Paradigm Regained:
Abstraction Mechanisms for Access Control

Mark S. Miller
Hewlett Packard Laboratories,
Johns Hopkins University
markm@caplet.com

Jonathan S. Shapiro
Johns Hopkins University
shap@cs.jhu.edu

Abstract. Access control systems must be evaluated in part on how well they enable one to distribute the access rights needed for cooperation, while simultaneously limiting the propagation of rights which would create vulnerabilities. Analysis to date implicitly assumes access is controlled only by manipulating a system's protection state—the arrangement of the access graph. Because of the limitations of this analysis, capability systems have been "proven" unable to enforce some basic policies: revocation, confinement, and the *-properties (explained in the text).

In actual practice, programmers build *access abstractions*—programs that help control access, extending the kinds of access control that can be expressed. Working in Dennis and van Horn's original capability model, we show how abstractions were used in actual capability systems to enforce the above policies. These simple, often tractable programs limited the rights of arbitrarily complex, untrusted programs. When analysis includes the possibility of access abstractions, as it must, the original capability model is shown to be stronger than is commonly supposed.

1. Introduction

We live in a world of insecure computing. Viruses regularly roam our networks causing damage. By exploiting a single bug in an ordinary server or application, an attacker may compromise a whole system. Bugs tend to grow with code size, so vulnerability usually *increases* over the life of a given software system. Lacking a readily available solution, users have turned to the perpetual stopgaps of virus checkers and firewalls. These stopgaps cannot solve the problem—they provide the defender no fundamental advantage over the attacker.

In large measure, these problems are failures of access control. All widely-deployed operating systems today—including Windows, UNIX variants, Macintosh, and PalmOS—routinely allow programs to execute with excessive and largely unnecessary authority. For example, when you run Solitaire, it needs only to render into its window, receive UI events, and perhaps save a game state to a file you specify. Under the *Principle of Least*

Authority (POLA—closely related to the Principle of Least Privilege [Saltzer75]), it would be limited to exactly these rights. Instead, today, it runs with all of your authority. It can scan your email for interesting tidbits and sell them on eBay to the highest bidder; all the while playing only within the rules of your system. Because applications are run with such excessive authority, they serve as powerful platforms from which viruses and human attackers penetrate systems and compromise data. The flaws exploited are not bugs in the usual sense. Each operating system is functioning as specified, and each specification is a valid embodiment of its access control paradigm. The flaws lie in the access control paradigm.

By *access control paradigm* we mean an access control model plus a way of thinking —a sense of what the model means, or could mean, to its practitioners, and of how its elements should be used.

For purposes of analysis, we pick a frame of reference—a boundary between a *base* system (e.g., a "kernel" or "TCB") creating the rules of permissible action, and programs running on that base, able to act only in permitted ways. In this paper, "program" refers only to programs running on the base, whose access is controlled by its rules.

Whether to enable cooperation or to limit vulnerability, we care about *authority* rather than *permissions*. Permissions determine what actions an individual program may perform on objects it can directly access. Authority describes effects a program may cause on objects it can access, either directly by permission, or indirectly by permitted interactions with other programs. To understand authority, we must reason about the interaction of program behavior and the arrangement of permissions. While Dennis and van Horn's 1966 paper, *Programming Semantics for Multiprogrammed Computations* [Dennis66] clearly suggested both the need and a basis for a unified semantic view of permissions and program behavior, we are unaware of any formal analysis pursuing this approach in the security, programming language, or operating system literature.

Over the last 30 years, the formal security literature has reasoned about bounds on authority exclusively from the evolution of state in protection graphs—the arrangement of permissions. This implicitly assumes all programs are hostile. While conservatively safe, this approach omits consideration of security enforcing programs. Like the access it controls, security policy emerges from the interaction between the behavior of programs and the underlying protection primitives. This omission has resulted in false negatives—mistaken infeasibility results—diverting attention from the possibility that an effective access control model has existed for 37 years.

In this paper, we offer a new look at the original capability model proposed by Dennis and van Horn [Dennis66]—here called *object-capabilities*. Our emphasis—which was also their emphasis—is on expressing policy by using abstraction to extend the expressiveness of object-capabilities. Using abstraction, object-capability practitioners have solved problems like revocation (withdrawing previously granted rights), overt confine-

ment (cooperatively isolating an untrusted subsystem),[1] and the *-properties (enabling one-way communication between clearance levels). We show the logic of these solutions, using only functionality available in Dennis and van Horn's 1966 Supervisor, hereafter referred to as "DVH." In the process, we show that many policies that have been "proven" impossible are in fact straightforward.

The balance of this paper proceeds as follows. In "Terminology and Distinctions", we explain our distinction between *permission* and *authority*. In "How Much Authority Does 'cp' Need?", we use a pair of Unix shell examples to contrast two paradigms of access control. In "The Object-Capability Paradigm", we explain the relationship between the object-capability paradigm and the object paradigm. We introduce the object-capability language ***E***, which we use to show access control abstractions. In "Confinement", we show how object-capability systems confine *programs* rather than *uncontrolled subjects*. We show how confinement enables a further pattern of abstraction, which we use to implement the *-properties.

2. Terminology and Distinctions

A *direct access right* to an *object* gives a *subject permission* to *invoke* the *behavior* of that object. Here, Alice has direct access to `/etc/passwd`, so she has permission to invoke any of its operations. She *accesses* the object, *invoking* its `read()` operation.

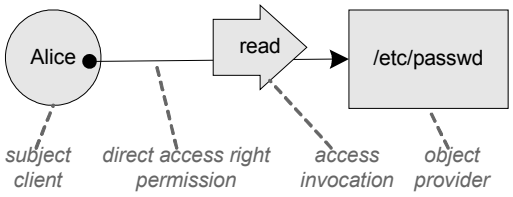

Fig 1. *Access diagrams* depict protection state.

By *subject* we mean the finest-grain unit of computation on a given system that may be given distinct direct access rights. Depending on the system, this could be anything from: all processes run by a given user account, all processes running a given program, an individual process, all instances of a given class, or an individual instance. To encourage anthropomorphism we use human names for subjects.

By *object*, we mean the finest-grain unit to which separate direct access rights may be provided, such as a file, a memory page, or another subject, depending on the system. Without loss of generality, we model restricted access to an object, such as read-only access to `/etc/passwd`, as simple access to another object whose behavior embodies the restriction, such as access to the read-only facet of `/etc/passwd` which responds only to queries.

[1] Semantic models, specifications, and correct programs deal only in *overt* causation. Since this paper examines only models, not implementations, we ignore covert and side channels. In this paper, except where noted, the "overt" qualifier should be assumed.

Any discussion of access must carefully distinguish between *permission* and *authority* (adapted from Bishop and Snyder's distinction between *de jure* and *de facto* transfer [Bishop79]). Alice can directly read `/etc/passwd` by calling `read(…)` when the system's protection state says she has adequate *permission*. Bob (unshown), who does not have permission, can indirectly read `/etc/passwd` so long as Alice sends him copies of the text. When Alice and Bob arrange this relying only on the "legal" overt rules of the system, we say Alice is providing Bob with an *indirect access right* to read `/etc/passwd`, that she is acting as his proxy, and that Bob thereby has *authority* to read it. Bob's authority derives from the arrangement of permissions (Alice's read permission, Alice's permission to talk to Bob), and from the behavior of subjects and objects on permitted causal pathways (Alice's proxying behavior). The thin black arrows in our access diagrams depict permissions. We will explain the resulting authority relationships in the text.

The protection state of a system is the arrangement of permissions at some instant in time, i.e., the topology of the access graph. Whether Bob currently has permission to access `/etc/passwd` depends only on the current arrangements of permissions. Whether Bob eventually gains permission depends on this arrangement and on the state and behavior of all subjects and objects that might cause Bob to be granted permission. We cannot generally predict if Bob will gain this permission, but a conservative bound can give us a reliable "no" or "maybe".

From a given system's *update rules*—rules governing permission to alter permissions—one might be able to calculate a bound on possible future arrangements by reasoning only from the current arrangement.[2] This corresponds to Bishop and Snyder's potential *de jure* analysis, and gives us an *arrangement-only bound on permission*. With more knowledge, one can set tighter bounds. By taking the state and behavior of some subjects and objects into account, we may calculate a tighter *partially behavioral bound on permission*.

Bob's eventual authority to `/etc/passwd` depends on the arrangement of permissions, and on the state and behavior of all subjects and objects on permitted causal pathways between Bob and `/etc/passwd`. One can derive a bound on possible overt causality by reasoning only from the current arrangement of permissions. This corresponds to Bishop and Snyder's potential *de facto* analysis, and gives us an *arrangement-only bound on authority*. Likewise, by taking some state and behavior into account, we may calculate a tighter *partially behavioral bound on authority*.

Systems have many levels of abstraction. At any moment our frame of reference is a boundary between a *base* system that creates rules and the subjects hosted on that base, restricted to play by those rules. By definition, a base system manipulates only permis-

---

[2] The Harrison Ruzzo Ullman paper [Harrison76] is often misunderstood to say this calculation is never decidable. HRU actually says it is possible (indeed, depressingly easy) to design a set of update rules which are undecidable. At least three protection systems have been shown to be decidably safe [Jones76, Shapiro00, Motwani00].

sions. Subjects extend the expressiveness of a base system by building abstractions whose behavior further limits the authority it provides to others. Taking this behavior into account, one can calculate usefully tighter bounds on authority. As our description ascends levels of abstraction [Neumann80], the authority manipulated by the extensions of one level becomes the permissions manipulated by the primitives of the next higher base. Permission is relative to a frame of reference. Authority is invariant.

It is unclear whether Saltzer and Schroeder's *Principle of Least Privilege* is best interpreted as least permission or least authority. As we will see, there is an enormous difference between the two.

3. How Much Authority Does "cp" Need?

Consider how the following Unix shell command works:

```
$ cp foo.txt bar.txt
```

Here, your shell passes to the `cp` program the two strings `"foo.txt"` and `"bar.txt"`. By these strings, you mean particular files in your file system—your namespace of files. In order for `cp` to open the files you name, it must use your namespace, and it must be able to read and write any file you might name that you can read and write. Not only does `cp` operate with all your authority, it must. Given this way of using names, `cp`'s *least authority* still includes all of your authority to the file system. So long as we normally install and run applications in this manner, both security and reliability are hopeless.

By contrast, consider:

```
$ cat < foo.txt > bar.txt
```

This shell command brings about the same end effect. Although `cat` also runs with all your authority, for this example at least, it does not need to. As with function calls in any lexically scoped language (even FORTRAN), the names used to designate arguments are evaluated in the caller's namespace prior to the call (here, by opening files). The callee gets direct access to the first-class anonymous objects passed in, and designates them with parameter "names" bound in its own private name space (here, file descriptor numbers). The file descriptors are granted only to this individual process, so only this process can use these file descriptors to access these files. In this case, the two file descriptors passed in are all the authority `cat` needs to perform this request.

Today's widely deployed systems use both styles of access control. They grant permission to open a file on a per-account basis, creating the pools of authority on which viruses grow. These same systems flexibly grant access to a file descriptor on a per-process basis. Ironically, only their support of the first style is explained *as* their access control system. Object-capability systems differ from current systems more by the elimination of the first style than by the elaboration of the second.

If support for the first style were eliminated and `cat` ran with access *only* to the file descriptors passed in, it could still do its job, and we could more easily reason about our vulnerabilities to its malice or bugs. In our experience of object-capability programming, these radical reductions of authority and vulnerability mostly happen naturally.

4. The Object-Capability Paradigm

In the object model of computation [Goldberg76, Hewitt73], there is no distinction between subjects and objects. A non-primitive object, or *instance*, is a combination of code and state, where *state* is a mutable collection of *references* to objects. The *computational system* is the dynamic reference graph of objects. Objects—behaving according to their code—interact by sending messages on references. Messages carry references as arguments, thereby changing the connectivity of the reference graph.

Matter tells space how to curve.
Space tells matter how to move.
—John Archibald Wheeler

The *object-capability* model uses the reference graph *as* the access graph, requiring that objects can interact *only* by sending messages on references. To get from objects to object-capabilities we need merely prohibit certain primitive abilities which are not part of the object model anyway, but which the object model by itself doesn't require us to prohibit—such as forged pointers, direct access to another's private state, and mutable static state [Kahn88, Rees96, Miller00]. For example, C++, with its ability to cast integers into pointers, is still within the object model but not the object-capability model. Smalltalk and Java fall outside the object-capability model because their mutable static variables enable objects to interact outside the reference graph.

Whereas the *functionality* of an object program depends only on the abilities provided by its underlying system, the *security* of an object-capability program depends on underlying inabilities as well. In a graph of mutually suspicious objects, one object's correctness depends not only on what the rules of the game say it can do, but also on what the rules say its potential adversaries cannot do.

4.1. The Object-Capability Model

The following model is an idealization of various object languages and object-capability operating systems. All its access control abilities are present in DVH (Dennis and van Horn's Supervisor) and most other object-capability systems.[3] Object-capability systems differ regarding concurrency control, storage management, equality, typing, and the primitiveness of messages, so we avoid these issues in our model. Our model does as-

[3] Our object-capability model is essentially the untyped lambda calculus with applicative-order local side effects and a restricted form of `eval`—the model Actors and Scheme are based on. This correspondence of objects, lambda calculus, and capabilities was noticed several times by 1973 [Goldberg76, Hewitt73, Morris73], and investigated explicitly in [Tribble95, Rees96].

Model Term	Capability OS Terms	Object Language Terms
instance	process, domain	instance, closure
code	non-kernel program + literal data	lambda expression, class file, method table
state	address space + c-list (capability list)	environment, instance variable frame
index	virtual memory address, c-list index	lexical name, variable offset, argument position
loader	domain creator, `exec`	`eval`, ClassLoader

Table 1: Object / Capability Corresponding Concepts

sume reusable references, so it may not fit object-capability systems based on concurrent logic/constraint programming [Miller87, Kahn88, Roy02]. However, our examples may easily be adapted to any object-capability system despite these differences.

The static state of the reference graph is composed of the following elements.

- An object is either a *primitive* or an *instance*. Later, we explain three kinds of primitives: *data*, *devices*, and a *loader*. Data is immutable.
- An *instance* is a combination of *code* and *state*. We say it is an *instance of* the behavior described by its code. For example, we say an operating system process is an instance of its program.
- An instance's *state* is a mutable map from *indexes* to *references*.
- A *reference* provides access to an object, indivisibly combining designation of the object, the permission to access it, and the means to access it. The permission arrows on our access diagrams now depict references.
- A *capability* is a reference to non-data.
- *Code* is some form of data (such as instructions) used to describe an instance's behavior to a loader, as explained below. Code also contains literal data.
- Code describes how a *receiving instance* (or "self") reacts to an *incoming message*.
- While an instance is reacting, its *addressable references* are those in the incoming message, in the receiving instance's state, and in the literal data of the receiving instance's code. The *directly accessible objects* are those designated by addressable references.
- An *index* is some form of data used by code to indicate which addressable reference to use, or where in the receiving instance's state to store an addressable reference. Depending on the system, an index into state may be an instance variable name or offset, a virtual memory address, or a capability-list index (a c-list index, like a file descriptor number). An index into a message may be an argument position, argument keyword, or parameter name.

Message passing and object creation dynamically change the graph's connectivity.

In the initial conditions of Figure 2, Bob and Carol are *directly accessible* to Alice. When Alice sends Bob the message "`foo(carol)`", she is both accessing Bob and permitting Bob to access Carol.

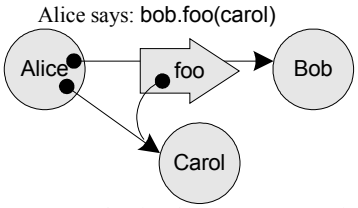

Fig 2: Introduction by Message Passing

Alice can cause effects on the world outside herself only by sending messages to objects directly accessible to her (Bob), where she may include, at distinct argument indexes, references to objects directly accessible to her (Carol). We model a call-return pattern as two messages. For example, Alice gains information from Bob by causing Bob (with a query) to cause her to be informed (with a return).

Bob is affected by the world outside himself only by the arrival of messages sent by those with access to him. On arrival, the arguments of the message (Carol) become directly accessible to Bob. Within the limits set by these rules, and by what Bob may feasibly know or compute, Bob reacts to an incoming message only according to his code. All computation happens only in reaction to messages.

We distinguish three kinds of primitive objects.

- *Data* objects, such as the number 3. Access to these are knowledge limited rather than permission limited. If Alice can figure out which integer she wants, whether 3 or your private key, she can have it. Data provides only information, not access. Because data is immutable, we need not distinguish between a reference to data and the data itself. (In an OS context, we model user-mode compute instructions as data operations.)
- *Devices*. For purposes of analysis we divide the world into a *computational system* containing all objects of potential interest, and an *external world*. On the boundary are primitive *devices*, causally connected to the external world by unexplained means. A non-device object can only affect the external world by sending a message to an accessible output device. A non-device object can only be affected by the external world by receiving a message from an input device that has access to it.
- A *loader* makes new instances. The creation request to a loader has two arguments: code describing the behavior of the new instance, and an index => reference map providing *all* the instance's initial state. A loader must ensure (whether by code verification or hardware protection) that the instance's behavior cannot violate the rules of our model. A loader returns the *only* reference to the new instance. (Below, we use a loader to model nested lambda evaluation.) As a safe simplification of actual systems, this paper assumes a single, immutable, universally accessible loader.

By these rules, *only connectivity begets connectivity*—all access must derive from previous access. Two disjoint subgraphs cannot become connected as no one can introduce them. Arrangement-based analysis of bounds on permission proceeds by graph

reachability arguments. Overt causation, carried only by messages, flows only along permitted pathways, so we may again use reachability arguments to reason about bounds on authority and causality. The transparency of garbage collection relies on such arguments.

The object-capability model recognizes the security properties latent in the object model. All the restrictions above are consistent with good object programming practice even when security is of no concern.

4.2. A Taste of *E*

To illustrate how the object-capability model is used to solve access control problems, we use a subset of the ***E*** language as our notation. This subset directly embodies our object-capability model. All the functionality it provides is present in DVH. Full ***E*** extends the capability paradigm beyond the model presented above. Using a cryptographic capability protocol among mutually suspicious machines, ***E*** creates a distributed persistent reference graph, supporting decentralized access control with somewhat weaker properties than are possible within a single machine. These issues are beyond the scope of this paper. For the rest of this paper, "***E***" refers to our subset of ***E***.

In ***E***, an instance is a single-parameter closure instantiated by lambda evaluation. The single (implicit) parameter is for the incoming message. A message send applies an object-as-closure to a message-as-argument. ***E*** combines Scheme-like semantics [Kelsey98] with syntax for message sending, method dispatch, and soft type checking, explained below. Here is a simple data abstraction.

```
def pointMaker {
    to make(x :int, y :int) :any {
        def point {
            to getX() :int { return x }
            to getY() :int { return y }
            to add(otherPt) :any {
                return pointMaker.make(x.add(otherPt.getX()),
                                       y.add(otherPt.getY()))
            }
        }
        return point
    }
}
```

The expressions defining `pointMaker` and `point` are object definitions—both a lambda expression and a variable definition. An object definition evaluates to a closure whose behavior is described by its body, and it defines a variable (shown in bold italics) to hold this value. The body consists of `to` clauses defining methods, followed by an optional `match` clause as we will see later. Because an object is always applied to a message, the message parameter is implicit, as is the dispatch on the message name to select a method. The `pointMaker` has a single method, `make`, that defines and returns a new `point`.

Method definitions (shown in bold) and variable definitions (shown in italics) can have an optional soft type declaration [Cartwright91], shown as a ":" followed by a guard expression. A guard determines which values may pass. The any guard used above allows any value to pass as is.

The nested definition of point uses x and y freely. These are its instance variables, and together form its state. The state maps from indexes "x" and "y" to the associated values from point's creation context.

Using the loader described above, we can transform the above code to

```
def pointMaker {
    to make(x :int, y :int) :any {
        def point := loader.load("def point {…}",
                                 ["x" => x, "y" => y])
        return point
    }
}
```

Rather than a source string, a realistic loader would accept some form of separately compiled code.

The expression ["x" => x, "y" => y] builds a map of index => reference associations. All "linking" happens only by virtue of these associations—only connectivity begets connectivity.

Applying this transformation recursively would unnest all object definitions. Nested object definitions better explain instantiation in object languages. The loader better explains process or domain creation in operating systems. In ***E***, we almost always use object definitions, but we use the loader below to achieve confinement.

4.3. Revocation: Redell's 1974 Caretaker Pattern

> *Capability systems modeled as unforgeable references present the other extreme, where delegation is trivial, and revocation is infeasible.*
>
> —Chander, Dean, Mitchell [Chander01]

When Alice says bob.foo(carol), she gives Bob unconditional, full, and perpetual access to Carol. Given the purpose of Alice's message to Bob, such access may dangerously exceed least authority. In order to practice POLA, Alice might need to somehow restrict the rights she grants to Bob. For example, she might want to ensure she can revoke access at a later time. But in a capability system, capabilities themselves are the only representation of permission, and they provide only unconditional, full, perpetual access to the objects they designate.

What is Alice to do? She can use (a slight simplification of) Redell's Caretaker pattern for revoking access [Redell74], shown here using additional elements of ***E*** we explain below.

```
def caretakerMaker {
    to make(var target) :any {
        def caretaker {
            match [verb :String, args :any[]] {
                E.call(target, verb, args)
            }
        }
        def revoker {
            to revoke() :void {
                target := null
            }
        }
        return [caretaker, revoker]
    }
}
```

Instead of saying "`bob.foo(carol)`", Alice can instead say:

```
def [carol2, carol2Rvkr] := caretakerMaker.make(carol)
bob.foo(carol2)
```

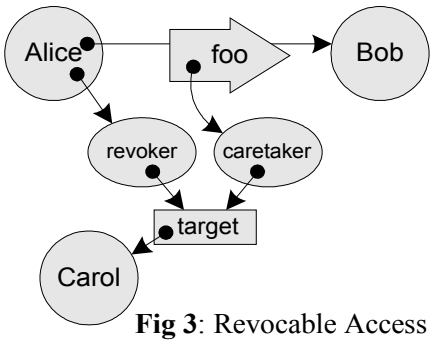

Fig 3: Revocable Access

The Caretaker `carol2` transparently forwards messages it receives to `target`'s current value. The Revoker `carol2Rvkr` changes what that current value is. Alice can later revoke the effect of her grant to Bob by saying "`carol2Rvkr.revoke()`".

Variables in ***E*** are non-assignable by default. The `var` keyword means `target` can be assigned. (`var` is the opposite of Java's `final`.) Within the scope of `target`'s definition, `make` defines two objects, `caretaker` and `revoker`, and returns them to its caller in a two element list. Alice receives this pair, defines `carol2` to be the new Caretaker, and defines `carol2Rvkr` to be the corresponding Revoker. Both objects use `target` freely, so they both share access to the same assignable `target` variable (which is therefore a separate object).

What happens when Bob invokes `carol2`, thinking he's invoking the kind of thing Carol is? An object definition contains methods and an optional `match` clause defining a matcher. If an incoming message (`x.add(3)`) doesn't match any of the methods, it is given to the matcher. The `verb` parameter is bound to the message name (`"add"`) and the `args` to the argument list (`[3]`). This allows messages to be received generically without prior knowledge of their API, much like Smalltalk's `doesNotUnderstand:` or Java's `Proxy`. Messages are sent generically using "`E.call(…)`", much like Smalltalk's `perform:`, Java's "reflection", or Scheme's `apply`.

This Caretaker[4] provides a temporal restriction of authority. Similar patterns provide other restrictions, such as *filtering facets* that let only certain messages through. Even in systems not designed to support access abstraction, many simple patterns happen naturally. Under Unix, Alice might provide a filtering facet as a process reading a socket Bob can write. The facet process would access Carol using Alice's permissions.

4.4. Analysis and Blind Spots

Given Redell's existence proof in 1974, what are we to make of subsequent arguments that revocation is infeasible in capability systems? Of those who made this impossibility claim, as far as we are aware, *none* pointed to a flaw in Redell's reasoning. The key is the difference between permission and authority analysis. ([Chander01] analyzes, in our terms, only permission.) By such an analysis, Bob was never given permission to access Carol, so there was no access to Carol to be revoked! Bob was given permission to access `carol2`, and he still has it. No permissions were revoked.

> *A security officer investigating an incident needs to know who has access to a compromised object.*
>
> —Karger and Herbert [Karger84]

In their paper, Karger and Herbert propose to give a security officer a list of all subjects who are, in our terms, permitted to access Carol. This list will not include Bob's access to Carol, since this indirect access is represented only by the system's protection state taken together with the behavior of objects playing by the rules. Within their system, Alice, by restricting the authority given to Bob as she should, has inadvertently thwarted the security officer's ability to get a meaningful answer to his query.

To render a permission-only analysis useless, a threat model need not include either malice or accident; it need only include subjects following security best practices.

An arrangement-only bound on permission or authority would include the possibility of the Caretaker giving Bob direct access to Carol—precisely what the Caretaker was constructed not to do. Only by reasoning about behaviors can Alice see that the Caretaker is a "smart reference". Just as `pointMaker` extends our vocabulary of data types, raising the abstraction level at which we express solutions, so does the Caretaker extend our vocabulary for expressing access control. Alice (or her programmer) should use arrangement-only analysis for reasoning about what potential adversaries may do. But Alice also interacts with many objects, like the Caretaker, *because* she has some confidence she understands their actual behavior.

---

[4] The simple Caretaker shown here depends on Alice assuming that Carol will not provide Carol's clients with direct access to herself.
See www.erights.org/elib/capability/deadman.html for a more general treatment of revocation in ***E***.

4.5. Access Abstraction

The object-capability model does not describe access control as a separate concern, to be bolted on to computation organized by other means. Rather it is a model of modular computation with no separate access control mechanisms. All its support for access control is well enough motivated by the pursuit of abstraction and modularity. Parnas' principle of *information hiding* [Parnas72] in effect says our abstractions should hand out information only on a *need to know* basis. POLA simply adds that authority should be handed out only on a *need to do* basis [Crockford97]. Modularity and security each require both of these principles.

The *object-capability paradigm*, in the air by 1967 [Wilkes79, Fabry74], and well established by 1973 [Redell74, Hewitt73, Morris73, Wulf74, Wulf81], adds the observation that the abstraction mechanisms provided by the base model are not just for procedural, data, and control abstractions, but also for *access abstractions*, such as Redell's Caretaker. (These are "communications abstractions" in [Tribble95].)

Access abstraction is pervasive in actual capability practice, including filtering facets, unprivileged transparent remote messaging systems [Donnelley76, Sansom86, Doorn96, Miller00], reference monitors [Rajunas89], transfer, escrow, and trade of exclusive rights [Miller96, Miller00], and recent patterns like the Powerbox [Wagner02, Stiegler02]. Further, every non-security-oriented abstraction that usefully encapsulates its internal state provides, in effect, restricted authority to affect that internal state, as mediated by the logic of the abstraction.

The base system is also a security enforcing system, providing abstractions for controlling access. When all code is either trusted or untrusted, one can only extend the expressiveness of a protection system by adding code to the base, making everyone fully vulnerable to its possible misbehavior. By contrast, only Alice relies on the behavior of her Caretaker, and only to limit authority flowing between Bob and Carol. The risks to everyone, even Alice, from its misbehavior are limited because the Caretaker itself has limited authority. Alice can often bound her risk even from bugs in her own security enforcing programs.

5. Confinement

> *… a program can create a controlled environment within which another, possibly untrustworthy program, can be run safely… call the first program a customer and the second a service. … [the service] may leak, i.e. transmit … the input data which the customer gives it. … We will call the problem of constraining a service [from leaking data] the confinement problem.*
>
> —Lampson [Lampson73]

Once upon a time, in the days before wireless, you (a human customer) could buy a box containing a calculator (the service) from a manufacturer you might not trust. Although you might worry whether the calculations are correct, you can at least enter your finan-

cial data confident that the calculator cannot leak your secrets back to its manufacturer. How did the box solve the confinement problem? By letting you see that it comes with no strings attached. When the only causation to worry about would be carried by wires, the visible absence of wires emerging from the box—the isolation of the subgraph—is adequate evidence of confinement.

Here, we use this same technique to achieve confinement, substituting capabilities for wires. The presentation here is a simplification of confinement in actual object-capability systems [Hardy86, Shapiro99, Shapiro00, Wagner02, Yee03].

To solve confinement, assume that the manufacturer, Max, and customer, Cassie, have mutual access to a `[Factory, factoryMaker]` pair created by the following code. Assume that Cassie trusts that this pair of objects behaves according to this code.[5]

```
def [Factory, factoryMaker] := {
    interface Factory guards FactoryStamp {…}

    def factoryMaker {
        to make(code :String) :Factory {
            def factory implements FactoryStamp {
                to new(state) :any {
                    return loader.load(code, state)
                }
            }
            return factory
        }
    }
    [Factory, factoryMaker]
}
```

The `interface .. guards` expression evaluates to a (*trademark guard*, *stamp*) pair representing a new *trademark*, similar in purpose to an interface type. This syntax also defines variables to hold these objects, here named `Factory` and `FactoryStamp`. Here we use the `FactoryStamp` to mark instances of `factory`, and nothing else, as carrying this trademark. We use the `Factory` guard in soft type declarations, like "`:Factory`" above, to ensure that only objects carrying this trademark may pass. The block of code above evaluates to a `[Factory, factoryMaker]` pair. Only the `factoryMaker` of a pair can make objects, instances of `factory`, which will pass the `Factory` guard of that pair.[6]

---

[5] Given *mutual* trust in this pair, our same logic solves an important mutual suspicion problem. Max knows Cassie cannot "open the case"—cannot examine or modify his code.

[6] Such trademarking can be implemented in DVH and in our model of object-capability computation [Morris73, Miller87, Tribble95, Rees96], so object-capability systems which provide trademarking primitively [Wulf81, Hardy85, Shapiro99, Yee03] are still within our model.

Max uses a `factoryMaker` to package his proprietary calculator program in a box he sends to Cassie.

```
def calculatorFactory := factoryMaker.make("…code…")
cassie.acceptProduct(calculatorFactory)
```

In section 5.2 Cassie uses a "`:Factory`" declaration on the parameter of her `acceptProduct` method to ensure that she receives only an instance of the above `factory` definition. Inspection of the factory code shows that a factory's state contains only data (here, a String) and no capabilities—no access to the world outside itself. Cassie may therefore use the factory to make as many live calculators as she wants, confident that each calculator has only that access beyond itself that Cassie authorizes. They cannot even talk to each other unless Cassie allows them to.

With lambda evaluation, a new subject's code and state both come from the same parent. To solve the confinement problem, we combine code from Max with state from Cassie to give birth to a new calculator, and we enable Cassie to verify that she is the only state-providing parent. This state is an example of Lampson's "controlled environment". To Cassie, the calculator is a *controlled subject*—one Cassie knows is born into an environment controlled by her. By contrast, should Max introduce Cassie to an already instantiated calculation service, Cassie would not be able to tell whether it has prior connectivity. (Extending our analogy, suppose Max offers the calculation service from his web site.) The calculation service would be an *uncontrolled subject* to her.

We wish to reiterate that by "confinement", we refer to the overt subset of Lampson's problem, where the customer accepts only code ("a program") from the manufacturer and instantiates it in a controlled environment. We do not propose to confine information or authority given to uncontrolled subjects.

5.1. A Non-Discretionary Model

Capabilities are normally thought to be *discretionary*, and to be unable to enforce confinement. Our confinement logic above relies on the non-discretionary nature of object-capabilities. What does it mean for an access control system to be discretionary?

> *"Our discussion … rested on an unstated assumption: the principal that creates a file or other object in a computer system has unquestioned authority to authorize access to it by other principals. …We may characterize this control pattern as* discretionary*." [emphasis in the original]*
>
> —Saltzer and Schroeder [Saltzer75]

Object-capability systems have no principals. A human user, together with his shell and "home directory" of references, participates, in effect, as just another subject. With the substitution of "subject" for "principal", we will use this classic definition of "discretionary".

By this definition, *object-capabilities are not discretionary*. In our model, in DVH, and in most actual capability system implementations, even if Alice creates Carol, Alice

may still only authorize Bob to access Carol if Alice has authority to access Bob. If capabilities were discretionary, they would indeed be unable to enforce confinement. To illustrate the power of confinement, we use it below to enforce the *-properties.

5.2. The *-Properties

> *Boebert made clear in [[Boebert84]] that an unmodified or classic capability system cannot enforce the *-property or solve the confinement problem.*
>
> —Gong [Gong89]

Briefly, the *-properties taken together allow subjects with lower (such as "secret") clearance to communicate to subjects with higher (such as "top secret") clearance, but prohibit communication in the reverse direction [Bell74]. KeySafe is a concrete and realistic design for enforcing the *-properties on KeyKOS, a pure object-capability system [Rajunas89]. However, claims that capabilities cannot enforce the *-properties continue [Gong89, Kain87, Wallach97, Saraswat03], citing [Boebert84] as their support. Recently, referring to [Boebert84], Boebert writes:

> *The paper … remains, no more than an offhand remark. … The historical significance of the paper is that it prompted the writing of [[Kain87]]*
>
> —Boebert [Boebert03]

Boebert here defers to Kain and Landwehr's paper [Kain87]. Regarding object-capability systems, Kain and Landwehr's paper makes essentially the same impossibility claims, which they support only by citing and summarizing Boebert. To lay this matter to rest, we show how Cassie solves Boebert's challenge problem—how she provides a one way communication channel to subjects she doesn't trust, say Q and Bond, who she considers to have secret and top secret clearance respectively. Can Cassie prevent Boebert's attack, in which Q and Bond use the rights Cassie provides to build a reverse channel?

Completing our earlier confinement example, Cassie accepts a calculator factory from Max using this method.

```
to acceptProduct(calcFactory :Factory) :void {
    var diode :int := 0
    def diodeWriter {
        to write(val :int) :void { diode := val }
    }
    def diodeReader {
        to read() :int { return diode }
    }
    def q := calcFactory.new(["writeUp" => diodeWriter, …])
    def bond := calcFactory.new(["readDown" => diodeReader, …])
    …
}
```

Cassie creates two calculators to serve as Q and Bond. She builds a data diode by defining a `diodeWriter`, a `diodeReader`, and an assignable `diode` variable they share. She gives Q and Bond access to each other only through the data diode. Applied to Cassie's arrangement, Boebert's attack starts by observing that Q can send a capability as an argument in a message to the `diodeWriter`. An arrangement-only analysis of bounds on permissions or authority supports Boebert's case—the data diode might introduce this argument to Bond. Only by examining the behavior of the data diode can we see the tighter bounds it was built to enforce. It transmits data (here, integers) in only one direction and capabilities in neither. (Q cannot even read what he just wrote!) Cassie relies on the behavior of the factory and data diode abstractions to enforce the *-properties and prevent Boebert's attack. (See [Miller03] for further detail.)

5.3. The Arena and Terms of Entry

Policies like the *-properties are generally assumed to govern a computer system as a whole, to be enforced in collaboration with a human sys-admin or security officer. In a capability system, this is a matter of *initial conditions*. If the owner of the system wishes such a policy to govern the entire system, she can run such code when the system is first generated, and when new users join. But what happens after the big bang? Let's say Alice meets Bob, who is an uncontrolled subject to her. Alice can still enforce "additive" policies on Bob, e.g., she can give him revocable access to Carol, and then revoke it. But she cannot enforce a policy on Bob that requires removing prior rights from Bob, for that would violate Bob's security!

Instead, as we see in the example above, acting as Lampson's "customer", she sets up an *arena*—Lampson's "controlled environment"—with initial conditions she determines, governed by her rules, and over which she is the sys-admin. If her rules can be enforced on uncontrolled subjects, she can admit Bob onto her arena as a player. If her rules require the players not to have some rights, she must set *terms of entry*. "Please leave your cellphones at the door." A prospective participant (Max) provides a player (`calcFactory`) to represent his interests within the arena, where this player can pass the security check at the gate (here, `:Factory`). No rights were taken away from anyone; participation was voluntary.

The arena technique corresponds to *meta-linguistic abstraction*—an arena is a virtual machine built within a virtual machine [Abelson86, Safra86]. The resulting system can be described according to either level of abstraction—by the rules of the base level object-capability system or by the rules of the arena. The subjects built by the admitted factories are also subjects within the arena. At the base level, we would say Q has permission to send messages to `diodeWriter` and authority to send integers to Bond. At the arena level of description, we would say a data diode is a primitive part of the arena's protection state, and say Q has permission to send integers to Bond. Any base level uncontrolled subjects admitted into the arena are devices of the arena—they have mysterious connections to the arena's external world.

When the only inputs to a problem are data (here, code), any system capable of universal computation can solve any solvable problem, so questions of absolute possibility become useless for comparisons. Conventional language comparisons face the same dilemma, and language designers have learned to ask instead an engineering question: *Is this a good machine on which to build other machines?* How well did we do on Boebert's challenge? The code admitted was neither inspected nor transformed. Each arena level subject was also a base level subject. The behavior interposed by Cassie between the subjects was very thin. Mostly, we reused the security properties of the base level object-capability system to build the security properties of our new arena level machine.

5.4. Mutually Suspicious Composition

When mutually suspicious interests build a diversity of abstractions to express a diversity of co-existing policies, how do these extensions interact?

Let's say that Q builds a gizmo that might have bugs, so Q creates a Caretaker to give the gizmo revocable access to his `diodeWriter`. Q's policy relies on the behavior of his Caretaker but not necessarily on Cassie's `diodeWriter`. To Cassie, Q's gizmo and Caretaker are part of Q's subgraph and indistinguishable from Q. Cassie's policy relies on the behavior of her `diodeWriter`, but not on Q's Caretaker. They each do a partially behavioral analysis over the same graph, each from their own subjective perspective. This scenario shows how diverse expressions of policy often compose correctly even when none of the interested parties are aware this is happening.

6. Conclusion

Just as we should not expect a base programming language to provide us all the data types we need for computation, we should not expect a base access control system to provide us all the elements we need to express our protection policies. Both issues deserve the same kind of answer: We use the base to build abstractions, extending the vocabulary we use to express our solutions. In evaluating a protection model, one must examine how well it supports the extension of its own expressiveness by abstraction and composition.

Security in computational systems emerges from the interaction between primitive protection mechanisms and the behavior of security enforcing programs. As we have shown here, such programs are able to enforce restrictions on more general, untrusted programs by building on and abstracting more primitive protection mechanisms. To our knowledge, the object-capability model is the only protection model whose semantics can be readily expressed in programming language terms: approximately, lambda calculus with local side effects. This provides the necessary common semantic framework for reasoning about permission and program behavior together. Because security-enforcing programs are often simple, the required program analysis should frequently prove tractable, *provided* they are built on effective primitives.

By recognizing that program behavior can contribute towards access control, a lost paradigm for protection—abstraction—is restored to us, and a semantic basis for extensible protection is established. Diverse interests can each build abstractions to express their policies regarding new object types, new applications, new requirements, and each other, and these policies can co-exist and interact. This extensibility is well outside the scope of traditional access graph analyses.

Analyses based on the evolution of protection state are conservative approximations. A successful verification demonstrating the enforcement of a policy using only the protection graph (as in [Shapiro00]) is robust, in the sense that it does *not* rely on the cooperative behavior of programs. Verification *failures* are *not* robust – they may indicate a failure in the protection model, but they can also result from what might be called "failures of conservatism"—failures in which the policy is enforceable but the verification model has been simplified in a way that prevents successful verification.

We have shown by example how object-capability practitioners set tight bounds on authority by building abstractions and reasoning about their behavior, using conceptual tools similar to that used by object programmers to reason about any abstraction. We have shown, using only techniques easily implementable in Dennis and van Horn's 1966 Supervisor, how actual object-capability systems have used abstraction to solve problems that analyses using only protection state have "proven" impossible for capabilities.

The object-capability paradigm, with its pervasive, fine-grained, and extensible support for the principle of least authority, enables mutually suspicious interests to cooperate more intimately while being less vulnerable to each other. When more cooperation may be practiced with less vulnerability, we may find we have a more cooperative world.

7. Acknowledgments

We thank Darius Bacon, Tyler Close, K. Eric Drexler, Bill Frantz, Norm Hardy, Chris Hibbert, David Hopwood, Tad Hogg, Ken Kahn, Alan Karp, Terence Kelly, Lorens Kockum, Charles Landau, Mark Lillibridge, Chip Morningstar, Greg Nelson, Jonathan Rees, Vijay Saraswat, Terry Stanley, Marc Stiegler, E. Dean Tribble, Bill Tulloh, Kazunori Ueda, David Wagner, Bryce Wilcox-O'Hearn, the cap-talk and e-lang discussion lists, and especially Ka-Ping Yee, our co-author on [Miller03], whose writing had substantial influence on this paper.

8. References

[Abelson86] H. Abelson, G. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1986.

[Bell74] D.E. Bell, L. LaPadula. "Secure Computer Systems" ESD-TR-83-278, Mitre Corporation, vI and II (Nov 1973), vIII (Apr 1974).

[Bishop79] M. Bishop, L. Snyder. "The Transfer of Information and Authority in a Protection System" SOSP 1979 , p. 45–54.

[Boebert84] W. E. Boebert. “On the Inability of an Unmodified Capability Machine to Enforce the *-Property” *Proceedings of 7th DoD/NBS Computer Security Conference*, September 1984, p. 291–293. http://zesty.ca/capmyths/boebert.html

[Boebert03] (Comments on [Miller03]) http://www.eros-os.org/pipermail/cap-talk/2003-March/001133.html

[Cartwright91] R. Cartwright, M. Fagan, “Soft Typing”, *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*.

[Chander01] A. Chander, D. Dean, J. C. Mitchell. “A State-Transition Model of Trust Management and Access Control” *Proceedings of the 14th Computer Security Foundations Workshop*, June 2001, p. 27–43.

[Crockford97] Douglas Crockford, personal communications, 1997.

[Dennis66] J.B. Dennis, E.C. Van Horn. “Programming Semantics for Multiprogrammed Computations” *Communications of the ACM*, 9(3):143–155, March 1966.

[Donnelley76] J. E. Donnelley. “A Distributed Capability Computing System” *Third International Conference on Computer Communication*, Toronto, Canada, 1976.

[Doorn96] L. van Doorn, M. Abadi, M. Burrows, E. P. Wobber. “Secure Network Objects” *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, p. 211–221.

[Fabry74] R. S. Fabry. “Capability-based addressing” *Communications of the ACM*, 17(7), 1974, p. 403–412.

[Goldberg76] A. Goldberg, A. Kay. *Smalltalk-72 instruction manual*. Technical Report SSL 76-6, Learning Research Group, Xerox Palo, Alto Research Center, 1976. http://www.spies.com/~aek/pdf/xerox/alto/Smalltalk72_Manual.pdf

[Gong89] L. Gong. “A Secure Identity-Based Capability System” *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, p. 56–65.

[Hardy85] N. Hardy. “The KeyKOS Architecture” *ACM Operating Systems Review*, September 1985, p. 8–25. http://www.agorics.com/Library/KeyKos/architecture.html

[Hardy86] N. Hardy. *U.S. Patent 4,584,639: Computer Security System*,

[Harrison76] M.A. Harrison, M.L. Ruzzo, and J.D. Ullman. “Protection in operating systems” *Communications of the ACM*, 19(8) p. 461–471, 1976.

[Hewitt73] C. Hewitt, P. Bishop, R. Stieger. “A Universal Modular Actor Formalism for Artificial Intelligence” *Proceedings of the 1973 International Joint Conference on Artificial Intelligence*, p. 235–246.

[Jones76] A. K. Jones, R. J. Lipton, L. Snyder. “A Linear Time Algorithm for Deciding Security” *FOCS*, 1976, p. 33–41.

[Kahn88] K. Kahn, M. S. Miller. “Language Design and Open Systems”, *Ecology of Computation*, Bernardo Huberman (ed.), Elsevier Science Publishers, North-Holland, 1988.

[Kain87] R. Y. Kain, C. E. Landwehr. “On Access Checking in Capability-Based Systems” *IEEE Symposium on Security and Privacy*, 1987.

[Karger84] P. A. Karger, A. J. Herbert. “An Augmented Capability Architecture to Support Lattice Security and Traceability of Access” *Proc. of the 1984 IEEE Symposium on Security and Privacy*, p. 2–12.

[Kelsey98] R. Kelsey, (ed.), W. Clinger, (ed.), J. Rees, (ed.), “Revised^5 Report on the Algorithmic Language Scheme”ACM Sigplan Notices, 1998.

[Lampson73] B. W. Lampson, “A Note on the Confinement Problem” *CACM on Operating Systems*, 16(10), October, 1973.

[Miller87] M. S. Miller, D. G. Bobrow, E. D. Tribble, J. Levy, "Logical Secrets" *Concurrent Prolog: Collected Papers*, E. Shapiro (ed.), MIT Press, Cambridge, MA, 1987.

[Miller96] M. S. Miller, D. Krieger, N. Hardy, C. Hibbert, E. D. Tribble. "An Automatic Auction in ATM Network Bandwidth". *Market-based Control, A Paradigm for Distributed Resource Allocation*, S. H. Clearwater (ed.), World Scientific, Palo Alto, CA, 1996.

[Miller00] M. S. Miller, C. Morningstar, B. Frantz. "Capability-based Financial Instruments" *Proceedings Financial Cryptography 2000*, Springer Verlag. http://www.erights.org/elib/capability/ode/index.html

[Miller03] M. S. Miller, K. -P. Yee, J. S. Shapiro, "Capability Myths Demolished", HP Labs Technical Report, in preparation. http://zesty.ca/capmyths/usenix.pdf

[Morris73] J. H. Morris. "Protection in Programming Languages" *CACM* 16(1) p. 15–21, 1973. http://www.erights.org/history/morris73.pdf.

[Motwani00] R. Motwani, R. Panigrahy, V. Saraswat, S. Venkatasubramanian. "On the Decidability of Accessibility Problems" AT&T Labs – Research. http://www.research.att.com/~suresh/Papers/java.pdf

[Neumann80] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, L. Robinson. *A Provably Secure Operating System: The System, Its Applications, and Proofs*, CSL-116, Computer Science Laboratory, SRI International, Inc., May 1980.

[Parnas72] D. Parnas. "On the Criteria To Be Used in Decomposing Systems into Modules" *CACM* 15(12), December 1972. http://www.acm.org/classics/may96/.

[Rajunas89] S. A. Rajunas. "The KeyKOS/KeySAFE System Design" Key Logic, Inc., SEC009-01, March, 1989.

[Redell74] D. D. Redell. *Naming and Protection in Extendible Operating Systems*. Project MAC TR-140, MIT, November 1974. (Ph. D. thesis.)

[Rees96] J. Rees, *A Security Kernel Based on the Lambda-Calculus*. MIT AI Memo No. 1564. MIT, Cambridge, MA, 1996. http://mumble.net/jar/pubs/secureos/

[Safra86] M. Safra, E. Y. Shapiro. *Meta Interpreters for Real*. Information Processing86, H. -J. Kugler (ed.), North-Holland, Amsterdam, p. 271–278, 1986.

[Saltzer75] J. H. Saltzer, M. D. Schroeder. "The Protection of Information in Computer Systems" *Proceedings of the IEEE* **63**(9), September 1975, p. 1278–1308.

[Sansom86] R. D. Sansom, D. P. Julian, R. Rashid. "Extending a Capability Based System Into a Network Environment" *Research sponsored by DOD*, 1986, p. 265–274.

[Saraswat03] V. Saraswat, R. Jagadeesan. "Static support for capability-based programming in Java" http://www.cse.psu.edu/~saraswat/neighborhood.pdf

[Shapiro99] J. S. Shapiro, J. M. Smith, D. J. Farber. "EROS: A Fast Capability System" *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, December 1999, p. 170–185.

[Shapiro00] J. S. Shapiro, S. Weber. "Verifying the EROS Confinement Mechanism" *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, p. 166–176.

[Sitaker00] K. Sitaker. *Thoughts on Capability Security on the Web.* http://lists.canonical.org/pipermail/kragen-tol/2000-August/000619.html

[Stiegler02] M. Stiegler, M. Miller. "A Capability Based Client: The DarpaBrowser" http://www.combex.com/papers/darpa-report/index.html

[Tanenbaum86] A. S. Tanenbaum, S. J. Mullender, R. van Renesse. "Using Sparse Capabilities in a Distributed Operating System" *Proceedings of 6th International Conference on Distributed Computing Systems*, 1986, p. 558–563.

[Tribble95] E. D. Tribble, M. S. Miller, N. Hardy, D. Krieger, "Joule: Distributed Application Foundations", 1995. http://www.agorics.com/joule.html

[Roy02] P. Van Roy, S. Haridi, "Concepts, Techniques, and Models of Computer Programming" MIT Press, in preparation. http://www.info.ucl.ac.be/people/PVR/book.html

[Wagner02] D. Wagner, D. Tribble. *A Security Analysis of the Combex DarpaBrowser Architecture*. http://www.combex.com/papers/darpa-review/index.html

[Wallach97] D. S. Wallach, D. Balfanz, D. Dean, E. W. Felten. "Extensible Security Architectures for Java" *Proceedings of the 16th Symposium on Operating Systems Principles*, 1997, p. 116–128. http://www.cs.princeton.edu/sip/pub/sosp97.html

[Wilkes79] M. V. Wilkes, R. M. Needham. *The Cambridge CAP Computer and its Operating System*. Elsevier North Holland, 1979.

[Wulf74] William A. Wulf, Ellis S. Cohen, William M. Corwin, Anita K. Jones, Roy Levin, C. Pierson, and Fred J. Pollack. "HYDRA: The Kernel of a Multiprocessor Operating System" *Communications of the ACM*, **17**(6):337-345, 1974

[Wulf81] W. A. Wulf, R. Levin, S. P. Harbison. *HYDRA/C.mmp: An Experimental Computer System*, McGraw Hill, 1981.

[Yee03] K.-P. Yee, M. S. Miller. *Auditors: An Extensible, Dynamic Code Verification Mechanism*. http://www.erights.org/elang/kernel/auditors/index.html