Capability Myths Demolished

Mark S. Miller
Combex, Inc.
markm@caplet.com

Ka-Ping Yee
University of California, Berkeley
ping@zesty.ca

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

ABSTRACT

We address three common misconceptions about capability-based systems: the *Equivalence Myth* (access control list systems and capability systems are formally equivalent), the *Confinement Myth* (capability systems cannot enforce confinement), and the *Irrevocability Myth* (capability-based access cannot be revoked). The Equivalence Myth obscures the benefits of capabilities as compared to access control lists, while the Confinement Myth and the Irrevocability Myth lead people to see problems with capabilities that do not actually exist.

The prevalence of these myths is due to differing interpretations of the capability security model. To clear up the confusion, we examine three different models that have been used to describe capabilities, and define a set of seven security properties that capture the distinctions among them. Our analysis in terms of these properties shows that pure capability systems have significant advantages over access control list systems: capabilities provide much better support for least-privilege operation and for avoiding confused deputy problems.

INTRODUCTION

This paper is concerned with three misconceptions about capability systems that we believe to be prominent among students, researchers, and practitioners of computer security. Over the course of history, these three myths seem to have achieved the status of common wisdom (we do not assume anyone believes all three simultaneously):

- *The Equivalence Myth:* Access control list (ACL) systems and capability systems are formally equivalent.
- *The Confinement Myth:* Capability systems cannot enforce confinement.
- *The Irrevocability Myth:* Capability-based access cannot be revoked.

The first myth obscures important benefits of capability systems, and is based on the common perception[1] of ACL systems and capability systems as merely alternative perspectives on Lampson's access matrix [15]. The second and third myths state false limitations on what capability systems can do, and have been propagated by a series of research publications over the past 20 years (including [2, 3, 7, 24]). They have been cited as reasons to avoid adopting capability models and have even motivated some researchers to augment capability systems with extra access checks [7, 13] in attempts to fix problems that do not exist. The myths about what capability systems cannot do continue to spread, despite formal results [22] and practical systems [1, 9, 18, 21] demonstrating that they can do these supposedly impossible things.

We believe these severe misunderstandings are rooted in the fact that the term *capability* has come to be portrayed in terms of several very different security models. This paper describes three common interpretations, which we call *Model 2: Capabilities as Rows*, *Model 3: Capabilities as Keys*, and *Model 4: Object Capabilities*. We compare these to *Model 1: ACLs as Columns*.

We will begin by addressing the three myths directly. The Equivalence Myth is based on a comparison of the ACLs-as-columns and capabilities-as-rows models; we will refute this myth by explaining why the comparison it makes is not meaningful and pointing out specific differences between ACL and capability systems in practice. The other two myths concern what is feasible in a capability system; we examine them in the context of the object-capability model, which represents the vast majority of implemented "capability-based" or "pure capability" systems.

We will then examine and compare the three different interpretations of capabilities in more detail, to clear up the confusion among them. In particular, we will identify distinctions between the models in terms of some important security properties, and trace how these

[1] To support the claim that this perception is common, David Wagner has kindly agreed to go on record as having believed that ACL systems and capability systems were roughly equivalent in expressiveness, until convinced otherwise by one of the authors.

interpretations have made the capability myths appear to be reasonable or even obvious statements. Along the way, we will find that these properties not only dispel the myths, but in fact afford capability-based systems with significant advantages that are crucial for secure operation. We will conclude with a discussion of these advantages with respect to the *principle of least privilege* [20] and the *confused deputy problem* [10].

TERMINOLOGY

In order to discuss all of these models together, we need to adopt some consistent terminology. For the following discussion, we will use the word *authority* to mean the ability of a *subject* to access a *resource*. Note in particular that although sometimes the word *object* is used to mean "a thing to which an authority provides access", we instead use the word *resource*. We reserve the word *object* to mean an encapsulation of state and behaviour, as in object-oriented programming.

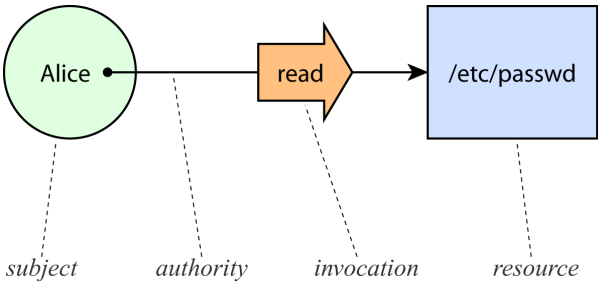

Figure 1. Graphical conventions used in this paper to depict access.

THE EQUIVALENCE MYTH

Our first myth is the common belief that ACLs and capabilities are formally equivalent. This usually stems from being introduced to ACLs and capabilities as different views on an *access matrix*.

The access matrix visualization of authority relationships, described by Lampson [15], is a table indexed by subjects and resources. The cells of the matrix contain *access attributes* that specify the kinds of access each subject has to each resource. For example, consider a simple system with three subjects, three resources, and the permission settings shown in Figure 2.

	`/etc/passwd`	`/u/markm/foo`	`/etc/motd`
Alice	{read}	{write}	{}
Bob	{read}	{}	{read}
Carol	{read}	{write}	{read}

Figure 2. Example of an access matrix.

For each resource, the corresponding column lists all the kinds of access any subjects have to that resource. An ACL system associates each resource with an access control list, which represents a column of this matrix. This organization of the access matrix is our *Model 1: ACLs as Columns*.

For each subject, the corresponding row lists all the kinds of access available to that subject. A capability system associates each subject with a list of capabilities (or C-list), which represents a row of this matrix. This row-based view is our *Model 2: Capabilities as Rows*.

Thus, ACLs and C-lists appear to represent the same information, differing only in whether the information is recorded in a by-column or by-row data structure. If the access matrix constitutes the entire story, then these two models would seem equivalent.

However, such a line of reasoning ignores a crucial factor: the difference between static models and dynamic systems.

Model is Not Dynamic

The access matrix is not usually assumed to include a precise specification of the rules that condition how the cells of the matrix may be updated. Although Lampson's presentation of the access matrix [15] actually did suggest a set of such rules, the lasting legacy of this model – what most everyone remembers and teaches to one generation after another – is the matrix itself as a static representation of access rights. In practice, what most people consider the definition of "ACL system" does not specifically impose Lampson's rules, or any canonical set of rules.[2]

Subtle changes in these rules for authority manipulation can have far-reaching effects on the security properties of the entire model. When the matrix is seen only as a static depiction of security state, the access matrix model becomes more general, which can be an advantage for some modelling purposes. However, no description of any security mechanism is complete without a specification of how access relationships are allowed to evolve over time. Thus, comparing ACL and capability models in terms of the static access matrix alone is insufficient to establish logical equivalence.

In short, the access matrix is indeed general enough to *accommodate* both types of security models, but that does not necessarily make them equivalent.

[2] Many ACL systems share some features with Lampson's rule set, such as an *owner* attribute. However, once such rules are taken into account, it is no longer possible to claim that ACL systems and capability systems are equivalent, because they use different update rules.

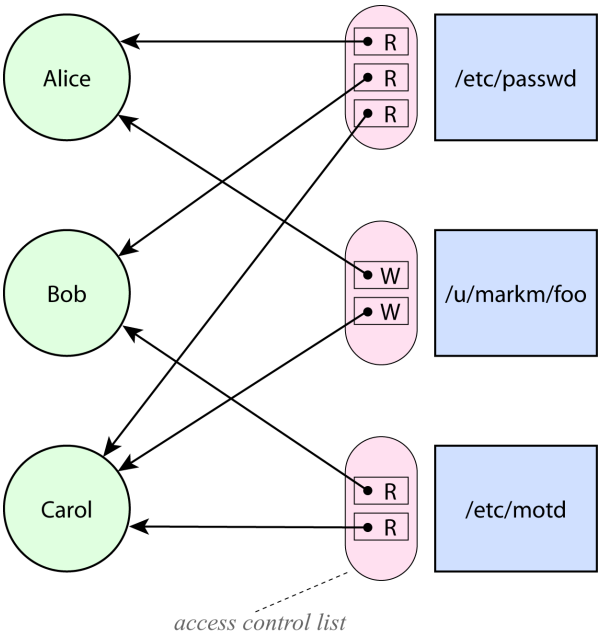

Figure 3. The ACLs-as-columns model, with references drawn as arrows.

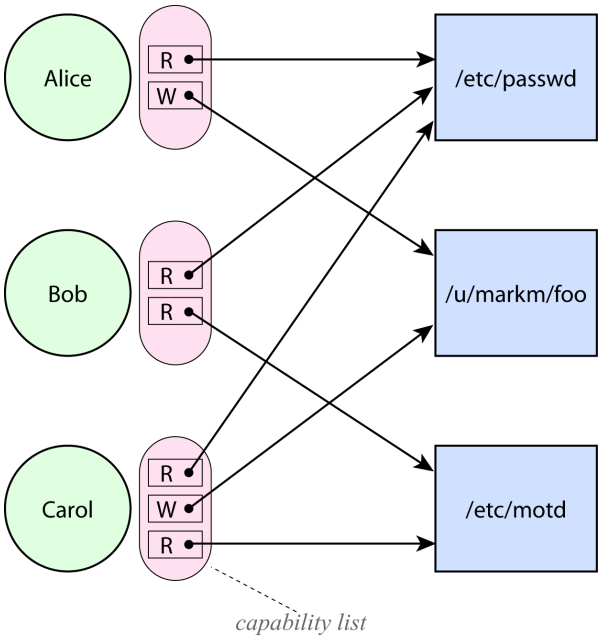

Figure 4. The capabilities-as-rows model, with references drawn as arrows.

A Shift in Perspective

Switching from a column-based to a row-based organization of the access matrix results in at least three real differences between the models. Visualizing the access matrix in another way will help make these differences apparent.

Figure 3 depicts the ACL-as-columns model, this time explicitly identifying the access control lists. The circles on the left are subjects, and the boxes on the far right are resources. Each access control list is shown as a tall oval next to its corresponding resource. When access control lists are stored with the resources, they must contain references to subjects, which are shown as the leftward-pointing arrows.

Now let us compare this to Figure 4, a depiction of the capabilities-as-rows model in the same style. Here, each subject on the left carries a C-list containing capabilities. Each capability has a reference to a resource, which we depict as a rightward-pointing arrow.

By shifting to this new visualization, where the references are explicitly visible, we see a difference between the models that was previously obscured: the direction of the arrows. To see why this matters, let us consider how subjects and resources refer to each other.

Designation and Authority

First, let us look at the arrows from the subject side. The ACL model presumes some namespace, such as the space of filenames, that subjects use to designate resources. This namespace must necessarily be separate from the representation of authority in an ACL system, since the arrows representing authorities point in the opposite direction. Looking only at the matrix without considering the direction of the arrows assumes away the designation problem, which is arguably one of the deepest problems in computer security. In order to use a resource designator, subjects must not only come to know the designator in some fashion; they must also understand their relationship to the designated resource in order to determine what actions they should perform on the resource.

In a capability system, each capability points from a subject to a resource. Consequently every capability can serve both to designate which resource to access, and to provide the authority to perform that access. This change provides us with the option to avoid introducing a shared namespace into the foundations of the model, and thereby avoid the complex issues involved in managing a shared namespace – issues rarely acknowledged as a cost of non-capability models.

We will refer to this distinction as *Property A: No Designation Without Authority*. As we have explained, no ACL system can possibly have this property, whereas a system described by the row-based view of the access matrix may or may not have this property. This property will be relevant to the issue of *confused deputy* problems [10], which we will discuss later in this paper.

Granularity of Subjects

Now, we will consider the arrows from the resource side. Another consequence of the direction of the arrows in the ACL model is that the ACLs also need a namespace in which to refer to subjects. In order for the ACL model to be coherent, the entities that update ACLs must maintain up-to-date knowledge of the set of subjects and their designations. Maintaining such consistency is difficult if subjects are frequently appearing and disappearing.[3] The frequency with which the set of subjects changes depends on the level of granularity at which subjects are distinguished.

Although Lampson's definition of the access matrix does not prohibit the possibility of fine-grained subjects[4], even in an ACL system, the difficulty just explained strongly motivates ACL systems to define subjects at a coarse granularity so as to keep the set of subjects relatively static. Thus, ACL systems in practice map processes to *principals* (broad equivalence classes of processes), such as user accounts[5], where the set of principals does not change in the course of normal operation. These principals then serve as the subjects of the access matrix. Subjects in an ACL system do not generally[6] have the ability to create an unbounded number of new subjects. The appeal of the ACL model rests on the ability of administrators to enumerate subjects whose identities they can know and reason about, such as humans.

In capability systems, a subject corresponds to an *instance* of a software component, such as an *object* (an instance of a class, as in a capability language) or a *process* (an instance of an executable program, as in a capability operating system). Instances are distinguished at a much finer granularity than user accounts – indeed, at finer granularity than users are normally even aware of. Because authorities are aggregated by subject, it is no problem for subjects to be defined at a fine granularity, even when that means subjects are creating new subjects all the time.

We will refer to this distinction as *Property B: Dynamic Subject Creation*. Strictly speaking, neither view of the access matrix imposes or prohibits this property, but we know of no ACL implementation that allows individual processes to be separate subjects, whereas all capability-based systems distinguish subjects at the instance level. The loss of subject granularity is a severe limitation on the expressiveness of ACL systems. This issue will bear on our later discussion of the *principle of least privilege* [20].

Power to Edit Authorities

Though policies for updating authorities vary from system to system, all ACL systems known to us define an *owner* or *edit permissions* attribute, which is set on a resource to give a subject the power to edit that resource's ACL. Consequently the power to edit authorities is aggregated by resource: the ability to change one permission generally comes together with the ability to edit an entire ACL. In capability systems, the power to edit authorities is aggregated by subject: subjects manipulate authorities in their own C-lists.

We will refer to this distinction as *Property C: Subject-Aggregated Authority Management*. As with *Property B*, the access matrix model does not force the presence or absence of this property, but in all cases of which we are aware, capability systems have this property and ACL systems do not.

Intermission

All of these differences between the capabilities-as-rows model and the ACLs-as-columns model should put to rest the Equivalence Myth. Object-capability systems are different from ACL systems in yet more ways, which will surface as we address the other myths.

THE CONFINEMENT MYTH

We come now to the Confinement Myth, which is sometimes stated as "capability systems cannot limit the propagation of authority" or "capability systems cannot solve the confinement problem". For example:

> *A major problem for capability systems is a fundamental inability to solve the confinement problem, ...*
>
> — Karger [13, p. 67]

This view continues to be held as recently as 2001:

> *Delegation in a trust management style of access control provides for bounds on propagation of access rights, a property which doesn't hold true for capabilities.*

---

[3] This situation is especially problematic if the design of the ACL system presumes that ACLs will typically be edited by humans.

[4] In fact, Lampson's presentation [15] explicitly suggested that domains (what we mean by *subjects* here) could be processes, whereupon access control lists would contain *access keys* to identify the domains. However, most implementations of access control lists contain subject names rather than access keys, which makes it hard for them to identify a dynamic, unbounded set of subjects. Lampson touches on the difficulty of handling fine-grained subjects in ACL systems when he writes "the procedure gets the domain's name as argument, and this cannot be forged. However, unique names may not be very convenient for the procedure to remember – access is more likely to be associated with a person or group of people, or perhaps with a program."

[5] Sometimes user accounts are created so that authorities can be specified separately for particular programs, but that is not the same as specifying authorities separately for each individual running instance of a program.

[6] Usually, only an administrator can create new principals.

> *... Trust management, modelled here without keys or name spaces, can strongly (one-to-one) simulate the other mechanisms, providing a tractable compromise between unrestricted capability passing from the capability model and easy revocation provided by access control lists.*
>
> — Chander, Dean, Mitchell [3]

Achieving Confinement

In order to explain the problems with this claim, we need to describe how authority is transferred in most capability systems in practice, where capabilities are object references [6]. We refer to this as *Model 4. Object Capabilities* (Model 3 is described below). The transfer of authority in these systems is depicted in Figure 5.

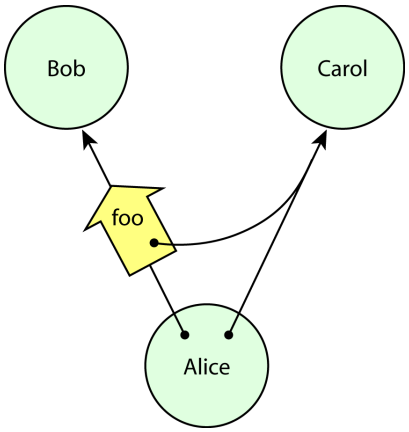

Figure 5. Delegation in object capability systems [8, 4].

In this diagram, Alice, a subject possessing a capability to Carol, is delegating this capability to another subject, Bob. The open arrow on the left represents an *invocation* or *message*. The message is riding on a capability from Alice to Bob, since messages may only travel along capabilities in an object-capability system.

In order for Alice to authorize Bob to access Carol, Alice must herself be authorized to access *both* Bob and Carol. The requirement for both capabilities makes confinement possible, since no capability transfer can introduce a new connection between two objects that were not already connected by some path. Confinement of authorities within a set of objects can be determined simply by observing that the subgraph containing the set of objects is not connected to the rest of the object graph.

Suppose, for example, we decide not to trust Bob. To prevent Alice from delegating to Bob, we simply refrain from giving Alice access to Bob. No subgraph of co-operating objects can delegate to Bob if Bob is not reachable from that subgraph to begin with. This simple insight is the basis for many capability-based confinement systems [9, 21, 25, 26].

Karger noted in 1988 that KeyKOS did, in fact, achieve confinement:

> *KeyKOS achieves confinement by a mechanism called factories. Essentially, a factory is a mechanism for creating new instances of protected subsystems.*
>
> — Karger [13, p. 75]

The discussion of KeyKOS goes on to compare its performance to that of another capability system, noting that occasionally KeyKOS requires two calls and returns instead of one. Regardless of the benchmarks, however, the fact remains that confinement is indeed achievable in a capability system.

It is unfortunate that the Confinement Myth continues to scare people away from capabilities so long after KeyKOS succeeded in confining programs. The myth was further amplified by an influential and widely cited paper on extensible security architectures for Java:

> *However, an important issue is confinement of privileges [26]. It should not generally be possible for one program to delegate a privilege to another program (that right should also be mediated by the system). This is the fundamental flaw in an unmodified capability system; two programs which can communicate object references can share their capabilities without system mediation. This means that any code which is granted a capability must be trusted to care for it properly. In a mobile code system, the number of such trusted programs is unbounded. Thus, it may be impossible to ever trust a simple capability system.*
>
> — Wallach, Balfanz, Dean, Felten [24]

Note that the above makes a slightly different error with respect to the confinement issue. Rather than stating that capability passing is unrestricted, it correctly acknowledges that the ability to communicate object references is necessary for two programs to share their capabilities. However, it appears to overlook the possibility that such an ability to communicate might be unavailable or restricted.

We will examine some possible reasons for the persistence of the delegation myth in more detail below, but let us first address the third myth before entering into that discussion.

THE IRREVOCABILITY MYTH

The third myth states that capabilities cannot revoke access. This is also quite widely believed, for example:

> *Capability systems modelled as unforgeable references present the other extreme, where delegation is trivial, and revocation is infeasible.*
>
> — Chander, Dean, Mitchell [3]

This belief stems from the fact that, once a subject holds a capability, no one but the subject can remove that capability, not even the creator of the corresponding resource. It is true that capabilities themselves are not literally revocable. Further, we know that the capability alone is sufficient to establish access to the resource. These two facts might lead one to reasonably believe that there is no opportunity to revoke access.

Revocable Access in Object-Capability Systems

In an object-capability system, however, capabilities can be composed in such a way as to provide revocable access. Suppose once again that Alice wants to give Bob access to Carol, but Alice also wants to have the option to revoke this access at some time in the future. To accomplish this, Alice could simply create a pair of forwarders, F and R, connected as shown in Figure 6. Of this pair, we may call F the *forwarding facet*, and R the *revoking facet*. Alice would send Bob access to F, and retain R for herself. Any messages sent to F get forwarded through R to Carol, so Bob may use F as if it were Carol. This works as long as inter-object interactions are mediated by messages, and messages are handled generically, so that a reusable mechanism can forward any message.

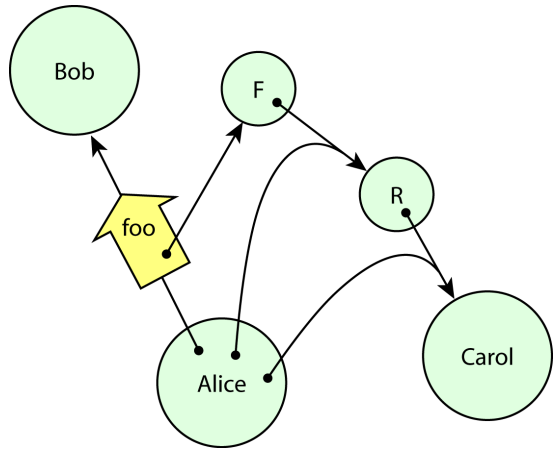

Figure 6. Alice provides Bob with revocable access to Carol.

When Alice wants to revoke Bob's access to Carol, she invokes R, telling it to stop forwarding. R then drops its pointer to Carol, and F becomes useless to Bob. Notice that no capabilities were themselves revoked, which is the truth supporting the myth. Bob still has access to F. However, access to F is now useless. Alice hasn't revoked Bob's capability itself, but she has revoked the access to Carol represented by that capability.

This revocation mechanism also works for delegated authorities. Suppose Bob delegates to Ted his access to Carol. Since Bob only ever has access to F, not to Carol herself, Bob can only give F to Ted (and Bob can only do so if Bob also has access to Ted). When Alice invokes R in order to disable F, this action prevents further access to Carol by Ted, or by anyone to whom Bob sent his capability, just as much as it prevents access by Bob.

This scheme is not a recent invention. Redell described exactly this method for revoking access in 1974 [18], and it was later implemented in the CAP-III system [12]. Karger addresses the concern that "indirection through a large number of revoker capabilities could adversely affect system performance" by suggesting that "a properly designed translation buffer, however, could cache the result of the capability indirections and make most references go quickly" [13, p. 111]. KeyKOS [9] and EROS [21] both employ this optimization technique.

ORIGINS OF THE MYTHS

Now we will examine the history of the last two myths, in order to understand the thinking that led to them.

An Argument Against Confinement

Our story begins in 1984, when Boebert presented a claim [2] that no *unmodified capability machine* can enforce the **-Property* (explained below). His paper defined an *unmodified capability machine* as one in which "capabilities are the sole mechanism for controlling access by programs to storage objects", and defined a capability as a distinguished object containing two parts: a reference to a storage object and an access mode such as *read* or *write*. His argument has been cited as evidence that capability systems cannot solve the confinement problem, for example:

> *Boebert made clear in [1] that an unmodified or classic capability system can not 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 [7]

(Gong's citation [1] is Boebert's 1984 paper, which corresponds to our citation [2]).

Because Boebert's argument [2] had such tremendous influence on later research, we will pay special attention to it here. The argument concerns the problem of enforcing two rules called the *Simple Security Property* and the **-Property*. The rules suppose a world in which subjects are assigned *clearance levels* (representing the trustworthiness of each subject) and resources are assigned *classification levels* (representing the sensitivity of the information they contain). The Simple Security Property requires that subjects have read access only to resources classified at or below their clearance level. The *-Property requires that subjects have write access only to resources classified at or above their clearance level. Together, these two rules confine information at higher classification levels and prevent it from escaping to lower levels.

Boebert's example consists of a capability system with two levels, where there is a subject (Alice) and a memory segment (Low) at the lower level, and a subject (Bob) and a memory segment (High) at the upper level. The two confinement rules are implemented by an oracle that grants capabilities of the appropriate modes to anyone requesting access, based on its special knowledge of the clearance of the subject and the classification of the object. The oracle grants read capabilities whenever the object's classification is at the same level or lower than the subject's clearance. The oracle grants write capabilities whenever the object's classification is at the same level or higher than the subject's clearance.

To make the argument easier to illustrate, we will depict subjects as carrying all the capabilities that such an oracle would have granted them. Figure 7 shows the starting condition corresponding to the oracle Boebert described.

In the attack that Boebert describes, Alice takes her Write-Low capability and writes it into the Low segment, as in Figure 8.

Then, Bob can read this capability out of the Low segment, as in Figure 9.

At this point, Bob can violate the *-Property by reading sensitive information out of the High segment and using his newly-acquired write capability to leak the information into the Low segment.

Boebert concludes correctly that the *particular* use of capabilities he described does not enforce the *-Property. However, the argument assumes that subjects can transmit capabilities anywhere they can transmit data, which is not the case in most capability systems.

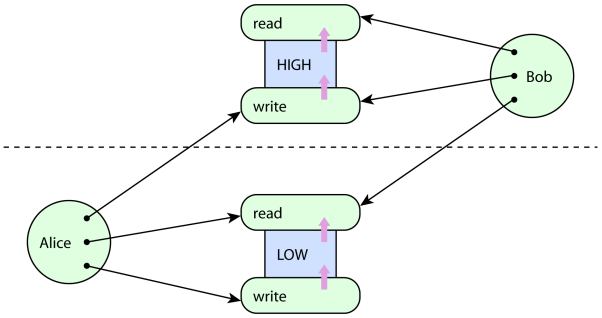

Figure 7. Initial condition for Boebert's attack.

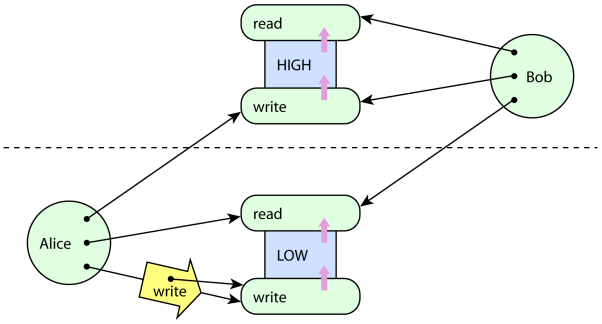

Figure 8. Alice writes her Write-Low capability into the Low segment.

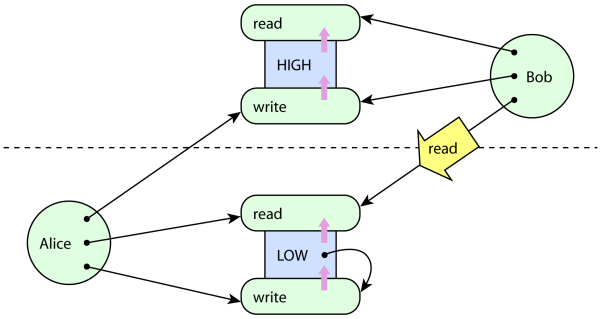

Figure 9. The Write-Low capability now resides in the Low segment, which Bob can read.

In partitioned or type-enforced capability systems such as KeyKOS [9], W7 [19], EROS [21], or E [4], capabilities and data are distinguished by the kernel or runtime. Reading and writing capabilities are necessarily distinguishable operations from reading and writing data. Nearly all capability systems make this distinction.

Suppose now that each of the two classification levels has separate read capabilities for capabilities and data, and separate write capabilities for capabilities and data. The oracle now hands out capabilities of both kinds, much as before, with the exception that it does not hand out capabilities that permit reading or writing capabilities between different levels.

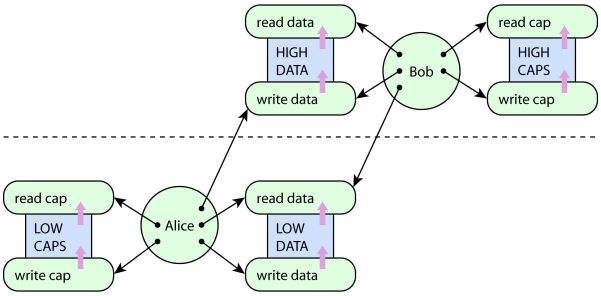

Figure 10. Enforcement of the *-Property in a pure capability system.

If we show each subject holding all the capabilities that it could possibly obtain from the oracle, we now have Figure 10.

Since no capability-carrying capabilities cross between levels, no capabilities can cross between levels. The only capability Bob holds to a lower level is a read capability, so the *-Property is enforced. The only capability Alice holds to a higher level is a write capability, so the Simple Security Property is enforced.

The result is an unmodified capability system (satisfying Boebert's own definition) that is perfectly capable of enforcing both the Simple Security Property and the *-Property.

The claim that capability systems *in general* cannot enforce the *-Property appears to be based on the misunderstanding that capabilities and data are not distinguishable. A similar misunderstanding is the belief that capabilities are equivalent to bit strings:

> *Since a capability is just a bit string, it can propagate in many ways without the detection of the kernel or the server....*
>
> *Generally a capability is a bit string and can propagate in many ways without detection...*
>
> — Gong [7]

The equivalence of capabilities and bit strings is a feature only present in a specialized category of capability systems known as *password capability systems*, such as Amoeba [17]. Boebert's result is valid in any capability system that cannot distinguish between data transfer and capability transfer. But partitioned and type-enforced capability systems do not have this problem, and password capability systems have been engineered to avoid this problem [1, 11]. Moreover, it has been formally verified that any capability system enforcing independent controls on data transfer and capability transfer can enforce both confinement and the *-Property [22].

The Capabilities-as-Keys Model

Sometimes capabilities are explained using the analogy that capabilities are like copyable, unforgeable keys (or copyable, unforgeable tickets) in the real world. This analogy is our *Model 3. Capabilities as Keys*. It turns out that the Confinement Myth and the Irrevocability Myth are both true statements within such a model, so we believe this model is a likely contributor to the propagation of these myths. Let us take apart the key analogy carefully to see how it compares with the object-capability model.

As the story goes, each resource is protected by a lock, perhaps a locked door. People carry keys on their key rings, and if they want to delegate authority, they can make copies of their keys to give to other people. The analogy usually assumes that they can give copies of their keys to anyone they like. To exercise an authority, a person must make two choices: a choice of key, and a choice of lock. The access attempt succeeds or fails depending on whether these choices put together a compatible key and lock.

As with object-capability systems, the keys are assumed to be unforgeable. The only way someone can get a key to a resource is either by being its creator, or if someone else who already has the key decides to give them a copy of the key. Subjects are also assumed to be encapsulated: if someone does not voluntarily decide to hand out a copy of a key, no one can steal it.

Note that one can hold a key without knowing the door to which it belongs, and one can also designate a door without holding the key that opens it. The analogy does not usually specify whether one chooses a particular key to use before seeking the appropriate door, or whether, upon encountering a locked door, one then tries all the keys on the key ring to find a key that works. This decision will be pertinent to the discussion of the confused deputy problem, below.

We will now compare this model to the other two capability-like models we have discussed.

Keys versus Rows: Ambient Authority

There is a crucial difference between the capabilities-as-keys model and the capabilities-as-rows model. In the key model, exercising an authority requires the selection of a key, but the capabilities-as-rows model has no such requirement.

We will use the term *ambient authority* to describe authority that is exercised, but not selected, by its user. In an ambient authority system, subjects are not required to indicate a specific authority in order to exercise it. The corresponding analogy is to imagine a world with doors but without keys. When a person walks up to a door, the door magically opens if it

deems the person worthy. For example, Unix filesystem permissions constitute an ambient authority mechanism, because the caller of a function such as `open()` does not choose any credentials to present with the request; the request merely succeeds or fails.

We will refer to the requirement that subjects select the authorities they exercise as *Property D: No Ambient Authority*. This distinction will also be relevant in our later discussion of the confused deputy problem.

Keys versus Objects

Although the capabilities-as-keys analogy is an appealing way to tell stories about capabilities, it is not an accurate representation of the object-capability model. In Figure 11, which represents the capabilities-as-keys model, the ability to read `/etc/motd` is behind a locked door. To read it, Alice must request access to `/etc/motd` for reading and present the key that unlocks the door. The diagram shows that Alice possesses such a key, represented by the thin black arrow pointing from Alice to the resource.

The wide arrow on the left represents a message from Alice to Bob, in which Alice sends Bob a copy of her key. Once Bob receives it, he will be able to read `/etc/motd` as well. By giving Bob a copy of her key, Alice authorizes Bob to read `/etc/motd`. The wide arrow containing the black arrow tail represents this act of authorization.

In the capabilities-as-keys model, subjects *authorize* subjects (the wide arrow on the left), whereas subjects *access* resources (the wide arrow carrying the operation name *read*), and these are two distinct kinds of action. As long as subjects and resources are partitioned into two separate type categories, authorization and access cannot be unified because the types of these two operators do not match.

Now compare Figure 11 to Figure 12, which illustrates authorization and access in the object-capability model. Three things are different.

First, the resource box has been replaced by a circle labelled Carol, to depict the fact that there is no artificial separation between subjects and resources. Every subject is a resource, and every resource is conceptually a subject (though some resources, such as the number 3, are primitively provided). Consequently, access and authorization can be unified. This uniformity makes the object-capability model *compositional*; networks of authority relationships can be composed to any depth. We will refer to this property as *Property E: Composability of Authorities*.

Second, the message from Alice to Bob is now riding on a capability from Alice to Bob. In the capabilities-as-keys model this access is not necessary (Alice can give her keys to anyone), but in the object-capability model such access is a prerequisite for delegation. We will refer to this requirement as *Property F: Access-Controlled Delegation Channels*.

Third, the message from Alice to Bob is labelled with an operation, *print*. (We imagine that Bob might be a printer, and Alice is asking Bob to print information from Carol.) Since authorization is just a form of access in this model, authorizations are conveyed in the context of a request. This will be relevant to our later discussion of the confused deputy problem.

Observe that without *Property E*, we cannot construct revocable forwarders to solve the revocation problem. And it is precisely the restriction of *Property F* that enables confinement. So, it is easy to see how staying within the capabilities-as-keys model could lead one to believe both the Irrevocability Myth and the Confinement Myth.

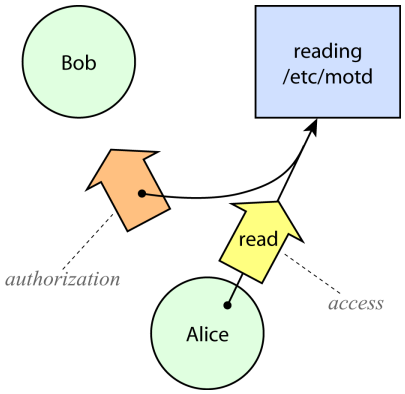

Figure 11. Authorization and access in the capabilities-as-keys model.

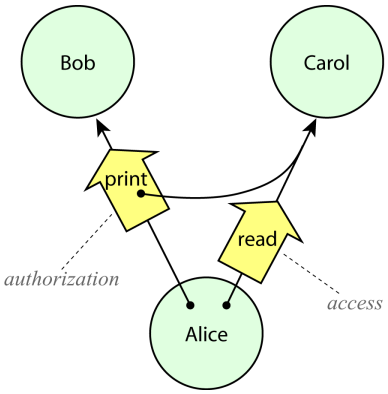

Figure 12. Authorization and access in the object-capability model.

Property	Test	Model 1. ACLs as columns	Model 2. capabilities as rows	Model 3. capabilities as keys	Model 4. object capabilities
A. No Designation Without Authority	Does designating a resource always convey its corresponding authority?	no (impossible)	unspecified (possible)	no	yes
B. Dynamic Subject Creation	Can subjects dynamically create new subjects?	no (in practice)	yes (in practice)	yes	yes
C. Subject-Aggregated Authority Management	Is the power to edit authorities aggregated by subject?	no (in practice)	yes (in practice)	yes	yes
D. No Ambient Authority	Must subjects select which authority to use when performing an access?	no	no	yes	yes
E. Composability of Authorities	Are resources also subjects?	unspecified	unspecified	no	yes
F. Access-Controlled Delegation Channels	Is an access relationship between two subjects X and Y required in order for X to pass an authority to Y?	unspecified	unspecified	no	yes

Figure 13. Comparison of the four security models. Model 1 represents ACLs in practice and approximately as commonly understood. Model 4 represents all the major capability systems in practice, in which capabilities are object references. Models 2 and 3 represent two different ways in which capabilities are sometimes explained: according to a naive static view of Lampson's access matrix ("capabilities as rows"), and according to the metaphor that capabilities are like unforgeable physical keys in the real world ("capabilities as keys").

SUMMARY OF THE FOUR MODELS

We have now described four different security models and identified several properties that distinguish them. These are summarized in Figure 13.

Chander et al. have presented and compared three of these models [3], which they identified as M_{acl}, M_{Crow}, and M_{Cref}. They call M_{acl} "access control lists", which corresponds to our Model 1, and call M_{Crow} either "Lampson matrix capabilities" or "capabilities as rows", which corresponds to our Model 2. They describe M_{Cref} alternately as "capabilities as unforgeable bit strings", "capabilities as tickets", or "capabilities as references", but M_{Cref} actually corresponds to our Model 3, since their definition of the Pass operation allows subjects to pass their capabilities without restriction (that is to say, M_{Cref} lacks *Property F*). Model 4 (capabilities as object references), which describes most of the capability systems in practice, is missing from their analysis.

SYSTEMS IN PRACTICE

For comparison, we will now look at some examples of security mechanisms in practice to see how they measure up against these properties.

Unix and NT Filesystems

Representative real-world systems that fit the ACLs-as-columns model include the standard Unix filesystem, Unix with a `setfacl()` extension, and the NT ACL system. All three of these mechanisms lack all six of Properties A through F.

POSIX Capabilities

The "POSIX capabilities" mechanism described in POSIX 1003.1e might seem to be a representative system for capabilities-as-rows, since authorities are indeed aggregated at the subject. Indeed, with respect to all of the six properties we have defined so far, the "POSIX capabilities" mechanism fits the capabilities-as-rows model. However, there is one very important difference: the set of resources (POSIX capability flags) is finite and fixed.

At the level of detail that we typically care about (individual files, programs, and so on) resources are created and destroyed all the time. There is a bounded set of POSIX capability flags only because they do not express authorities at this level of detail. For example, one of the POSIX capability flags is `CAP_CHOWN`, which represents the power to change the ownership of *any file on the entire system*. Just as with the subject granularity issue, we can consider this a granularity issue with respect to resources, where the ability to dynamically create new resources is the clear dividing line between "fine-grained" and "coarse-grained".

We will refer to this new distinction as *Property G: Dynamic Resource Creation*. All of the other models and examples mentioned here have this property, but the "POSIX capabilities" mechanism does not.

POSIX capability flags can be transmitted to a new process when it is created, but the ability to start new processes requires access to executable files. Such access depends on file permissions rather than POSIX capabilities, and is also affected by the view of the

filesystem (which a call to `chroot()` may have altered). Thus, with respect to *Property F: Delegation on Access-Controlled Channels*, delegation can be somewhat limited, but the complexity of Unix filesystem access makes the question of confinement unclear.

SPKI

SPKI [5] is a reasonable real-world representative for the capabilities-as-keys model. In SPKI, an authority is a signed certificate carried by a subject. The certificate specifies the resource and the kind of access, and the existence of a valid signature on the certificate conveys the authorization.

Authorities (certificates) are always bound to resource designators (names) in SPKI, because resource designators are embedded in the signed certificates. If the certificate is altered to break this binding, the signature becomes invalid. However, designators are not always bound to authorities. The designators are S-expressions that describe a path to the resource, and do not in themselves convey any authority.

Just as with the key analogy, authority in SPKI is not ambient: a subject must choose and present a certificate as part of an attempt to access any resource. Also as with the key analogy, propagation of SPKI certificates is unrestricted; the holder of a certificate may give a copy of that certificate to any other party.

Nothing prevents the construction of entities that can both accept and hold SPKI certificates, so, strictly speaking, authorities are composable in this scheme. However, we have not heard of this kind of indirection being done in practice; it is not clear whether such a use of SPKI could be made practical.

Unix File Descriptors

File descriptors in Unix are nearly, but not quite, equivalent to object capabilities. Since they are sometimes cited as an examples of capabilities, it is useful to compare them to the models we have discussed. Although files themselves cannot wield other file descriptors, a file descriptor could designate a pipe to another process that wields file descriptors. So file descriptors are composable as long as they are limited to the functionality of a pipe (e.g. pipes do not support random access). File descriptors differ from object capabilities in that the channel for transmitting file descriptors among processes (a Unix socket) is controlled by an ACL, not by a capability-like mechanism. Thus, while the file descriptor mechanism is similar to object capabilities in some ways, confining the propagation of file descriptors depends on the details of the ACL system.

Pure Capability Systems

A large number of capability systems in the history of security research have all seven of the security properties we have mentioned, and thus fit Model 4. These systems include Dennis and Van Horn's Supervisor, CAL-TSS, CAP, and Hydra (all described in Levy's survey of capability systems [16]), KeyKOS [9], W7 [19], Mungi [11], EROS [21], and E [4], as well as the password capability system by Anderson, Pose, and Wallace [1].

ADVANTAGES OF OBJECT-CAPABILITY SYSTEMS

Least-Privilege Operation

An essential design requirement for secure systems is the *principle of least privilege* [20]: every entity should operate using the minimal set of privileges necessary to complete its task. In terms of authority relationships between subjects and resources, we can look at this principle from two perspectives. Operating in least-privilege fashion demands that we provide access to minimal resources, and that we grant such access to minimal subjects.

In the course of comparing security models and systems so far, we have already encountered both granularity issues. *Property B: Dynamic Subject Creation* is necessary for limiting authority when starting new running instances of software components. In order for subjects to be able to create instances with limited authority, each instance must have its own separate set of authorities, and must therefore be a distinct subject. For example, *Property B* allows a user to invoke a program while granting it only the subset of the user's authority that it needs to carry out its proper duties. *Property G: Dynamic Resource Creation* is necessary in order for the model to be able to express access restrictions on objects (such as individual files) that can be created and destroyed. No security model limited to controlling a static set of resources can possibly have sufficient expressive detail to support least-privilege operation on a dynamic system.

So both *Property B* and *Property G* are necessary (though not sufficient) for least-privilege operation. The three capability-like models offer both of these properties, whereas the ACL model is missing *Property B*. The POSIX capabilities mechanism bears a weak resemblance to capability models but lacks *Property G*, so it cannot (on its own) support least-privilege operation.

Avoiding Confused Deputy Problems

A *deputy* is a program that must manage authorities coming from multiple sources. A *confused deputy* [10] is a deputy that has been manipulated into wielding its authority inappropriately. A frequent challenge in

computer security is to construct deputies that cannot be confused. Confused deputy problems are a common class of security incidents in many systems, including the World-Wide Web [23].

The classic story of the confused deputy [10] concerns a compiler in an ambient authority system.[7] The compiler is granted write access to a file called `BILL` in order to store billing information. Upon invoking the compiler, the user can specify the name of a file to receive debugging output. If the user specifies `BILL` as the name of the debugging file, the compiler is fooled into overwriting the billing information with debugging information.

The problem is not caused by the compiler using access that it should not have. The problem is that it exercises its authority to write to `BILL` for the wrong purpose.

While no security model can prevent people from writing bad programs, certain properties of the security model can have a profound effect on our likelihood of writing reliable programs. Let us consider the confused deputy problem with respect to two properties we have identified: *Property D: No Ambient Authority* and *Property A: No Designation Without Authority*.

Ambient Authority

The question of ambient authority determines whether subjects can identify the authorities they are using. If subjects cannot identify the authorities they are using, then they cannot associate with each authority the *purpose* for which it is to be used. Without such knowledge, a subject cannot safely use an authority on another party's behalf.

Suppose that we return to the compiler story with this property in mind. If the authority to write to `BILL` were not ambient, then the compiler could hold one key to `BILL` for the purpose of writing billing information, and accept another key from the user for the purpose of writing debugging information. Then, as long as the compiler uses each key for its intended purpose, the confused deputy problem cannot occur. The lack of distinguishable keys would prevent the compiler from having any way to draw this distinction.

Eliminating ambient authority helps make it *possible* to avoid confused deputies, but doesn't guarantee that deputies will never be confused. We mentioned earlier that it matters whether one chooses a key to use before attempting to open a door, or whether one goes to a door and then tries all available keys to find one that works. Even if one can distinguish the keys, deciding to try all available keys puts one at risk of becoming a confused deputy.

In order to avoid the confused deputy problem, a subject must be careful to maintain the association between each authority and its intended purpose. Using the key analogy, one could imagine immediately attaching a label to each key upon receiving it, where the label describes the purpose for which the key is to be used. In order to know the purpose for a key, the subject must understand the *context* in which the key is received; for example, labelling is not possible if keys magically appear on the key ring without the subject's knowledge.

Separable Designators

We mentioned earlier that designating resources is a tricky problem when designators are separated from authorities. When designators and authorities take separate paths through a system, their recombination is likely to lead to confused deputies.

Looking again at the scenario of the compiler as confused deputy, we see that an authorization given by one party is used to access a resource designated by a different party, bringing about an unintended transfer of authority. In ACL systems, because designation and authorization are necessarily separated, this confusion is difficult to escape. In a system where designation and authority are inseparable, this common type of confused deputy problem – in which a malicious party designates a resource they are not supposed to access – simply cannot occur.

In addition, if resource designations can never be separated from authorities, any request asking the deputy to access a resource necessarily includes the corresponding authorities, and places those authorities in the context of the request. This helps to provide the aforementioned context that a deputy needs in order to determine the proper purpose for each received authority.

Any request for access to a resource must designate the resource in some way. If designators are inseparable from authorities, any request for access must necessarily include the authority, which means that any subject requesting access always chooses the authority to exercise. So the presence of *Property A* implies the presence of *Property D*.

Figure 14 summarizes the arguments regarding the confused deputy issue in this section.

[7] Actually, the story we tell here is a simplified version of the original. In the original story, the compiler is given write access to a directory containing the billing file for the purpose of writing a different file. With respect to the confused deputy problem, the point is the same.

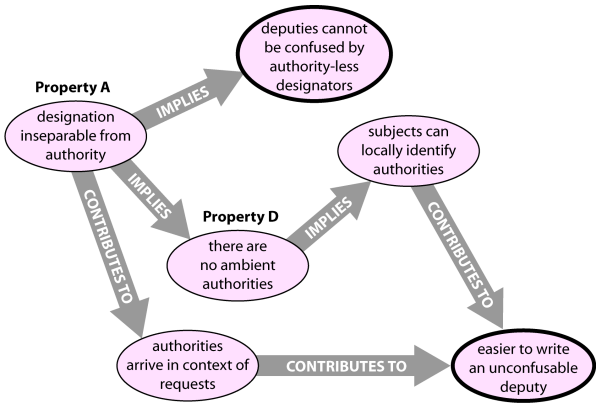

Figure 14. Design factors related to confused deputies.

Object-capability systems possess both *Property A* and *Property D*, so they enforce the combination of designation with authority, enable the assignment of local identifiers to authorities, and encourage the presence of context when authorities are conveyed. All three of these things contribute to establishing a chain of designation, running from the original creator of a resource, through the entity that exercises the resource, and finally to the resource itself. Maintaining this unbroken chain of designation greatly improves our ability to reason about the behaviour of trusted programs.

SPKI is an interesting case to analyze in these terms. Each certificate comes with an embedded nonlocal resource designator in cleartext, so designators can be separated from authorities. However, the construction of unconfusable deputies can still be feasible so long as (a) certificates arrive in the context of a request (in particular, an *undamaged* request) and (b) subjects maintain local identifiers for the certificates they hold. SPKI does not specifically provide for a certificate or tuple of certificates to be securely bound together with a message so that the receiver can determine the intended purpose for the conveyed authorities. Such a feature would be necessary to enable (a).

SUMMARY OF SYSTEMS AND PROPERTIES

The table in Figure 15 summarizes all of the models and systems we have mentioned, evaluating them in terms of the seven security properties, the two model-specific myths, and the issues of confused deputies and least-privilege operation. The columns are arranged roughly along a spectrum from ACLs to capabilities.

Property	**Model 1.** ACLs as columns	Unix fs, `setfacl()`, NT ACLs	**Model 2.** capabilities as rows	POSIX capabilities	**Model 3.** capabilities as keys	SPKI	Unix file descriptors	**Model 4.** object capabilities	CAP, Hydra, KeyKOS, W7, EROS, E, etc.
A. No Designation Without Authority	no (impossible)	no	unspecified (possible)	no	no	no	yes	yes	yes
B. Dynamic Subject Creation	no (in practice)	no	yes (in practice)	yes	yes	yes	yes	yes	yes
C. Subject-Aggregated Authority Management	no (in practice)	no	yes (in practice)	yes	yes	yes	yes	yes	yes
D. No Ambient Authority	no	no	no	no	yes	yes	yes	yes	yes
E. Composability of Authorities	unspecified	no	unspecified	no	no	possible, but unusual	yes, as pipes	yes	yes
F. Access-Controlled Delegation Channels	unspecified	no	unspecified	yes, but by ACL	no	no	yes, but by ACL	yes	yes
G. Dynamic Resource Creation	yes	yes	yes	no	yes	yes	yes	yes	yes
Consequence									
Irrevocability Myth (holds if B but not E)	false	false	true	true	true	false	depends	false	false
Delegation Myth (holds if B but not F)	false	false	true	unclear	true	true	unclear	false	false
Confused Deputy (hopeless without D, best if A)	danger	danger	danger	danger	better	better	best	best	best
Least Privilege (requires B and G)	infeasible	infeasible	better	infeasible	better	better	better	better	better

Figure 15. Comparison of various systems and models with respect to the seven security properties.

A NOTE ON THE WORD "CAPABILITY"

Given these various interpretations of the capability model, the reader may wonder what one should adopt as the most legitimate meaning for the term *capability*. We should also explain why we feel justified in declaring the Irrevocability Myth and Confinement Myth to be false, rather than merely false in certain cases. We would argue that the "true" capability model is the object-capability model, because all known major capability systems take the object-based approach (for examples, see [1, 4, 9, 11, 16, 17, 19, 21]). In all of these systems, a capability is an object reference – not something that behaves like a key or ticket in the real world. Definitive books on capability-based systems [6, 16] also describe these systems from the object-capability perspective, and explicitly characterize them as "object-based".

We know of no security mechanisms outside of the object-capability model that have described themselves using the word *capability* except for "POSIX capabilities", "Netscape capabilities", and "split capabilities" [14]. POSIX capabilities are not generally described as "capability-based security". The "Netscape capabilities" extensions to Java were fairly short-lived and have not been presented in the research literature as a capability system. Moreover, both "POSIX capabilities" and "Netscape capabilities" have never been presented as security mechanisms that can stand on their own, instead only serving as an extension to existing security systems. The split capabilities model is explicitly presented in contrast to the pure capability model [14].

CONCLUSION

We have described a progression of four security models from traditional ACLs to pure capabilities, while defining a set of seven properties that can be used to distinguish the models. We have also used the properties to evaluate and compare some real-world security systems that resemble the models.

The distinctions that we have drawn support our refutations of three common misconceptions concerning capability-based systems – the Equivalence Myth, the Confinement Myth, and the Irrevocability Myth. Although the myths have some truth in the intermediate security models that are often taken as interpretations of capabilities, they do not hold for the "pure capability" or "object-capability" model represented by the vast majority of capability systems. Furthermore, the properties we identified show that capability systems lack certain fatal flaws of ACL systems – namely, the susceptibility of ACLs to the confused deputy problems that are inherent in ambient authority systems, and the inability of ACLs to perform least-privilege delegation to new processes. Capability-based systems provide much stronger support for the precise, minimal, and meaningful delegation of authority, which is fundamental to secure operation.

ACKNOWLEDGEMENTS

Many thanks to Darius Bacon, Sue Butler, Tyler Close, Hal Finney, Bill Frantz, Norm Hardy, Chris Hibbert, Alan Karp, Ben Laurie, Terry Stanley, Marc Stiegler, E. Dean Tribble, Bill Tulloh, and Zooko for their detailed comments and assistance with this paper.

REFERENCES

1. M. Anderson, R. Pose, C. S. Wallace. A Password Capability System. The Computer Journal, 29(1), 1986, p. 1–8.
2. 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. Online at: http://zesty.ca/capmyths/boebert.html
3. A. Chander, D. Dean, J. C. Mitchell. Proceedings of the 14th Computer Security Foundations Workshop, June 2001, p. 27–43.
4. The E Language: Open Source Distributed Capabilities. http://erights.org/.
5. C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, T. Ylonen. SPKI Certificate Theory. IETF RFC 2693. Online at: http://www.ietf.org/rfc/rfc2693.txt
6. E. F. Gehringer. Capability Architectures and Small Objects. UMI Press, 1982.
7. L. Gong. A Secure Identity-Based Capability System. Proceedings of the 1989 IEEE Symposium on Security and Privacy, p. 56–65.
8. M. Granovetter. The Strength of Weak Ties. American Journal of Sociology 78, 1973, p. 1360–1380.
9. N. Hardy. The KeyKOS Architecture. ACM Operating Systems Review, September 1985, p. 8–25. Online at: http://www.agorics.com/Library/KeyKos/architecture.html
10. N. Hardy. The Confused Deputy (or why capabilities might have been invented). Operating Systems Review 22(4), October 1988, p. 36–38.
11. G. Heiser, K. Elphinstone, S. Russel, J. Vochteloo. Mungi: A Distributed Single Address-Space Operating System. Proceedings of the 17th Australasian Computer Science Conference, p. 271–280.
12. A. J. Herbert. A Microprogrammed Operating System Kernel. Ph. D. thesis, University of Cambridge Computer Laboratory, September 1982.
13. P. Karger. Improving Security and Performance for Capability Systems. Technical Report 149, University of Cambridge Computer Laboratory, 1988. (Ph. D. thesis.)
14. A. H. Karp, R. Gupta, G. J. Rozas, A. Banerji. Using Split Capabilities for Access Control. IEEE Software 20(1), January 2003, p. 42–49.

15. B. Lampson. Protection. Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems, 1971, p. 437–443.

16. H. Levy. Capability-Based Computer Systems. Digital Press, Bedford, Massachusetts, 1984. Online at: http://www.cs.washington.edu/homes/levy/capabook/

17. 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. Online at: ftp://ftp.cs.vu.nl/pub/papers/amoeba/dcs86.ps.Z

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

19. J. Rees. A Security Kernel Based on the Lambda-Calculus. Technical Report AIM-1564, MIT, March 1996. (Ph. D. thesis.)

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

21. 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.

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

23. K. Sitaker. Thoughts on Capability Security on the Web. Online at: http://lists.canonical.org/pipermail/kragen-tol/2000-August/000619.html

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

25. D. Wagner, D. Tribble. A Security Analysis of the Combex DarpaBrowser Architecture. Online at: http://www.combex.com/papers/darpa-review/

26. K.-P. Yee, M. S. Miller. Auditors: An Extensible, Dynamic Code Verification Mechanism. Online at: http://www.erights.org/elang/kernel/auditors/index.html