

Split Capabilities for Access Control

Alan H. Karp, Rajiv Gupta, Guillermo Rozas^{*}, Arindam Banerji
Hewlett-Packard Labs
Palo Alto, California

Abstract

The access control mechanisms we use when sharing resources over the Internet were designed in the days when networking computers was a rarity. Many of the security breakdowns occurring today come from the resulting mismatch between today's realities and the assumptions made in designing those mechanisms. Although Access Control Lists are the most commonly used mechanism, Capability Lists have a number of advantages when used in a distributed environment. However, traditional capabilities have problems with revocation and scalability that we address with our implementation of split capabilities.

1. Introduction

The fundamental problem of access control is to limit what a process can do to an object when.¹ For example, whether or not to honor a request to read or write a particular file is the question that any access control mechanism² must be able to answer. Specifying the access policy is conventionally described as one of populating an *access control matrix*, which has a row for each resource and a column for each user. An element of this matrix is the set of access rights on that resource being granted to that user.

The access control matrix is sparsely populated, so people developed two representations, *access control lists* (ACLs) and *capability lists* (CLs). An ACL has an entry for each resource containing a list of the access rights for each user. A CL is a list of capabilities. Each capability specifies access rights being granted to a resource. ACLs are held by the computing infrastructure, called the Trusted Computing Base (TCB), often the operating system kernel but sometimes the entity controlling the resources. CLs are held by the user's process.

There are several problems when using ACLs in a networked environment [10,11]. First of all, access control is necessarily based on the identity of the requester. The unfortunate effect is that every request carries the full privileges of the requester. That's why opening an email attachment can unleash a virus; the virus, running in the user's

^{*} Present address: Transmeta Corporation, Santa Clara, California

¹ We often say incorrectly "who can do what to whom when". Not properly understanding this distinction leads to a variety of security lapses.

² We use the term *access control mechanism* to denote the way control of access to resources is enforced. *Security policy* is a term we use to describe who gets what access rights. We address only the former issue.

process, has all the privileges of that user. A second problem is one of scalability. An ACL must have an entry for every user or access must be denied. That means that the ACL must be modified every time a potential user becomes an actual user, but the ACL is a resource critical to the security of the system. Hence, only a limited number of people should be able to modify it, and they can become overwhelmed in a dynamic environment like the Internet.

CLs don't have these problems. A requester can specify the exact set of capabilities to be submitted with a request. Thus, when opening email, the "execute" capability can be withheld, and the virus program won't run. Further, capabilities are easily copied (but not forged, of course), so one user can pass the capability to others that are trusted.

Four different types of capability structures have been used. Representatives of these types, described in Section 7, are

1. Traditional capabilities [1]
2. Simple Public Key Infrastructure (SPKI) capability certificates [7]
3. E-language capabilities [2]
4. Split capabilities

This paper compares traditional capabilities with split capabilities, but many of the points made apply to SPKI capabilities, too. Space limitations preclude a comparison with E-language capabilities. Capabilities have been used to control access to hardware resources, such as memory segments, ephemeral resources, such as processor time, and software resources, such as files. This paper concentrates on the last of these.

Each traditional capability is an unforgeable, indivisible pair containing a unique identifier for the resource and a list of access rights that are authorized by that capability [1, page3]. The capability itself can either be held by the user's process, in which case it is cryptographically protected against tampering and forgery, or by the TCB, in which case the user's process is given only a handle to it.

Traditional capabilities have two problems. It is hard to revoke a capability [1, page 149], and a capability is needed for each separately controlled access right on each resource. Split capabilities have the advantages of traditional capabilities without their limitations. The basic idea is to provide a handle to the resource being accessed and a handle to a separate object representing the access rights being requested. While such separation of name from authority is potentially problematic [8], in our system these two elements are brought together in the TCB of the resource.

The essential idea is that this separation lets one access rights object represent different permissions on different resources at the discretion of the resource owner, making it surprisingly easy to enforce some complex security policies. For example, military style (multi-level) security [4] assigns security levels to people and resources and specifies the access rules. The so-called **-property* says that you can read documents at the same or lower security level and write documents at the same or higher level. We enforce this

property by giving someone with a Secret clearance a "Secret" access right. This object can be associated with read permission for unclassified documents, read and write permissions for secret documents, and write permission for Top Secret documents. This same access right object could be used to grant access to a physical resource, such as a network port.

Here we describe a particular implementation of split capabilities. The basic work on split capabilities was done at HP Labs in 1996 as part of the Client Utility project, and split capabilities appeared as the basic access control mechanism in the open source versions of e-speak through the Beta 2.2 release [6]. Subsequent versions of e-speak used SPKI capability certificates [7], as they were deemed more appropriate for business-to-business platforms. In order to avoid confusion with this latter system, we'll use the term *Client Utility* or *CU* when describing the platform using split capabilities.

Security has many aspects – physical security of the hardware, authentication of machines and users, privacy, and access control. The CU and e-speak architectures [6, 7] deal with all of these issues, but only the last subject is addressed in this paper. Indeed, many aspects of the CU architecture contribute to the security of the system, but space limitations restrict the present discussion to those aspects directly attributable to split capabilities.

Section 2 gives a brief overview of the CU architecture, to make more concrete the discussion of split capabilities on a single machine in Section 3 and across machines in Section 4. CU visibility tests to control naming are described in Section 5, and various attacks against the system are described in Section 6. Other capability systems are described in Section 7, and Section 8 summarizes the key points.

2. Architecture Overview

CU was designed with access control built in from the very beginning. Hence, in order to explain how CU uses split capabilities, it is necessary to first describe the architecture. More details have been published elsewhere [6, 7].

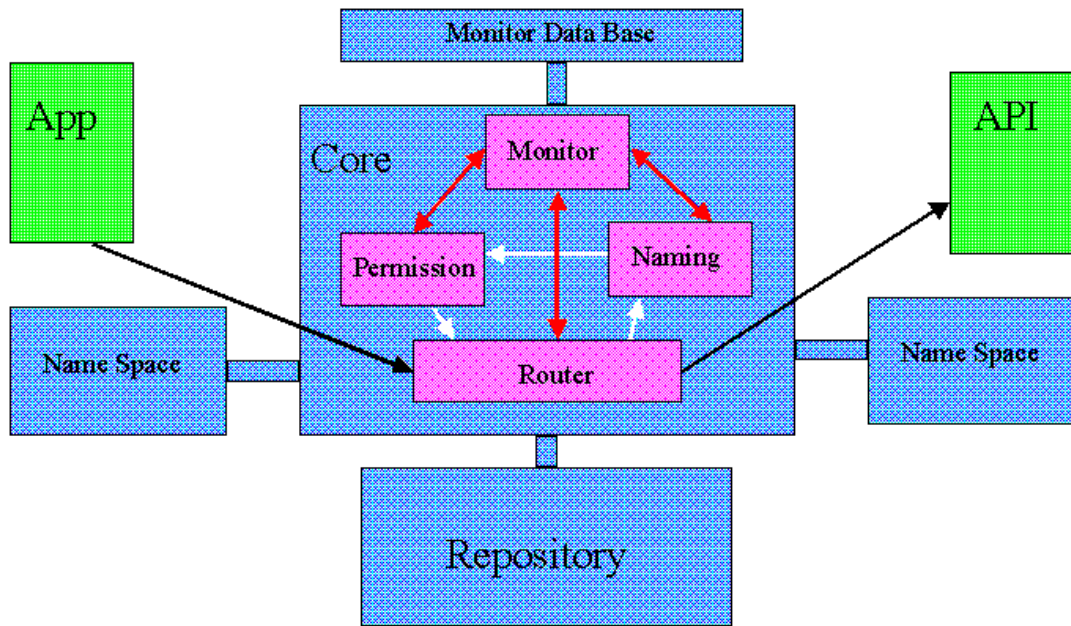


Figure 1: Single machine view.

Figure 1 shows the basic operations within a single *Logical Machine* (the TCB) consisting of a *core* and a *repository* holding information on resources managed by the core. The core is typically a process running on top of a conventional operating system, such as Unix or Windows. Processes running in their own address spaces that interact with the CU protocol are called *clients* of the core.

The basic unit of control is a *resource*. Before a resource can be managed by the core, it must be *registered* in the repository, at which time it is assigned a repository handle. Repository handles are unique within a repository and are never reused. A number of fields are recorded in the repository entry, including a designation of which client will act as *handler* for the resource, a field of data delivered only to the resource handler, and fields containing access control information.

The fact that you can't hurt what you can't name is central to CU's use of split capabilities. Name visibility is controlled by a *protection domain* the core maintains in its address space on behalf of each client. Each protection domain includes, among other things, a name space for the use of the client. Clients interact by sending messages to the core consisting of an envelope, which contains information used by the core, and a payload, which contains application specific information. The core never looks at the payload, which means that end-to-end security can be obtained by encrypting and/or digitally signing the payload.

When a client wants to access a resource, it constructs a message naming the resource. The core looks up this name in the client's name space to find the repository handle bound to the name. The core uses the information in the corresponding repository entry

to process the message, which it then delivers to the handler for processing. Section 3 contains an example.

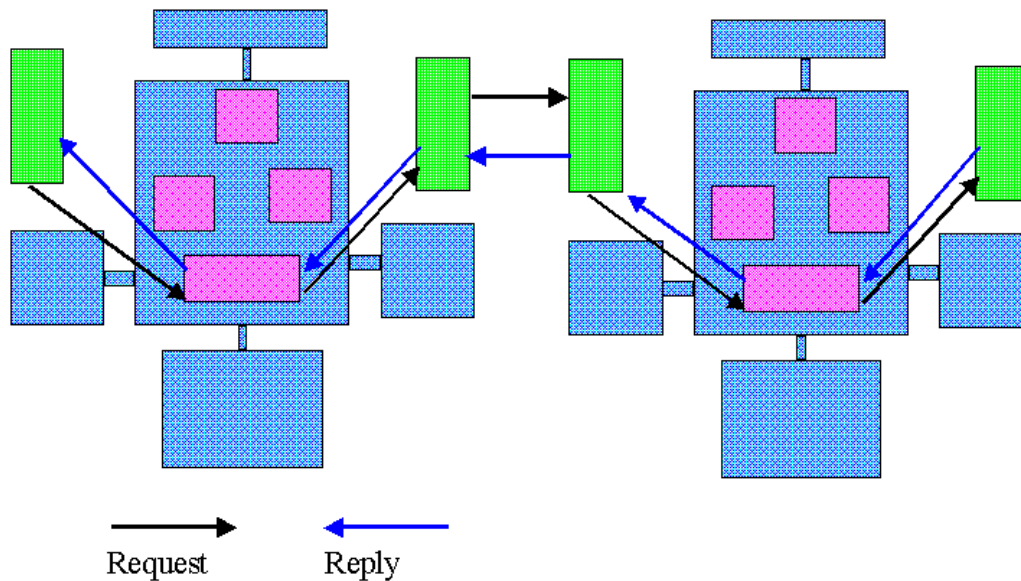


Figure 2: Accessing a remote resource.

This same pattern is used for remote access, as shown in Figure 2. When two machines first connect, they each start a client to act as a proxy for users on the other machine. The connection protocol includes protocol negotiation, mutual authentication, and exchange of encryption keys to be used on the link between machines. Each proxy also registers resources *imported* from the other machine, listing itself as the handler.

A request from a client on one logical machine for a resource on another follows the path shown in Figure 2. The client's core does exactly what it does in the single machine case, forwarding the request to the handler, which is the proxy in this case. The proxy sends the request to its counterpart on the exporting machine. That proxy simply repeats the request as a local message to its core. Eventually, the request reaches the handler where it is processed. An example is shown in Section 4.

3. Split Capabilities on One Machine

As noted in Section 2, CU splits a capability into two parts, a name for the resource being accessed and a name for a resource denoting rights to the resource being accessed. The client's only means to designate a resource is by naming it. Hence, the first half of the split capability is a name bound to a resource's repository handle in the client's name space. The desired access right is denoted by naming another resource, a key (as in a key

that opens a lock, not a cryptographic key). To avoid confusion, we'll call this resource a *CUKey*.

A resource handler, such as a file system, wishing to have a resource managed by CU would submit a request to register the resource. For example³

```
Handler:      myInbox
Private Data: "/u/bill/boss"
Permissions:  (readKey,"read")
              (writeKey,"write")
              (executeKey,"execute")
```

where items not in quotes are names bound to repository handles in the name space of the client acting as the resource handler. For example, the handler is denoted by the resource representing the mailbox where it receives messages. Requests for every file come to this handler, and the private data tells the handler which file is being accessed. The corresponding repository entry might look like

```
Repository Handle: 8594
Handler:           44A5
Private Data:      "/u/bill/data"
Permissions:       28CF "read"
                  3F32 "write"
                  AD97 "execute"
```

The 4-digit numbers represent repository handles; actual handles are 64 bits long.

A client now wishes to read this file. First, it needs a name bound to the repository handle of this file, which it can get in a number of ways. For example, the binding could be part of the client's initial environment. The client also needs a name bound to the repository handle of a *CUKey* associated with read permission.

In our example, the client message might be⁴

```
Envelope( Name:      BossIsDumb,
Label:      "BossIsSmart",
Keys:      Key1)
Payload( "read BossIsSmart")
```

where you can see how CU provides a means for a client to keep its local names private. (There are no security implications of revealing these names, only privacy issues, as in the example.) The core looks in the client's name space, sees that this name is bound to repository handle 8594, finds that the request includes *CUKey* 28CF and delivers the following message to the mailbox 44A5.

³ Only fields relevant to this discussion are shown.

⁴ The encoding can be in any industry standard format, such as XML.

```
Envelope( Name: BossIsSmart,  
          Private: "/u/bill/boss",  
          Permissions: "read")  
Payload( "read BossIsSmart")
```

The handler now knows to grant whatever permission on this file the string “read” designates. Numerous details have been left out for clarity.

The client accesses a resource by naming it, and the name is associated with a repository handle in the client’s name space. Should the client use a name that doesn’t appear in its name space, the client is told that the resource does not exist. Note that the names in the client’s name space are specific to the client. Hence, there is no way to guess the name of a resource or for names to be communicated out of band, say by email. A name has meaning only within the CU system and is specific to the binding in a particular client’s name space.

Traditional capability systems require at a minimum a capability for each resource. However, if we want to be able to control which rights accompany which request, we need a capability for each access right on each resource. The resulting number of capabilities scales with N , the number of resources, times M , the number of access rights. For example, there must be a capability each for read, write, and execute permissions for each file.

Split capabilities provide these same semantics with just a few CUKeys. We need one name bound to each resource, N names, and a CUKey for each access right, M of them. The result is $N+M$ items to be managed. We still have the ability to decide which access rights accompany which requests because only the resources named in the request and the permissions unlocked by the CUKeys presented are used by the handler. For Unix file semantics, the client needs 3 CUKeys for world access, 3 CUKeys for each group it is in, and 3 CUKeys for owner permissions. Thus, a typical client needs only 9 CUKeys, independent of the number of files.

Our implementation of split capabilities has another advantage over traditional capabilities: revocation. Recall that traditional capabilities are held by the user and may be passed from one to another outside of the control of the TCB. Anyone presenting the capability is granted the access rights it denotes. However, once Alice has passed a traditional capability to Bob, there is no way for her to revoke Bob’s privilege.

CU provides several ways for Alice to revoke a privilege she granted to Bob. One is to grant the privilege by giving Bob a name binding to a clone of the key. Clones of a key are different resources that open the same lock. Revocation is as simple as destroying the clone. If Alice is a system administrator, she can also be granted access to modify Bob’s name space, with a CUKey of course, and can revoke access by removing the appropriate name binding.

4. Split Capabilities between Machines

Thus far we've talked about access control within a machine. CU would be quite limited unless it permitted access to resources on other machines. This section shows how split capabilities are used when resources are shared with users on other machines.

When two machines want to share resources with each other, they connect with the protocol described in Section 2. Exporting a resource involves creating an *export form* for the data in the resource's registry entry. The export form of our sample resource from Section 3 might look like

```
Repository Handle:    resource1
Permissions:         CUKey1
                   CUKey2
                   CUKey3
```

The export form is the same as the data in the repository entry with a few exceptions. First of all, the private data field is not included. Secondly, the repository handles in the repository entry are converted to names bound to these handles in the proxy's name space. Also, since the exporting machine is not expecting the importing machine to enforce the exporter's access control policies, there is no need to put the permissions into the export form.

This export form is passed to the importing proxy, which adds its own private data field and creates new permission fields. The new permission fields use a CUKey created by the proxy and a permission that is the exporting proxy's name for the CUKey. So, the repository entry of the sample resource might look like

```
Repository Handle:    73BF
Handler:              3330
Private Data:         "resource1"
Permissions:          F032 "CUKey1"
                   5228 "CUKey2"
                   B3A8 "CUKey3"
```

The importing machine can add its own permissions to enable enforcement of local policies. It may be, for example, that only managers are allowed to access remote resources. Of course, the proxy must be able to interpret these permissions to use this feature.

When a client accesses a remote resource, the Core forwards the request to the resource handler, in this case the proxy acting on behalf the machine that owns the resource. Any permissions that get unlocked result in the proxy's name for the CUKey being delivered to the proxy. If the client submits the same request as in Section 3, the proxy sees

```
Envelope( Name:      BossIsSmart,
          Private:   "resource1",
          Permissions: "CUKey1")
Payload("read BossIsSmart")
```

The proxy forwards the request across the wire to its counterpart. That client submits the names of the CUKeys presented with a request to its core. In this case, the proxy names `resource1` and submits the key named `CUKey1` in a request of its local Core. The name `resource1` is bound to repository handle `8594`, and `CUKey1` is bound to `28CF`, so the resource handler sees the same request as in the single machine example.

5. Visibility

Control of what a client can name is an important part of CU access control. However, we can do even better at controlling the use of names. In this section we'll see how CU can make it difficult for someone to obtain certain name bindings or use them should they be made available.

Actually, the CU architecture requires something better than described so far for a reason that has not been explained yet. So far, the only method described for getting name bindings was to have them available as part of the client's initial environment. There are two other methods.

When a message is received, name bindings for all the resources other than CUkeys listed in the original message are put into the name space of the recipient.

A field in the repository entry describes the resource so it can be discovered with a look-up request to the core.

Security rules could be violated if one client inadvertently sent a name binding to a client who should not be allowed access to the resource or if a client discovered such a resource. This situation is a particular problem with bindings to CUKeys.

Our solution is to add two fields to the repository entry of each resource, called *allow* and *deny*. Each contains a list of locks. Each look-up and each attempt to use a name in a message is checked by comparing the CUKeys accompanying the request with the locks listed in these two fields. If any of the deny locks is opened, or if none of the allow locks is opened, the system acts as if the resource does not exist. These CUKeys can be the same ones used to grant access rights.

Of course, it's natural to ask why any client would submit CUKeys that might deny access to some resources. There are several reasons. The request might be part of a test run of a program, and the user doesn't want to risk damage to the resource. Also, the CUKey that denies access to one resource may be needed because it grants a desired access right or appears in the allow field of another resource that the client wants to access. Finally, CU includes *mandatory keys* in each protection domain that are implicitly included in each request. This last allows a system administrator to enforce a

variety of policies by putting certain CUKeys in each client's protection domain. We saw one such example in Section 1, where we wanted to enforce multi-level security [4]. Putting the "Secret" CUKey in the client's protection domain guarantees that it is included with every request.

The visibility tests allow us to enforce some common security policies in a rather straightforward manner. One is compartmentalization in which a client in one compartment may not access resources in another compartment. An example might be a consulting company that wants to assure a customer from company ABC that people working on its contract can't accidentally mix resources with those working on the contract for a competitor, say company XYZ. The administrator can put a lock corresponding to one CUKey in the allow field of resources associated with company ABC and the same lock in the deny field of those resources associated with company XYZ. If this CUKey is put in the protection domain of one of the consultant's employees working on a project for ABC, that person will not be able to find out that resources associated with company XYZ exist.

Compartmentalization is an example of the problem of *rights amplification*, the ability to do something only if two, separate access rights are available. For example, if I give Alice access to the can opener and Bob access to the can, my tuna is safe as long as the capabilities cannot be presented together. Normally, we don't care if a client gives a capability to another client. After all, Alice could read a file and send Bob the results, so it doesn't matter if Alice gives Bob the capability to read the file. However, if Alice can give Bob access to the can opener, they can share the tuna. Split capabilities with visibility control can prevent such rights amplification.

Visibility controls can also be used to implement the *restricted *-property* of multi-level security, a policy that allows a user to read and/or write resources only at the client's security level. Putting a lock in the allow field of every confidential resource, a different one in the allow field of every secret resource, and a third in the allow field of every top secret resource allows this level of control. Putting the CUKey corresponding to the user's security level in the protection domain of a client guarantees that the client can only access resources at its level. Although less flexible than the full *-property, the restricted form is enforced by the Core, while the full form is enforced by the resource handler.

6. Attacks

We don't know how well the system will stand up to real attacks, but we can look at a number of possibilities to see if we can identify weaknesses. Here, we'll look only at attempts to perform unauthorized actions. Many denial of service attacks are also dealt with by the CU architecture, but they are not relevant to the present discussion. We won't worry about social problems, such as poorly chosen passwords or people who write their PINs on their ATM cards. There is also nothing we can do about attacks against the underlying operating system or its components.

Social engineering is always a concern. An attacker might get a user to reveal the names of certain resources, particularly CUKeys. This information does the attacker no good, since the user's names have meaning only through the bindings in the user's name space. The attacker might also go directly after the core and attempt to learn the repository handles of certain resources. Again, the information does the attacker no good, since the core does not accept repository handles from clients. The attacker could also try to get the name of a resource exported to someone else. This information is useless, because the proxy acting on behalf of the attacker has name bindings only for resources exported to the attacker.

Next, consider a malicious user on a single machine who would like to do some unauthorized operation. Since any user has access to a process with enough resources to complete a valid login attempt, the attack could start there. This task will have a protection domain with some named resources. The attacker has no names for anything else, so the initial attack will be against those resources. We can make sure that the attack does no harm by only giving this process access to CUKeys that would allow it to modify resources when needed by the login procedure.

Our attacker now attempts to get additional resources into the protection domain of this login process. However, without access to CUKeys granting permission to add these resources, this attack fails. For example, the initial set of CUKeys made available to anonymous clients need not include permission to add bindings to the client's name space. Additionally, the mandatory keys in this protection domain can include a CUKey corresponding to a lock in the deny field of every resource not needed to complete the login process.

Say that the attacker has logged in, either by guessing a password or because the attacker is an authorized user. The attacker can now modify any resources available to the active account. The attacker can also add to its task's name space bindings to any resources made available to it by others. However, there is no way for the attacker to even name a resource that doesn't have a binding in its name space. Since critical system resources, such as the system log files, are controlled by the system administrator, getting access to them is harder than if name guessing is possible.

If we're dealing with more than one machine, the attacker can try to get the other machine to do something unauthorized on its behalf. However, it doesn't matter if the attacker is running on a machine with a corrupted core, modified operating system, or customized hardware. Even if the attacker refuses to honor the permissions in the exported resources, the attack fails because these permissions are checked when the proxy on the machine that owns the resources attempts the access.

7. Related Work

A complete review of access control mechanisms, or even only capabilities, is beyond the scope of this paper. The early history [1] and modern developments [3] of capabilities

are well documented elsewhere. Here, we'll briefly describe the archetypes of the different capability systems mentioned in Section 1.

SPKI capabilities [7] contain a name for the resource and one or more access rights. Wildcards can be used to grant access to a group of similarly named resources, such as files in a directory. SPKI certificates are open documents and are digitally signed to prevent forgery or tampering. These signatures are checked against the resource handler's trust assumptions before granting access. The submitter of the capability must prove knowledge of the private key associated with the public key assigned to the capability. Anyone with a SPKI certificate can create a new capability containing a subset of the access rights. Hence, SPKI capabilities don't have all the scalability problems of traditional capabilities, but revocation is a problem.

E-language capabilities [2] are quite different. Each is a name for a facet of a particular object, *i.e.*, a handle. A facet is treated exactly like the object, having state and methods, but the facet has a subset of the methods. For example, a file object may have read, write, and execute methods, but a facet of that object may have only a read method. The access rights are implicit in the facet being accessed, in contrast to a traditional capability in which they are listed explicitly. Since the E-language capability is entirely in the name, forgery is prevented by making the names unguessable using cryptographic techniques and encrypting them when they are passed between machines.

There is a close analogy to split capabilities, namely virtual memory. Each "name" is a virtual address useful only to the process using it. In some systems [9], access to the pages of the virtual memory is controlled by *storage protection keys*. Split capabilities as we have used them are more general, applying to any kind of resource, and need not involve the operating system.

8. Summary

Capabilities have a number of advantages over access control lists [10], and our implementation of split capabilities has advantages over traditional capabilities, namely

1. Revocation is easy.
2. The number of elements to be managed scales only with the sum of the number of resources and the number of access rights instead of with their product.
3. Using split capabilities with visibility controls allows the simple specification of complex security policies.

We have only implemented split capabilities within the CU architecture. It may be possible to use them in other environments, for example, those with a global name space. Split capabilities could also be used as the basis for building a capability secure operating system such as Eros [**Error! Reference source not found.**].

Acknowledgements

I'd like to thank Harumi Kuno, Nigel Edwards, Jonathan Shapiro, and Chris Hibbert for helping me improve the presentation.

References

1. Henry M. Levy, *Capability-Based Computer Systems*, Digital Press, Bedford, MA (1984)
2. <http://www.skyhunter.com/marcs/ewalnut.html>
3. <http://www.skyhunter.com/marcs/capabilityIntro/index.html>
4. David E. Bell and Leonard La Padula, *Secure Computer System: Unified Exposition and Multics Interpretation*, ESD-TR-75-306, ESD/AFSC, Hanscom AFB, Bedford, MA 01731 (1975) [DTIC AD-A023588]
<http://csrc.nist.gov/publications/history/bell76.pdf> (1975)
5. Alan H. Karp, Rajiv Gupta, Guillermo Rozas, Arindam Banerji, "The Client Utility Architecture: The Precursor to E-speak", HP Labs Technical Report, HPL-2001-136, (2001) available at
http://www.hpl.hp.com/personal/Alan_Karp/cuarch/arch_overview_TR.html
6. "E-speak Architectural Specification: Beta 2.2",
<http://www.e-speak.net/library/pdfs/E-speakArch.pdf> (1999)
7. <http://www.e-speak.net/library/pdfs/a.0/Architecture.pdf> (2001)
8. Norman Hardy, "The Confused Deputy", *Operating Systems Reviews*, **22**, #4, (1988)
9. *System/370 Principles of Operation*, GA22-7000-9, IBM (1983)
10. <http://www.erights.org/elib/capability/consensus-9feb01.html> (2001)
11. <http://www.eros-os.org/essays/ACLSvCaps.html>
12. J. S. Shapiro, J. M. Smith, and D. J. Farber, "EROS: A Fast Capability System", in *Proc. 17th ACM Symposium on Operating System Principles*, pp. 170-185, Kiawah Island, near Charleston, SC, December (1999), see also
<http://www.eros-os.org/>