MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1564 March, 1996

A Security Kernel Based on the Lambda-Calculus

Jonathan A. Rees

This publication can be retrieved by anonymous ftp to publications.ai.mit.edu.

Abstract

Cooperation between independent agents depends upon establishing a degree of security. Each of the cooperating agents needs assurance that the cooperation will not endanger resources of value to that agent. In a computer system, a computational mechanism can assure safe cooperation among the system's users by mediating resource access according to desired security policy. Such a mechanism, which is called a *security kernel*, lies at the heart of many operating systems and programming environments.

This report describes Scheme 48, a programming environment whose design is guided by established principles of operating system security. Scheme 48's security kernel is small, consisting of the call-by-value λ-calculus with a few simple extensions to support abstract data types, object mutation, and access to hardware resources. Each agent (user or subsystem) has a separate evaluation environment that holds objects representing privileges granted to that agent. Because environments ultimately determine availability of object references, protection and sharing can be controlled largely by the way in which environments are constructed.

I will describe experience with Scheme 48 that shows how it serves as a robust and flexible experimental platform. Two successful applications of Scheme 48 are the programming environment for the Cornell mobile robots, where Scheme 48 runs with no (other) operating system support; and a secure multi-user environment that runs on workstations.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for this research was provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-92-J-4097.

1 Introduction

Cooperation between independent agents depends upon establishing a degree of security. Each of the cooperating agents needs assurance that the cooperation will not endanger resources of value to that agent. In a computer system, a computational mechanism can assure safe cooperation among the system's users by mediating resource access according to desired security policy. Such a mechanism, which is called a *security kernel*, lies at the heart of many operating systems and programming environments.

I claim that the λ-calculus can serve as the central component of a simple and flexible security kernel. The present report supports this thesis by motivating and describing such a λ-calculus-based security kernel and by giving several lines of evidence of the kernel's effectiveness.

The W7 security kernel[1] consists of the call-by-value λ-calculus with a few simple extensions to support abstract data types, object mutation, and access to hardware resources. Within W7, each agent (user or subsystem) has a separate evaluation environment that holds objects representing privileges granted to that agent. Because environments ultimately determine availability of object references, protection and sharing can be controlled largely by the way in which environments are constructed.

The effectiveness of W7 as a security kernel is demonstrated through three lines of evidence:

1. its ability to address certain fundamental security problems that are important for cooperation (Sections 2.2–2.3);
2. a structural correspondence with familiar operating system kernels (Section 2.4); and
3. the success of Scheme 48, a complete implementation of the Scheme programming language built on W7, as a basis for secure, robust, and flexible programming systems (Section 3).

1.1 Security Kernel Based on λ-calculus

The λ-calculus is a calculus of functions, and is concerned with how computations are abstracted and instantiated and how names come to have meanings. W7 is a λ-calculus of *procedures,* which are generalized functions capable of performing side effects. Procedures correspond to what in an operating system would be programs and servers. Side effects include access to input and output devices and to memory cells. It is through side effects that communication, and therefore cooperation, is possible. However, side effects can be harmful. For example, a computer-controlled robot arm can easily hurt a person who gets in its way.

The purpose of a security kernel is to allow control over access to objects. The challenge in designing a security kernel is not to support sharing or protection per se, but rather to allow flexible control over the extent to which an object is shared or protected. One way in which the λ-calculus and W7 provide protection is through *closure*: a procedure is not just a program but a program coupled with its environment of origin. A procedure cannot access the environment of its call, and its caller cannot access the procedure's environment of origin. The caller and callee are therefore protected from one another. Sharing is accomplished through shared portions of environments, which may include procedures that allow still other objects to be shared.

To address a number of authentication and certification problems, W7 includes an abstract data type facility. (Usually this term refers to type abstraction enforced through compile-time type checking, but here it means a dynamic information hiding mechanism.) The facility is akin to digital signatures: a subsystem may sign an object in such a way that the signed object may be recognized as having been definitely signed by that subsystem. In particular, a compiler might use a particular signature to mean that the signed procedure is one that is "harmless" (in a technical sense) and is therefore safe to apply to fragile arguments.

1.2 Scheme 48

Scheme 48 is a complete Scheme system that tests W7's capacity to support safe cooperation. Scheme 48 was a major design and implementation effort and therefore constitutes the heart of the project. Section 3 gives an overview of Scheme 48, but most of the information on it is to be found in the related reports [13, 23, 6, 19].

A large amount of engineering goes into making a practical programming environment, and in Scheme 48 security has been a concern in nearly every component. Major facilities whose design has been shaped by security concerns include the following:

- The module system [19]. Modules are truly encapsulated, just as procedures are, allowing them to be shared safely.
- The macro facility [6]. Macros are also closed, like procedures. This allows a form of compile-time security in which a module may export a macro while protecting objects used in the macro's implementation.
- Dynamic variables. Information can be communicated from caller to callee through an implicit dynamic environment. However, a dynamic variable must be accessed via a key, and such keys can be protected.

A major theme running through the design of Scheme 48 is avoidance or minimization of shared global state. For example, the virtual machine (byte-code interpreter) has only an essential minimum set of registers; there are no registers that hold global symbol tables or environment structure as there is in most Lisp and Scheme implementations. Another example is in the run-time system modules, which never alter global state (in Scheme terms, no top-level variable is ever assigned with `set!`). Data structures manipulated by these modules can be instantiated multiple times to avoid conflict over their use.

[1] This name was chosen to have no mnemonic or cuteness value.

Finally, the success of Scheme 48 (and therefore of W7) is demonstrated by its use in a number of applications. These include the programming environment for the Cornell mobile robots, where Scheme 48 runs with no (other) operating system support [23], and a secure multi-user environment that runs on workstations.

2 The Security Kernel

This section is an exposition, starting from first principles, of the problem of secure cooperation between independent agents. It describes a simple idealized security kernel that addresses this problem. The presentation is intended to show the essential unity of security concerns in operating systems and programming languages.

The idealized kernel is based on a graph of encapsulated objects. Accessibility of one object from another is constrained by the connectivity of the object graph and further limited by object-specific gatekeeper programs. The kernel enables the natural construction of a variety of mechanisms that support secure cooperation between agents. In particular, an agent can securely call an untrusted agent's program on a sensitive input.

The kernel is similar to those of the capability-based operating systems of the 1970's [15, 34]. The main differences are that this kernel is simpler, more abstract, and more clearly connected with programming language concepts than are classical capability systems.

2.1 Safe Computer-Mediated Cooperation

The participants in a cooperative interaction carry out a joint activity that uses or combines the resources that each provides. Resources might include energy, information, skills, or equipment.

Each agent relinquishes control to some extent over the resources that the agent brings. If the agent values a resource, relinquishing control over it is dangerous, because the resource may come to harm or may be used to cause harm.

To assure a resource's safety when control over it is relinquished, it is desirable for an agent to be able to dictate precisely the extent to which control is relinquished. A trusted intermediary can be helpful in this situation. The resource is handed over to the intermediary, who performs actions as specified by the recipient subject to restrictions imposed by the source.

A computer system may be useful in a cooperation not only because of the resources it may provide (programmable processor, memory, network communications, programs, and so on), but also because of its potential to act as a trusted intermediary. The agent providing a resource can dictate use restrictions in the form of a program, and the resource's recipient can specify actions to be performed with the resource in the form of a program.

Program invocation therefore plays a critical role in computer-mediated cooperation. But when a program is untrusted (as most programs should be!), invoking it is fraught with peril. The invoking agent puts at risk resources of value, such as inputs and the use of parts of the computer system. Similarly, the agent who supplied the program may have endowed it with (access to) resources of value to that agent, so those resources are also put at risk when the program is invoked.

Consider the following scenario:

> Bart writes a program that sorts a list of numbers. Being a generous fellow, he gives this useful program to Lisa, who has expressed an interest in using such a program. But Lisa is hesitant to use Bart's program, since Bart may have arranged for the program to do sneaky things. For example, the program might covertly send the list to be sorted back to Bart via electronic mail. That would be unfortunate, since Lisa wants to sort a list of credit card numbers, and she would like to keep these secret. What should Bart and Lisa do?

There are two distinct approaches to solving the safe invocation problem — that is, the general situation in which each agent in a cooperation has resources that should be protected when one agent's program is invoked in a context (inputs and computer system) given by another agent. In the first approach, the program runs in a limited computing environment, one in which dangerous operations (such as sending mail) are prohibited. In the second approach, the second agent rejects the program unless it bears a recognized certificate of safety. The next two sections look at the two approaches.

2.1.1 Limited Environments

One way to avoid disclosure of the numbers (an instance of the *confinement problem* [14]) would be for Lisa to isolate a computer running Bart's program, making it physically impossible for the program to do any harm. This would require detaching the computer from the network, removing disks that should not be read or written, setting up a special network or disk for transmitting the inputs and results, and so on.

These measures are inconvenient, time-consuming, and expensive. They are also crude; for example, it should be permissible for Bart's program to fetch information from the network or disks, but not write information. This selective restriction would be impossible using hardware configurations.

The basic idea is sound, however. The trick is to ensure that all accesses to sensitive resources are intercepted by a special supervisor program. The supervisor program checks the validity of the access attempt and allows it to go through only if it should.

The most straightforward method by which this is achieved is through the use of an *emulator*. An emulator is a program that mimics the role that the processor hardware usually plays in decoding and executing programs. For instructions that are always safe, the emulator simulates the hardware precisely; for dangerous instructions, the supervisor program is invoked.

Use of an emulator makes programs run slowly, so a more common alternative is the use of a special hardware feature available on many computers called "user mode". When the processor is in user mode, all dangerous operations are preempted, causing the supervisor program to be invoked. The supervisor program runs in an ordinary (non-user or "supervisor") mode, so it may implement whatever protection policy it likes.

Here is how safe program invocation might be accomplished using an architecture with user-mode support:

1. The invoker runs the program in user mode, telling the supervisor program which operations are to be permitted.
2. The supervisor program monitors all potentially dangerous operations, refusing to perform operations that are not permitted.
3. When the program is done, it executes a special operation to return to supervisor mode.

2.1.2 Screening and Certification

Another way to avoid disclosure of her list of numbers would be for Lisa to screen Bart's program to verify that it contains no dubious operations (such as calls to the send-email program). This would probably require her receiving the program in source form, not as a binary; the program would have to be of modest size and written in a language familiar to Lisa. She would also need access to a compiler and time on her hands to run the compiler. But even if Lisa were willing and able to go to this trouble, Bart might be unwilling to give the source program to her because it contains information that he wants to keep secret, such as passwords, material subject to copyright, trade secrets, or personal information.

This dilemma could be solved with the help of a trusted third party. There are many different ways in which a third party might help; the following is a variant of the screening method suggested above that doesn't require Lisa to obtain Bart's source program.

> Suppose that Bart and Lisa both trust Ned. Bart gives his source program to Ned with the instructions that Ned is not to give the source program to Lisa, but he may answer the question of whether the program is obviously harmless to run. ("Obviously harmless" could mean that the program uses no potentially harmful operations, or it could be a more sophisticated test. Because harmlessness is uncomputable, any such test will be an approximation.) Lisa obtains the machine program as before, and asks Ned whether the machine program is obviously harmless to run; if he says it is, she runs it with assurance.

In the screening approach, the agent invoking the program rejects it unless it bears a recognized certificate of safety. Following this initial check, invoking the program is just as simple as invoking any fully trusted program. There is no need for an emulator program or user-mode hardware support.

Here is how safe program invocation might be accomplished using the screening approach:

1. The invoker checks to see whether the program is definitely restricted to operations permitted by the invoker.
2. The invoker refuses to run the program if it appears to use any unpermitted operations.
3. If not, the invoker runs the program on the real machine. Harmful operations needn't be prevented, since they won't occur (if the certifier is correct).

Screening has the drawback that some cooperation is required from the person supplying the program. He must go to the trouble of obtaining certification, and perhaps even change the program so that it is certifiable.

2.2 A Simple Kernel

A *security kernel* is the innermost layer of an operating system or programming language, the fundamental component responsible for protecting valued resources from unwanted disclosure or manipulation. Isolating the security kernel from the rest of the system helps to ensure reliability and trustworthiness. Simplicity is important in a kernel because the simpler the kernel, the more easily it can be tested and verified.

What follows is just one of many ways to define a security kernel. This design, which I'll call *W7*, is intended to be suitable for use in either an operating system or a programming language. W7 might also be seen as a theory of security that can be used to describe security in existing programming languages and operating systems.

W7 organizes the computer's resources into a network of logical entities that I'll call *objects*.[2] Objects are the basic units of protection in W7. A link in the network from one object to another means that the second is accessible from the first. The objects to which a given object has access are its *successors,* and the objects that have access to a given object are its *predecessors.*

Objects may be thought of as representing particular privileges, capabilities, or resources. For example, an object might represent the ability to draw pictures on a display, to transmit messages over a communications line, to read information from a data file, or to run programs on a processing element.

New nodes enter the object network when a running program requests the creation of a new object. In order to avoid the dangers of dangling references (links to deleted nodes) and memory leaks, object deletion is left in the hands of the kernel. The kernel may delete an object as soon as there is no path through the network to it from the current computation. Absence of such a path is detected by an automatic mechanism such as reference counts or garbage collection.

There are several different kinds of objects. These will be introduced as they are needed in the presentation.

2.2.1 On the Choice of Notation

In order to elucidate the kernel design and describe its consequences, it will be necessary to present specimen programs, and for this purpose I must choose a notation — that is, a programming language. The choice is in principle arbitrary, but is important because it affects the exposition.

[2] Unfortunately, "object" has become a loaded term in computer science. To many people it implies complicated phenomena such as methods, classes and inheritance. I don't mean to suggest any of this baggage. It may be best to take it as an undefined term, as is "point" in geometry.

If I were most interested in comparing W7 to currently popular and familiar security kernels in operating systems, I would present my examples as programs that perform kernel operations in the traditional way, using system calls. Programs would be written in an Algol-like language, or perhaps in a machine-level language such as C. This would be a viable approach, since there is nothing in the theory that precludes it (see Section 4.3.3).

However, this approach makes some of the examples awkward to write and to understand. Particularly awkward is the frequent case where in creating a new object, a program must specify a second program. Most traditional languages don't have a good way for a program to specify another entire program. Even given such a notation, the division of labor between the language and the security kernel might be difficult to tease apart.

For this reason I prefer to use a notation in which kernel operations have natural forms of expression. The language of the Unix "shell" approaches this goal since program invocation has a natural syntax, and operations on programs and processes are more concise than they would be in Algol or C.

However, I would like to go a step further in the direction of languages with integrated kernel support, and use a version of Scheme [7] in which all Scheme values, including procedures, are identified with objects known to the W7 kernel. ML [16] would have been another reasonable choice of language.

2.2.2 On the Notation

To avoid confusion with other Scheme dialects, I'll refer to the small dialect of Scheme to be used in examples as "Scheme$^-$." The grammar given in Figure 1 summarizes Scheme$^-$. E stands for an expression.

The meaning of most of the constructs is as in full Scheme. The first group of constructs is Scheme$^-$'s λ-calculus core: variable reference, procedure abstraction (`lambda`), and application. Each of the second group of constructs (`if`, etc.) has a special evaluation rule.

The third group is a set of primitive operators. In every case all of the operand expressions are evaluated before the operator is applied. Most of these are familiar, but a few require explanation:

A *cell* is a mutable object with a single outgoing access link (field). `(new-cell)` creates a new, uninitialized cell, `(cell-ref` *cell*`)` returns the object to which *cell* currently has a link, and `(cell-set!` *cell obj*`)` redirects *cell*'s outgoing link to *obj*.

The `enclose` operator takes a program and a specification of a successor set and converts them to a procedure. `enclose` might be called on the output of a Scheme$^-$ or Algol compiler. Its details are unimportant. See Section 2.2.4.

The `control` operator controls hardware devices. For example,

```
(control keyboard 'read-char)
```

might read a character from a particular keyboard. Its arguments are interpreted differently for different kinds of devices.

To reduce the complexity of the exposition, pairs will be assumed to be immutable, and `eq?` will be assumed to be applicable only to symbols and cells.

I'll use the term *value* to describe integers, booleans, symbols, and other entities that carry only information, not privileges. Whether or not values are considered to be objects is unimportant.

2.2.3 Procedures

The W7 kernel is principally concerned with *procedures.* Every procedure has an associated program. The procedure's program can access the procedure's successors and use them in various operations. Access is not transitive, however: access to a procedure does not imply the ability to access the procedure's successors. Any such access is necessarily controlled by the procedure's program. In this sense, the procedure's program is a "gatekeeper" that protects the procedure's successors.

The primary operation on a procedure is application to an argument sequence. When a procedure is applied to arguments (which may include other objects or values, such as integers), its program is run. In addition to the procedure's successors, the program has access to the arguments. It may use any of these objects, but no others, in making new objects or in performing further applications. Applications are written in Scheme$^-$ using the syntax

`(`*procedure argument* `...)`.

For example, if `f` names a link to a procedure, then `(f x y)` denotes an application of that procedure to `x` and `y`.

The essential source of security in W7 is that a procedure's program is *absolutely limited* to using the procedure's successors and the objects that are in the application's argument sequence.

Scheme$^-$ `lambda`-expressions are a concise notation for specifying procedures. When a program evaluates a `lambda`-expression, a new procedure is created. The new procedure's successors are the objects that are the values of the `lambda`-expression's free variables, and its program is specified by the body of the `lambda`-expression.

Here is an example illustrating `lambda` and application.

```
(lambda (x) (g (f x)))
```

specifies a procedure with two successors, which it knows as `f` and `g`. When the resulting procedure (say, h) is applied to an argument, it applies `f` to that argument, and then applies `g` to the result obtained from applying `f`. Finally, `g`'s result becomes the result of the call to h. h therefore acts as the functional composition of `f` and `g`.

An procedure's program could be written in a language other than Scheme$^-$, for example, Algol, C, or a machine language. I will consider this possibility later (Section 4.3.3).

2.2.4 Initial program

Suppose that someone — let's call her Marge — obtains a brand-new machine running W7 as its operating system kernel. She turns it on, and the machine begins executing an initial program. The initial program is an ordinary program with no special relation to the kernel

```
E ::= var
    | (lambda (var ...) E)
    | (E E ...)

    | constant
    | (if E E E)
    | (begin E ...)
    | (let ((var E) ...) E)
    | (let var ((var E) ...) E)

    | (arith E E)
    | (cons E E) | (car E) | (cdr E)
    | (null? E) | (pair? E) | (list E ...)
    | (symbol? E) | (eq? E E)
    | (new-cell) | (cell-ref E) | (cell-set!\ E E)
    | (enclose E E) | (control E E)

arith ::= + | - | * | / | < | = | >
```

Figure 1: A grammar for Scheme$^-$.

other than that it is executed on power-up. In principle, the initial program is arbitrary; it is whatever the manufacturer has chosen to provide.

The initial program is given access to objects that represent all of the computer system's attached hardware resources, which might include keyboards, displays, facsimile machines, traffic lights, etc. (Hardware may be manipulated with the `control` primitive operator, whose details are not important here.) Because these resources are objects, any given program will only have access to those hardware resources to which it has been explicitly granted access by the initial program (perhaps indirectly through a series of other programs).

In order to make the computer as generally useful as possible, the manufacturer has installed an initial program that executes commands entered from the keyboard. One kind of command is simply a Scheme$^-$ expression. When an expression is seen, the initial program compiles and executes the expression and displays its result. For example, the command

```
(+ 2 3)
```

causes the number 5 to be displayed.

A second kind of command is a *definition*, which looks like

```
(define var E).
```

Definitions give a way for a user to extend the environment so that new objects can be given names that subsequent commands can see. For example, Marge can write

```
(define square (lambda (x) (* x x)))
```

to define a squaring procedure, and

```
(square 17)
```

to invoke it and display the result.

To be able to use existing resources, commands need to have access to them. For this reason they are processed relative to an *environment,* or set of variable bindings, allowing objects to be accessed by name. The environment initially includes two kinds of resources: I/O devices (with names like `the-display` and `the-fax-machine`), and handy utilities that the user could have written but the manufacturer has thoughtfully supplied. The utilities are the usual Scheme procedures such as `assoc`, `write`, `read`, and `string-append`, as well as a few others to be described below.

The initial program just described, including utilities and command processor, might have been written entirely in Scheme$^-$, in which case it would resemble the program of Figure 2: `eval` is a utility that executes a Scheme$^-$ expression, of which a representation has been obtained from the user or otherwise received or computed. It makes use of a compiler that translates the expression into a form acceptable to the kernel's primitive `enclose` operator. The translated expression is given access to a specified environment, and executed.

```
(define eval
  (lambda (expression env)
    ((enclose (compile-scheme-expression
                (map car env))
              (map cdr env)))))
```

Definitions are handled using `bind`, which extends a given environment by adding a binding of a variable to a value. With environments represented as association lists, `bind` could be defined with

```
(define bind
  (lambda (name value env)
    (cons (cons name value) env))).
```

2.2.5 Administration

Now we have enough mechanism at our disposal to consider some examples.

Marge intends to set up her machine so that Lisa and Bart can use it and its resources. As a means for the machine's users to share objects with one another, she defines a simple object repository:

```
(define *repository* (new-cell))
(cell-set! *repository* '())
```

```
(define command-processor
  (lambda (env source sink)
    (let loop ((env env))
      (let ((command (read source)))
        (if (definition? command)
            (loop (bind (cadr command)
                        (eval (caddr command) env)
                        env))
            (begin (write (eval command env) sink)
                   (loop env)))))))

(define definition?
  (lambda (command)
    (if (pair? command)
        (eq? (car command) 'define)
        #f)))
```

Figure 2: A simple initial program.

```
(define lookup
  (lambda (name)
    (let ((probe (assoc name *repository*)))
      (if probe (cdr probe) #f))))

(define publish!
  (lambda (name object)
    (cell-set! *repository*
      (cons (cons name object)
            *repository*))))
```

Anyone with access to Marge's `lookup` procedure can obtain values from the repository, while anyone with access to `publish!` can store values in the repository.[3] (`assoc` is assumed to be Lisp's traditional association list search function.)

Next, Marge makes an environment for Bart's command processor. It must include all of the objects that Bart is to be able to access initially. There is no harm in including all of the system's utility procedures (`assoc`, `read`, etc.), since no harm can come from his using them. However, she needs to be careful about the I/O devices. For the sake of safety, she includes only Bart's own I/O devices in `Bart-env`:

```
(define make-user-env
  (lambda (from-user to-user)
    (bind 'standard-input from-user
       (bind 'standard-output to-user
             (bind 'lookup lookup
                   (bind 'publish! publish!
                         utilities-env))))))
(define Bart-env
  (make-user-env from-Bart to-Bart))
```

Bart's command processor can now be initiated with

```
(command-processor Bart-env
                   from-Bart to-Bart)
```

[3] For simplicity, it is assumed that concurrency is not an issue. In a multiprocessing or multitasking system, access to the `*repository*` cell would have to be serialized.

Similarly for Lisa's:

```
(command-processor Lisa-env
                   from-Lisa to-Lisa)
```

Security results from the omission of vulnerable objects from users' initial environments. For example, Lisa's I/O devices (`from-Lisa` and `to-Lisa`) aren't accessible from Bart's environment, so Bart won't be able to cause any mischief by reading or writing them.

Bart and Lisa are able to share objects with each other through the repository:

```
Bart:
(define really-sort
  (lambda (list-of-numbers)
    (if (null? list-of-numbers)
        '()
        (insert (car list-of-numbers)
                (really-sort
                 (cdr list-of-numbers))))))

(define insert
  (lambda (x l)
    (let recur ((l l))
      (if (null? l)
          (list x)
          (if (< x (car l))
              (cons x l)
              (cons (car l)
                    (recur (cdr l))))))))

(publish! 'sort sort)

Lisa:
(define Bart-sort (lookup 'sort))
(Bart-sort '(9 2 7))  ⟶  '(2 7 9)
```

Of course, Bart and Lisa must both trust Marge, who still has full control over all resources. They must treat her administrative structure as part of the machine's trusted substrate. (Trust cannot be avoided. Bart and Lisa must trust Marge's software and machine just as Marge trusts the manufacturer's installed software, the manufacturer trusts the semiconductor chip factories, the chip factories trust the solid state physicists, and so on.)

2.2.6 Trusted Third Party Establishes Safety

There is already a great deal of safety built in to the structure of the W7 kernel. When a procedure is invoked, the only privileges the invoked procedure's program has are (1) those that it was given when created ("installed"), and (2) those that are passed as arguments. If the two privilege sets contain no dangerous privileges, then there is no way that any harm can come from the invocation. However, sometimes there is reason to pass privileges or sensitive information to an unknown program. This can be a problem if when installed the program was given access to channels over which the information might be transmitted, or to a cell in which a privilege or information may be saved for later unexpected use.

To return to the main example from the introduction: Lisa wants to call Bart's `sort` program without risking disclosure of the input, which may contain sensitive information. Specifically, the risk she runs is that Bart has written something like

```
(define *list-of-numbers* (new-cell))

(define sort
  (lambda (list-of-numbers)
    (begin (cell-set! *list-of-numbers*
                      list-of-numbers)
           (really-sort list-of-numbers))))

(define really-sort
  (lambda (list-of-numbers)
    (if (null? list-of-numbers)
        '()
        (insert (car list-of-numbers)
                (really-sort
                  (cdr list-of-numbers))))))
(publish! 'sort sort)
```

This deceit would allow Bart to extract the most recent input passed to `sort` by evaluating `(cell-ref *list-of-numbers*)`.

Sketch of a rudimentary solution:

1. Ned, who both Bart and Lisa trust, sets up a "safe compilation" service.
2. Bart submits the source code for his `sort` program to Ned's service.
3. Ned's service analyzes Bart's source code to determine whether the program might divulge its input to Bart or to anyone else. (For a definition of "might divulge," see below.)
4. Ned evaluates (for initialization) Bart's program in a fresh environment and extracts access to the resulting `sort` procedure.
5. Ned makes `sort` available to Lisa.
6. Lisa obtains the program from Ned, and runs it with assurance that the input will not be divulged.

Whether a program "might divulge" its input is undecidable, so any test for this property is necessarily conservative: such a test will reject perfectly benign, useful programs that can't easily be cast into a recognizably safe form.

There are many possible tests. One simple test is the following one: a program might divulge its input if the program is not obviously applicative; and a program is obviously applicative iff none of its `lambda`-expressions contains a `(cell-set! ...)` expression.

One can clearly do better than this; some methods will be discussed later (in the context of the module system, section 3.2).

To see how this safe compilation service might work, let's continue with the scenario begun above in which Marge has established a public object repository. The first problem is that Lisa needs a way to determine authorship of a repository entry, to distinguish Bart's entries from Ned's. Assuming that all objects published by Ned are safe, she must make sure that the `sort` procedure she uses is published by Ned, not by Bart.

This is a fundamental flaw in the repository, so Marge must fix it. See Figure 3. Each object in the repository is now accompanied by the name of the user who put it there, and each user's environment now has its own `publish!` procedure. Bart's `publish!` procedure, for example, will store entries of the form `(cons 'Bart` *object*`)` into the repository.

Here is the complete scenario:

```
Ned:
(define publish-if-safe!
  (lambda (name program)
    (if (safe? program)
        (publish! name (eval program))
        'not-obviously-safe)))
(define safe?
  (lambda (program) ...))
(publish! 'publish-if-safe! publish-if-safe!)

Bart:
(define sort-program
  '(begin (define sort ...)
          (define insert ...)
          sort))
(define publish-if-safe!
  (cdr (lookup 'publish-if-safe!)))
(publish-if-safe! 'safe-sort sort-program)

Lisa:
(define safe-sort-entry (lookup 'safe-sort))
(define safe-sort
  (if (eq? (car safe-sort-entry) 'Ned)
      (cdr safe-sort-entry)
      "safe-sort not published by Ned"))
(safe-sort '(9 2 7))
```

There is nothing Bart can do to trick Lisa. If he tries to publish an unsafe `safe-sort` procedure, he will have to use his own `publish!` procedure to do so, since Ned's `publish!` isn't accessible to him. In this case, when Lisa does `(lookup 'safe-sort)`, the result will be marked

```
(define make-publish!
  (lambda (submitter)
    (lambda (name object)
      (publish! name (cons submitter object)))))

(define make-user-env
  (lambda (user-name from-user to-user)
    (bind 'standard-input from-user
          (bind 'standard-output to-user
                (bind 'lookup lookup
                      (bind 'publish!
                            (make-publish! user-name)
                            utilities-env))))))

(define Bart-env
   (make-user-env 'Bart from-Bart to-Bart))
(define Lisa-env
   (make-user-env 'Lisa from-Lisa to-Lisa))
(define Ned-env
   (make-user-env 'Ned from-Ned to-Ned))
```

Figure 3: Marge's improved repository.

with Bart's name, not Ned's, and Lisa will discard it.

2.3 Authentication

This section treats the problem of *authentication* and considers its solution in W7. Broadly speaking, authentication is any procedure or test that determines whether an object is trustworthy or genuine. For example, checking a student identification card authenticates the person presenting the card as being a student as indicated on the card. Having been found authentic, the person may be granted privileges appropriate to that status, such as permission to use an ice rink.

Authentication is an important capability of secure computer systems. Some examples of authentication in such systems are:

- A request received from an untrusted source such as a public communications network must be authenticated as originating from an agent that has the right to perform the action specified by the request.
- In a dynamically typed programming language such as Lisp or Snobol, a value must be authenticated as being of the correct type for an operator receiving it as an operand.
- The solution to the safe invocation example of Section 2.2 involves a test for the authenticity of a putatively safe or trustworthy object (Bart's program).

Authentication is necessary for reliable transmission of an object using an untrusted messenger (or channel). To authenticate an object transmitted from a sender to receiver, the sender must label or package the object so that it can be recognized (authenticated) by the receiver, and this labeling or packaging must be done in an unforgeable and tamper-proof manner. I'll use the word *capsule* for an object so labeled or packaged. A physical analogy for a capsule would be a locked box to which only the receiver has a key.

When the object is digital information, such as a sequence of characters or numbers, well-known digital cryptographic techniques can be used to implement authenticated transmission [24]. However, object transmission in a security kernel that limits object access according to an accessibility graph requires a different mechanism. With such a kernel, transmitting a name for an object, whether encrypted or not, is never the same as transmitting access to that object, since interpretation of names — and therefore accessibility of named objects — is local to each object. Access can only be transmitted through approved channels such as argument and result transmission in a procedure invocation.

2.3.1 Abstract Data Types

Authentication is required whenever the results of one procedure are intended to constitute the only valid inputs to another procedure. These objects, the results and valid inputs for a related set of procedures, are called the instances of a *type* or *abstract data type*. The set of related procedures is called a *module* or *cluster* in the context of programming languages, or a *protected subsystem* or *server* in classical operating systems parlance.

For example, consider a central accounting office that issues account objects to clients. Clients may create new accounts using a `new-account` procedure and transfer money between accounts using a `transfer` procedure. The integrity of accounts is of great importance to the accounting office; it would be unfortunate if a client created a counterfeit account and then transferred money from it into a valid account. Therefore, `transfer` needs to authenticate the accounts it's asked to manipulate as genuine — that is, created by `new-account`.

From the clients' point of view, accounts are objects that clients may manipulate a certain set of operators.

From the accounting office's point of view, accounts are capsules transmitted from `new-account` to `transfer` via an untrusted network of interacting clients.

2.3.2 Key-based Authentication

Section 2.2.6's solution to the safe invocation problem relies on the exchange of object *names*. Bart tells Lisa the name of his object in the trusted repository (`safe-sort`), and Lisa obtains from the repository (1) the fact that the object is authentically safe (i.e. placed there by Ned), and (2) the object itself, which Lisa can then use.

If Bart wishes to keep his object a secret between him and Lisa, he may give it a secret key in the form of an unguessable name, i.e. a password, and transmit that name to Lisa. Passwords can be made long enough and random enough to make the probability of unauthorized discovery arbitrarily small. (Of course, this assumes that there is no way for untrusted agents to obtain a list of all names defined in the repository.)

Instead of passwords, it is possible to use cells (Section 2.2.2) as keys, as suggested by Morris [18]. Like passwords, cells are objects with unique, recognizable identities. Unlike passwords, they have all the security advantages of objects: access to them can only be transferred through kernel-supported operations.

There is no need to use a single repository, such as Marge's, to store all objects that might need to be authenticated. Each separate abstract data type can have its own mapping from keys to objects.

Figure 4 gives an implementation of a general authentication facility (essentially the same as that described by Morris [18]). Each call to `new-seal` returns a new triple of objects ⟨*seal, unseal, sealed?*⟩. Each such triple represents a new abstract data type. *seal* encloses an object in a capsule that can be authenticated; *unseal* is an extraction operator that reveals the encapsulated object iff the capsule is authentic (i.e. was made by this *seal*); and *sealed?* is a predicate that returns true iff the capsule is authentic.

It is assumed that `assq` (association list lookup) can determine cell identity. Cells (other than the one holding the association list) are used only for their identities, not for holding references to objects. As an example, the program of Figure 5 is an implementation of the accounting system described above. It's very important *not* to publish `account-cell`, since doing so would allow clients of the accounting system to set account balances to arbitrary values.

2.3.3 The Case for Kernel Support

As seen above, existing kernel mechanisms (cells, procedures) can be used to implement authentication. However, I would like to argue in favor of a direct authentication mechanism implemented in the W7 kernel. An argument in favor of this is required due to the stated goal of keeping the kernel as simple as possible.

A key-based authentication mechanism has several practical problems.

- Performance: It's inefficient to have to search the central table on every use of an instance.
- Memory management: How can the kernel know whether it is safe to delete an object? Straightforward garbage collection techniques don't work because the central table holds links to all of the type's instances, and the garbage collector is ignorant of the association between the key and the object.
- Semantic obscurity: It would be unfortunate if creating an object of an abstract data type were necessarily a side effect, as it would be if a key had to be generated. The side effect defeats optimizations such as subgraph sharing.

A direct approach, not involving a keyed table, requires kernel support. The following argument shows that in the W7 kernel without abstract data type support, a recipient can be fooled no matter what packaging method is used.

> Consider a candidate encapsulation and authentication technique. Without loss of generality, we can assume that capsules are procedures. In order to authenticate a capsule, the recipient has no choice but to apply it to some argument sequence, since there is nothing else that can be done with a procedure. This application can appropriately be called a "challenge," since examining its result distinguishes authentic capsules from nonauthentic ones. If the challenge is correctly met, then an additional application of the capsule (or of some related object returned by the challenge) will then either return or perform some operation on the underlying object.
>
> But given an authentic capsule, an untrusted messenger may easily make a counterfeit, as follows: the counterfeit answers challenges by consulting the authentic capsule and returning what it returns, and handles other applications in any way it likes.

This argument doesn't even consider the problem that the challenge might diverge, signal an error, or cause troublesome side effects. The annoyance of handling these situations (see section 4.3.1) would argue against using procedures, even were there a way to do so.

Kernel support can be provided in W7 by adding `new-seal` as a new primitive operator:

E `::= (new-seal)`

Capsules are a new kind of object that can be implemented efficiently. A capsule can be represented as a record with two fields, one containing the unique seal, and the other holding the encapsulated object.[4]

[4] It is an inelegant aspect of this design that there are two distinct encapsulation mechanisms (procedures and capsules). There are various ways to remedy this; for example, instead of introducing capsules as a separate kind of object, `lambda` and application could be extended to take unique markers, with the requirement that a procedure's marker must match the marker specified in an application of the procedure:

```
(define (new-seal)
  (let ((instances (new-cell)))
    (cell-set! instances '())
    (let ((seal
           (lambda (rep)
             (let ((abs (new-cell)))
               (cell-set! instances
                          (cons (cons abs rep)
                                (cell-ref instances)))
               abs)))
          (unseal
           (lambda (abs)
             (let ((probe (assq abs (cell-ref instances))))
               (if probe
                   (cdr probe)
                   (error "invalid argument" abs)))))
          (sealed?
           (lambda (x)
             (if (assq x (cell-ref instances))
                 #t
                 #f))))
      (list seal unseal sealed?))))
```

Figure 4: Implementation of seals.

```
(define account-operators (new-seal))
(define make-account (car account-operators))
(define account-cell (cadr account-operators))
(define account? (caddr account-operators))

(define new-account
  (lambda ()
    (make-account (new-cell 0))))

(define transfer
  (lambda (amount from to)
    (let ((from-cell (account-cell from))
          (to-cell   (account-cell to)))
      (if (>= (cell-ref from-cell) amount)
          (begin (cell-set! from-cell
                            (- (cell-ref from-cell) amount))
                 (cell-set! to-cell
                            (+ (cell-ref to-cell) amount)))
          (error (error "insufficient funds"))))))

(publish! 'new-account new-account)
(publish! 'transfer transfer)
```

Figure 5: Accounting module.

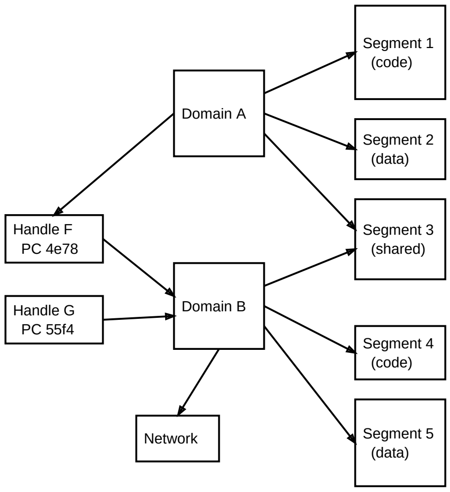

Figure 6: In an operating system, accessibility of resources is controlled by kernel data structures called *protection domains* (or simply *domains*). A domain includes a segment table or page table that specifies which memory segments are directly accessible by a program running in that domain. A domain also specifies availability of other resources, such as I/O devices (here represented by a network) and procedures residing in other domains. (References to such procedures are called *remote procedure handles* or *inter-process communication handles* in the literature.)

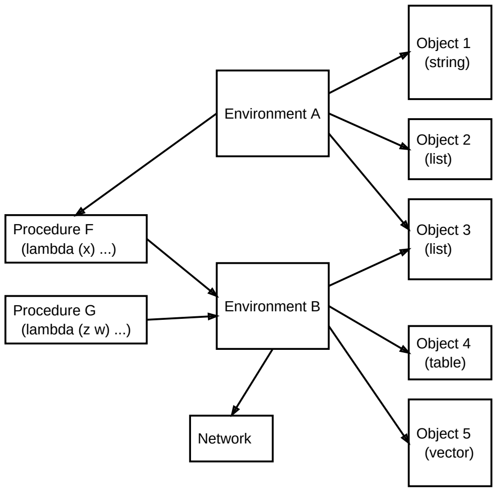

Figure 7: In a λ-calculus interpreter, accessibility of resources is controlled by interpreter data structures called *environments*. An environment contains references to data structures, such as lists, that are directly accessible to a program running in that environment. An environment also specifies availability of other resources, such as I/O devices (here represented by a network) and procedures connected to other environments.

2.4 Protection Domains

This section attempts to explain the correspondence between W7 and more conventional operating system kernels.

In a typical secure operating system, the objects that are immediately accessible to a running program constitute its *protection domain* (or simply *domain*). These objects are of various sorts, including the following (there may be others):

1. Mapped memory segments. These contain both programs and the data stuctures that programs directly manipulate.

2. Descriptors for hardware devices and files. Descriptors control access to devices and the file system, and may include information such as buffers or position markers.

3. In some operating systems, references to procedures residing in other domains (variously called *gateways, inter-process communication handles,* or *remote procedure handles*); I will write simply *handle.* A handle consists of a domain together with an address within that domain specifying the procedure's entry point.

The program refers to objects using short names (numbers). For example, a load or store machine instruction uses a memory address to access a memory segment object, while a system call instruction requesting that information be read from a file uses a file descriptor number to specify the file.

The domain determines how these names are to be interpreted. The domain contains a segment table or page table for use by the hardware in interpreting memory addresses, and tables of descriptors and handles for use by system call handlers in interpreting I/O and cross-domain procedure call requests.

Just as a domain maps a name (address or descriptor number) to an accessible object (memory segment or descriptor), an environment in W7 maps a name (identifier) to its denotation (value or object). The various object types are parallel in the two frameworks as well (see Figures 6 and 7):

1. Mapped memory segments correspond to lists and cells. (Full Scheme also has vector and string data types, which are better approximations to memory segments than are lists and cells. A more complete version W7 might support memory segments directly.)

2. Device and file descriptors correspond to W7 devices.

3. Handles correspond to W7 procedures. Just as a handle is a domain coupled with an entry point, a procedure is an environment coupled with executable code.

`(lambda` *marker* `(`*var* `...)` *body*`)`
`(applicate` *marker procedure arg* `...)`

A number of other unifications have been proposed [22, 1, 25].

3 Implementation Experience

A variant of the W7 kernel forms the basis of a complete Scheme implementation called Scheme 48. This section describes Scheme 48, analyzes it from a security standpoint, and discusses experience with its use in various applications in which security is important.

The implementation of Scheme 48 consists of two parts: a virtual machine that manages memory and executes a byte-code instruction set, and a set of modules that are executed by the virtual machine. Virtual machine code could in principle be replaced by real machine code, but this would require a significant compiler construction effort, and that is beyond the scope of the project at this time. The virtual machine approach has the virtues of robustness, small size, and ease of portability.

(A thorough discussion of Scheme 48 as a Scheme system can be found elsewhere [13].)

3.1 Scheme 48 Security Kernel

Scheme 48's security kernel is implemented in part by the virtual machine and in part by a set of run-time system modules defined using privileged instructions supplied by the virtual machine. The VM component includes instructions such as `car`, which is secure because the VM raises an exception if it is applied to anything other than a pair, and application, which raises an exception when given a non-procedure. The run-time modules include the exception system and the byte-code compiler. Both of these export interfaces whose use is unrestricted, but they are defined in terms of lower-level operations that are restricted.

Type safety for instructions such as `car` and procedure application rely on an assiduously followed tagging discipline. All values, both immediate values such as small integers and access links to objects stored in memory, are represented as *descriptors.* A descriptor has a tag that distinguishes immediate values from links. Arithmetic operations require immediate-tagged operands and affix immediate tags to results. The non-tag portion of a descriptor for a link holds the hardware address of the stored object. A stored object is represented as a contiguous series of memory locations. The first memory location holds a header specifying the object's size and type (procedure, capsule, cell, etc.), while subsequent locations hold descriptors (which include the links to the object's successors in the accessibility network).

The tagging discipline guarantees that object addresses are never seen by the programmer. Not only does this promote security, it also gives the memory manager freedom to move objects from one place to another, guaranteeing that such relocations will not affect any running computations.

The security kernel does not provide direct access to the byte-code interpreter. Instead, new programs must be given as Scheme programs that are byte-compiled by `eval`. The compiler translates the program to a byte-code instruction stream, and then uses a privileged `make-closure` instruction to make a procedure. Use of the `make-closure` instruction is restricted for two reasons:

- An agent could defeat security by constructing and executing code streams containing privileged instructions. For example, the `closure-env` instruction allows unrestricted access to all the successors of an arbitrary procedure.
- An agent could execute an object access instruction with an index specifying a location beyond the end of the object being accessed. This might fetch a reference to an object that shouldn't be seen. (For performance reasons, some instructions perform no bounds checking. They rely on the byte-code compiler to ensure that indexes are valid.)

Abstract data types (Section 2.3.2) are used heavily in the Scheme 48 run-time system. They are provided in the form of a record facility, which is more convenient and efficient than the theoretically equivalent but more elegant `new-seal` would be. In the common case where a capsule is a tuple (that is, has several fields), only a single object (a record) is created for both the tuple and the capsule. Not only is less memory consumed, but field references require only one indirection instead of two.

The features of Scheme 48's security kernel beyond what W7 defines are the following:

- The basic features of standard Scheme [12]: characters, vectors, the full suite of numeric operators, strings, string/symbol conversion. None of these has security implications; they belong to the kernel for the sake of efficiency.
- Exception handling. It is important for the agent initiating a computation to be able to gain control when the computation leads to an exceptional event such as division by zero. A special construct allows all exceptions or exceptions of certain types to be intercepted.
- Fluid variables. Every application implicitly passes an environment that maps unique tokens (called *fluid variables*) to cells. A special construct extends the environment with a new binding. The interesting aspect from a security perspective is that if the fluid variable (unique token) is protected, then bindings of the fluid variable will be protected. Thus if X calls Y calls Z, then it can be the case that Z can access fluid bindings established by X, while Y can't. This contrasts with a similar facility in Unix (process environments), which has no such protection.
- Multiple threads of control. A new thread of control may be started with `(spawn` *thunk*`)`. Synchronization mechanisms include locks and condition variables. Each thread has its own separate exception context and fluid environment. (Threads are not really secure; see Section 4.3.1.)
- Immutability. This feature is somewhat of a frill, but useful. A pair, vector, or string can be made read-only, as if it had been created by `(quote ...)`. Immutable objects may be passed to untrusted procedures without worry that they might be altered, since an attempted mutation

(set-car!, vector-set!, etc.) will raise an exception.

Variables and pairs are mutable, as in Scheme.

3.2 Module System

Scheme 48 provides operators for constructing and querying environments. Environment control is as important for determining what is in an environment as it is for advertising what is *not* in an environment. Roughly speaking, an environment that excludes dangerous things can't be dangerous.

The environments that are manipulated as objects and passed to the compiler are called *top-level* environments or *packages.* Packages have both static and dynamic components. The static component defines information important for compilation: syntactic keywords, type information, and early binding information such as definitions of in-line procedures. The dynamic component of a package determines how variables obtain their values at run time.

Bindings move between packages via another kind of environment-like object called a *structure*. A structure is a particular package's implementation of a particular *interface*; and an interface is a set of names, with optional types attached:

(make-simple-interface *names*)
  ⟶ *interface*
(make-structure *package* *interface*)
  ⟶ *structure*

Dually, a package receives bindings from other packages via their structures:

(make-package *structures*) ⟶ *package*

Borrowing terminology from Standard ML, I'll say that a package *opens* the structures from which it receives bindings.

Here is a simple example. Assume that `scheme` is defined to be a structure holding useful bindings such as `define`, `lambda`, and so forth.

```
(define repository-package
  (make-package (list scheme)))
(eval '(begin
         (define *repository* ...)
         (define lookup ...)
         (define publish! ...))
      repository-package)
(define repository-interface
  '(lookup publish!))
(define repository
  (make-structure repository-package
                  repository-interface))
...
(define repository-client
  (make-package (list scheme repository)))
```

(I describe the Scheme 48 module system in more detail elsewhere [19].)

3.2.1 Checking Structures for Safety

As seen in Section 2.2.6, it is useful to be able to determine whether a procedure is safe. In particular, it is useful to know whether a procedure has access to any resource that could be used as a communication channel to the outside world. If it were possible to traverse the access graph starting from the procedure, it would be possible to determine the answer to this question. However, such a traversal cannot be done outside the security kernel.

In Scheme 48, information necessary to perform certain safety checks is contained in the network of structures and packages. This network can be considered a quotient (summary) of the true access network. A procedure is considered safe when the structure that exports it is safe; a structure is safe if its underlying package is safe; and a package is safe if every structure that it opens is safe. This definition of "safe" is recursive, so it can be made relative to a basis set of structures that are assumed safe. Following is a crude implementation of such a safety predicate:

```
(define (safe? struct assumed-safe)
  (cond ((memq struct assumed-safe) #t)
        ((structure? struct)
         (every (lambda (o)
                  (safe? o assumed-safe))
                (package-opens
                 (structure-package struct))))
        (else #f)))
```

This isn't a precise test, but it has the virtue of being simple and intuitive. Rather than rely on sophisticated techniques such as types, effects, verification, or other kinds of code analysis, it only involves visibility of values.

The usual Scheme 48 run-time system provides a number of structures, some of which may be considered safe for the purpose of guaranteeing the absence of communication channels. The structure implementing the standard Scheme dialect is not one of these because it is so easy to use it to create such channels. A different `safe-scheme` structure is defined that eliminates the possibility of such channels, as described following:

One source of channels is assignments to top-level variables. For example:

```
(define *last-sorted-list* #f)
(define (sort l)
  (set! *last-sorted-list* l)
  (really-sort l))
```

We cannot get rid of top-level definitions, so to prevent this, `set!` must be disallowed on top-level variables (except possibly during initialization). The current solution is to exclude `set!` entirely from `safe-scheme`, since otherwise the compiler would have to be modified, and project time constraints didn't allow this.

Mutations to data structures reachable from top-level variables also must be prohibited, but now instead of excluding mutations entirely (as suggested in Section 2.2.6), we exclude top-level variables that hold mutable data structures. This is done by excluding normal Scheme `define` and replacing it with a variant that makes the right assurances:

- (define (*var var* ...) *body*) is allowed.

- `(define` *var exp*`)` is allowed iff *exp* evaluates to a verifiably immutable object.

Verifiably immutable objects include scalars such as numbers and characters, immutable strings, and immutable pairs, vectors, and capsules that have verifiably immutable components. Note that the category doesn't include procedures, since procedures can easily hide channels.

Of course, `safe-scheme` also excludes all operations that access the file system. It includes I/O routines that take explicit port arguments (such as `(display` *x port*`)`) but excludes their implicit-port variants (`(display` *x*`)`).

3.2.2 Static Privileges

The fact that static bindings of syntactic keywords, macros, and primitive operators are all determined relative to a package means that any such entity may be considered a privilege to be granted or not. For example, a particular agent (user or subsystem) might be denied access to all operators that have side effects just by excluding those operators from environments available to the agent. Scoping of all syntactic and static entities effectively allows an equation between languages and modules, with the derivative ideas of safe languages and modules that are safe because they are written in safe languages.

Macros are secure in a precise sense: names that a macro introduces into expanded program text are resolved in the environment of the macro's definition, not in the environment of its use. For example, a package might define `letrec` in terms of `set!` and then export `letrec` without exporting `set!`. The structure with `letrec` and its clients can be considered applicative (or safe) even though the process of compiling it involves source to source rewrites containing imperative operators.

(Further explanation of the problem of lexically scoped macros and the algorithm used by Scheme 48 to implement them can be found elsewhere [6].)

3.3 Deployed Configurations

The Scheme 48 configurations described here are the more interesting ones with respect to protection problems. Other configurations include the single-user development environment (by far the most evolved), a multiprocessor version written at MIT by Bob Brown, and a distributed version developed by Richard Kelsey and others at NEC.

3.3.1 Mobile Robot System

Scheme 48 is the operating system running on four mobile robots at the Cornell Computer Science Robotics and Vision Lab (CSRVL). The main on-board computer system is a 16 MHz MC68000 with 500K RAM and 250K EPROM.

User code is isolated from operating system internals by Scheme's type and bounds checking; this means that there's nothing that a user program can do to harm the robot. This is important because the robots are programmed by undergraduates taking the robotics course, many of whom are relatively novice programmers.

The Scheme 48 virtual machine running on the robot communicates with a development environment running in either Scheme 48 or Common Lisp on a workstation. The byte-code compiler runs on the workstation and sends byte codes to be executed on the robot. This "teledebugging" link offloads to the workstation all debugging and programming environment support, including the byte-code compiler. This division of labor frees up precious memory on the robot. The tether, which can be a physical encumbrance, can be detached without affecting the virtual machine, and reattached at any time for debugging or downloading.

(A more detailed description of the mobile robot system can be found elsewhere [23].)

3.3.2 Multi-User System

The Scheme 48 development environment can be configured to be accessed over the Internet by multiple users, with each user given a separate initial evaluation environment and thread of control. This configuration, developed by Franklyn Turbak and Dan Winship and called Museme, is a multi-user simulation environment (MUSE) similar to LambdaMOO [8].

3.3.3 WWW Evaluation Server

Scheme 48 can be configured to be run by a World-Wide Web server (`httpd`) in response to an appropriate request arriving over the Internet. This service aims to promote Scheme by giving the general network public a chance to experiment with it easily. A request contains a Scheme program and a Scheme expression, which execute on the server. The server replies with the printed representation of the result of evaluating the expression. The environment in which the programs and expressions run has been explicitly reduced from the default environment in order to limit the capabilities of programs, which, because they come from anyone on the Internet, shouldn't be trusted.

3.4 Security in Standard Scheme

Scheme 48 implements the Scheme standard [12]. In order to maintain security, however, users are given different instantiations of some of the built-in procedures, much as a conventional time-sharing system would provides users with different address spaces. If shared between agents, the following procedures would spell trouble for safe cooperation:

- Any built-in procedure that opens a file, such as `open-output-file`. A buggy or malevolent program could overwrite important files or read sensitive information from files.
- Anything that binds or accesses the current input and output ports, such as `display` with no explicit port argument. Use of such procedures would allow "spoofing" — misleading output that could be mistaken for valid messages coming from legitimate source such as an error handler or command processor.
- `load`, which not only accesses the file system, but also reads and writes a "current" interaction environment.

Multi-agent Scheme 48 systems need to ensure non-conflicting use of the external file system (assuming there is an external file system). Each agent has her own instantiations of file-opening procedures; the private versions implement file system access specific to that agent.

Because file opening procedures have access to an agent's files, a utility that opens files must be given not file *names,* but rather *access* to the files in the form of procedures, ports, or some appropriate data abstraction. This transition from name-oriented to value-oriented protection is exactly what is necessary in order to implement the principle of least privilege [26], which is also behind the analogous shift from Lisp 1.5's dynamic scoping to Scheme's lexical scoping.

As with files, one agent's `current-output-port` procedure is not allowed to access an output port that matters to another agent. Each agent has his own version of `current-output-port`, `display`, `write-char`, `read`, etc.

The `call-with-current-continuation` procedure raises a fundamental question: is the ability to invoke one's continuation multiple times a privilege that should be available to all agents? The answer isn't obvious; see Section 4.3.2.

The following features accepted by the Scheme report authors for inclusion in a Revised[5] report [20] are also troubling:

- `(interaction-environment)`. Given `eval`, this has the same problem as `load`.
- `dynamic-wind`. This operator is supposed to establish a dynamic context such that all entries into and exits from the context are guarded by specified actions. A mischievous program could initiate long-running or continuation-invoking computations in an unwind action, and/or set up an arbitrarily large number of nested `dynamic-wind`s, perhaps defeating time-sharing or mechanisms established for regaining control after an error or abort request.

4 Conclusion

The basic premise of this work is that program exchange and interaction are powerful modes of cooperation, and should be encouraged by a computational infrastructure that makes them as risk-free as possible. Program exchange and interaction are becoming easier and more common, thanks to advances in hardware infrastructure (increasing numbers of computers and growing network connectivity), but they are hindered because most computer systems provide little protection against the danger that might be inflicted by unknown programs.

The techniques proposed here to attain safety are:

1. Employ a security kernel with a simple but powerful semantics that allows fine-grained control over privileges.
2. Grant an invoked program only the privileges it needs to do its job. Rights to secondary storage and I/O devices shouldn't be implicitly inherited from the invoking agents' privilege set.
3. Certify that the program has passed a reliable safety test.
4. Label or "seal" objects so that they can be authenticated later.

None of these ideas is particularly new. Some can be derived from Saltzer and Schroeder's 1975 paper on protection in computer systems [26], which lays down desiderata for secure systems. (1) is their principle of economy of mechanism, (2) is the principle of least privilege, and (4) is the principle of separation of privilege.

Rather, the main novelty in the present work is the demonstration that a spare security kernel, derived from first principles and spanning the operating system / programming language gulf, can be simultaneously secure and practical.

4.1 Previous Work

4.1.1 Actors

The connection between lambda-calculus and message passing or "actor" semantics is well known [29]. Application in lambda-calculus corresponds to message passing in actor systems, with argument sequences playing the role of messages.

The actor languages [2] and Scheme were both developed as programming language frameworks, with little explicit attention to security concerns or cooperation between mutually suspicious agents. Security is of course implicit in the fact that an actor (procedure) can only be sent a message, not inspected, and that on a transfer of control only the argument sequence is transmitted, not any other part of the caller's environment.

There is no provision for authentication (actor recognition or abstract data types) either in the actor languages or in Scheme.

4.1.2 MUSEs

A "multi-user simulation environment" (MUSE, also MUD (multi-user dungeon) or MOO (MUD object-oriented)) simulates a world with a number of places (rooms) and inhabited by users (players or characters). A MUSE typically runs on a network and accepts connections from many users at once. Users move from place to place, communicate with each other, and manipulate simulated physical objects. Because objects may have value to users and users and their programs may attempt actions that can harm objects, and because of their generally open door policy, security is an important issue in MUSE design and administration.

Programming languages for MUSEs (in particular, that for LambdaMOO [8]) have the feature that a caller's privileges aren't implicitly passed on to a called program. This is certainly an improvement over the behavior of mainstream operating systems, which give all of the caller's privileges to the callee. The correct behavior is dictated by the demands of the environment: users encounter strange objects and do things to them; this causes invocation of programs belonging to the objects (or rather their creators).

MUSEs are generally based on ad hoc programming languages with many peculiar features that lead to security problems [3], inflexibility, or both. An overall lack of security in MUSEs is betrayed by the fact that it is a special privilege to be a "programmer." New members of a MUSE must convince an administrator that they are worthy before they are allowed to write new programs.

4.1.3 Oooz

Strandh's Oooz system [30] resembles Scheme 48 in aspiring to be a simple, multi-user, Scheme-based programming environment and operating system. The similarity ends there. Oooz extends Scheme with a hierarchical, globally accessible namespace and an object-oriented programming framework. The global namespace is analogous to a conventional file system, with access controlled by access control lists attached to objects. There is no obvious way in Oooz to implement abstract data types or authentication.

4.2 Discussion

4.2.1 Schroeder's Thesis

Schroeder studied the problem of cooperation between mutually suspicious principals in his 1972 dissertation [27]. He describes an extension to the Multics operating system in which caller and callee in a cross-domain call may protect resources from one another.

He was aware of the problem of protecting parameters from unauthorized transmission, but had nothing in particular to say about it:

> In some cases it may be desirable to guarantee that a protected subsystem invoked by another cannot remember and later divulge the values of input parameters. This problem appears to be very difficult to solve in a purely technical way and will not be considered in this thesis. ([27], page 16)

4.2.2 Operating System Security

Status quo operating systems (VMS, Unix, DOS Windows, etc.) don't effectively address safe invocation problems. When a program is invoked, it inherits all of the privileges of the invoker. The assumption is that every program that a user runs is absolutely trustworthy.

Safe invocation could be implemented in some versions of Unix, but it would require some heavy machinery. One way would be to have a privileged program that invokes a given program with the privileges of a specially unprivileged user. Such a program is not standard and cannot be created by an ordinary user. A different implementation would be to have a network server dedicated to the purpose, but the result would be strangely limited and awkward to use (file descriptors couldn't be passed, interrupts might act strangely, etc.).

Unix does have a facility whereby a program's owner can mark the program in such a way that when the program is invoked, it runs with the program's owner's privileges instead of with the invoker's (except for arguments). This sounds very similar to safe invocation of procedures, until one reads the fine print, which points out that such a program can obtain all of the invoker's privileges simply by doing a `setuid` system call.

Most common operating systems distinguish persistent objects (files) from volatile objects (in Unix, file descriptors). An `open` system call coerces a persistent object to a volatile one. Persistent objects have global names, while volatile objects have short numeric indexes that are interpreted locally to a running program. This "scoping" makes them resemble a procedure's successor links. Volatile objects can be passed from one program (process) to another that it invokes, but not in any other way.

Many operating system designs support secure program invocation in ways similar to what I describe, but these designs aren't deployed.

It is remarkable that we get by without secure cooperation. We do so only because people who use programs place such a high level of trust in the people who write those programs. The basic reason for this high level of trust is that computer systems are isolated from one another. Without communication, there can be no theft, since stolen goods must be communicated back to the thief. Any harm that arises is either vandalism or accident. Vandalism (e.g. the current epidemic of computer viruses) sophisticated enough to be untraceable is difficult to carry off and has little payoff for the perpetrator, while really harmful accidents (as when a commercial software product accidentally erases a disk) are mercifully rare.

4.3 Future Work

This section points out various shortcomings of W7 and Scheme 48, and attempts to suggest ways of fixing them. One of the dreams driving this work is to extend Scheme 48 to cover all operating system services, including device drivers. At that point it should be possible to dispense with the host operating system and run Scheme 48 stand-alone on a workstation, as it does on the mobile robots.

4.3.1 Preemption

A means to preempt a running process is necessary in any operating system. An agent, in invoking an unknown object, runs the risk that the invocation will be nonterminating; therefore the agent must have some way to request that the invocation be halted on some condition, such as receipt of special input from outside the processor (an abort key or button) or the passage of a predetermined amount of time.

Preemption is also desirable in that it is necessary and, together with support for first-class continuations or coroutines, sufficient for constructing a scheduler that simulates multiple hardware processing elements in software. One particularly elegant design for a timed preemption facility is a mechanism known as *engines*, of which Dybvig and Hieb have published a general implementation [11, 9]. Engines abstract over timed preemption by providing a way to run a computation for a specified amount of time. They are sufficient to construct a user-mode task scheduler.

Dybvig and Hieb's engine implementation has two problems for a system in which caller and callee are mutually suspicious:

1. Response time is sensitive to engine nesting depth; thus a malevolent callee could pile up a very deeply

nested sequence of engines, making response to an outer engine (which should have priority) arbitrarily sluggish.

2. No design is given specifying how engines should interact with waits and interrupts associated with concurrent activities (such as I/O).

If these problems can be solved, and I believe they can, then engines should serve well as part of W7.

(Scheme 48 supports a multitasking scheduler via a threads system, but threads are inferior to engines in several ways. Threads are less secure than engines, since one's share of the processor is proportional to how many threads you have; if you want to take over a processor, you need only create lots of threads. Scheme 48's threads give no reliable way to monitor the execution time of a supervised untrusted computation, since the computation can spawn new threads. And there is no way to limit or even monitor the amount of processor time allocated to a thread, since the threads system doesn't keep track of processor time used per thread.)

4.3.2 Continuations

First-class continuations are troublesome when caller and callee are mutually untrusting. With Scheme's `call-with-current-continuation` operator, the callee can obtain the continuation and invoke it twice. An unwary caller who has continuation code that performs a side effect is then vulnerable to having the side effect happen twice, which may be unexpected and undesired. Must all code that invokes untrusted objects be prepared for this possibility? To deal with the contingency requires code similar to the following:

```
(let ((returned? (new-cell)))
  (cell-set! returned? #f)
  (let ((result (untrusted arg ...)))
    (if (cell-ref returned?)
        (error "I didn't expect this")
        (begin (cell-set! returned? #t)
               (side-effect-to-be-done-
                         only-once!)
               ...))))
```

An unwillingly retained continuation is also undesirable from a resource allocation standpoint, since the continuation's creator might be penalized for tying down resources (space) consumed by the continuation.

4.3.3 Machine Programs

Supporting execution of compiler-generated machine code programs is important for two reasons: it would vastly improve performance relative to Scheme 48's current byte-code interpreter; and, by using existing compilers, Scheme 48 could be made to run useful programs written in a variety of languages (such as Pascal and C).

The main difficulty in supporting machine programs is ensuring that the kernel's security policy is followed; the machine program must not obtain access to any resources that it shouldn't have access to. This could very easily happen, since machine programs can construct arbitrary addresses and attempt to dereference them, or even construct arbitrary machine code sequences and jump to them. Any object that uses memory in the machine program's address space, or that is accessible via any sequence of machine instructions, is at risk.

There are several approaches to eliminating the risks associated with machine programs:

- Limitation: switch into a limited hardware-provided "user mode" when invoking the machine program. This is the approach used by most operating systems. In user mode, memory not belonging to the program is protected from access, and no hardware I/O is permitted. Some instructions, such as those that alter memory protection registers, are disabled. A protected environment is thus established, and all instructions are either permitted or actively prohibited.
- Verification: a trusted program scans the machine program to make sure that it doesn't do anything that it shouldn't. Unverified programs are rejected.
- Sandboxing: similar to verification, except that extra code is inserted around all troublesome instructions to dynamically ensure that security policy is respected [33]. Some programs may still be rejected, but fewer are than would be with verification.
- Trusted compilation: a program generated by a compiler may respect security policy by construction; if we trust the compiler, we will trust its output.

The type and array bounds safety of many compilers, such as many Scheme and Pascal compilers, are sufficient to guarantee that the kernel's security policy is respected. Such compilers may be used without change.

Limitation may be the only option if one has little influence over the compiler and program being compiled. For example, most C programs use pointers in ways that are difficult to guarantee safe by verification or sandboxing, and few C compilers generate object code that checks validity of pointers. Limitation is to be avoided because transfers into and out of user mode are expensive in many hardware architectures. On the other hand, for programs that do few control transfers to other programs, use of memory protection hardware may be the most efficient way to detect invalid pointers and array indexes (that is, to implement kernel security policy).

An important part of the implementation of machine program support is the interface between machine programs and the kernel. That is, how does a machine program perform a kernel operation, such as creating or invoking a procedure, and how are the operands — generally object references — specified? This will be answered differently depending on the extent to which the machine program can be trusted.

When the machine program is trusted, object references can be represented as pointers to data structures representing objects, and kernel operations can be implemented either as subroutine calls (e.g. object creation can be accomplished by a call to an allocation routine) or as code sequences occurring directly in the program (e.g. invocation of a W7 procedure might be compiled similarly to local procedure call).

When the machine program is untrusted, transfers of control outside of the program must be generally accomplished by a trap or system call. (A few hardware architectures support general calls across protection domains). Object references must be amenable to validation on use, since the program may present an arbitrary bit pattern as a putative object reference. The method used by most operating systems in similar circumstances is to associate, with each activation of an untrusted machine program, a table mapping small integer indexes to objects. (In Unix, the indexes are known as "file descriptors.") The program presents an index, which the kernel interprets as specifying the object at that position in the table. Validation consists of a simple range check to determine that the index is within the table.

Alternatively, objects may be given unique names (perhaps their addresses in memory), and these names can be used by untrusted machine programs. When the program presents a name in a kernel operation, the kernel validates the name by determining whether it occurs in an object table specific to the program's activation (as above). This approach is similar to the "capability-based" approach to protection [15]. For the W7 kernel, there is little reason to prefer this over the small-index approach, since table searches and unique name maintenance are likely to be complicated and inefficient relative to use of indexes, which can be determined when a program is compiled or installed.

4.3.4 Reflection

Reflection is reasoning about self [28]. In a computing system, the concept of reflection comprehends examining the internal structure of objects and continuations for purposes of debugging, analysis, or optimization. Reflection interacts with security issues in ways that to my knowledge have not been researched, much less resolved.

For example, Scheme 48's debugger has access to special reflective operators that break all protection boundaries; it can examine the internals of procedures, records, and continuations. This is very useful, but unfortunately the debugger is egregiously insecure, since it allows any user access to any value transitively accessible from an accessible object. It is easy enough to achieve security in Scheme 48 by disabling the debugger, but a better solution would be for the kernel to provide a simple, safe way to examine continuations and procedures, so that an unprivileged debugger could be built. I believe that this can be done in such a way that a user is able to see what she ought to be entitled to see, but nothing else. In particular, a user should be able to see any of the information that exists in object representations, as long as that information might have been available to her had she included extra debugging hooks in her own programs.

4.3.5 Other Issues

Persistence. Some support is needed for keeping objects in secondary storage efficiently, and there should be some guarantees about which objects will necessarily survive crashes. Currently Scheme 48 relies on an external file system to store information persistently; this is not integrated with the internal protection system beyond the fact that access to the file system as a whole can be limited.

Quotas. Limits should be imposed on the amount of memory agents should be allowed to use. Without this, a malevolent or buggy program can consume all available space, making the system unuseable. Memory limitation is more difficult than execution time limitation; the allowed space decreases as memory is allocated, but must increase as memory is reclaimed by garbage collection. With each object, then, must be associated an "account" to be credited when the object is reclaimed. This could be done either by putting an extra field in each object indicating the account, or by maintaining distinct regions of memory for different accounts. The latter is reminiscent of multi-stage garbage collection , and could perhaps be unified with it.

Accountability. When something goes wrong, it's nice to know who is responsible, if not exactly how it went wrong. But responsibility is difficult to assign in cooperative enterprises. In particular, if something goes wrong when one agent invokes another agent's program, who is responsible? If a server receives a request from a client, is it the server or the client who is responsible for ensuring that the request is a reasonable and safe one?

Revocation. There is no way in W7 or Scheme 48 to revoke access to an object. Objects simulating revokable links could be defined using cells, but it's not obvious that this would be sufficient. One would have to decide when giving out access to an object. The times when one would want to revoke a link might be precisely the times when one didn't anticipate that one would want to.

Distribution. A network of encapsulated objects lends itself to distribution over a network of computer systems. One major component of a distributed operating system that is missing from W7/Scheme 48 is a remote invocation mechanism (classically called RPC, or remote procedure call). Such a mechanism should have the following properties:

- Objects can be passed as arguments and returned as results. This requires automatic creation of "stub" or delegate objects that forward calls over the network.
- Calls are properly tail recursive: in a call from node A to node B, B can make a call to node C specifying that the result is to be sent to A, not B.
- Exceptions are distributed transparently, in that an exception on one machine can find the correct exception handler even if it's on another machine.

Alan Bawden has described and implemented something relevant in his PhD thesis [4].

4.4 Contributions

The success of Scheme 48 shows that a spare security kernel can provide flexible and solid security without becoming difficult to use or to program.

Is W7 simpler than all other security kernels? Much of the complexity in operating system kernels arises from performance concerns that W7 has yet to address, so a rational comparison is difficult. It is difficult to see how it could be smaller, since all of the services it provides are (as is shown) necessary: program (object) creation and invocation; object marking and authentication; primitive access to I/O devices.

There are some minimalists in the Scheme community who believe that procedures, perhaps together with some primitive data types such as symbols and cells, serve as a basis for the construction of all other useful programming constructs. I hope that the discussion of authentication and abstract data types (Section 2.3.3) will be seen to refute this position and to show the need for a built-in authentication mechanism in minimal programming languages.

The ease with which Trojan horses and viruses may infiltrate computer systems is appalling, as is the extent to which users must blindly trust software provided by vendors. I hope I have contributed a bit to the currently unpopular cause of principled solutions to security problems such as these.

I hope the document helps to break down the artificial distinction between programming languages and operating systems. Progress in both of these areas is hindered by a failure to recognize that the concerns of both are fundamentally the same. What is needed is operating systems that provide services that are useful to languages, and languages that give convenient access to operating system services.

Acknowledgements

This report is extracted from a dissertation [21], of which the complete acknowledgements section is reproduced here.

Joseph Weizenbaum, for generously allowing me the use of his office during the long final stretch.

Gerry Sussman of 4AI, Bruce Donald of Cornell, and Gregor Kiczales of Xerox PARC, for moral and financial support, guidance, and ideas.

Hal Abelson and Tom Knight, for serving on my committee.

Richard Kelsey, for hard work and heroic deeds that helped to bring Scheme 48 to life.

Norman Adams and Will Clinger, for productive collaborations.

Alan Bawden, David Espinosa, Dave Gifford, Philip Greenspun, Ian Horswill, Kleanthes Koniaris, Dave McAllester, Jim O'Toole, Brian Reistad, Bill Rozas, Mark Sheldon, Olin Shivers, Franklyn Turbak, and many others at Tech Square, for making life at MIT not just tolerable but interesting and even amusing.

Russell Brown, Jim Jennings, Daniela Rus, and others at the Cornell Computer Science Robotics and Vision Laboratory, for their good humor and courageous robotry.

Kathleen Akins, Mike Dixon, John Lamping, Brian Smith, and others at Xerox PARC, for creating an environment in which peculiar ideas flourish.

Butler Lampson, for his dazzlingly clear explanations.

Hal Abelson, Oakley Hoerth, Frans Kaashoek, Tom Knight, Mark Sheldon, Olin Shivers, Gerry Sussman, and Franklyn Turbak, for their comments and suggestions on drafts of the document.

Marilyn Pierce, for making the administrative nightmares go away.

Rebecca Bisbee, for making everything work.

Albert Meyer, for urging expedience.

Tom Collett, for giving me a reason to finish.

Laura Burns, Katy Heine, Kris Jacobson, and Geoff Linburn, for their generosity and emotional support in strange times.

ARPA and NSF, for grants to the projects that employed me.

Others too numerous to mention.

Finally, Oakley Hoerth, for the diverse array of model arthropods: glow-in-the-dark cicada, wind-up beetle, pop-up and puppet ants, cricket stickers, and many others.

References

[1] Norman I. Adams IV and Jonathan A. Rees.
Object-oriented programming in Scheme.
Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, pages 277–288.

[2] Gul A. Agha.
Actors: A Model of Concurrent Computation in Distributed Systems.
MIT Press, Cambridge, MA, 1986.

[3] David B. Albert.
Issues in MUSE security.
Manuscript (?), 1994.

[4] Alan Bawden.
Linear Graph Reduction: Confronting the Cost of Naming.
PhD thesis, MIT, 1992.

[5] Nathaniel Borenstein and Marshall T. Rose.
MIME extensions for mail-enabled applications: application/Safe-TCL and multipart/enabled-mail.
Working draft, 1993.
Internet:
`ftp://ftp.fv.com/pub/code/other/safe-tcl.tar`.

[6] William Clinger and Jonathan Rees.
Macros that work.
Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, pages 155–162, January 1991.

[7] William Clinger and Jonathan Rees (editors).
Revised^4 report on the algorithmic language Scheme.
LISP Pointers IV(3):1–55, July–September 1991.

[8] Pavel Curtis.
LambdaMOO programmer's manual for LambdaMOO version 1.7.6.
Xerox Corp., `ftp://parcftp.xerox.com/pub/MOO/ProgrammersManual.txt`, 1993.

[9] R. Kent Dybvig and Robert Hieb.
Engines from Continuations.
Computer Languages 14(2):109–123, 1989.

[10] Adele Goldberg and David Robson.
Smalltalk-80: The Language and its Implementation.
Addison-Wesley, 1983.

[11] Christopher P. Haynes and Daniel P. Friedman.
Engines build process abstractions.
Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, pages 18–24.

[12] IEEE Std 1178-1990.
IEEE Standard for the Scheme Programming Language.
Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[13] Richard Kelsey and Jonathan Rees.
A tractable Scheme implementation.
Lisp and Symbolic Computation 7(4):315–335, 1995 (to appear).

[14] Butler W. Lampson.
A note on the confinement problem.
CACM 16(10):613–615, 1973.

[15] Henry M. Levy.
Capability-based Computer Systems.
Bedford, MA: Digital Press, 1984.

[16] Robin Milner, Mads Tofte, and Robert Harper.
The Definition of Standard ML.
MIT Press, 1990.

[17] David Moon.
Genera retrospective.
International Workshop on Object-Oriented Operating Systems, 1991.

[18] James H. Morris Jr.
Protection in Programming Languages.
CACM 16(1):15–21, 1973.

[19] Jonathan Rees.
Another module system for Scheme.
In `ftp://ftp-swiss.ai.mit.edu/pub/s48/scheme48-0.36.tar.gz`, March 1994.

[20] Jonathan A. Rees.
The June 1992 Meeting.
Lisp Pointers V(4):40–45, October–December 1992.

[21] Jonathan A. Rees.
A Security Kernel Based on the Lambda-Calculus.
Ph.D. dissertation, MIT, 1995.

[22] Jonathan A. Rees and Norman I. Adams.
T: A dialect of Lisp or, LAMBDA: the ultimate software tool.
In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming,* pages 114–122.

[23] Jonathan Rees and Bruce Donald.
Program mobile robots in Scheme.
Proceedings of the 1992 IEEE International Conference on Robotics and Automation, pages 2681–2688.

[24] R. L. Rivest, A. Shamir, and L. Adleman.
A method for obtaining digital signatures and public-key cryptosystems.
Communication of the ACM 21:120–126, Feb. 1978.

[25] Guillermo J. Rozas.
Transluscent Procedures, Abstraction Without Opacity.
PhD thesis, MIT, May 1993.

[26] Jerome H. Saltzer and M. D. Schroeder.
The protection of information in computer systems.
Proceedings of the IEEE 63(9):1278–1308, 1975.

[27] Michael D. Schroeder.
Cooperation of Mutually Suspicious Subsystems in a Computer Utility.
Ph.D. thesis, MIT Project MAC TR-104, 1972.

[28] Brian C. Smith.
Reflection and Semantics in LISP.
In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Programming Languages,* pages 23–35, Salt Lake City, January 1986.

[29] Guy Lewis Steele Jr. and Gerald Jay Sussman.
Lambda: the ultimate imperative.
MIT AI Memo 353, 1976.

[30] Robert Strandh.
Oooz, a multi-user programming environment based on Scheme.
BIGRE Bulletin 65, IRISA, Campus de Beaulieu, F-35042, Rennex Cédex, France, July 1989.

[31] Gerald Jay Sussman and Guy Lewis Steele Jr.
Scheme: an interpreter for extended lambda calculus.
MIT AI Memo 349, 1975.

[32] Swinehart et al.
Cedar.
TOPLAS 4(8), 1986.

[33] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham.
Efficient software-based fault isolation.
Proceedings of the Fourteenth ACM Symposium on Operating System Principles, pages 203–216, December 1993.

[34] William A. Wulf, Roy Levin, and Samuel P. Harbison.
HYDRA/C.mmp: An Experimental Computer System.
McGraw-Hill, 1981.