Chapter 7

Access Control

Confidentiality and integrity are often enforced using a form of authorization known as *access control*.

- Predefined *operations* are assumed to be the sole means by which principals can learn or update information.

- Some reference monitor is consulted whenever one of these predefined operations is invoked; the operation is allowed to proceed only if the invoker holds the required *privileges*.

Confidentiality is then achieved by restricting if and when each principal is authorized to execute operations that reveal information; integrity is achieved by analogously restricting operations that perform updates.

An *access control policy* specifies when a *subject* may perform the operations associated with a given *object*, including operations that change the policy. The subjects can correspond to any entities to which execution can be attributed—users, processes, threads, or even procedure activations. And the objects can be any entity on which operations can be defined—storage abstractions, such as memory or files (with read, write, and execute operations), and code abstractions, such as modules or services (with operations to initiate or suspend execution). Distinct operation typically require distinct privileges so, for example, a subject S can perform a read operation on an object O only when S holds the read privilege for O.

The Principle of Least Privilege is best served by having fine-grained subjects, objects, and operations; the Principle of Failsafe Defaults favors defining an access control policy by enumerating privileges rather than prohibitions. That, however, is only a small part of the picture, and there is much ground to cover in this chapter.

Copyright 2010. Fred B. Schneider. All rights reserved.

subject	object		
	`c1.tex`	`c2.tex`	`invtry.xls`
`fbs`	`r`, `w`	`r`, `w`	`r`
`mmb`			`r`, `w`
`jhk`			`r`

Figure 7.1: Example DAC Policy

7.1 Discretionary Access Control

A *discretionary access control* (DAC) policy is an access control policy in which the initial assignment and subsequent propagation of privileges associated with each object are controlled by the *owner* of the object and/or other subjects whose authority can be traced back to the owner. Rather simple DAC policies are often implemented by commercial operating systems for their file systems. The users are subjects; a user who creates a file is the owner and specifies which other users are authorized to read, write, and/or execute the file.

The assignment of privileges by a DAC policy can be depicted with a table having a row for each subject and a column for each object. The entry in the cell associated with a subject S and an object O lists privileges corresponding to those operations on O that are authorized if invoked by execution being attributed to S. Figure 7.1, for example, gives a table that assigns privileges for users `fbs`, `mmb` and `jhk` to perform operations on files `c1.tex`, `c2.tex`, and `invtry.xls`. Only execution attributed to `fbs` can read (`r`) or write (`w`) `c1.tex` and `c2.tex`; only execution attributed to `mmb` can write `invtry.xls`; execution attributed to any of the three users can read `invtry.xls`.

Figure 7.1 depicts what has been called an *access control matrix*.[1] However, using the term “matrix” here is misleading, because row and column ordering in a matrix has significance but the ordering of rows and columns in our table does not. What the table actually specifies a set *Auth* of triples, where $\langle S, O, op\rangle \in Auth$ holds if and only if execution attributed to subject S is granted privilege op for object O; we will call *Auth* an *authorization relation*.

Any DAC policy is easily defeated if subjects can make arbitrary changes to *Auth*. Yet as execution of a system proceeds, changes will invariably be needed, because new objects must be accommodated and because trust relationships between principals evolve. To characterize which changes to *Auth* are allowed, a DAC policy includes *commands*. Each command has a Boolean *precondition* and an *action* to change *Auth*. If the command is invoked and the precondition holds, then the action is performed; if the precondition does not hold, then the command fails. Evaluation of the precondition and performing the action is assumed to be indivisible.[2]

[1] Some authors prefer the term a *protection matrix*.

[2] In practice, checking a boolean and executing the code for the action is likely to involve multiple atomic actions. The effect must nevertheless somehow be made to appear indivisible with respect to execution of other commands.

Copyright 2010. Fred B. Schneider. All rights reserved.

As an example, here is a command that might be found on a system where subjects are users.

$$\begin{array}{l} \mathit{addPriv}(U, U', O, op)\text{: } \textbf{command} \\ \quad \textbf{pre}\text{: } \mathit{invoker}(U) \ \wedge \ \langle U, O, \texttt{owner}\rangle \in \mathit{Auth} \ \wedge \ op \neq \texttt{owner} \\ \quad \textbf{action}\text{: } \mathit{Auth} := \mathit{Auth} \cup \{\langle U', O, op\rangle\} \end{array}$$

The precondition in *addPriv* is

$$\mathit{invoker}(U) \ \wedge \ \langle U, O, \texttt{owner}\rangle \in \mathit{Auth} \ \wedge \ op \neq \texttt{owner}$$

where predicate $\mathit{invoker}(U)$ is satisfied if and only if *addPriv* is invoked by execution attributed to user U; so the precondition implies that the invoker of *addPriv* has the `owner` privilege for O and that op is not `owner`. The action in *addPriv* is assignment statement

$$\mathit{Auth} := \mathit{Auth} \cup \{\langle U', O, op\rangle\}$$

which adds to *Auth* a triple authorizing execution by U' to invoke operation op on object O. So, *addPriv* is consistent with the defining characteristic for a DAC policy—the owner of object O grants privileges for operations on O.

Separation of Privilege suggests that it is better for a distinct privilege[3] say op^* to be required for granting a privilege op than having a single generic privilege, like `owner`, that empowers granting any privilege. So a better command than *addPriv* would be:

$$\begin{array}{l} \mathit{grantPriv}(U, U', O, op)\text{: } \textbf{command} \\ \quad \textbf{pre}\text{: } \mathit{invoker}(U) \ \wedge \ \langle U, O, op^*\rangle \in \mathit{Auth} \\ \quad \textbf{action}\text{: } \mathit{Auth} := \mathit{Auth} \cup \{\langle U', O, op\rangle\} \end{array}$$

Finer-Grained Subjects: Protection Domains. The Principle of Least Privilege implies that the set of operations a principal is authorized to execute ought to differ for different tasks. Users are thus too coarse-grained to serve us well as the subjects in DAC policies. This leads to the idea of a *protection domain*; it specifies sets of privileges for objects and, therefore, can serve as the basis for distinguishing subjects in an authorization relation. We then associate a protection domain with each thread of control, possibly associating different protection domains at different points of execution. Thus, different privileges can be associated with the execution required for different tasks, so the Principal of Least Privilege can now be instantiated in *Auth*.

Protection domains implemented by a system can be characterized abstractly using a function $\mathcal{D}$. For each thread of control P, $\mathcal{D}(H_P)$ gives the protection domain in which P is currently executing, where H_P is the sequence of system states that execution by P has thus far produced. A transition between protection domains occurs whenever further execution by P extends H_P with a new state, producing H'_P satisfying $\mathcal{D}(H'_P) \neq \mathcal{D}(H_P)$.

[3]Privilege $op*$ is called the *copy flag* for op.

Copyright 2010. Fred B. Schneider. All rights reserved.

$\mathcal{D}$ abstracts what is actually implemented by some run-time environment. Engineering realities constrain $\mathcal{D}$ to associate protection-domain transitions with events that the run-time environment can detect efficiently and also constrain any state or history information used for evaluating $\mathcal{D}(H_P)$ to being information that is cheap for the run-time environment to collect and maintain. For example, protection-domain transitions that coincide with certain kinds of control transfer (e.g., invoking a program) are typically inexpensive to support by a run-time environment, as are those that coincide with certain other[4] kinds of state changes (e.g., changing from user mode to system mode). But few run-time environments support entry to a new protection domain being triggered by branching to a new location or by changing the value of an arbitrary variable, simply because detecting such events would be expensive.

An operating system is often the run-time environment where protection domains are supported. The usual approach is to include protection-domain transitions in the semantics of certain system calls. System calls for invoking a program or changing from user mode to system mode are obvious candidates. But some operating systems simply provide an explicit domain-change system call rather than implicitly linking domain changes to other functionality; the application programmer or a compiler's code generator is then required to decide when to invoke this system call.

Since distinct tasks are typically implemented by distinct pieces of code, the Principle of Least Privilege is better served if we have a way to associate different protection domains with different code segments. We might, for example, contemplate having a protection domain *U/pgm* for each code segment *pgm* executing on behalf of user U. Here, *pgm* could be an entire program, a method, a procedure, or a block of statements; it might be executed by a process started by user U or by a process started by some other user in response to a request from U. Ideally *U/pgm* would only include the minimum privileges U needs to execute *pgm*.

Figure 7.2 illustrates, reformulating the access control matrix of Figure 7.1 in terms of subjects that are protection domains associated with code segments corresponding to entire programs: a shell (`sh`), a text editor (`edit`), and a spreadsheet application (`excel`). Notice, `c1.tex` and `c2.tex` can be written by user `fbs` only while executing `edit`, and `invtry.xls` can be accessed only by executing `excel` (with `mmb` still the only user who can perform write operations to that object).

Executing in a given protection domain might or might not be appropriate when working on a given task and, therefore, according to the Principle of Least Privilege, transitions ought to be permitted only between certain pairs of protection domains. For example, we would expect that execution in a shell should be allowed to start either a text editor or a spreadsheet application, but execution in a text editor should not be allowed to start a mailer. We might specify such restrictions by defining an `enter` (`e`) privilege for each protection

[4]Transfers of control change the program counter, which is part of the system state. So transfers of control are really just a special kind of state change.

Copyright 2010. Fred B. Schneider. All rights reserved.

domain	object c1.tex	c2.tex	invtry.xls
fbs/sh			
fbs/edit	r, w	r, w	
fbs/excel			r
mmb/sh			
mmb/edit			
mmb/excel			r, w
jhk/sh			
jhk/edit			
jhk/excel			r

Figure 7.2: Example DAC Policy for Domains

domain and by adding protection domains to the set of objects governed by *Auth*. A protection domain D must possess the `enter` privilege for a protection domain D'—that is, $\langle D,\ D', \texttt{enter}\rangle \in Auth$ must hold—for execution in D' to be started by execution in D.[5] Figure 7.3 adds the constraints noted above about starting a shell, text editor and spreadsheet.

We thus far have not constrained how privileges change at a transition between protection domains. In practice, though, the set of privileges before and after a transition are likely to be related.

> **Attenuation of Privilege.** Suppose execution in a protection domain D initiates a subtask, and that subtask is executed in protection domain D'. Then D', having a more circumscribed scope, should not grant all of the privileges D has. We use the term *attenuation of privilege* when execution in a protection domain is not granted all of the privileges its initiator has. □

> **Amplification of Privilege.** Suppose execution in a protection domain D' implements an operation on some object O, as a service to execution in protection domain D. Then D' should grant privileges for O that D does not. We use the term *amplification of privilege* when a protection domain D' grants privileges not held by a protection domain D it serves. □

Notice that attenuation of privilege and amplification of privilege both can help with the Principle of Least Privilege. Moreover, amplification of privilege also enables *data abstraction* by ensuring that users of an object are kept ignorant of how that object is implemented.

[5] An alternative to having protection domains be objects is having code segments (independent of user) be objects. With this alternative, protection domains from which a user U is allowed to next start execution of code segment *pgm* are granted an `execute` privilege for object *pgm*. This approach is less expressive than the approach outlined in the text, but it is simpler to implement and thus tempting in practice.

Copyright 2010. Fred B. Schneider. All rights reserved.

	object											
domain	`c1.tex`	`c2.tex`	`invtry.xls`	`fbs/sh`	`fbs/edit`	`fbs/excel`	`mmb/sh`	`mmb/edit`	`mmb/excel`	`jhk/sh`	`jhk/edit`	`jhk/excel`
`fbs/sh`					`e`	`e`						
`fbs/edit`	`r`, `w`	`r`, `w`										
`fbs/excel`			`r`									
`mmb/sh`								`e`	`e`			
`mmb/edit`												
`mmb/excel`			`r`, `w`									
`jhk/sh`											`e`	`e`
`jhk/edit`												
`jhk/excel`			`r`									

Figure 7.3: Example DAC Policy with Domain Entry

The Confused Deputy. Amplification of privilege brings the risk of a *confused deputy* attack. Here, one subject S requests execution by another subject S' in a way that abuses privileges granted to S' but not granted to S.

Consider a server that, in servicing each client request, reads from some client-named input file, computes results, writes these to a client-named output file, and records billing information in file `charges.txt`. Further, suppose request processing is performed by a server having a `write` privilege for `charges.txt`, and the server also inherits from the requestor a `read` privilege (if present) for the client-named input file and a `write` privilege (if present) for the client-named output file.

We might expect that the processing of client requests cannot corrupt file `charges.txt`, because clients lack `write` privileges for this file. But that expectation is naive. A client naming `charges.txt` as the output file for a request would cause the server to corrupt the billing information stored in `charges.txt`. What happened was the server functioned as a deputy to the client, and it became "confused" by the client's request.

An obvious defense would be to expect programs, like the server, to validate that each client has appropriate privileges for the client-named file in a request. This defense, however, requires programmers to include checks in every program that might invoke operations on objects provided by another. Many programmers would regard adding all those checks as onerous and not bother. So the defense would not be very effective.

A more elegant defense is to combine naming and authorization. Instead of names for objects (like files), programs use unforgeable *bundles* comprising the name for an object along with privileges for that object. Bundles are assumed to be the sole means for programs to name, hence access, objects. In the example

Copyright 2010. Fred B. Schneider. All rights reserved.

above, each client request would convey two bundles—one for the input file and one for the output file—and the server would use these client-supplied bundles for reading the input file and writing to the output file. The server would also have a bundle for `charges.txt`, with a `write` privilege. Since the client does not have a `write` privilege for `charges.txt`, a client-supplied bundle for `charges.txt` would not include a `write` privilege—attempts by the server to perform write operations to `charges.txt` using that client-supplied bundle would fail. The confused deputy is no longer duped into writing the wrong content into `charges.txt`.

Implementation of DAC. At the heart of any implementation of DAC will be a scheme for representing authorization relation *Auth*. That scheme must provide the means to

- evaluate whether $\langle S, O, op\rangle \in Auth$ holds and, therefore, privileges are present for a subject S to perform an operation *op* on some object O,
- change *Auth* in accordance with commands the DAC policy defines, and
- associate a protection domain with each thread of control, providing for transitions between protection domains as execution proceeds.

In addition, support for two kinds of *review* is also desirable: (i) listing, for a given subject, its privileges for each object, and (ii) listing, for a given object, the subjects and their privileges for that object.

The obvious scheme for representing *Auth* is to employ a 2-dimensional array-like data structure resembling an access control matrix. However, access control matrices are likely to be sparse, because the typical subject has privileges for only a small fraction of all objects in a system. Implementors thus favor data structures that store only the non-empty cells of the access control matrix. We explore these in what follows.

7.1.1 Access Control Lists

An *access control list* for a given object O is a list

$$\langle S_1, Privs_1\rangle\ \langle S_2, Privs_2\rangle\ \dots\ \langle S_n, Privs_2\rangle$$

of *ACL-entries*. Each ACL-entry $\langle S_i, Privs_i\rangle$ is a pair, where S_i is a subject, $Privs_i$ is a non-empty set of privileges, and $op \in Privs_i$ holds if and only if $\langle S_i, O, op\rangle \in Auth$ holds. Thus, an access control list encodes the non-empty cells in some column of the access control matrix. For example, the access control list for `invtry.xls` in Figure 7.1 is

$$\langle \texttt{fbs}, \{\texttt{r}\}\rangle\ \langle \texttt{mmb}, \{\texttt{r}, \texttt{w}\}\rangle\ \langle \texttt{jhk}, \{\texttt{r}\}\rangle.$$

Long access control lists are difficult for people to understand and keep updated, as well as expensive for enforcement mechanisms to check when authorizing access requests. Therefore, various extensions have focused on shortening

Copyright 2010. Fred B. Schneider. All rights reserved.

the number of ACL-entries in an access control list and/or making important but complicated kinds of updates easier to perform. We now summarize these.

Groups of Principals. Particularly in corporate and institutional settings, users might be granted privileges by virtue of membership in a group. Students who are taking a course, for example, are given access to that semester's class notes and assignments; employees of a company are given access to descriptions of operating procedures.

Group memberships change over time. If membership in a group confers privileges for many objects, then adding or deleting a member requires separately updating multiple access control lists. Updating all those access control lists could be tiresome. Moreover, updating an individual access control list can be subtle. Suppose, for example, user U is in group G, and membership in G should confer privilege *op* for object O. If U is dropped from G then you might be tempted to delete *op* from the ACL-entry naming U in the access control list for O. But this ignores the possibility that U might also be a member of some other group that also confers *op* for O on its members, in which case U being dropped from G should not cause U to lose *op* for O.

We can avoid these difficulties by supporting indirection in access control lists. To do so, we introduce and use groups of subjects.

> **Groups in Access Control Lists.**
> - A *group declaration* associates a *group name* with a set of subjects. Membership in the set is specified either by enumerating its elements or by giving a predicate that all subjects in the set must satisfy.[6]
> - An ACL-entry $\langle G, \mathit{Privs} \rangle$, where G is a group name and *Privs* is a set of privileges, is used to grant all privileges in *Privs* to all subjects S such that $S \in G$ holds. □

Notice how the indirection eliminates the need to update multiple access control lists when a group's membership is changed—only the associated group declaration must be changed. Moreover, for an ACL-entry $\langle G, \mathit{Privs} \rangle$ on the access control list for an object O, deleting S from G does not revoke S's privileges to O if S also appears elsewhere on that access control list (directly or through membership in some other group).

Permission and Prohibition. Sometimes, not conferring a specified privilege on certain subjects is what's important. We might, for example, be focused on constructing a DAC policy to prevent students from reading solution sets for future assignments. Given the semantics presented above for access control lists, in order to conclude that S has not been granted *op* for an object O, we would have to: (i) enumerate all subjects granted *op* by the access control list

[6]An enumeration should be short enough so that subjects are unlikely to be overlooked; a predicate for characterizing a set should be sufficiently transparent so that it defines all of the intended subjects and no others.

Copyright 2010. Fred B. Schneider. All rights reserved.

for O, (ii) check that S is not among them, and (iii) invoke the Principle of Fail-safe Defaults. This is rather complicated, so some systems define a *prohibition* $\overline{op}$ corresponding to each privilege *op*; prohibitions are then allowed to appear directly in the set of privileges an ACL-entry specifies.

The exact interpretation given to access control lists containing prohibitions varies from system to system. But it is not uncommon for the relative order of ACL-entries in an access control list to resolve conflicts when one ACL-entry confers *op* and another confers $\overline{op}$ on a given subject:

> **Prohibitions in Access Control Lists.** S is deemed to have privilege *op* for O if and only if the access control list for O (i) contains an ACL-entry that confers *op* on S and (ii) neither that ACL-entry nor any earlier one in the access control list confers $\overline{op}$ on S. □

Each individual access control list thus specifies—through the sequence of ACL-entries—whether, for each given subject, prohibition trumps privilege or *vice versa*. Also, note that now reading only a prefix of the access control list often suffices for deciding whether to allow an access to proceed; humans and reference monitors alike benefit from that.

Engineering Considerations. Designers and implementors of access control mechanisms are typically concerned with three things: flexibility, cost, and understandability. The usual tensions are present. Without sufficient flexibility, we might not be able to specify the security policy we intend. But support for flexibility usually entails complexity, which introduces the risk that people will be unable or disinclined to write or understand policies; complexity also tends to erode assurance in an access control mechanism's implementation. Finally, high run-time costs cause people to shun stronger access-control policies in favor of those involving less restrictive, hence cheaper, checking.

Subjects. Our success in expressing a policy by using access control lists will depend, in part, on what can be a subject. That set of subjects, in turn, is constrained by the availability of efficient means for attributing (authenticating) accesses, since the name of a subject making a request is what's needed for checking an access control list.

Operating systems typically offer cheap mechanisms for authenticating users and processes. Some language run-time environments do even better, allowing execution to be associated with a chain of nested procedure invocations—for example, defining a protection domain $U/pgm_1/pgm_2/pgm_3$ to serve as the subject for execution of pgm_3 invoked by a call within pgm_2, itself invoked by a call from pgm_1, running on behalf of user U.

Independent of what constitutes a subject, care should be exercised in recycling subject names. Otherwise, some future incarnation of a subject name could inadvertently receive privileges granted to a past incarnation. One solution is simply not to reuse subject names for different subjects, but this (i) requires saving enough state to ensure no future name duplicates a past name and

Copyright 2010. Fred B. Schneider. All rights reserved.

(ii) constrains the choice of subject names, potentially making policies harder to understand. The more widely-adopted solution is, as part of the process for deleting a subject from the system, to delete that subject's name from all access control lists.

Objects. The choice of objects imposes another constraint on our ability to express policy with access control lists. Each object requires a reference monitor and an access control list. The reference monitor must intercept every access to the object, and that restricts what can be an object. Some implementations require that all accesses cause traps; others require that checks be in-lined before each access. An access control list must be stored in a way that its integrity is protected, and two solutions here are common: (i) store the access control list with the object it, so updates to the access control list are checked by the reference monitor; (ii) store the access control list with the reference monitor that reads it, so the mechanism protecting the integrity of the reference monitor also protects the integrity of the access control list.

Operating system abstractions are thus particularly well suited to serve as objects. First, system calls are then the only way to access an object, and a reference monitor is easily embedded in the operating system routine that handles a system call. Second, operating system abstractions are typically large enough (e.g., files) to accommodate storing their own access control lists or are relatively few in number (e.g., locks or ports) so that the operating system's memory can be used to store the access control lists.

Encodings. Various schemes have been proposed to reduce the amount of computation a reference monitor requires for determining whether an access should be allowed to proceed. Many start from a debatable premise[7] that checking shorter access control lists is faster, and they employ various encodings. One class allows patterns and wildcards in names of subjects or privileges, so that a single ACL-entry can replace many; another involves replacing a set of ACL-entries that grants privileges with a set of ACL-entries imposing prohibitions on the compliment, or *vice versa* whichever is shorter. Note, however, access control lists that employ these encodings might be shorter but are often harder for humans to understand. There is a trade-off: representing security policies in a form that humans can validate versus reduced execution time for enforcement. Cheap enforcement of policies that nobody understands is arguably a dubious goal.

7.1.2 Capabilities

A *capability* to grant privileges *Privs* for operations on an object O corresponds to a pair $\langle O, Privs \rangle$; any subject *holding* that capability is granted the speci-

[7] If an access control list is shortened by using an encoding scheme or indirection then additional computation or lookups might be required to determine what principals each ACL-entry governs. The cost of the additional computation per ACL-entry might well exceed the savings of processing fewer ACL-entries.

Copyright 2010. Fred B. Schneider. All rights reserved.

fied privileges for the named object. Thus, each capability held by a subject S encodes some non-empty cell in S's row of the access control matrix, and an authorization relation *Auth* is faithfully represented by a set of capabilities where, for all subjects S and objects O, capability $\langle O, \mathit{Privs} \rangle$ is held by S if and only if $\langle S, O, \mathit{Privs} \rangle \in \mathit{Auth}$ is satisfied.

We restrict execution to comply with *Auth* by employing a run-time environment in which

- each subject S invoking an operation *op* on an object O must hold a capability $\langle O, \mathit{Privs} \rangle$ where $op \in \mathit{Privs}$ is satisfied, and
- no subject is able to counterfeit a capability, nor is any subject that holds a capability *cap* able unilaterally to change what object is named in *cap* or what privileges *cap* grants.

And we support changes to *Auth* allowed by the commands in a DAC policy through run-time environment routines that enable an authorized subject to

- create a new object and, as a result, receive a capability for that object,
- transfer capabilities it holds, with attenuation and/or amplification of privilege applied when constructing the capability the recipient gets, or
- revoke capabilities derived from capabilities it holds.

All capabilities for an object O are thus derived from the capability held by subject that creates (owns) O.[8] So, consistent with the defining characteristics of DAC, access privileges are controlled by the owner or by subjects whose authority can be traced to the owner.

Since a defining characteristic of subjects and of threads of control is to make requests, identifying instances of the former with instances of the latter is sensible. We can then define the protection domain for a subject to be the set of capabilities addressable by some corresponding thread of control. The address space could map names for capabilities to main memory addresses, if some form of memory protection prevents each thread of control from accessing memory outside its address space. Or, it could map names for capabilities to abstractions implemented by routines in the run-time environment (hence, not directly addressable by the thread of control). Implementors of a run-time environments thus have considerable flexibility. Moreover, notice that our definition of protection domain simplifies supporting the common kinds of protection-domain transitions—specifically, invoking or returning from a procedure, initiating request processing in another process, or executing a system routine—because these actions already coincide with switching address spaces.

[8]Some run-time environments impose further restrictions and allow transferring only those capabilities that contain a *copy-capability* privilege; some allow transferring a privilege *op* only if there is a corresponding a copy-flag privilege (i.e., $op*$) present.

Copyright 2010. Fred B. Schneider. All rights reserved.

Object Naming

Unless object name O in a capability $\langle O, Privs\rangle$ continually refers to the same unique object—independent of what subject is exercising $\langle O, Privs\rangle$ or when—then transferring capabilities or even storing them could grant unintended privileges. For example, if an object name O designating object Obj_1 is later recycled to designate object Obj_2, then holding $\langle O, Privs\rangle$ eventually grants privileges to access Obj_2 (even though access only to Obj_1 was authorized).[9] And if an object name O designates object Obj_1 in one address space but Obj_2 in another, then transferring $\langle O, Privs\rangle$ from one thread of control to another can give the recipient privileges to a Obj_2 (even though it is Obj_1 that the sender is authorized to access).

With *capability-based addressing*, capabilities become the sole means for accessing objects.[10] The most direct implementation uses the virtual address where an object is stored as the name that appears in a capability for that object. Virtual address spaces found on today's processors are large enough (64 bits) to make this feasible. With 64 bit names, distinct objects can each be assigned globally distinct virtual addresses for now until (almost) eternity. Thus, object names (virtual addresses) now never need to be recycled; different threads of control use the same name only if they are referring to the same object.

Address-translation hardware for support virtual memory, however, is not the only way to implement a mapping between names in capabilities and addresses where objects are stored. A software run-time environment can be used, albeit with some overhead for object accesses. The run-time environment maintains a table that maps object names to memory addresses; and it provides routines that subjects call for invoking operations on objects, where the address to which control is transferred for the invocation is obtained by the run-time from the table. Notice, when this scheme is being used, an object can be relocated to another address as execution proceeds, because updating only a single run-time table entry—rather than the name field in every capability for that object—suffices. Also, using this software-based indirection scheme, the address where an object is stored can be real or virtual.

Capability Authenticity

To preserve *capability authenticity*, we must prevent unauthorized creation of new capabilities and prevent unauthorized changes to existing capabilities. This we seem to require an authorization mechanism for implementing an authorization mechanism! Fortunately, only a very simple form of authorization is required for protecting capabilities—a form of write-restricted main memory. And a variety of approaches, which we describe below, can implement that.

[9] This problem can be avoided if the run-time environment's object-deletion routine also eliminates or revokes all capabilities naming the object. But, as will become clear, all implementations of capabilities are not equally amenable to the bookkeeping necessary for this.

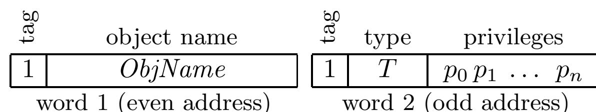
[10] The bundles proposed in the solution to the confused deputy problem are thus a form of capability-based addressing.

Copyright 2010. Fred B. Schneider. All rights reserved.

However, a subject not being executed has no footprint in main memory, raising a question about where capabilities are stored for long-lived objects (e.g., files) that no subject is accessing. The answer is the same for all approaches to capability authenticity. Capabilities for long-lived objects are themselves archived in long-lived objects (e.g., directories). A trusted subject[11] runs whenever the system is executing; this trusted subject holds the capabilities necessary to read these archives and to transfer those capabilities to the appropriate subject.

Capabilities in Hardware-Implemented Tagged Memory. Worth understanding because the approach is both elegant and illuminating, the required hardware for supporting tagged memory is rarely found in commodity computers. So commercial deployments of the approach have been few and far between.

Each register and word of memory stores a 1-bit tag in addition to ordinary data. Capabilities, but no other words, have tags that equal 1. For example, assume 64-bit words, so that object names can be 63-bit virtual memory addresses. The hardware might then define a capability to be any two consecutive words starting on an even address and whose tags equal 1:



Here, T identifies the *type* of object *ObjName*, which in turn defines how to interpret bit string $p_0\, p_1\ \ldots\ p_n$ in the privileges field. For some types, each bit p_i specifies whether the capability grants its holder corresponding privilege $priv_i$ for *ObjName*, and $priv_i$ might well be a different privilege for objects having different types; with other types (e.g., memory segments), pre-defined substrings of $p_0\, p_1\ \ldots\ p_n$ might give other properties of *ObjName* (e.g., the segment length) needed for enforcing an access control policy.

Tag bits alone are not sufficient to ensure that capabilities cannot be counterfeited or corrupted, though. The processor's instruction set also must be defined with capability authenticity in mind. Typically, this includes a few special-purpose instructions as well as restrictions on updates to memory words whose tag values equal 1.

For example, a special-purpose user-mode instruction[12]

`cap_copy` @*src*, @*dest*

might be supported for copying two consecutive words of memory from a source address @*src* to a destination address @*dest*, where execution fails (causing a trap) unless (i) the source and destination both start on an even address, (ii) the tags on both words at the source equal 1, and (iii) the subject has read access at the source addresses and write access at the destination addresses.

[11] This trusted subject may, in fact, be the run time environment.
[12] We write @w to denote the address of w.

Copyright 2010. Fred B. Schneider. All rights reserved.

And user-mode `invoke` and `return` instructions might be provided for invoking operations on objects. If *cap* is a capability for an object named *ObjName*, *op* is an integer satisfying $0 \leq op \leq n$, and bit p_i equals 1, then execution of

`invoke` *op*, @*cap*

(i) loads integer *op* into some well known register (say) `r1`, (ii) constructs a capability *retCap* having as its object name the address of the instruction following the `invoke` and having `return` as its type, (iii) stores *retCap* someplace accessible to execution by *ObjName* (e.g., on a run-time stack) leaving @*retCap* in a well known register (say) `r2`, and (iv) starts executing at address *ObjName*. In short, the `invoke` transfers control to *ObjName* only if the subject holds a capability that authorizes the invocation; it causes a trap otherwise.

The effect of executing `invoke` presumes code at (virtual address) *ObjName* implements a `case` statement that, based on the value found in `r1`, transfers control to the code for *op*. Later, when the code for *op* completes, it can execute

`return` @*retCap*

which, provided address @*retCap* is accessible and *retCap* is a capability whose type is `return`, loads the program counter with the address appearing as the object name in *retCap* (presumably the instruction in the caller following the `invoke`) and also (to prevent reuse) sets the tags in *retCap* to 0. So the `return` transfers control back to the invoker, and may do so only once. As might be expected, `return` causes a trap if capability *retCap* at @*retCap* cannot be read or if *retCap* is not a capability having type `return`.

Special instructions for constructing and modifying capabilities are not needed, provided (i) the processor permits instructions executed in system mode to read or change the tag or contents of any memory location, and (ii) executing such instructions in user mode causes a trap. This is because a run-time environment can then provide system routines for creating and changing capabilities. These system routines execute in system mode but can be invoked by executing in user mode.

- New objects and their capabilities are created by invoking a system routine that instantiates the object, generates a corresponding capability *cap*, stores *cap* in the caller's address space, and returns @*cap* to the caller.

- Capabilities are propagated from one subject to another when the subjects do not share an address space (so `cap_copy` cannot be used) by invoking system routines to send and receive capabilities. The run-time environment presumably has access to every subject's address space and can execute the two `cap_copy` instructions needed.

- The functionality of the `cap_copy` instruction is extended with attenuation and amplification, where desired, by having the source and destination invoke system routines.

Copyright 2010. Fred B. Schneider. All rights reserved.

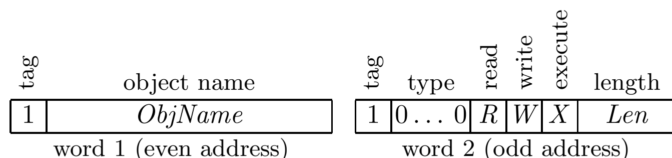


Figure 7.4: Example Format of Capability for a Memory Segment

- Attenuation is supported by a system routine that takes as inputs (i) a set *Rmv* of privileges to remove and (ii) the address of a capability having some set *Privs* of privileges; it returns the address of another capability for the same object but with $Privs - Rmv$ as its set of privileges.
- Amplification is supported by a system routine that takes as inputs the addresses for two capabilities: one capability names an object O with some type T and a set $Privs_O$ of privileges, and the second capability gives type T as its name[13] (and `type` as its type) and a set $Privs_A$ of privileges; it returns the address of a new capability having $Privs_O \cup Privs_A$ as its privileges.

The approach just outlined can even be used to ensure that appropriate privileges be held for each and every memory access a process makes. To implement this, capabilities are associated with memory segments.[14] Figure 7.4 suggests a format for such a capability; it is an instance of the general capability format proposed above. The type (0) indicates that the capability is for a memory segment; *ObjName* gives the starting address of the memory segment; privileges bits p_0, p_1, and p_2 (labeled R, W, and X in Figure 7.4) specify privileges for operations read, write, and execute; and suffix $p_3\, p_4\, \ldots\, p_n$ of the privileges specifies the segment length.

The processor hardware might then incorporate memory segment capabilities into memory access, as follows.

- The processor provides a set of *segment capability registers* to store capabilities for memory segments.[15] A memory access is allowed to proceed only if the operation being requested (i) names a word in some memory segment whose capability is currently found in a segment capability register and (ii) the requested operation (read, write, or execute) is one for which the corresponding privilege bit is set in that capability. The memory access is blocked and an *access-fault* trap occurs, otherwise.

[13]When typed objects are not supported, then the hardware might require that the two input capabilities name the same object.

[14]A *memory segment* is a contiguous region of an address space; it is defined by a starting address and a length.

[15]In some architectures, these register might contain a capability for a segment that itself contains capabilities for segments. This additional level of indirection allows a small number of registers to support accessing a significantly larges number of segments.

Copyright 2010. Fred B. Schneider. All rights reserved.

- The processor provides a system-mode instruction

  `load_scr` *scr*, @*cap*

  for loading a segment capability register *scr* with the memory segment capability stored at address @*cap*; when in user mode or when *cap* is not a capability whose type is 0, executing this instruction causes a trap.

The run-time environment then provides system routines that allow user-mode execution to *map* a memory segment and to *unmap* a memory segment. The set of memory segments that are mapped establishes the current protection domain by defining what memory, hence what set of capabilities, can be addressed by the executing subject. In some systems, the set of mapped memory segments for a given subject is partitioned into subsets: memory accessible to every subject, memory accessible only to this subject throughout its execution, and memory accessible because some operation on a given object is being executed.

A run-time environment might allow more memory segments to be mapped at a given time than there are segment capability registers. To accomplish this, run-time routines multiplex the segment capability registers in much the same way an operating system multiplexes a small set of page frames to create a much larger virtual memory. Specifically, the run-time environment maintains a set *MappedSegs* of the capabilities for memory segments that currently are mapped. Whenever an access fault trap occurs, the corresponding trap-handler checks whether *MappedSegs* contains a capability *seg_cap* (say) for the memory segment encompassing the address that caused the access-fault. If *MappedSegs* does, then the trap handler replaces the contents of some segment capability register with *seg_cap*; otherwise, the memory access attempt is deemed to violate the security policy.

Capabilities in Protected Address-Spaces. Processors intended to run operating systems invariably support some form of memory protection, if only to isolate operating system code and data from errant applications. The hardware allows memory to be partitioned into one or more regions and, for each, enforces access restrictions on all words in that region. Although coarse-grained, this kind of memory protection can suffice for implementing capability authenticity. Software plays the dominant role in this approach. So higher run-time costs are incurred, but practical experience suggests that these overheads need not be prohibitive.

The basic strategy is to segregate capabilities and store them in memory regions that cannot be written in user mode. Run-time environment routines, which execute in system mode, are granted write-access to these memory regions. And all functionality that requires creating or modifying capabilities is then supported by such routines (rather than by special-purpose instructions, as for tagged memory). So there are run-time environment routines for instantiating a new object (and its corresponding capability), copying capabilities, a subject's sending or receiving capabilities, and performing protection-domain

Copyright 2010. Fred B. Schneider. All rights reserved.

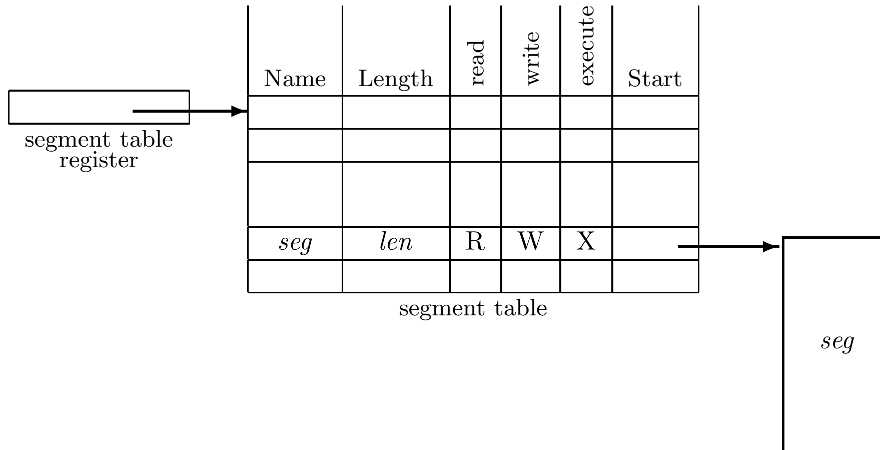
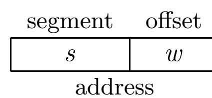


Figure 7.5: Addressing with a Segment Table

transitions such as invocation and return from operations on objects (with attenuation and amplification).

Storing Capabilities in Virtual Memory Segments. One version of this protected address-space approach builds on the memory segment abstraction provided by contemporary segmented virtual memories. A *virtual address* here is a bit string; some predefined, fixed-length prefix of that bit string is interpreted as an integer s that *names* a segment, and the remaining suffix specifies an integer offset for a word w in the segment:



A *segment table*, which comprises a set of *segment descriptors*, is used during execution to translate virtual addresses into real addresses. See Figure 7.5. Each segment descriptor gives the name, length, and start for a segment, as well as *access bits* (R, W, and X) that indicate whether words in the segment can be read, written, and/or executed. The segment table thus defines an address space and access restrictions on the contents of that address space.

The operating system's process abstraction associates some threads of execution with a segment table. Whenever one of these threads is executing, the (real) address of the corresponding segment table is held in the processor's *segment table register*, which is considered part of the processor context.[16] And

[16] The *processor context* comprises the general-purpose registers, the program counter, and any other processor state that must be saved and restored when a processor is being time-multiplexed by a collection of tasks.

Copyright 2010. Fred B. Schneider. All rights reserved.

whenever a trap or interrupt occurs, the hardware (i) stores in memory the contents of the segment table register and (ii) loads the segment table register with a pre-specified address for a new segment table. A system-mode instruction is also provided for loading the segment table register. We can thus arrange for execution by the run-time environment and for execution by each process to use different segment tables and, therefore, to have different (virtual) address spaces and/or different access restrictions being enforced.

The use of segments to implement memory regions that exclusively store capabilities should now come as no surprise. By using access bits in segment descriptors, we can construct virtual address spaces for which writes to segments storing capabilities are prohibited; user-mode execution occurs using such a segment table and, therefore, cannot counterfeit or corrupt capabilities. Moreover, if we identify subjects with user-mode processes, then we can define the capabilities held by a subject to be the set capabilities that the corresponding user-mode process can read:

- Processes execute in user mode with a segment table in effect that allows read (but not write) access to segments storing the capabilities held by the corresponding subject. Read access to other segments storing capabilities is not allowed by the segment table.

- Run-time environment routines execute in system mode with a segment table in effect that allows read and write to all segments storing capabilities (i.e., for any process).

The size and format of capabilities being implemented here is constrained only by the software, which must be able to interpret them unambiguously. And the choice of what segments are assigned to store capabilities is unconstrained, provided the run time environment can determine whether a given segment is dedicated to storing capabilities or storing other content. Often a convention is adopted (e.g., only segments named with low-numbers store capabilities), but alternatively the run-time environment could explicitly save the names of all segments dedicated to storing capabilities. Segment descriptors on some architectures contain bits unused by the address-translation hardware, and a run-time environment might employ these to indicate whether a segment stores capabilities.

The use of segments to store capabilities does not preclude capabilities from being used to control memory access. As with our tagged-memory implementation of capabilities, the run-time environment would provide routines to map or unmap a memory segment, which might or might not store capabilities. The map and unmap routines for a segment s would check that the invoker holds a capability cap_s for s, where cap_s specifies appropriate read, write and/or execute access privileges. If the invoker does hold such a capability, then map (unmap) modifies the invoker's segment table and adds (deletes) the segment descriptor for s. The segment table thus simulates the segment capability registers, which explains why the information found in a segment descriptor is so similar to what is found in a memory segment capability.

Copyright 2010. Fred B. Schneider. All rights reserved.

When capability authenticity is implemented by using memory segments, a segment table specifies which capabilities the executing process holds and which other data it can access (because those data segments have been mapped). So a segment table defines a protection domain, and protection-domain transitions require changing that segment table.

For example, a protection-domain transition typically accompanies invoking an operation. A run-time environment routine for that might expect to be passed two arguments:[17] the name *op* of the operation and the address @*obj_cap* of a capability *obj_cap* for an object on which *op* is to be performed. Execution of the run-time routine then proceeds as follows.

1. *Authorization.* Validate that the segment named in @*obj_cap* is dedicated to storing capabilities (so *obj_cap* is a capability), the caller can read *obj_cap* (and thus holds that capability), and *obj_cap* grants permission for *op*.

2. *Segment Table Construction.* If the authorization conditions were found to be valid in step 1, then build a segment table containing segments to store capabilities and state that should be accessible when performing *op* on the object named by *obj_cap*:

   - Segments that should not be accessible when *op* executes would be deleted from the caller's segment table.
   - Segments would be added to the caller's segment table for data and capabilities that should become accessible, including amplified or attenuated versions of designated capabilities held by the caller.

3. *Control Transfer.* In software, orchestrate the designated transfers of control to and from *op* in *obj*: (i) construct a capability of type `return` for the return address, (ii) store that capability at a well known virtual address in a segment of capabilities readable when executing in the invoked operation, (iii) load the segment table register with the address of the new segment table, and (iv) transfer control to the code for *op* in *obj*.

Run-time environment support for other functionality involving capabilities would be built along similar lines.

Storing Capabilities in Kernel Memory. Even when segmented virtual memory is not supported by the processor, some form of memory isolation usually will be. Hardware support for a single memory region (for the operating system) that cannot be corrupted by user-mode execution is typical. For implementing capabilities in this setting, it suffices that all reads and writes to that region be mediated by a run-time environment:

- The memory region would store all capabilities and, for each, identify the subject(s) holding that capability.

[17]These arguments give the same information that the `invoke` instruction expects to find in registers for the tagged-memory implementation of capabilities discussed above.

Copyright 2010. Fred B. Schneider. All rights reserved.

- The run-time environment would ensure that its routines are the sole means by which user-mode execution can read or change the memory region.

The operating system kernel now is an obvious choice of run-time environment for implementing capabilities. Moreover, this choice synergizes with the usual practice of using the operating system kernel to implement the process abstraction. Processes are the sole source of requests that must be authorized. So when support for capabilities is being provided by a kernel, it is natural to identify subjects with user-mode processes.

Various schemes are then employed for storing in the kernel the capabilities processes hold. Invariably, these schemes associate some kind of a table, herein called a *c-list* (for "capability list"), with each process. Some schemes have separate c-lists for every process; in others, c-lists are shared by multiple processes. Each entry in a c-list stores a single capability and is identified by a unique index, typically a small integer. All references to capabilities by processes are thus indirect, specified relative to the c-list implicitly associated with the process making the reference.

Kernel calls then provide the sole means for creating, examining, and manipulating capabilities and the c-lists that store them. For example, send and receive kernel calls might be provided to pass capabilities from one process to another, whether or not those processes share a c-list. Specifically, a process P might invoke send, giving a destination process P' and indexes for some capabilities that would then be buffered at P' for receipt; by invoking receive, P' would move any buffered capabilities to its c-list and obtain indexes for where they are stored.[18]

The kernel would also provide routines for creating and managing c-lists. The kernel call for new process creation would presumably take an argument specifying whether the new process will share the caller's c-list or have a new one (and, for a new c-list, what subset of the caller's capabilities and privileges to include). And invoke/return operations might be implemented as kernel calls, in order to allow execution of an invoked routine to use a new c-list. This new c-list would be populated from the caller's c-list with copies of capabilities the caller indicates in arguments as well as capabilities obtained though attenuation and amplification of capabilities the caller indicates.

Cryptographically Protected Capabilities. The protections required for capability authenticity are well matched to the security properties that digital signatures provide. Consequently, digital signatures provide an alternative to hardware-implemented tags or protected memory for supporting capability authenticity. The costs—both in compute time and space—limit the applicability of this approach, but digital signatures are a good way to implement capabilities in some settings.

[18]A kernel call to examine the capability stored for each index would then be used by P' to determine what the new capabilities authorize.

Copyright 2010. Fred B. Schneider. All rights reserved.

Our starting point is a digital signature scheme comprising algorithms to generate and to validate signed bit strings, where the following properties hold.

- *Unforgeability.* For any bit string b, only those principals that know private key k can generate k-signed bit string $\mathcal{S}_k(b)$.
- *Tamper Resistance.* Principals that do not know k find it infeasible to modify $\mathcal{S}_k(b)$ and produce a different k-signed bit string $\mathcal{S}_k(b')$.
- *Validity Checking.* Any principal that knows the public key K corresponding to a private key k can validate whether something is a k-signed bit string.

We then implement capabilities as signed bit strings, because the Unforgeability and Tamper Resistance properties imply capability authenticity[19] and the Validity Checking property gives a way for system components to ascertain whether a bit string purporting to be a capability is authentic.

Specifically, for *cap* a bit string that gives the name, type, and privileges for an object O, the k-signed bit string $\mathcal{S}_k(cap)$ serves as a capability that grants its holders the specified privileges for O provided:

(i) private key k is known only by component(s) authorized to generate capability $\mathcal{S}_k(cap)$ for O, and

(ii) corresponding public key K is available to any principal needing to check the authenticity of a capability $\mathcal{S}_k(cap)$.

Notice that code to generate capabilities or to check capability authenticity need not execute in system mode—user mode works just fine for performing the necessary cryptographic calculations. Confidentiality of private keys used in such user-mode computations is required, but this confidentiality can be implemented through memory isolation typically provided by a run-time environment. And digital certificates signed by some well-known trusted authority are an obvious means for making public keys available for Validity Checking.

Any component that has a private key can generate capabilities. If, as usual, each component has a distinct private key, then capabilities generated by different components can be distinguished according to the public key that validates them. This built-in capacity for attribution allows capabilities to be ignored when they have been generated by components that lack the authority. In some systems, only one component, such as the operating system kernel, is authorized to create capabilities. So a single public key checks the authenticity of

[19]More precisely, this implementation of capability authenticity requires a digital signature scheme that is secure against selective forgery under a known message attack. Security against a known message attack accounts for the possibility that attackers might have access to some authentic k-signed bit strings (i.e., capabilities) but would not be able to get an arbitrary bit string signed (because any reasonable security policy the system enforces should preclude generating arbitrary capabilities on demand). Selective forgery is the right concern here, because we want to prevent an attacker from generating capabilities that grant specific privileges for specific objects.

Copyright 2010. Fred B. Schneider. All rights reserved.

all capabilities. In other systems, a well known mapping defines which public key validates capabilities for each given object; the mapping typically uses some characteristic(s) of an object, such as its type, to select that public key. Capabilities for all objects of each given type T might, for example, be validated by a corresponding public key K_T.

Capabilities implemented as signed bit strings are easy to transfer between subjects, protection domains, and even between computers. It is just a matter of copying the bits (assuming infrastructure is in place to disseminate public keys that can be trusted for checking capability authenticity). However, performing amplification and attenuation for a capability $\mathcal{S}_k(cap)$ being transferred is another matter. The Tamper Resistance property implies that a component with knowledge of private key k must be involved—either to generate from scratch a capability with modified privileges or to modify the privileges in $\mathcal{S}_k(cap)$ directly. But disseminating private keys is risky, and cryptographic computations create performance bottlenecks. So designers often respond by avoiding amplification and attenuation in their system designs. An impoverished mechanism thus forces the designer to eschew the Principle of Least Privilege, resulting in a system that is not as secure as it could be.

Three costs are noteworthy when a digital signature scheme is used to support capability authenticity: the amount of space required to store a k-signed bit string, the amount of time required to generate one, and the amount of time required to validity check one. To facilitate comparisons with schemes that use hardware-implemented tagged memory or protected address-spaces, we assume that the name, type, and privileges conveyed by a capability can be represented in 64 bits. Also, we adopt as the unit of measure for execution the required time to make a kernel call, since that run-time cost dominates the execution for the alternatives.

For a digital signature scheme being deployed in 2010, NIST recommends 2048-bit RSA with SHA-256. The cost estimates that follow are derived from available implementations of those algorithms on commodity hardware, although the conclusions hold for other digital signature algorithms as well.

- To create a k-signed bit string b, a tag is appended to b. The length of this tag depends on the RSA key size and not on how long b is; for 2048-bit keys, that tag will be approximately 2048 bits. Thus, our implementation of capabilities as signed bit strings entails a substantial space overhead—a 2048 bit tag is required in order to protect 64 bits of content.

- The execution time required to create or check the validity of a tag for a 64 bit string b is dominated by the RSA key size. Creation of a k-signed bit string $\mathcal{S}_k(b)$ using a 2048-bit RSA key takes orders of magnitude longer than the time required for a kernel call; validity checking takes somewhat less time but the execution time still is orders of magnitude longer than the time required for a kernel call.

Needless to say, cryptographic protection is a relatively expensive way to implement capability authenticity.

Copyright 2010. Fred B. Schneider. All rights reserved.

Cryptographic protection, however, can be quite attractive for use in distributed systems. Consider the costs of the alternatives in that setting. If capabilities can migrate between computers then implementing capabilities by using hardware-implemented tags or protected memory regions leads to communication with remote run-time environments. The integrity and authenticity of such messages must be ensured. Digital signatures are the usual defense here, but signature generation and validity checking now add to the costs of operations implemented by a remote run-time environment. So implementing capabilities using hardware-implemented tags or protected memory regions leads to run-time costs comparable to the costs of cryptographic protection.

In contrast, cryptographic protection allows the authenticity of a capability to be checked locally in user mode, which will be considerably cheaper than querying a run-time environment on the (possibly remote) computer that generated that capability. Also, transferring a capability between two principals executing on the same computer does not require a remote run-time environment to serve as an intermediary. So when cryptographic protection is used, the execution times for capability authenticity checking or transfer operations need not incur the expense of inter-processor communication. The costs for cryptographically protected capabilities in a distributed system therefore are lower than the costs with hardware-implemented tags and protected memory regions.

Capabilities Protected by Type Safety. Programs written in type-safe programming languages declare a *type* for each variable. Execution is then restricted accordingly. Since support for capabilities also involves enforcing restrictions on execution, a natural question is whether the restrictions type safety introduces can be used to implement the restrictions capabilities require. We answer in the affirmative here by defining types for capabilities, where type safety implies (i) possession of a suitable capability is necessary for executing each operation defined on an object, and (ii) capability authenticity is enforced.

Type Safe Execution. A type T defines (i) a set $vals_T$ containing values that includes the special constant $\perp$ indicating uninitialized, and (ii) a set ops_T of *operations* defined on values in $vals_T$. The following restrictions are then enforced for *type-safe execution*:

> **Type-Safe Assignment Restriction.** Throughout execution, variables declared to have type T only store elements of $vals_T$. □

> **Type-Safe Invocation Restriction.** Throughout exececution, only operations in ops_T are invoked for values in $vals_T$. □

And for any types T and T', relation $T \preceq T'$ is defined to hold if and only if $vals_T \supseteq vals_{T'}$ and $ops_T \subseteq ops_{T'}$ both hold. Thus, $T \preceq T'$ characterizes when the type-safe execution restrictions above are not violated by storing values of type T' in variables declared to have type T.

Copyright 2010. Fred B. Schneider. All rights reserved.

A static check of the program text can establish that execution of a given assignment statement always complies with the Type-Safe Assignment Restriction. Assignment statement $v := Expr$ evaluates $Expr$ and stores the resulting value in variable v. Letting $type(x)$ denote the type of a variable or expression x, the following condition implies that $Expr \in vals_{type(v)}$ holds, which is what Type-Safe Assignment Restriction requires for executions of $v := Expr$.

> **Type-Safe Assignment.** Assignment statement $v := Expr$ exhibits type-safe execution provided $type(v) \preceq type(Expr)$ holds. □

The condition is statically checkable, because the declaration of v provides $type(v)$, and the declarations of the variables and operators in $Expr$ suffice for deducing a type for the value $Expr$ produces.

Next, consider an *invocation statement*

$$\textbf{call } obj.\mathrm{op}(Expr_1, \ldots, Expr_i, \ldots, Expr_N) \tag{7.1}$$

since these will figure prominently in our language-based treatment of capabilities. Here, obj is a program variable that designates some object. The definition for $type(obj)$ presumably contains declarations for methods; these implement the operations supported on instances of $type(obj)$, where a declaration for a method op would have the following form.

$$\mathrm{op}: \textbf{method}(parm_1 \colon T_1,\ \ldots,\ parm_i \colon T_i,\ \ldots,\ parm_N \colon T_N)\ \ body_{\mathrm{op}}\ \ \textbf{end}$$

Execution of invocation statement (7.1) assigns the value of each argument $Expr_i$ to the corresponding formal parameter $parm_i$ and then executes $body_{\mathrm{op}}$.

To ensure type-safe execution for invocation statement (7.1), we must be concerned with Type-Safe Assignment Restriction and with Type-Safe Invocation Restriction. Assume that checking has established $body_{\mathrm{op}}$ will exhibit type-safe execution when started in a state where $parm_i = Expr_i$ holds for $1 \leq i \leq N$. Type-Safe Assignment Restriction for (7.1) is then implied by Type-Safe Assignment Restriction for $parm_i := Expr_i$ where $1 \leq i \leq N$ which, according to Type-Safe Assignment, requires $T_i \preceq type(Expr_i)$ to hold for $1 \leq i \leq N$. And Type-Safe Invocation Restriction requires that $obj \neq \bot$ and $\mathrm{op} \in ops_{type(obj)}$ hold. Three conditions thus characterize type-safe invocation statements.

> **Type-Safe Invocation.** An invocation statement
>
> $$\textbf{call } obj.\mathrm{op}(Expr_1,\ Expr_2,\ \ldots,\ Expr_N)$$
>
> for a method
>
> $$\mathrm{op}: \textbf{method}(parm_1 \colon T_1,\ \ldots,\ parm_i \colon T_i,\ \ldots,\ parm_N \colon T_N)\ \ body_{\mathrm{op}}\ \ \textbf{end}$$
>
> exhibits type-safe execution provided the following hold:
>
> – $obj \neq \bot$
>
> – $\mathrm{op} \in ops_{type(obj)}$

Copyright 2010. Fred B. Schneider. All rights reserved.

– $type(parm_i) \preceq type(Expr_i)$ for $1 \leq i \leq N$. □

Condition, $obj \neq \bot$, might have to be checked at run-time but only if analyzing the program text cannot guarantee that $obj \neq \bot$ will always hold prior to reaching the invocation statement; the other two conditions can be discharged by using the type declarations present in the program text.

Support for Capabilities. We now explore how capabilities can be implemented using types. For a type T whose values are objects and whose operations include m_1, m_2, ..., m_N (and perhaps others too), the *capability type*

$$\mathbf{cap}(T)\{m_1, m_2, \ldots, m_N\}$$

defines a set of values identical to the values of type T and defines a set of operations that contains only those operations both supported by T and appearing in list m_1, m_2, ..., m_N of operations the capability type authorizes:

$$\begin{aligned} vals_{\mathbf{cap}(T)\{m_1,m_2,\ldots,m_N\}} &= vals_T \\ ops_{\mathbf{cap}(T)\{m_1,m_2,\ldots,m_N\}} &= ops_T \cap \{m_1, m_2, \ldots, m_N\} \end{aligned}$$

Substitution into the definition of $\preceq$, we get that

$$\mathbf{cap}(T)\{m_1, m_2, \ldots, m_P\} \;\preceq\; \mathbf{cap}(T')\{m_1{}', m_2{}', \ldots, m_Q{}'\}$$

holds if and only if $T \preceq T'$ and $\{m_1, m_2, \ldots, m_P\} \subseteq \{m_1{}', m_2{}', \ldots, m_Q{}'\}$ hold. Therefore, if $C \preceq C'$ holds for capability types C and C' then a Type-Safe Invocation for operation op of an object designated by variable *obj* having type C will exhibit type-safe execution even if an object having type C' is stored in *obj*.

To understand type-safety for two capabilities, consider capabilities *cap1* and *cap2* for objects of type *dbase*, which supports three operations: read(x, *val*), update(x, *val*), and reset().

$$\begin{aligned} \mathbf{var}\;\; & cap1 : \mathbf{cap}(dbase)\{\text{read}, \text{update}\} \\ & cap2 : \mathbf{cap}(dbase)\{\text{read}\} \end{aligned}$$

Assume that *cap1* designates object *db1* and *cap1* designates object *db2*.

An invocation statement **call** *cap1*.update(...), when its argument values have suitable types, satisfies Type-Safe Invocation because $cap1 \neq \bot$ holds (by assumption) and *cap1* is declared to have a capability type that includes operation update (since $ops_{type(cap1)} = \{\text{read}, \text{update}\}$). But **call** *cap2*.update(...) cannot satisfy Type-Safe Invocation, since update $\in ops_{type(cap2)}$ does not hold; and this is exactly what we should desire—*cap2* does not convey priviliges for operation update and type-safety is prohibiting the attempt to invoke update through *cap2*.

Assignment statement $cap2 := cap1$ satisfies Type-Safe Assignment because $type(cap2) \preceq type(cap1)$ holds. This assignment stores into *cap2* a capability for *db1* that authorizes fewer operations than *cap1* does; the assignment statement

Copyright 2010. Fred B. Schneider. All rights reserved.

implements attenuation of privilige. Assignment statement $cap1 := cap2$ does not satisfy Type-Safe Assignment since $cap2 \preceq cap1$ does not hold. This is desirable, because it means that program fragment

$$cap1 := cap2; \;\; \textbf{call}\; cap1.\text{update}(\ldots)$$

is not type safe, as we should want—allowing the fragment to execute would enable a subject holding a capability (*cap2*) for *db2* that authorizes only read operations to invoke an update operation on *db2* (since invocation statement **call** *cap1*.update(...) is type safe).

Notice how knowing that $obj \neq \perp$ will hold before a type-safe invocation statement is executed suffices to conclude that a subject possesses not just any capability but a suitable capability to proceed with the invocation. No run-time checks concerning priviliges need to be performed; that aspect of capability semantics is enforced through type-checking analysis of program text before program execution starts. Moreover, when a program-flow analysis determines that $obj \neq \perp$ is necessarily true prior to reaching a given invocation statement, then no run-time check at all is required for an invocation.

The other defining characteristic for an implementation of capabilities is that subjects are prevented from altering or forging capabilities—capability authenticity. Type-Safe Assignment is what prevents a subject from altering capabilities and increasing its priviliges. In particular, Type-Safe Assignment ensures that an assignment statement never stores a capability into a variable allowing more operations on some object than were allowed by the variable originally storing the capability.

We prevent subjects from forging capabilities by restricting the class of expressions whose evaluation produce values with capability types. This class of *capability expressions* must include (i) expressions that manufacture a capability whenever a new object is created and (ii) expressions that produce a capability already held by the subject evaluating the expression. To address (i), we introduce a capability expression **new**(T); it returns a capability authorizing all methods on a new object of type T that the run-time environment creates when **new**(T) is executed. And, to address (ii), we define all variables declared with capability types to be capability expressions, thereby allowing existing capabilities to be retrieved for copying (perhaps between subjects).

> **Capability-Valued Expressions.** An expression *Expr* is defined to have capability type $\textbf{cap}(T)\{m_1, m_2, \ldots, m_N\}$ if
>
> – *Expr* is an invocation of the built-in function **new**(T) and m_1, m_2, ... , m_N is the list of all methods that objects of type T support.
>
> – *Expr* is a variable or function application declared to have type $\textbf{cap}(T)\{m_1, m_2, \ldots, m_N\}$. □

Type-Safe Assignment allows attenuation (as was noted above), but it does not allow amplification. However, a form of amplification is intrinsic in the usual scope rule for variables declared within an object. This scope rule stipulates that

Copyright 2010. Fred B. Schneider. All rights reserved.

```
type dbase = object
        var dbCntn : map
        read: method(x:field, var val:field)
                val := dbCntn[x]
                end read
        update: method(x:field, val:field)
                dbCntn[x] := val
                end update
        reset: method()
                dbCntn := ∅
                end reset
    end dbase
```

Figure 7.6: Definition of type *dbase*

such variables may be named within that object's methods but not outside. For example, Figure 1 gives a definition for type *dbase*. It declares a single variable *dbCntnt*, which stores the state of a *dbase* instance; the scope rule allows *dbCntnt* to be named within the body of operations read, update, and reset but not elsewhere. A form of amplification thus occurs during execution of the methods, because the body of a method can directly access the object's variables. Moreover, if variables with capability types are declared within an object, then only by executing a method can these capabilities be exercised. So, a subject executing a method has amplified priviliges relative to what it had when executing outside the method.

Copyright 2010. Fred B. Schneider. All rights reserved.