Submitted to the *Eighth Asian Computing Science Conference* (ASIAN'03)
Tata Institute of Fundamental Research, Mumbai India, December 10-13 2003

Paradigm Regained: Abstraction Mechanisms for Access Control

SRL Technical Report SRL2003-03
Department of Computer Science
Johns Hopkins University

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 support the Principle of Least Authority (POLA), i.e., how well they enable the distribution of appropriate access rights needed for cooperation, while simultaneously limiting the inappropriate proliferation of access rights which would create vulnerabilities. POLA may be practiced by *arrangement* of permissions and by *abstraction* of access. To date, access control systems have been evaluated only by their effectiveness at POLA-by-arrangement.

Working in the original capability model proposed by Dennis and van Horn, we show how actual systems have used abstraction to enforce revocation, confinement, and the *-properties—policies whose enforcement has been "proven" impossible by arrangement-only analysis. To account for these abilities, analysis must also examine the behavior of security-enforcing programs (which are usually simple) to see how they limit the authority of arbitrarily complex untrusted programs. The original capability model, analyzed in these terms, 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 substantial damage. By exploiting a single bug in an ordinary server or application, an attacker is able to compromise a whole system. Bugs scale linearly 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 do not offer any viable long-term solution: they fail to provide the defender with any 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 dangerously 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 Princi-

ple 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.

As system designers, we are concerned with *authority* rather than *permissions*. Permissions determine what actions an individual program may perform on objects it can directly access. Authority describes the effects a program may cause as determined both by the arrangement of permissions and by the permitted actions of other programs. To form a coherent understanding of access it is therefore necessary to reason about the interactions between permissions and program behavior in a unified way. If we are unable to reason about access, it seems self-evident that we are equally unable to reason about access *control*. 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 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—under the implicit assumption that all programs are hostile. While conservatively safe, this approach explicitly 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. Unfortunately, the prevailing formal analysis approach is unable to capture the further bounds trusted programs place on the authority available to untrusted programs. This results in "false negatives" that have led a number of researchers to mistaken conclusions about the feasibility of policy enforcement in certain access control models. In the process, these results have diverted attention from the possibility that a more 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 primitive expressiveness of object-capabilities. Using abstraction, object-capability practitioners have solved problems like revocation, overt confinement,[1] and the *-properties. We show here the logic of these solutions, using only functionality available in Dennis and van Horn's 1966 Supervisor, hereafter referred to as "DVH."

---

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

To understand how object-capability practitioners build abstractions to solve these problems, we show how simple and tractable programs can be used to bound the authority available to arbitrarily complex, unanalyzed, and potentially hostile programs. 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*, adapted from the take-grant distinction between *de jure* and *de facto* transfer. In "How Much Authority Does 'cp' Need?", we use a pair of Unix shell examples to illuminate the stark contrast between ACLs and object-capabilities. 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

For the following discussion, we say a *direct access right* to an *object* gives *permission* to a *subject* 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.

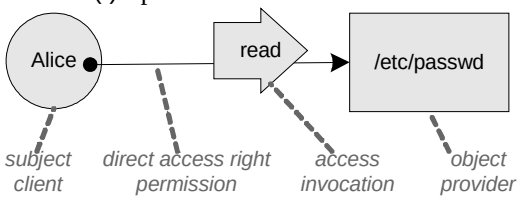

Fig 1. *Access diagrams* depict protection state.

By *subject* we mean the finest grain unit of computation on a given system that may be given distinct 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 grained unit to which separate 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.

Any discussion of access must carefully distinguish between *direct* and *indirect* access (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 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 can't generally predict if Bob will gain this permission, but our safety may demand that we ensure he cannot.

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. When the state and behavior of some subjects and objects are also taken into account, we have 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 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, if we take some state and behavior into account, we have 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 imposes rules *vs.* the subjects hosted on that base, restricted by those rules. By definition, the operators of a base access control system can only manipulate permissions. When subjects extend the expressiveness of a system by abstraction, authority serves as virtual permission. Only by understanding the behavior of an abstraction can we see how it further tightens bounds on authority. As we ascend layers of abstraction, as the extensions of one layer become the implementation of the next higher, the boundary between permission and authority shifts accordingly. Permission is relative to a frame of reference. Authority is invariant.

---

[2] The Harrison Ruzzo Ullman paper [Harrison76] is often misunderstood to say this calculation is never decidable. HRU actually says it is possible (in fact, 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].

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 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 objects (files) in your namespace. In order for `cp` to do its job, it must use your namespace, and must be able to read and write any file you might name that you can read and write. Not only does it operate with all your authority, it must. Given this ACL-oriented way of using names, its *least authority* would still include all of your authority to the file system. So long as we install and run normal 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). In this case, the two open files passed in are all the least authority needed to perform this request. If `cat` ran as a separate subject with *only* these authorities, it could still do its job, and we could more easily reason about our possible 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 pure object model of computation [Goldberg76, Hewitt73], there is no distinction between subjects and objects. A *non-primitive object* is an encapsulated combination of code and state, where the *state* is an addressable and mutable collection of references to objects. A reference provides access to an object, indivisibly combining designation of the object, the right to access it, and the means to access it. Depending on the system, an *address* may be an instance variable name or index, a memory address, or a capability-list index (a c-list index, like a file descriptor number). The *computational system* is the dynamic access graph of objects held together by these references. Within the limits set by the access graph, objects, behaving according to their code, interact only by sending messages carrying an addressable collection of arguments (e.g., by argument position), thereby changing the access graph.

Model Term	Cap OS Term	OO Language Term
object	process, domain, page	object, closure, instance
code	program	lambda expression
state	address space + c-list (capability list)	environment, instance variable frame
address	memory address, c-list index	lexical name, argument position

The *object-capability* model of secure computation recognizes the security inherent in the object model. 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 (like forged pointers, direct access to another's private state, 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.

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 powers of access control are present in DVH.[3]

In the *initial* conditions of Figure 2, Alice's state includes references to Bob and Carol—Bob and Carol are *directly accessible* from Alice.

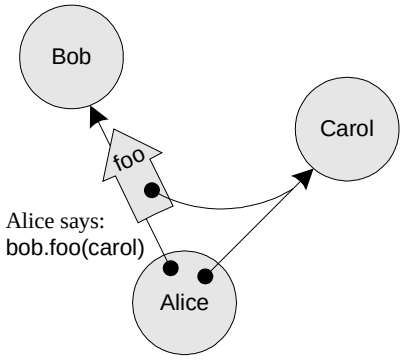

Fig 2: Introduction by Message Passing

Alice causes an effect on the world outside herself only by sending *messages* to objects directly accessible to her (such as Bob), where she may include, at distinct argument addresses, references to any objects accessible to her (such as Carol). A call-return pattern consists of 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) also become accessible to Bob. Within the limits set by these rules, and by what Bob may

[3] The object-capability model is essentially the untyped lambda calculus with local side effects (the model Scheme is based on), where a message send applies an object-as-closure to a message-as-argument. This three-way correspondence of objects, lambda calculus, and capabilities was noticed several times by 1973 [Goldberg76, Hewitt73, Morris73], and investigated explicitly in [Rees96].

feasibly know or compute, Bob's reaction to an incoming message is only according to his *code* (see below). All computation happens only in response to messages.

We distinguish three kinds of primitive objects.

- *Immutable 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. We refer to immutable data objects collectively as *bits*, as they provide only information, no access. Because data is immutable, we need not distinguish between a reference to data and the data itself. A reference to non-data is a *capability*.
- *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.
- *Creator*. We assume universal access to an object creation service, which any object can invoke with code and state to create a new object. The creator accepts only immutable bits as code, or alternatively, it makes a private bits-only copy for use as the new object's code. Code uses an *address* to indicate which accessible reference to use, or where in its state to store an accessible reference. The state is a mutable map from addresses to references. The new object is an *instance* of its code. The creator returns the *only* reference to the new object. (Below, we explain nested lambda evaluation as built from use of the creator.)

By these rules, *only connectivity begets connectivity*—all access relationships must derive from previous access relationships. Two disjoint subgraphs cannot become connected as no one can introduce them. Arrangement-based analysis of permission bounds proceeds by graph reachability arguments. Overt causation, carried only by messages, may flow only along permitted pathways. We may again use reachability arguments to reason about authority bounds, i.e., overt causality. The transparency of garbage collection relies on such arguments.

4.2. A Taste of *E*

E is a simple object-capability language in which objects are closures instantiated by lambda evaluation. We use only a subset of ***E*** providing functionality present in DVH. This has the semantics of lambda calculus with local side effects, much like Scheme [Kelsey98], combined with syntactic support for message sending and method dispatch. Here is a simple data abstraction.

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

This defines an object, `pointMaker`, used to make new points. The keyword "`to`" is used to define its single method, `make`, that, when invoked, defines and returns a new `point`. (The "`:any`" *vs.* "`:void`" on a method declares whether it returns a value.) The code for `point` uses `x` and `y` freely. These are its instance variables, and together form its state. The state maps from addresses `"x"` and `"y"` to the associated values from point's creation context. This example defines a polymorphic abstract data type—points respond to the same `add` message used to add integers.

Given the universal creator specified above, we could transform the above code to

```
def pointMaker {
    to make(x,y) :any {
        def point := creator.create("def point {…}",
                                    ["x" => x, "y" => y])
}   }
```

(The expression `["x" => x, "y" => y]` builds a map (hashtable) of key => value associations.)

Applying this transformation recursively would unnest all lambda expressions. Nested lambda evaluation better explains instantiation in object languages. The `creator` better explains process or domain creation in object-capability OSes. In ***E***, we almost always use lambda, but we use the `creator` 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 is giving Bob unconditional, full, and perpetual access to Carol.[4] 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 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 object they designate.

What is Alice to do? Using a slight simplification of Redell's Caretaker pattern for revoking access [Redell74]

[4] Object-capability operating systems such as KeyKOS [Hardy85] and EROS [Shapiro99] implement explicit storage management. In these, access to an object is universally rescinded whenever an object is destroyed.

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

Alice instead says

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

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

Variables in ***E*** are final by default. `var` simply means that `target` is non-final, it can be assigned to. Within the scope of `target`'s definition, `make` defines two objects, `caretaker` and `revoker`, and returns both of these 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 "`to`" clauses defining 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`). `E.call(…)` allows messages to be sent generically (like Smalltalk's `perform:` or Java's "reflection").[5]

The Caretaker itself 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 run 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 im-

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

possibility 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 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.

4.5. The Object-Capability Paradigm

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. Modularity and security each require both of these principles.

The *object-capability paradigm*, in the air by 1967 [Wlkes79, 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.

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.

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 (the customer) could buy 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 financial data confident that the calculator can't leak your secrets back to its manufacturer. How did the calculator solve the confinement problem? By letting you see that no strings were attached. In a world where the only causation of concern would be carried by wires, the visible absence of wires emerging from the box—the isolation of the subgraph—is adequate evidence of confinement. (Wires within the box are of no concern.)

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

To solve confinement, assume that the manufacturer and customer have mutual access to a (`Factory`, `factoryMaker`) pair created by the following code, and assume that the customer trusts that this pair of objects behaves according to this code:[6]

```
{   interface Factory guards FactoryStamp {…}

    def factoryMaker {
        to make(code) :Factory {
            def factory implements FactoryStamp {
                to build(state) :any {
                    creator.create(code, state)
    }   }   }   }
    [Factory, factoryMaker]
}
```

The keywords `interface` and `guards` create a pair of objects, `Factory` and `FactoryStamp`, representing a new *trademark*, similar in purpose to an interface type. The `FactoryStamp` is used here to mark instances of the `factory` definition, and nothing else, as carrying this trademark. The `Factory` is used in soft type decla-

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

rations, like "`:Factory`" above, to ensure that only objects carrying this trademark may pass. (Although it may not be obvious, such trademarking can be implemented in DVH and in our model of object-capability computation [Wulf81, Morris73, Miller87, Rees95, Yee03, Hardy85, Shapiro99].)

Using the `factoryMaker`, the manufacturer puts his proprietary calculator program in an impenetrable shell and sends it to the customer.

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

The customer uses a "`:Factory`" declaration to ensure that the product she receives has no external wires. She then uses it to make as many live calculators as she wants. Each has only that access beyond themselves that the customer provides, so they can't even talk to each other unless the customer allows them to, as shown in the next example. (The customer does not care about access relationships ("wires") within the calculator—that's the calculator's business.)

With lambda evaluation, a new subject's code and state both come from the same parent. To solve the confinement problem, we need to combine code (including immutable data) from the manufacturer with state from the customer to give birth to a new calculator, and to enable the customer to verify that she is the only state-providing parent. This state is an example of Lampson's "controlled environment".[7] By participating in the instantiation of the calculator, the customer has special knowledge of what other access rights the calculator does not have. We say the calculator is a *controlled subject* to her—one born into an environment controlled by her. By contrast, should the manufacturer introduce the customer to an already instantiated calculation service, the customer would not be able to tell whether it has strings attached. (Extending our analogy, suppose the manufacturer offers a 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. The customer creates, in our terms, a controlled subject, whose authority she confines.

5.1. A Non-Discretionary Model

Our confinement logic depends on the non-discretionary nature of object-capabilities.

> *"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]

[7] KeyKOS, EROS, and ***E*** provide more flexible support for confinement than we show here. For example, the manufacturer can also be permitted to supply capabilities that convey transitively read-only authority, to be included in each calculator's initial state [Shapiro00].

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. Without this constraint, our confinement logic would not work.

5.2. The *-Properties

To illustrate the power of confinement, we use it to enforce the *-properties.

> *Boebert made clear in [[Boebert84]] that an unmodified or classic capability system cannot enforce the *-property or solve the confinement problem. The main pitfall of a classic capability system is that "the right to exercise access carries with it the right to grant access".*
>
> —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 a system providing *-properties, etc., that was to be implemented 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 for this claim. Recently, referring to [Boebert84], Boebert writes:

> *The paper in question was, and 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 see no choice but to accept Boebert's challenge problem.

Here is the method our customer uses to accept the calculator product from the manufacturer.

```
to acceptProduct(calcFactory :Factory) :void {
    var diode :int := 0
    def diodeWriter { to write(val :int) :void {diode := val}}
    def diodeReader { to read() :int { diode }}

    def q := calcFactory.build(["writeUp" => diodeWriter])
    def bond := calcFactory.build(["readDown" => diodeReader])
    …
}
```

Our customer creates two calculators, Q and Bond, that she might consider to have secret and top secret clearance, respectively. 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. An arrangement-only bound on permissions or authority supports Boebert's case—the data diode might introduce Q and Bond. By examining its behavior we see the tighter bounds it was built to enforce. We see it transmits bits (here, integers) in only one direction and capabilities in neither. (Q cannot even read what he just wrote!) With these tools, one can easily implement the *-properties, and prevent Boebert's attack. The right to exercise access to the data diode does not carry with it the right to grant access. (See [Miller03] for further details.)[8]

By using confinement, our customer can enforce policies, like the *-properties, that depend on the participants having no external communications channels beyond those enabled by the customer.

5.3. A Partitioned Model

> *Since a capability is just a bit string, it can propagate in many ways without the detection of the kernel or the server …*
>
> —Gong [Gong89]

Our ability to enforce the *-properties depends on the partitioned nature of object-capabilities.

Above, Q may communicate any bits he knows to Bond—he may share all his knowledge with Bond—but he is unable to share any of his permissions with Bond. In object-capability systems including DVH, bits and permissions are *partitioned.* Because Bond and Q cannot create a causal pathway from Bond to Q, Q is severely limited in his ability to proxy for Bond, and therefore in his ability to share even his authority with Bond. By contrast, in a *pure password capability system* like Amoeba [Tanenbaum86], a capability is just a bit string—permission is obtained merely by demonstrating knowledge of secrets.[9] In a pure password capability system, it is indeed impossible to both allow Q to communicate to Bond while preventing Q from sharing his permissions with Bond.

5.4. 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

---

[8] DVH and many other capability systems provide primitive data-read-only capabilities, enabling them to solve Boebert's problem only by arranging permissions. By using abstraction instead, our example presents a pattern that can be adapted to express more complex policies.

[9] By our classification, the Monash system is a semi-partitioned semi-password capability system [Anderson86].

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", Alice 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 (the manufacturer) 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 bits (here, code), any system capable of universal computation can solve any solvable problem, so questions of absolute possibility become useless. 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 this supposedly impossible example? The code admitted was neither inspected nor transformed. Each arena level subject was also a base level subject. The behavior interposed by the customer between the subjects was very thin. Altogether, we mostly just reused the security properties of the base level object-capability system to build the security properties of our new arena level machine.

6. Conclusion

Security in computational systems emerges from the interaction between primitive protection mechanisms and the behavior of trusted programs. To form a coherent understanding of access it is therefore necessary to extend our reasoning across the permissions/behavior boundary. Because of their restricted function, the required program analysis frequently proves tractable for security-enforcing programs, *provided* they are built on effective primitives. As we have shown here, such trusted 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: lambda calculus with side effects. This provides the necessary common semantic framework for reasoning across the permissions/behavior boundary.

By extending our analysis to include the behavior of trusted programs, a critically missing paradigm for protection—abstraction—is restored to us. The effectiveness of the object-capability model in this framework is readily apparent. Perhaps more important, a semantic basis for extensible protection systems is established. We can, if we like, extend the space of security-enforcing programs to meet the security requirements of new object types, new applications, and new requirements—all of which are outside the scope of what can be described using the traditional access graph.

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. As we have shown by illustration here, several of the impossibility proofs concerning object capability systems may be seen as failures of conservatism.

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 security 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 use of abstraction.

In this paper, 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 subsequent permission-only analyses 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 parties to cooperate more intimately while being less vulnerable to each other. When more coop-

eration may be practiced with less vulnerability, we may find we have a more cooperative world.

7. Acknowledgments

We thank Norm Hardy, Alan Karp, Jonathan Rees, Vijay Saraswat, Terry Stanley, Marc Stiegler, E. Dean Tribble, Bryce Wilcox-O'Hearn, the e-lang discussion list, 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.

[Anderson86] M. Anderson, R. Pose, C. S. Wallace. "A Password Capability System." *The Computer Journal*, 29(1), 1986, p. 1–8.

[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

[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.

[Dennis66] J.B. Dennis, E.C. Van Horn. "Programming Semantics for Multiprogrammed Computations." *Communications of the ACM*, 9(3):143–155, March 1966.
http://citeseer.nj.nec.com/dennis66programming.html

[Donnelley76] J. E. Donnelley. "A Distributed Capability Computing System." *Third International Conference on Computer Communication*, Toronto, Canada, 1976.
http://www.nersc.gov/~jed/papers/DCCS/

[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.
ftp://ftp.digital.com/pub/DEC/SRC/publications/wobber/sno.ps

[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.

[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*,
http://www.cis.upenn.edu/~KeyKOS/Patent/Patent.html

[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.
http://citeseer.nj.nec.com/kelsey98revised.html

[Lampson73] B. W. Lampson, "A Note on the Confinement Problem." *CACM on Operating Systems*, 16(10), October, 1973.
http://citeseer.nj.nec.com/lampson73note.html

[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.
http://www.cis.upenn.edu/~KeyKOS/agorics/KeyKos/keysafe/Keysafe.html

[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.
http://www.cap-lore.com/CapTheory/ProtInf/

[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.
ftp://ftp.cs.vu.nl/pub/papers/amoeba/dcs86.ps.Z

[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 \mbox{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