCC may be used to enforce address-based protection, or it can express safety through a type system. Necula et. al. described packet filters whose safety predicates were at least partly address-based [NL96]. PCC can also be used to express the correctness of software fault isolation. More recent work by Necula and Lee [NL98] describe the translation of a type-safe subset of C to proof-carrying code, where the proofs express guarantees provided by the type system.

2.3 General type-based mechanisms

Language-based protection is closely related to capability systems [EB79, Lev84, PCH+82, Red74, WLH81]. In a capability system, data values are distinguished from pointers (capabilities) and the system enforces that data values cannot be used as capabilities. Some systems add a tag bit to every word in memory to distinguish data and pointers [CKD94], others partition memory into data and capability segments, and some software implementations use cryptography techniques to distinguish capabilities from data.

Both safe languages and capability systems work by distinguishing several types of values in memory. Therefore we consider both capability systems and language-based protection systems part of a general class of *type-based* protection mechanisms. We summarize the class of protection enforcement mechanisms considered so far in the following figure:



3 The Case

This section compares the advantages and disadvantage of language-based protection over address-based protection. While one could argue that identical protection abstractions can be built over both hardware and software primitives, so that the distinction between the two is irrelevant, the nature of the primitives does have a strong impact on the power and performance of higher-level abstractions. For example, naive emulation of language primitives in hardware and of hardware primitives in a safe language both perform poorly.

3.1 (Stereo-)Typical advantages of language-based mechanisms

Language-based protection is generally thought of as having a number of advantages over address-based protection [BSP95]. Due to Java, *portability* is probably the most obvious. But the fact that the language can *precisely* specify the data items to be protected gives it another advantage over address-based protection, where the precision (sometimes called granularity) of protection is generally a memory page frame. Language-based mechanisms also allow cheap control transfers into code with selectively *amplified rights*. The fact that in language-based protection the entire program is analyzed before execution confers it more *power*: a language-based system can enforce types of security policies that a system based only on observed program behavior cannot.

3.1.1 Precision

Language-based protection mechanisms allow access rights to be specified with more precision than address-based mechanisms: the data items to which access is permitted as well as the types of accesses permitted can be specified more finely. For example, in Java, access can be granted precisely to individual objects and even to only certain object fields using the `public` qualifier.

The precision with which access rights can be specified is important in enforcing what Saltzer and Schroeder call the "principle of least privilege" [SS75]: each principal in a system should get access only to the data and operations it needs to perform its task. When sharing data structures between tasks, language-based protection mechanisms make it possible to share just the required elements of the data structure. In contrast, with address-based protection access to a shared data structure can only be controlled to the extent that elements of the data structure map onto the memory regions of the protection mechanism.

Fundamentally, it is difficult to achieve high precision of access rights with address-based mechanisms because the mechanisms do not match the elements that programming languages manipulate. While a number of hardware projects have tried to build hardware that implements the elements manipulated by the programming language directly, these efforts have mostly lead to excessively complex hardware [Org73, PCH+82].

3.1.2 Rights amplification

A major advantage of the language-based approach is that certain forms of selective rights amplification are very cheap. For example, in Java, when calling a method on an object, the code of that method gains rights to access the private fields and methods of that object as well as the static fields of the class. The called method may in addition be able to call methods of classes that the original caller does not have access to.

Expressed in the terms of capabilities, the caller holds the capability to an object and the callee holds a template such that, when called, it can amplify the rights of the capability to allow access to the private fields of the object in addition to the public ones. In general, with language-based protection, every transfer to a new "piece of code" can amplify rights, however, the nature of the amplification (the types it applies to and the rights conferred) is determined statically.

The interesting aspect of this rights amplification is that it is very specific and localized; all rights are relative to data types and/or objects. In contrast, conventional hardware has only two privilege levels, and even multiple levels do not provide the same level of precision.

3.1.3 Portability

One of the main features of Java is that it is portable. In general, this refers to the fact that applications are portable, which relies on compiling to a machine-independent bytecode as well as standard APIs. However, the Java run-time itself with its protection mechanism implemented in software is portable as well. This applies to all software based protection mechanisms: SFI, Java, TAL can all be ported to a variety of existing architectures and operating systems, while hardware-based protection mechanisms by definition cannot.

3.1.4 Power of security policies

While this paper primarily examines access rights, it is interesting to note that, as Schneider [Sch98] points out, language-based protection mechanisms allow more general security policies to be implemented than mechanisms that rely only on run-time checks of program behavior. The reason is that language-based protection is based on an analysis of the entire program and not on checking the validity of a specific program execution, e.g., using a reference monitor. This means that language-based protection can enforce security policies that conceptually require all possible executions of a program to be examined and not just a single one. Examples of policies that cannot be enforced by monitoring a single execution include many availability policies (specifying that a lock will eventually be relinquished, for instance), and information flow policies in general.

Even for policies that are enforceable by a reference monitor, it may be more practical to enforce some policies by examining a program rather than monitoring the program's execution. For example, a policy requiring a program to release a lock in less than 20 machine instructions after acquiring the lock can be enforced at run-time in principle, but in practice may be more easily enforced by program analysis.

Note that this has nothing to do with language mechanisms being type-based; rather, it is a result of programs being statically analyzable. Software fault isolation is also based on static analysis, while hardware mechanisms to support capabilities are type-based, but do not involve static analysis.

3.2 (Stereo-)Typical problems with language-based mechanisms

Language-based protection mechanisms do have a number of drawbacks. With all the publicity made around bugs in the various Java VM implementations, the lack of strong security guarantees is quick to come to mind. While this may well be a problem with Java, it is not fundamental to language-based protection. To examine this issue we attempt to compare the sizes and complexity of the *trusted computing base* (TCB) in the two cases, i.e., the parts of the system that the protection mechanisms rely on working correctly. Another typical criticism of language-based protection is that, being implemented at the language level, it restricts the programmer to a *single high-level language*. In addition, language-based protection requires *garbage collected memory* management and the *revocation* of rights is difficult. Finally, we examine whether the advantages of language-based protection are acquired at the expense of the *performance of regular code*, e.g., whether checks or other code generation constraints slow down the execution overall.

3.2.1 TCB complexity

The interesting issue regarding the "trustworthiness" of the various protection mechanisms is whether enforcement by hardware or software is better. The Java approach should probably be dismissed as rather unsafe from the outset: the operation of the verifier enforcing the protection mechanism is not formally specified and the verified code is subsequently transformed by a just-in-time compiler. Even if the bytecode is interpreted the complexity of that interpreter is substantial. SPIN's approach using a trusted Modula-3 compiler is similarly problematic and includes the entire compiler in the TCB.

Comparing SFI or TAL with a conventional hardware-based approach requires a closer investigation of the sources of faults and of the verification tools that can be applied to the system. While hardware is sometimes argued to be "inherently more reliable" than software, this is not necessarily true. In the case of address-based protection implemented in hardware, virtually the entire processor must be considered part of the TCB. This includes the operation of all regular instructions, the operation of all illegal instructions, all synchronous and asynchronous exceptions in the pipeline, and of course the address translation and checking hardware itself. Assuming that the protection checks can be isolated to a small portion of the processor is at odds with current designs: addresses are translated in multiple pipeline stages and the validity of the protection checks depends intricately on the correct detection and propagation of exceptions throughout the pipeline.

In the case of SFI, the code generated by the compiler can be verified by a separate program. In addition to this verification program, the hardware implementing regular legal instructions must be considered as part of the TCB. This is because SFI is based on a specification of each machine instruction and assumes that the hardware implements that specification correctly. The major gain is that illegal instructions do not need to be handled correctly by the hardware as SFI can guarantee that none will ever be executed. To witness the latest Pentium "Invalid CMPXCH8B Instruction" bug [Int97] in which an illegal instruction can lock up the processor this may be a significant win.

In the case of TAL, the TCB is larger than for SFI. It includes the type-checker, the garbage collector, and any TAL-to-native instruction translator being used. The properties enforced by the type-checker can all be verified formally, but that has not yet been done for the type checker code itself. Alternatives to garbage collection, such as regions [TT94] and linear objects [Bak92] exist, but their practicality is unclear.

A system that relies entirely on TAL for protection without any hardware support for address translation or protection, without any hardware privilege levels, and without any synchronous exceptions could arrive at an arguably smaller TCB than current hardware approaches.

Finally, one advantage of the software-based protection approaches is that a larger portion of the TCB, namely all the software, can be inspected by whomever places the system into operation.[1] In the hardware case, only the manufacturer of the microprocessor can verify its operation by inspection and even so, it is doubtful that a single person (or even a small team) can actually comprehend its operation in sufficient detail to make guarantees about its trustworthiness.

3.2.2 Single-language restriction

Traditionally systems using language-based protection require that all code be written in the same high-level language. The Pilot system [RDH80] and Oberon [WG89] are examples where this is the case. It is possible to compile one safe language down to another safe language – for instance, Ada95 and Pizza can be compiled to Java bytecode [OW97, Taf96], but mismatches between the language features often result in sub-optimal performance when this is done.

TAL provides a more universal platform for multiple languages to compile to, but its type system[2] may still lead to mismatches. Subtyping and modules are two language features that are difficult to efficiently compile down to TAL without making extensions to TAL's type system.

It may be possible to compile "nearly-safe" languages, such as Pascal, down to a safe language, but languages like C may be impossible to efficiently translate to a safe language without significant programmer-visible restrictions.

3.2.3 Garbage collection

Type-based systems require that types be correctly associated with values stored in memory. One danger is the possibility of aliasing: if two different types were ever associated with the same memory location, then a value of the first type could be written into the memory location, and subsequently incorrectly interpreted as a value of the second type. This issue arises when one object is deallocated and a second object is allocated into the space previously occupied by the first object. To prevent "dangling pointers" to the deallocated object, all common language-based systems use a garbage collector. Although garbage collection is often perceived as a welcome help to the programmer, most garbage collectors introduce pause times into the system. Incremental collectors have short, bounded pause times, but impose other overheads on reads or writes to memory.

Note that the dangling pointer problem occurs for both language-based protection systems and hardware-based capability systems. Capability systems deal with the issue either by using garbage collection of

[1] This is a "security by inspection" argument. If "security by obscurity" is desired, the software in the TCB can be kept hidden.

[2] TAL's type system is based on System F (the polymorphic lambda calculus) by Girard and Reynolds, with products, recursion on terms, and existential types added.

capabilities, or by generating pointers that are never repeated (by augmenting every pointer with a unique time-stamp, for instance).

3.2.4 Revocation

Revocation is easier in an address-based system, because indirection and/or checking is built into every memory access. With virtual memory hardware, a process' access to memory can easily be revoked by changing the page tables. In the case of SFI, a fault domain's access to memory can be revoked by changing its bounds values, assuming these are kept in registers and are not hard-wired into the code.

Revocation has historically been a problem for type-based systems, because pointers point directly to objects, without indirection. Revocation requires either adding a level of indirection, or somehow invalidating the direct pointers to an object. Redell's thesis [Red74] and Ekanadham et. al. [EB79] discuss many design choices available in the context of capability systems.

3.2.5 Performance of regular code

An important problem with language-based protection (and with software-based protection in general) is the performance penalty imposed on "regular" code, i.e., code that does not make use of any special sharing or protection feature. These slowdowns come from three sources: dynamic checks performed to enforce protection, restrictions on the instruction sequences available to the code generator, and additional operations necessary due to type system constraints.

In the case of SFI, dynamic checks are necessary whenever the verification cannot statically determine that a memory address is within bounds or that a jump target is a legal entry point. In Java or TAL, dynamic checks are additionally required to enforce type safety. Array bounds checks, Java's `instanceof`, and null pointer checks are examples for such dynamic checks that are not necessary when SFI or hardware-based protection are used (but that might be included anyway for software engineering purposes).

Writing type-safe code sometimes requires additional operations. An example is the reception of a packet containing an object from the network: the program obtains a pointer to an array of bytes holding the packet and needs to turn that into a handle onto an object with the same data. In C, this can be accomplished by a simple type cast while in Java, an object must be allocated and the data must be copied into it from the byte array. Hsieh et. al. describe an extension to Modula-3 to optimize this particular case [HFG+96].

Sirer et al [SFP96] report that for several small benchmarks Modula-3 performance ranges from 20% slower to 8% faster than optimized C code. Wahbe et. al. [WLA+93] discuss an efficient SFI technique called sandboxing, which slow down various SPEC92 and Splash benchmarks around 20% on average on RISC processors when both loads and stores are checked. Adl-Tabatabai et. al. [ALL+96] report slowdowns of 5-20% on RISC and x86 architectures for SFI that protects against errant stores but not loads, and Small [Sma97] reports 1-9% slowdowns when protecting against errant stores on x86, and 40-300% slowdowns when also checking loads.

3.3 Summary

The question whether language-based or address-based protection is better cannot be resolved in absolute. It is clear that the most recent developments in language-based protection, such as TAL and PCC, make it an attractive alternative to the traditional hardware-based schemes, and not some inferior choice that can be tolerated under special circumstances. Language-based protection offers more flexibility, higher precision, and lower costs to cross boundaries than address-based protection. Whether it is more or less trustworthy than the latter is debatable and depends heavily on the characteristics of the underlying hardware. It is clear that language-based protection is somewhat more restrictive on the programming languages it can support, but the least understood issue is its performance impact.

4 Building a micro-kernel using language-based protection

The protection mechanisms described in the previous section all operate with a task-centric view: they allow a task to define what it exports to other tasks and what it imports from them. To complete a system, an operating system kernel needs to manage the collection of tasks and allocate physical resources to them.

In an address-based approach, the kernel allocates processor time in the form of time-slices and it allocates physical memory in the form of page frames. It is the kernel that runs in supervisor level and ensures that the usage of the protection mechanisms by the various tasks forms a coherent system.

For example, the L4 micro-kernel provides abstractions for tasks, composed of pages and threads, and for inter-task communication. Wherever a resource can be granted to a process, it can also be revoked. For instance, if one L4 task maps one of its pages into another task, it can always revoke that mapping later. In addition, L4 tracks a process' resource usage, and allows limits to be set.

With language-based protection, the need for an OS kernel might not seem obvious, partly because there is no supervisor level. Yet, the same requirements of allocating processor time, of distributing physical memory, and of ensuring a coherent system remain. We thus argue that it makes as much sense to include a micro-kernel into a system using language-based protection as into a system using address-based protection.

As a concrete example, consider a standard Java virtual machine (JVM) inside a web browser. The JVM provides simple primitives for applets to allocate memory and processor time, but revocation of resources is ill-supported. The JVM provides hierarchical thread groups and, in principle, a parent thread group can terminate a child thread group. However, the two available methods for terminating threads are problematic. The first, *stop*, causes an exception to be thrown in all threads of the group allowing clean up code to execute. However, termination of the clean-up code is not enforced. The second method, *destroy*, does not allow for clean-up code and in consequence may disrupt unrelated code. For instance, if the thread being destroyed is in the middle of a call into a system class, that call might be aborted without being able to free a critical system lock.

Memory revocation is not supported in JVMs either. If the only roots to a task's objects are in its threads, then killing the threads allows its memory to be garbage collected. However, if tasks can share objects, many or all of a task's objects may be reachable from another task (there are back doors in the Java API that allow applets to share objects; ironically, thread groups are one such back door). In the worst case, a system class can hold pointers to a task's objects, which consequently may never be garbage collected.

These problems show that just adding facilities for thread groups and garbage collection is not sufficient for resource management—a more integrated, system-centric approach is needed to fairly multiplex resources among tasks. The rest of this section describes approaches to building a coherent micro-kernel level on top of language protection mechanisms. Alternatively, the micro-kernel facilities could be provided by a special run-time system, or integrated into the language itself.

4.1 Resource management in the J-Kernel

One sample system, which layers revocation on top of the language-based protection mechanisms, is the J-Kernel [HCC+98], a micro-kernel written in Java that runs on standard Java virtual machines and allows multiple tasks to run protected from each other. It uses the Java class loader and the safety of Java's type system to isolate the tasks from each other, not unlike browsers do with applets. However, the J-Kernel also allows the tasks to communicate safely with each other, a feature not provided to applets, and ensures that all resources allocated to tasks can be revoked.

In order to retain control over the resources used by each task, the J-Kernel only allows forms of communication and sharing between tasks that can be revoked. This precludes the direct passing of object references from one task to another. Instead, the J-Kernel allows tasks to export services to other tasks through an extra level of indirection: a task shares an object indirectly through a *capability*, which is a stub generated dynamically by the J-Kernel that points to the original object. Any method invocation on the capability is translated into an invocation on the original object[3]. To ensure that only capabilities are shared between tasks, the stub passes all non-capability arguments by copy rather than by reference. Tasks then interact with one another by exchanging capabilities and invoking methods on the capabilities.

---

[3] Capabilities can be viewed as implementing a form of LRPC [BAL89], or, more precisely, LRMI (local remote-method-invocation).

The J-Kernel keeps track of all capabilities created by a task and it can revoke any of them by setting the indirection pointers to nil. If a method of a revoked capability is invoked, the stub generates an exception that the caller can catch to initiate error recovery. Revocation of all capabilities created by a task also allows the J-Kernel to terminate a task cleanly, completely severing it from other tasks. The main problem with the approach taken in the J-Kernel is that data structures cannot be shared directly among tasks, thus not realizing one of the major potential gains of a language-based protection model.

4.2 A micro-kernel for TAL

The same approach pursued in the J-Kernel can be applied to TAL, but at a lower level and with better performance. We built a run-time library for TAL that supports user-level threads, tasks, revocable capabilities, and inter-task communication.

The run-time system is based on the notion of revocation handles, which keep lists of resources to revoke. Each time a resource (a thread, task, or capability) is created, it is added to the list of a revocation handle specified by the resource creator. When the revocation handle is "activated", all of the resources on its list are deallocated[4]. Revocation handles are organized hierarchically, so that when a parent revocation handle is activated, all of its child revocation handles are activated as well. This leads to a simple framework that ensures when a task is revoked, all of its resources are deallocated, including any threads, capabilities, or sub-tasks that it might have created.

As in the J-Kernel, tasks can share capabilities directly, but cannot share arbitrary objects. All inter-task communication is performed through invocations on capabilities. To avoid the problems caused in Java when a thread crosses multiple tasks and cannot be killed cleanly, every inter-task communication switches threads, so that threads do not cross over task boundaries. Many features are lacking in this simple run-time system. Perhaps the most important is that there is no mechanism to enforce limits on the resources that a task can use. However, all resources that are allocated can always be freed.

A capability invocation and return call with 4 integer arguments and 4 integer return values, measured in a tight loop, takes about 41 cycles on a 266 MHz Pentium II, and about 60-70 cycles on a 133 MHz Pentium, which is within an order of magnitude of the cost of a function call. This is significantly faster than the fastest micro-kernels. L4, for instance, takes around 242 cycles for a round-trip RPC by using two IPC's on a 166 MHz Pentium (passing up to 3 words of data each way in registers) [LES+95].

With this low capability invocation overhead, the most significant cost is likely to be the cost of copying arguments. This cost depends on many factors, such as the speed of memory allocation and garbage collection, language-specific costs such as calling constructors for each copied object, and whether special provisions need to be made for copying cyclic data structures. The copying costs in the J-Kernel running on off-the-shelf Java virtual machines are high [HCC+]. Because of these costs, we are investigating how data can be shared directly among tasks without sacrificing revocation and resource reclamation. The most promising approach seems to be to extend the TAL type system to talk about resource revocation directly; regions [TT94] provide a promising framework for this.

5 Conclusion

The main tenet of this paper is that state-of-the-art language-based protection is an attractive alternative to more traditional approaches. Claims that address-based approaches offer all of the advantages of language-based approaches, without being restricted to a single language or requiring trust in a compiler [JLI98] are outdated and do not consider all the issues. Language-based mechanisms offer efficient fine grain protection: boundaries can be specified precisely and crossed with low overhead. In addition, safety need not be inferior to hardware solutions; language-based protection can no longer be called a "high-risk approach" [Che94].

How an operating system kernel built on language-based protection primitives should be structured remains an open question. A return to capability systems, as proposed in the J-Kernel, seems attractive but raises a

[4] Revocation handles are similar to Ekanadham's "conditional capabilities" [EB79], except that revocation handles point to capabilities and not the other way around, and that revocation cannot be undone.

number of issues. In particular, it is not yet clear whether revocation and resource management should be implemented in the language primitives or layered on top of them in the kernel.

The performance of language-based protection remains its major problem. Little data exists in the literature to compare the performance of safe and unsafe languages, and, while TAL and PCC purportedly support fully optimizing code generation, the performance achieved using real compilers remains to be determined. Our experiments show, however, that capabilities, revocation, and cross-domain calls can be implemented efficiently.

6 References

[ALL+96] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. *Efficient and Language-Independent Mobile Programs*. Conference on Programming Language Design and Implementation, p. 127–136, Philadelphia, PA, May 1996.

[Bak92] H.G. Baker. *Lively Linear Lisp - 'Look Ma, No Garbage!'*. ACM Sigplan Notices, Volume 27, Number 8, p. 89-98, August 1992.

[BAL89] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. *Lightweight Remote Procedure Call*. 12th ACM Symposium on Operating Systems Principles, p. 102-113, Lichtfield Park, AZ, December 1989.

[BSP95] B. N. Bershad, S. Savage, and P. Pardyak. *Protection is a Software Issue*. Fifth Workshop on Hot Topics in Operating Systems, Orcas Island, WA, May 1995.

[BSP+95] 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*. 15th ACM Symposium on Operating Systems Principles, p. 267–284, Copper Mountain, CO, December 1995.

[Che94] D. R. Cheriton. *Low and High Risk Operating System Architectures*. First USENIX Symposium on Operating Systems Design and Implementation, p. 197, Monterey, CA, November 1994.

[CKD94] N. P. Carter, S. W. Keckler, and W. J. Dally. *Hardware Support for Fast Capability-based Addressing*. Sixth Int'l Conference on Architectural Support for Programming Languages and Operating Systems, p. 319–327, San Jose, CA, 1994.

[EB79] K. Ekanadham and A. J. Bernstein. *Conditional Capabilities*. IEEE Transactions on Software Engineering, p. 458–464, September 1979.

[EKO95] R. Engler, M. F. Kaashoek, and J. James O'Toole. *Exokernel: An Operating System Architecture for Application-Level Resource Management*. 15th ACM Symposium on Operating Systems Principles, p. 251–266, Copper Mountain, CO, December 1995.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java language specification*. Addison-Wesley, 1996.

[GS98] L. Gong and R. Schemers. *Implementing Protection Domains in the Java Development Kit 1.2*. Internet Society Symposium on Network and Distributed System Security, San Diego, CA, March 1998.

[Här97] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. *The Performance of μ-Kernel-Based Systems*. 16th ACM Symposium on Operating Systems Principles, p. 66-77, Saint-Malo, France, October, 1997.

[HCC+98] C. Hawblitzel, C. C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. *Implementing Multiple Protection Domains in Java*. To appear. USENIX Annual Technical Conference, New Orleans, LA, June 1998.

[HFG+96] W. C. Hsieh, M. Fiuczynski, C. Garrett, S. Savage, D. Becker, and B. N. Bershad. *Language Support for Extensible Operating Systems*. First Workshop on Compiler Support for System Software, Tucson, AZ, February 1996.

[Int97] Intel Corporation. *Pentium Processor Invalid Operand with Locked Compare and Exchange 8Byte (CMPXCHG8B) Instruction Erratum*, http://intel.com/support/processors/pentium/ppiie/, November 20 1997.

[JL78] A. K. Jones and B. H. Liskov. *A Language Extension for Expressing Constraints on Data Access*. Communications of the ACM, Volume 21, Number 5, p. 358–367, May 1978.

[JLI98] T. Jaeger, J. Liedtke, and N. Islam. *Operating System Protection for Fine-Grained Programs*. 7th USENIX Security Symposium, San Antonio, TX, January 1998.

[JW75] A. K. Jones and W. A. Wulf. *Towards the Design of Secure Systems*. Software Practice and Experience, Volume 5, Number 4, p. 321–336, 1975.

[LES+95] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, and T. Jaeger. *Achieved IPC Performance*. 6th Workshop on Hot Topics in Operating Systems, p. 28–31, Chatham, MA, May 1997.

[Lev84] H. M. Levy. *Capability-Based Computer Systems*. Digital Press, Bedford, Massachusetts, 1984.

[Lie95] J. Liedtke. *On μ-kernel Construction.* 15th ACM Symposium on Operating Systems Principles, p. 237–250, Copper Mountain, CO, December 1995.

[LY96] T. Lindholm, and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[Mor73] J. H. Morris Jr. *Protection in Programming Languages*. Communications of the ACM, Volume 16, Number 1, p. 15–21, January 1973.

[ML97] A. Myers and B. Liskov. *A Decentralized Model for Information Flow Control*. 16th ACM Symposium on Operating System Principles, p. 129–142, Saint Malo, France, October 1997.

[MWC+98] G. Morrisett, D. Walker, K. Crary, and N. Glew. *From System F to Typed Assembly Language*. 25th ACM Symposium on Principles of Programming Languages. San Diego, CA, January 1998.

[Nec97] G. Necula. *Proof-carrying code*. 24th ACM Symposium on Principles of Programming Languages, p. 106-119, Paris, January 1997.

[NL96] G. Necula and P. Lee. *Safe Kernel Extensions Without Run-Time Checking.* 2nd USENIX Symposium on Operating Systems Design and Implementation, p. 229–243, Seattle, WA, October 1996.

[NL98] G. Necula and P. Lee. *The Design and Implementation of a Certifying Compiler*. To appear. ACM Conference on Programming Language Design and Implementation. Montreal, Canada, June 1998.

[Org73] E. I. Organick, *Computer System Organization, The B5700/B6700 Series*. Academic Press, 1973.

[OW97] M. Odersky and P. Wadler. *Pizza into Java: Translating Theory into Practice*. 24th ACM Symposium on Principles of Programming Languages, p. 146–159, Paris, France, January 1997.

[PCH+82] F. J. Pollack, G. W. Cox, D. W. Hammerstrom, K. C. Kahn, K. K. Lai, and J. R. Rattner. *Supporting Ada Memory Management in the iAPX-432*. Symposium on Architectural Support for Programming Languages and Operating Systems, p. 117-131, Palo Alto, CA, 1982.

[Red74] D. D. Redell. *Naming and Protection in Extendible Operating Systems*. Technical Report 140, Project MAC, MIT 1974.

[RDH80] D. Redell, Y. Dalal, T. Horsley, H. Lauer, W. Lynch, P. McJones, H. Murray, and S. Purcell. *Pilot: An operating system for a personal computer*. Communications of the ACM, Volume 23, Number 2, p. 81–92, February 1980.

[Sch98] F. B. Schneider. *Enforceable Security Policies*. Cornell University Computer Science Technical Report TR98-1664, January 1998.

[SFP96] E. G. Sirer, M. Fiuczynski, and P. Pardyak. *Writing an Operating System with Modula-3*. First Workshop on Compiler Support for System Software, Tucson, AZ, February 1996.

[Sma97] C. Small. *MiSFIT: A Tool For Constructing Safe Extensible C++ Systems*. Third USENIX Conference on Object-Oriented Technologies, June 1997.

[SS75] J. H. Saltzer and M. Schroeder. *The Protection of Information in Computer System*. Proceedings of the IEEE, Volume 63, Number 9, p. 1278–1308, September 1975.

[Taf96] T. Taft. *Programming the Internet in Ada 95*. http://www.inmet.com/appletmagic/ajpaper/index.html. May 1996.

[TT94] M. Tofte and J.P. Talpin. *Implementation of the Typed Call-by-Value Lambda Calculus using a Stack of Regions*. 21st ACM Symposium on Principles of Programming Languages, p. 188–201, Portland, OR, January 1994.

[WBD+97] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. *Extensible Security Architectures for Java.* 16th ACM Symposium on Operating Systems Principles, p. 116–128, Saint-Malo, France, October 1997.

[WG89] N. Wirth, J. Gutknecht. *The Oberon System*. Software Practice and Experience, Volume 19, Number 9, p. 857–893, 1989.

[WLA+93] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. *Efficient Software-Based Fault Isolation*. 14th ACM Symposium on Operating Systems Principles, p. 203–216, Asheville, NC, December 1993.

[WLH81] W. A. Wulf, R. Levin, and S. P. Harbsion. *Hydra/C.mmp:An Experimental Computer System*. McGraw-Hill, 1981.