Computer Security and the Internet: Tools and Jewels

Chapter 5: Protection in Operating Systems

P.C. van Oorschot

Aug 24, 2018

Comments, corrections, and suggestions for improvements are welcome and appreciated.
Please send by email to: paulv@scs.carleton.ca

Chapter 5

Protection in Operating Systems

Mass-produced computers emerged in the 1950s. Security requirements were brought to focus by 1960s time-sharing systems. 1965-1975 was the golden design age for operating system protection mechanisms, hardware protection features and address translation. Although the threat environment was considerably simpler—e.g., computer networks were largely non-existent, and the number and sources of software programs were fewer—the challenges were largely the same that face us today: maintaining separation of processes while selectively allowing sharing of resources, protecting programs from one another on the same machine, and restricting access to resources. "Protection" largely meant controlling access to memory locations. This is more powerful than it first appears. Since both data and programs are stored in memory, this controls access to running processes; input/output devices and communications channels are also accessed through memory addresses and files. Files are simply logical units of data in memory. Thus access control of memory and files provides a basis for access control to objects and devices in general.

Initially, protection meant limiting memory addresses accessible to processes, in conjunction with early virtual memory address translation, and access control lists were developed to enable resource sharing. These remain protection fundamentals. Beginning with such protection in operating systems provides a solid basis for understanding computer security. Aside from Unix, we base our discussion in large part on Multics; its segmented virtual addressing, access control, and protection rings heavily influenced later systems including Unix. Providing security-related details of all major operating systems is not our goal—rather, considering features of a few specific, real systems allows a coherent coverage highlighting principles and exposing core issues important in any system design. Unix of course has many flavours and cousins including Linux, making it a good choice. Regarding Multics, security influenced it from early design (1964-67) through later commercial availability. It was one of the most carefully engineered widely-known systems ever, and distinctive among early systems, its rich and detailed technical literature explaining its motivation and design provides invaluable learning resources.

5.1 Memory protection, supervisor mode, and accountability

MEMORY ISOLATION NEEDED. Early computers were large and expensive—and simple compared to later systems. When they were used to run single computer programs one after the other, the delay between running one and the next wasted valuable computer time. This motivated *batch processing*—programs were prepared ahead of time and submitted for later processing. Submitted jobs were "batched" together, one after the other, and run by an operator. This reduced idle CPU time and costs—but was not particularly convenient. The *time-sharing systems* emerging in the early 1960s offered an alternative for shared use of a computer, and gave users the view of running a program on their own machine in real time. While programs appeared to run concurrently, the innovation was to organize them as processes which the CPU alternated between. This is how single-user computers work today, albeit with one user running multitudes of programs concurrently.

A security issue arises immediately once more than one process runs "concurrently": the protection needed to avoid resource conflicts. A main original concern was computer memory—some isolation mechanism is needed to prevent one process writing into the memory used by another. It is useful to enforce modularity, such that benign errors in one program do not impact another (let's not mention malicious programs just yet). Even for computers running single programs one at a time, if a user process could access the computer's full memory range, errors might disrupt OS data or code—a lack of basic protection subjects even a debugger program to disruption by faulty code being debugged. Another early commercial motivation was to allow execution by one program of proprietary functionality of another, without the first having read access to the second.

SUPERVISOR PROGRAM, PRIVILEGED BIT, DESCRIPTOR REGISTER. A typical isolation mechanism began as follows. All memory references went through a hardware *descriptor register* holding a *memory descriptor* consisting of a pair `(base, bound)` of values. These indicate the lowest physical memory address accessible to the active process, and the number of addressable memory words from that point. To control the memory address range visible to a process, only the unique *supervisor program*, which runs with a *privileged mode* (supervisor mode) bit on, can load the descriptor register. In earlier machines, all programs ran this mode—there was only one mode. User programs can change the mode bit only through a single machine instruction which then immediately passes execution control back to the supervisor. The design protects memory descriptors by storing them in memory managed exclusively by the supervisor. This starting basis for process isolation thus consists of a simple three-component memory protection scheme:

1. a descriptor register constrains the address range a process can access (the supervisor maintains a descriptor for each process, loading this register as a process runs);

2. the privileged bit must be set in order for the descriptor register to be loadable (only the supervisor process runs with this bit set); and

3. the supervisor process is the only process that can alter the privileged bit, aside from a user process in the solitary case of using the dedicated machine instruction that

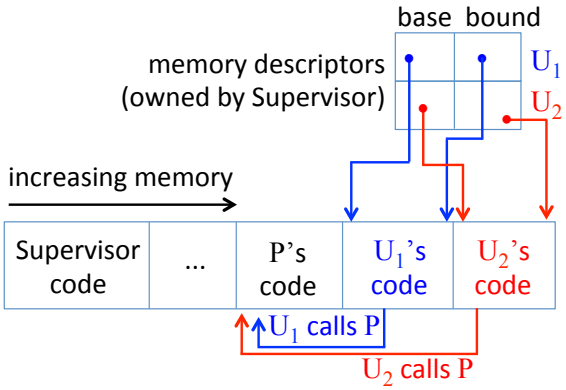

Figure 5.1: Two programs calling the same sub-program P (model circa 1970).

immediately transfers execution to the supervisor after turning the bit on.

LIMITATIONS OF MEMORY-RANGE BASED PROTECTION. With this basic memory protection scheme, the supervisor prevents user processes from altering supervisor code or data by reserving memory which user processes cannot access. This provides an *all-or-nothing* solution in the sense of either full access to everything as supervisor, or no cross-process sharing at all as a user process. This allows full isolation, but not selective cross-process sharing of memory—and selectivity matters, given the concern of unauthorized use or alteration of one process's memory by another.

Consider a service program P in memory, to be called by user processes U_1 and U_2; see Figure 5.1. Both must access P's code memory. Simple memory-range limits of a descriptor register are no longer adequate, e.g., because P needs memory to write temporary results for each of U_1, U_2. A step forward would be more specific *access permissions* allowing, say, separate read, write and execute permissions for a specified memory region. This motivates enhancements, as follows.

SEGMENT ADDRESSING WITH ACCESS PERMISSIONS. A memory *segment*, supported by *segment addressing hardware*, is a continuous block of words, representing a logical unit of information. (This approaches the definition of a *file*, a term that would soon prove to become popular.) A memory word is now addressed by a pair of values (S,W), the segment number S and word number offset W therein. Thus a level of indirection now separates this early *virtual memory* descriptor from a segment's physical address. The OS maintains a special per-process *descriptor segment* which holds a table of *segment descriptors* defining the physical memory addressable by the process. The addressing scheme controls access—a process can't access a segment that it can't "see" (i.e., reference). A processor *descriptor base register* (DBR) points to the descriptor segment of the active process. S is an index into this table, and each segment descriptor therein details a segment's physical starting address, current size, and an *access control indicator* specifying permission bits for this memory segment, for example (see Figure 5.2):

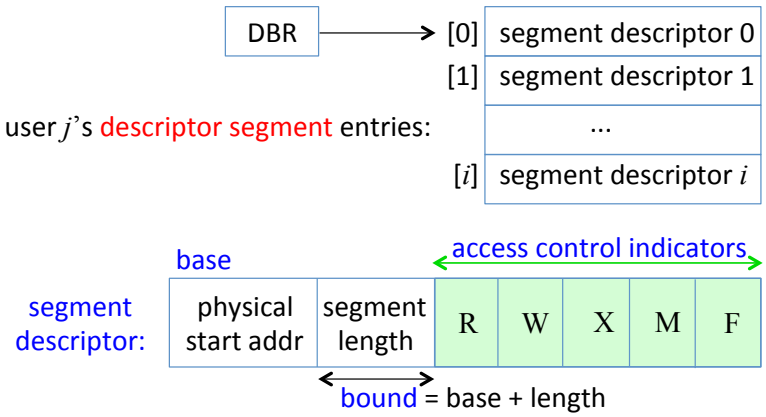

Figure 5.2: Segment descriptors, and descriptor segment holding them. When user j runs, DBR (Descriptor Base Register) points to j's descriptor segment, and indexes descriptor i therein as DBR[i]; here i corresponds to the segment S in the memory address pair (S, W).

- R: read (if off, only the supervisor has access; if on, all processes have read access)
- W: write (the segment is writable if this bit on and the execute bit is off)
- X: execute (if this bit is on and W is off, the code is not self-modifying)
- M: mode when executing in segment (supervisor or user; valid only if X is set)
- F: fault bit (if nonzero, all access attempts trap; overrides all other bits)

Note that by this design, now the same physical address region can be given (for different processes) different access permissions, through different segment descriptors.

‡**Exercise** (Memory protection design). In segment addressing above, the access control bits are essentially appended to a base-bound pair to form a descriptor. If the access control indicator bits were instead stored in (a protected part of) the physical storage associated with the target segment, what disadvantage arises?

The segment descriptors above contain physical addresses and access permissions. An improvement associates the access permissions directly with a (virtual) segment identifier, i.e., a label uniquely identifying the segment independent of physical memory details. As one motivation, permissions logically relate to virtual, not physical memory. As another, this facilitates combining the resulting (physical) descriptor segment with memory allocation schemes in some designs.

ACCOUNTABILITY, USERIDS AND PRINCIPALS. Each user on a system is given a unique identifier—their *account name*, or `userid` within the OS. To gain login access to their account, the user enters their userid and a corresponding *password*. The latter is intended to be known only to the authorized user; a matching userid-password pair is

accepted as evidence of being the legitimate user associated with userid. This is the basic authentication mechanism.[1]

The formal term *principal* is often used to abstract the "entity" responsible for code execution resulting from user (or consequent program) actions. The OS keeps track of the `userid` associated with each process; this may be viewed as the principal *accountable* for the actions of the process. The userid affects the resource *privileges* the OS grants—i.e., the permissions associated with the set of objects the process is authorized to access, or the *domain* in access control terminology.[2] The userid is also useful for administrative and billing purposes, and in debugging, audit trails, and forensics. A separate numeric `processID` is used as a process identifier for OS-internal purposes like scheduling.

A user may function in several roles—e.g., both as a regular user, and occasionally as an administrator. In this case, by the principle of least privilege, it is good practice for the user to be assigned more than one userid, and switch userid when acting in a role requiring the privileges of a different domain. Abstractly, the distinct userid may be considered a distinct principal. The converse, i.e., several users making use of the same account (same `userid`), is generally frowned upon or considered *poor security hygiene*, as it hinders accountability among other drawbacks. For shared resources, a preferred alternative is to use *protection groups* as discussed in Section 5.3.

5.2 The reference monitor, access matrix, and security kernel

Before progressing beyond basic protection, we pause to introduce several concepts needed in the sections which follow. The first, the *reference monitor concept*, was proposed in 1972 as a model for building secure systems for government use in the context of defending against malicious users. The basic notion was stated thus: *all references by any program to any program, data or device are validated against a list of authorized types of reference based on user and/or program function.* See Figure 5.3.

ACCESS MATRIX. The reference monitor is a subject-object model. A *subject*[3] is a system entity that may request access to a system object. An *object* is any item that a subject may request to use or alter—e.g., active processes, memory addresses or segments, code and data (pages in main memory, swapped pages, files in secondary memory), peripheral devices such as terminals and printers (often involving input/output, memory or media), privileged instructions, etc.

In the model, a system first identifies all subjects and objects. For each object, the types of accesses (*access attributes*) are determined, each corresponding to an access permission or privilege. Then for each subject-object pair, the system pre-defines the authorized access permissions of that subject to that object. Examples of types of accesses are read or write for a data item or memory address, execute for code, wakeup or terminate for a process, search for a directory, delete for a file, etc. The authorization of privileges

[1] For detailed discussion of passwords, see the chapter on User Authentication.

[2] Further details on how subjects, processes and domains are related are given in Section 5.7.

[3] Cf. *principal* above. A more precise definition of *subject* is given in Section 5.7.

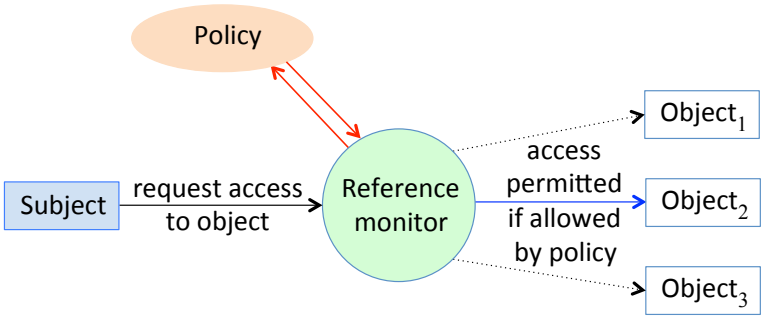

Figure 5.3: Reference monitor model. The policy check may involve, e.g., consulting an access control matrix.

across subjects and objects is modeled as an *access matrix* with rows i indexed by subjects, columns j by objects, and entries $A(i, j)$, called *access control entries* (ACEs), specifying access permissions subject i has to object j (Figure 5.4). The ACE will typically contain a collection of permissions, but we may for ease of discussion refer to the entry as a single permission with value z, and if $A(i, j) = z$ then say that subject i has z-access to object j.

Note that the access matrix itself is policy-independent; access control policy is determined by the permission content specified in matrix entries, not the matrix structure.

REFERENCE MONITOR IMPLEMENTATION. The reference monitor model is implemented and enforced by a software-hardware *reference validation mechanism*. It is conceptualized as a single monitor, but in practice would be a collection of monitors each protecting different classes of objects. Every access request made to any system object by any subject is mediated as follows. When subject i attempts z-access to object j, the request information "(i, j, z)" is intercepted—by a monitor handling objects of j's class—and the monitor checks if entry $A(i, j)$ permits z, granting access only if authorized.

For this validation mechanism to address known threats, the requirements specified as necessary are that it be:

1. tamper-proof;
2. always invoked (not circumventable); and
3. *verifiable*, or more specifically, *"small enough to be subject to analysis and tests, the completeness of which can be assured"*.

Such a validation mechanism forms the heart of a secure system, and in the context of these requirements is called a *security kernel*. The idea is to centralize, in this nucleus, the minimal control structures and code needed to enforce and verify access control and all other core security-related functions. Even 45 years later, such security kernels are uncommon—mainstream operating systems remain typically monolithic. Nonetheless, reference monitor ideas heavily influenced computer security research and practice, helping popularize the key concepts of access control and access control lists (see below).

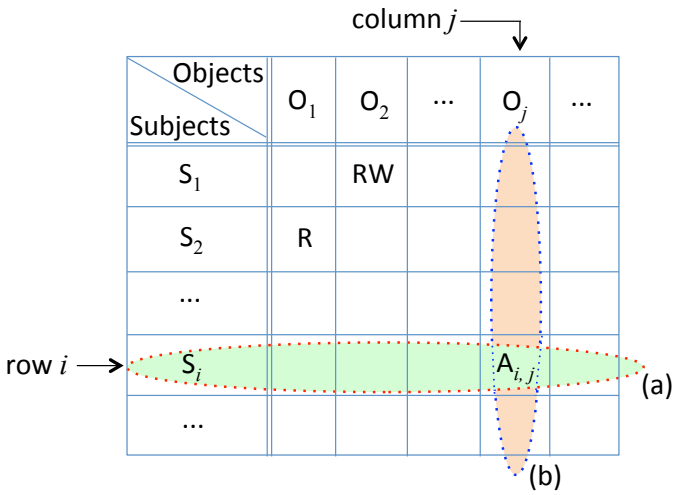

Figure 5.4: Access control matrix. $A(i,j)$ is an ACE specifying the permissions that subject i has on object j. (a) Row i can be used to build a *capabilities list* (C-list) for subject i. (b) Column j can be used to build an *access control list* (ACL) for object j.

REFERENCE MONITOR DEPENDENCIES. Aside from a properly functioning reference validation mechanism, the reference monitor depends heavily on a number of supporting functions: a trustworthy authentication system (the access matrix assumes legitimate identified subjects), properly operating hardware, physical security of this hardware and system (including storage media and any devices accessing memory), and security of the input-output communication paths between users and the system. For high confidence in the security of a computing system, its entire manufacturing chain, including the production environment of all software and hardware components, and individuals involved, must be trustworthy. The challenge is that complex computer systems require integration of components from countless international suppliers.

ACCESS MATRIX IS A MODEL ONLY. In practice, access control is often implemented by storing permissions within access matrix entries in lists organized either by rows or columns. Mechanisms doing so pre-date the matrix model. Direct implementation of a 2D access matrix is inefficient—the matrix is typically large and sparse; alternatively, naively storing long lists of entries $(i,j,A(i,j))$ makes searching inefficient.

CAPABILITIES (C-LISTS). Decomposition by row gives focus to an individual subject, detailing all the access privileges it holds. In Figure 5.4(a), for a fixed row i^*, taking the non-empty cells (objects j that i^* can access) as a list of tuples $(j,A(i^*,j))$ provides a *capabilities list* (*C-list*) for subject i^*. Each entry specifies allowed access permissions, or i^*'s capabilities on different objects j. Such lists can be held by or associated with individual subjects, referenced as needed. Care is required to prevent unauthorized copying of capabilities; design alternatives support this, e.g., storage in supervisor memory.

ACCESS CONTROL LISTS. Decomposition by fixed column j^* gives focus to an in-

dividual object; see Figure 5.4(b). Taking non-empty column cells defines a list of pairs $(i, A(i, j^*))$ ranging down all subjects i permitted access to j^*. Such a list, with entries specifying permitted access modes on j^* by different subjects i, can be constructed, perhaps co-located with j^*, and made available for use on requests to access j^*. This and variations are called *access control lists* (ACLs) for object j^*. An ACE (entry) might conceptually contain fields `(subject; access-type-id = permission-bits)` for a file object, with example values `(adamjones; RWX = 001)` where a 1-bit grants permission. Granting privileges to groups of principals rather than individual subjects may be done using ACEs and Unix-like definition of protection groups (see Section 5.3).

Exercise (Access control of ACLs). (a) Is an ACE itself subject to access control? (b) In systems where a file creator controls the file's ACE, how can a user process alter the ACE if the ACE is stored in supervisor memory? (c) A disadvantage of creator-owner schemes is that if the object owner is unavailable to alter permissions, there is no elegant way to reassign access control. In one *hierarchical access control* alternative, the principal creating an object designates a distinct primary access control principal, and a secondary access control principal subordinate to it an access control hierarchy. From this sketch, outline a full working scheme. (Hint: see [20, Fig.13].)

TICKET-ORIENTED VS. ID-LIST-ORIENTED PROTECTION. Protection mechanisms are sometimes categorized as either *ticket-oriented* or *ID-list-based* (authorization based). The first model is that of a ticket, coupon, access token, or *bearer token* allowing entry to an event, regardless of the identity of the ticket holder, provided that the ticket is recognizable as authentic. In the second model, a guard at the door does an identity-check, for example using photo ID cards in the physical world, and any entity whose identity is verified and on an authorized list is allowed access. Tickets are held by subjects; authorization lists (based on identity verification) are held by an object's guard. Important properties are that tickets must be unforgeable; identities must be unspoofable.

Exercise (Categorizing C-lists and ACLs). (a) Are ACLs ticket-oriented or ID-list-oriented? (b) Same question for C-lists. (c) What category do segment descriptor-based addressing schemes fall into? Is the same true for virtual memory translation tables in general? (d) What category do passwords fall into? Clarify as necessary. (e) What protection category does access control by encryption fall into?

Exercise (Access control and USB drives). When a USB flash drive is inserted into a personal computer, which system accounts or processes are given access to the content on this storage device? When files are copied from the USB drive into the filesystem of an account on the host, what file permissions result on the copied files (assume a Unix-type system)? Discuss possible system choices and their implications.

BASIS FOR AUDIT TRAILS. The basic reference monitor idea of mediating every access provides a natural basis from which to build fine-grained audit trails. Audit logs support not only debugging and accountability, but intruder detection and forensic investigations. Whether or not audit records must be tamper-proof depends on intended use.

Exercise (Access control through encryption and key release). Access control to documents can be implemented through web servers and encryption. Give a technical overview of one such architecture, including how it supports an audit trail indicating times

and subjects who access documents. (Hint: see [7].)

TIME-OF-CHECK TIME-OF-USE RACE CONDITIONS. A race condition which arises in mediating access to objects can sometimes be exploited. Suppose an access control permission check is made at time t_1, and the object is later accessed at time t_2. It is often implicitly assumed that nothing changes between t_1 and t_2, i.e., from time-of-check to time-of-use (TOCTOU). However in multi-processing systems with interrupts, many things can change, e.g., file permissions and owners (meta-data), file data content, the object a filename resolves to, arguments passed to called routines. The chances of changes occurring may be increased by a malicious entity creating favourable circumstances. When an implicit assumption is made at one point in time which is relied on at a later point, and there is a timing-dependent chance that in the interval some change can invalidate the assumption, then a race condition occurs. When this happens in the context of access control, it is called a *TOCTOU-problem*.

One thought is to address this by disabling interrupts, but this has its own problems. Not all interrupts can be safely disabled; not all processes can tolerate interrupts being disabled, or waiting while other processes run uninterrupted; and in multi-processor systems which share memory and other resources, this may require disabling interrupts on all processors. The TOCTOU problem is challenging, arises in many situations, and requires attention by systems designers and developers.[4]

5.3 Object permissions and file-based access control

The *access control indicator* bits in Section 5.1 were a good start, helping us understand basic issues.[5] Early protection based on memory segments gave way to setting permissions on abstract objects, and use of access control lists (Section 5.2). For object-based access control in a subject-object permission framework, after specifying subjects and objects, the task is to identify the types of access operations (modes) for objects and frame these as permissions for consideration.

FILE-BASED ACCESS CONTROL. To understand object-level access permissions, it helps to consider logical files—for both pedagogical and implementation reasons. In this and the next section, we discuss file-based access control as pioneered in Unix systems. Beyond a file's data contents, file systems maintain per-file meta-data specifying access permissions. In Unix, to simplify input-output operations across a multitude of peripheral devices, an early design principle was to treat everything as a file, and design a corresponding file system. For example, if printing a file is done by writing a stream of bytes to an address identified with a printing device, then access permissions to the printer are "file permissions". Thus the study of file permissions generalizes to access control on resources. This explains in part why file permissions are a main focus when access control is taught in computer security.

[4] A TOCTOU example is deferred to Chapter 8.

[5] Filesystem management of access control was already mentioned in 1967 [10], related to early Multics.

The simple permission mechanisms in early systems provided basic functionality. Many common operating systems now support full ACLs (Section 5.2) for system objects including files. ACLs are powerful and offer fine-grained precision—but also have disadvantages. ACLs can be as long as the list of system principals, costing memory and search time; they may require frequent updates; it can be inconvenient to list all principals requiring access to a file. An alternative less expressive than ACLs, the `ugo` architecture, became popular long ago, and remains widely used. We discuss it shortly, but first mention file ownership.

FILE OWNER AND GROUP. In Unix systems, each file has an *owner* and is assigned to a *protection group*. Default values are set on file creation—the owner as the `userid` of the creating process, and the group as the primary group that this userid is assigned to according to the `group` identifier in its entry in */etc/passwd*.

USER-GROUP-OTHERS PERMISSION ARCHITECTURE. The `ugo` architecture can now be explained. It assigns permissions based on three categories of principals: `(user, group, others)`. The `user` category refers to the principal that is file owner. (Why not call it `owner`? It seems `others` won the battle for "o".) The `group` category enables sharing of resources among small to medium-sized sets of users (say, project groups), with relatively simple permissions management.[6] The third category, `others`, is the universal or world group for "everyone else". It defines permissions for all users not addressed by the first two categories—as a means to grant non-null file permissions to users neither being file owner nor in the file's `group`. This provides a compact and efficient way to handle an object for which many (but not all) users should be given the same privileges.

This `ugo` architecture allows fixed-size filesystem meta-data entries, and saves storage and processing time. Whereas ACLs may involve arbitrary-length lists, here permission checking involves bit-operations on sets of just three categories of principals; the downside is a significant loss in expressiveness.

Example *(group permissions).* Suppose a group identifier `accounting` is set up to include `userA`, `userB` and `userC`, and a group `executives` to include `userC`, `userD` and `userE`. Then `userC` will have, e.g., RX access to any file-based resource if the file's `group` is `accounting` or `executives`, and the file's `group` permissions are also RX. Group identifiers and membership may be defined by a single line in the file */etc/group*.

META-DATA AND FILE PERMISSIONS. The above user-group-others mechanism is supported by a per-file filesystem data structure, which also holds other "accounting details" related to a file, such as the address of the file contents. A commonly used such data structure contains the following protection-related fields:[7]

- `user` indicating the userid of the file owner.
- `group` indicating the groupid of the file.

[6] This handling of groups may be contrasted to role-based access control—see RBAC in a later chapter.

[7] To illustrate concepts concretely, details in this section are based on Unix, including its filesystem index-node or `inode`. Other `inode` fields unrelated to permissions indicate file size, last-modified time, number of directory entries that link to the file, and fields to signal if file is a directory or special Unix I/O device file.

- nine (9) RWX protection bits, arranged in 3 groupings for `(user, group, others)`. For regular files, their meaning is relatively obvious. R (read): the file content may be read. W (write): an existing file may be modified (but not removed from the directory—Section 5.5 explains why). X (execute): a binary file may be executed (to execute a shell script requires reading the file first and thus R and X). For a non-executable file, X permission is meaningless (execution will fail).

- three (3) further protection bits: `setuid`, `setgid`, `t-bit` (Sections 5.4 and 5.5).

The superuser process (`root` on Unix, with numeric userid `0`) has access to all files, independent of protection settings. In some circumstances, `root` may need to determine if a particular user process, with a given `userid` and `group-ID`, is authorized to access a file resource, say *filepath*. The system call access() can be used to answer this question.

USE OF PROTECTION BITS. When a user process requests access to a file object, the system checks if the process has the requested access privilege, based on the permissions (privileges) indicated by this data structure. The checks are made in sequence—user, group, others—and the first qualifying category determines privileges. So for a process which seeks R access and is file owner, if the `user` category does not grant R then the request fails even if `others` grants R.

PERMISSION DISPLAY STRINGS. A common visual display format for file permissions is a 10-character string, such as `-rwxr-xr--` with the first character conveying information on file type (e.g., a leading dash indicates a nondirectory file). The next 9 characters, in groups of 3, convey permissions for the `ugo` categories in order. A substring `rwx` corresponds to binary `111` indicating read, write and execute, while a dash "–" conveys a `0`-bit denoting the corresponding permission absent. Information about additional permissions can be overlaid onto these 10 characters (a character can clearly convey more than one bit), e.g., when the `setuid` bit is set as explained below.

PROTECTION BIT INITIAL VALUES (NON-DIRECTORY FILES). On file creation, the 9 RWX bits are set according to a system default post-modified by `umask`, the process' 9-bit user file creation mask. The mask's positional 1-bits specify permissions to remove (if present), allowing a more conservative default. The mask is typically set in a user startup file; a process inherits its parent's mask. For display brevity, each 3-bit string for a RWX group is often written as one octal digit (e.g., `101` for `r-x` is `5`). In this format, for regular files, using a common system default for permissions of `666` (RW for all categories), and common mask default of `022` (removing W from group and others), the combined default permission would be `644` (RW for user, R only for group and others).

Exercise (setting/modifying file permissions). The initial value of a Unix mask can be modified where set in a user startup file, or later by the **umask** command. (a) Experiment to discover default file permissions on your system by creating a new file with the command **touch**, and examining its permissions with `ls -l`. Change your mask setting (restore it afterwards!) using **umask** and create a few more files to see the effect. (b) The command **chmod** allows the file owner to change a file's 9-bit protection mode. Summarize its functionality for a specific flavour of Unix or Linux.

Exercise (modifying file owner and group). Unix commands for modifying file attributes include **chown** (change owner of file) and **chgrp** (change a file's group). Some systems allow file ownership changes only by superuser; others allow a file owner to **chown** their owned files to any other user. Some systems allow a file owner to change the file's group to any group that the file owner belongs to; others allow any group whatsoever. Summarize the functionality and syntax of these commands for some Unix flavour.

Exercise (access control in swapped memory). *Paging* is common in computer systems, with data in main memory temporarily stored out to secondary memory ("swapped to disk"). What protection mechanisms apply to swapped memory? Discuss.

FILE PERMISSIONS AUGMENTED BY ACLS. The `ugo` permission architecture above is often augmented by ACLs (Section 5.2). On access requests, the OS then checks if the associated `userid` is in an ACL entry with appropriate permissions.

Exercise (file ACL commands). For a Unix-type system of your choice (specify the OS version), summarize the design of file ACL protection. In particular, explain what information is provided by the command **getfacl**, and the syntax for the **setfacl** command.

5.4 Effective userid and setuid

SETUID PERMISSION BIT. The `setuid` bit is an extra permission bit on files. A Unix file owner can turn on this bit for any binary executable file owned, say *file1*. Then when a process with execute permission thereon runs *file1*, the OS will temporarily set—while executing *file1*—that process' (effective) `userid` to be that of *file1*'s owner. This allows *file1* to access resources which the calling process might not itself have sufficient privilege to access. The bargain made is that the original process now has more access than it otherwise would, but only under the constraint of a (hopefully very) carefully designed program written by someone else. A major use is for security-critical programs, owned by `root`, which access system resources. This design makes such programs of special interest to attackers, especially when the file owner is `root`. However `setuid` programs owned by regular users are also of use, e.g., to allow others controlled access to that user's files; analogous `setgid` programs (below) allow shared file access within groups.

REAL USERID AND EFFECTIVE USERID. To support the `setuid` bit and related functions, the OS tracks three process-related userid values: an original or real userid (`rUID`), an effective userid (`eUID`), and a saved userid (`sUID`). The latter is used to remember a previous userid value, e.g., if a privileged program wishes to temporarily drop privileges (change `eUID`) and later restore them. The `eUID` value is used to determine privileges on resource access requests, and to set file owner on created files. Supporting system calls getuid() and geteuid() respectively return `rUID` and `eUID`. Privileged functions such as setuid(), seteuid() and setreuid() allow setting `rUID`, `eUID` and/or `sUID`, employing `sUID` as needed (details are beyond our scope). Successful use of the **su** (switch user) command changes `rUID`, `eUID`, `sUID` to the userid of the specified user.

SETGID PERMISSION BIT. For nondirectory files, `setgid` similarly conveys privileges of a file's group, with analogous system values `rGID`, `eGID`, `sGID`. Supporting system

calls getgid(), getegid() and setgid() likewise exist. For directory files, functionality of the `setgid` bit differs as explained in Section 5.5.

Example *(setuid and passwd).* Unix users initiate password changes with the **passwd** command. On many systems, */usr/bin/passwd* is root-owned and setuid.[8] This enables write access to the system password file—but the program checks `rUID`, to enforce that only the password entry of the userid of the calling process can be changed.

INHERITING USERIDS. Process creation in Unix occurs by the system call fork(). This replicates the calling process (*parent*), creating a *child*. Both run the same code. The child inherits the parent userids (`rUID`, `eUID`, `sUID`), but gets a new process-id. If the child process was forked in order to run a separate executable, then it identifies itself as child by seeing a fork() return code of 0, and cedes control by executing that program via one of the exec() system calls. That program continues with the child's process-id; and inherits its userids (`rUID`, `eUID`, `sUID`), except in the case that the executable is `setuid`.

Example *(userids and login).* On Unix systems, when a user logs in, the root-owned setuid **login** program prompts the user for userid-password, does password verification, sets itself to have the userid (`rUID` and `eUID`) and groupid specified in the verified user's password file entry, and after various other intializations gives up control by executing the user's declared shell. The shell inherits the userid and groupid of this parent.

DISPLAYING SETUID AND SETGID BITS. When a file's `setuid` or `setgid` bit is set, the 10-character display string reflects this by changing the `user` or `group` execute character (respectively, position 4 or 7) from `x` to `s`. If execute was not set then the corresponding "–" is changed to `S` (this conveys the status, but is not useful without execute permission). So `-rwsr-xr-x` indicates a file executable by all, with `setuid` bit set; and `-rwSr--r--` indicates a file `setuid`, but not executable (thus not useful; the non-standard capital S signals this).

Exercise (`setuid`). Explain how the `setuid` functionality is of use for user access to a printer daemon, and explain in general, what new risks `setuid` creates.[9]

Exercise (`sudo command`). Look up and explain the design and use of the Unix command **sudo** (superuser do/switch-user do), including its effect on `rUID`, `eUID`, `sUID`.

5.5 Directory permissions, inode example, and linking

To understand permissions and ownership for directory files, and how they relate to those for regular files, an example detailing how the filesystem is implemented helps. We begin with that, and then discuss RWX permissions for directory files in Unix-types systems.

UNIX DIRECTORY STRUCTURE. Consider a Unix-style filesystem modeled as a rooted tree with top named "/". Each leaf node is a regular file, and each interior node a directory file (dirfile). Both file types are implemented using an inode for meta-data (described earlier) plus a datablock for content (recall that a Unix file is an object represented

[8] On other systems this executable itself is not setuid; e.g., Mac OS X uses an alternate means.

[9] We will return to consider this question later.

by an inode). Meta-data relevant to our discussion is permissions data, a flag distinguishing dirfiles, and a pointer (datalink) to the datablock. The datablock for a dirfile contains filesystem data for the entries (hierarchical children) of that directory node, structured as a list of entries `dir-entry = (d-name, d-inode)`. A string value populating `d-name` names a (regular or directory) file; a `d-inode` value identifies that file's `inode`. The first two dir-entries are always for the directory node itself with string name ".", and for its parent (parent-dir) which is given the string name ".." by convention.

‡**DETAILED INODE EXAMPLE.** The following, and Figure 5.5, explain the internal actions that occur in creating a new directory node, and provide a detailed example of the filesystem (directory) structure summarized above. This may help in understanding how directory-node RWX permissions (below) work. Suppose that in an existing directory */NBAwest*, we wish to create a new sub-directory */NBAwest/Warriors*. Peer sub-directories *Lakers* and *Spurs* already exist. While in current directory */NBAwest*, a user types the shell command: `mkdir Warriors`. The resulting operating system actions begin with a system call to mknod() with parameters indicating filepath */NBAwest/Warriors* and that this should be a new directory. The following then occurs.

1. Creation of a new `inode` instance for *Warriors*, flagged as a directory, initially with null datalink. Let `pathlink1` be a pointer to this new inode.

2. Creation of a new datablock with two directory entries, `(".", NULL)` and `("..", NULL)`. The null datalink of the inode at `pathlink1` is set to point to this datablock.

3. So that the new directory node for *Warriors* can be reached from its parent-dir, the entry `("Warriors", pathlink1)` is added to the list of dir-entries in the datablock of that parent-dir. This datablock will already have entries for `"."` and `".."`, plus peers `"Lakers"` and `"Spurs"` as noted above. The system can obtain a pointer (call it `pathlink2`, for use also below) to the parent-dir inode from available parameters.

4. The system makes two system calls to link() to fix the two null dir-entry pointers above. The first results in the datablock pointer for `"."` in the *Warriors* datablock dir-entry being set to `pathlink1`, i.e., pointing to its own inode; the second results in the datablock pointer for `".."` in the *Warriors* datablock dir-entry being set to `pathlink2`, i.e., pointing to its parent inode, the inode for *NBAwest*. So the datablock entries are now `(".", pathlink1)` and `("..", pathlink2)`.

The user now creates (regular) files named *curry*, *durant*, *thompson*, etc., for storing player statistics. As a side detail, note that the string `"Warriors"` appears nowhere in the two data structures representing the logical file *Warriors*, and only in the parent's dir-entry for this file. That is how the file is referenced: as a child under its parent.

Well, wasn't reading through all that a lot fun! If you found this hard to follow, read it again a second time—but start with a blank sheet of paper in front of you, and draw out what happens at each step. (Who knew computer security could be so enjoyable?)

DIRECTORY PERMISSIONS. The explanation of the 9 RWX permission bits and three others was for regular (non-directory) files. For Unix directory files, these have very

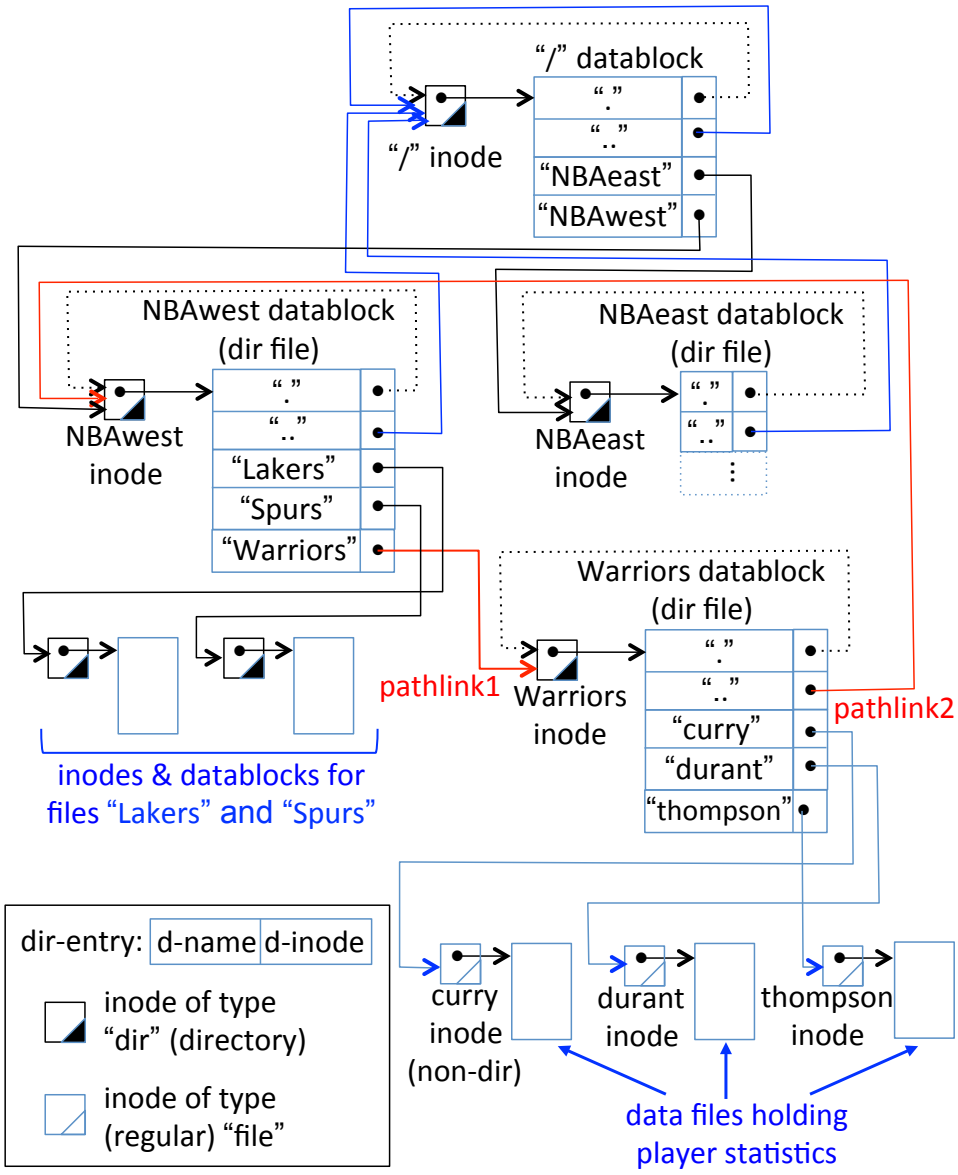

Figure 5.5: Directory structure and example (Unix filesystem with inode structure). See inline discussion of example for notes on `pathlink1`, `pathlnk2`. Compare to Figure 5.6.

different meanings, better reflected by calling them VAT bits (view-list, alter, traverse). Here is a summary for a user with the specified permissions on a directory (e.g., having R permission for the first category `user`, `group` or `other` they belong to).

- R: the user may view (V) the directory content, i.e., list of filenames that are `d-name` fields of `dir-entry` items (above). R alone on a directory gives no access to the contents of files therein (they have their own permissions), nor path-access to their properties (see X below). Access to a file's properties (inode meta-data including permissions), to its content, and to its name (as a directory entry) are all distinct. Access to a file's properties—whether a directory or non-directory (regular) file—requires only path-access (X) to that inode; R, W or X permission on that file itself is not required and pertains to file content, not meta-data.

  Example *(Permissions).* Suppose *dir1* has entries *dir2* and *file2*. Path-access to *dir1* (X permission on a directory holding *dir1*) is required to access properties of *dir1* itself. R on *dir1* allows "visibility" of (the names) *dir2* and *file2*. X on *dir1* allows seeing the properties of *dir2* and *file2* (e.g., via **ls**, below), but to read their content requires R on the individual files.

- W: the user may alter (A) directory content, i.e., filenames within it—renaming or deleting existing entries, or creating new ones thus allowing file creation—provided X permission is also held.[10] The system will modify, remove, or newly add a corresponding `dir-entry`. Thus permission to delete a file is controlled not by the file's permissions, but by those of the directory the file is in. While "deleting a file" removes the filename from the directory's list of filenames, the file itself, i.e., its inode and datablock, is removed from the filesystem only if this was the last directory entry referencing the file's inode. A file's permissions control privileges to its content—and in the case of a directory, file content is its list of filenames.

- X: the user may *traverse* (T) the directory. This includes executing commands that explore or search its properties, e.g., **find**; entering the directory or setting it as the working/default directory for filesystem references; and path-access to the files named in the directory list (traversing this filesystem node to get to those files). Lack of X permission on the directory denies path-access to the directory's files independent of permissions on the individual files—but their filenames (as directory content) remain visible if R permission is held. By Unix *path-based permissions*, to read a file's content requires both X permission on all directories in the path to the file and R on the file itself. Having X but not R on a directory allows path-access to a file therein if the filename is already known.

- `setuid`: this typically has no meaning for directories in Unix and Linux.

- `setgid`: the `group` value initially assigned to newly created (non-directory and directory) files therein is set to the group of the directory itself (rather than the

[10] A new filename could result from creating a new file in the ordinary sense, or from a link (below).

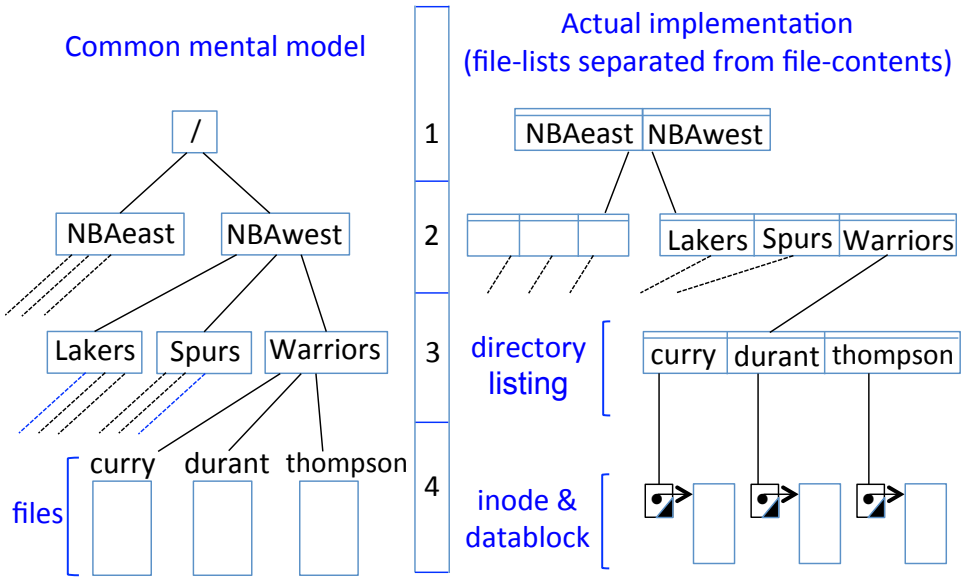

Figure 5.6: Common mental model of Unix directory structure. As typically implemented, a file's inode and datablock themselves contain no filename. Instead, they are data structures referenced by a directory entry, which specifies the filename; this also allows the same structures to be referenced by different names from distinct (multiple) directory entries. The poor fit of mental model to implementation may lead to misunderstanding directory permissions. See also Figure 5.5.

primary group of the creating user); and a newly created sub-directory in addition inherits the directory's `setuid` bit. The intent is easier sharing of files in groups.

- `t-bit` (*text* or *sticky* bit): a set `t-bit` on a directory prevents a user from deleting or renaming any file therein owned by another user. The directory owner and `root` retain their usual privileges. A main use is for directories like */tmp* which are world-writable (below), where an attacker could otherwise remove and replace a file with a malicious one of his choosing and the same filename. When set, a `t` replaces `x` in position 10 of 10-character permission strings. For non-directory files, this bit had a different original purpose, but is little-used today.

- protection bit settings do not impede the supervisor from accessing a directory.

Figure 5.6 may help some readers better understand these directory permission semantics.

PROTECTION BIT INITIAL VALUES (DIRECTORY FILES). As for regular files (above), when a new Unix directory is created, its 9 RWX bits are assigned. A common system default for directories is `777`, and for the common default mask `022`, the combined default permission for a directory is `755` (RWX for user, RX for group and others).

DIRECTORY LISTINGS. Typing `ls -l` to a Unix shell lists the contents of a directory (`-l` for long format, including permissions); use `ls -ld` for meta-data on the directory node itself. As before, the output is a sequence of lines each beginning with a 10-character sequence such as `drwxrwxr-x`, each line giving information about a file listed. A leading `d` signals a directory file. The next 9 characters, in groups of 3, reflect permissions for the categories `user, group, others`; dash indicates no privilege. The string `drwxrwsr-x` indicates a `setgid` bit set for a directory file (recall Section 5.3).

Exercise (viewing directory properties). On a Unix-like system, explore the properties and protection bits of various directories using the **ls** (list) command and various options like `ls -l`. (Note: on a Mac system, to get a command interpreter shell to the underlying Unix system, run the Terminal utility, often found in *Applications/Utilities*.)

SYMBOLIC LINKS AND HARD LINKS. Since security issues can arise related to symbolic links, security practitioners should understand them. In Unix, the same nondirectory file can appear in multiple directories, optionally with different names. This is done by *linking* using the **ln** command. A link can be either a *symlink* (symbolic/soft link or *indirect alias*) or a *hard link* (*direct alias*).

Example *(Hard link).* Consider a file with pathname *existing*. Then the command `ln existing new1` results in a `dir-entry` specifying the string *new1* as the name of a file object whose directory entry references the inode for *existing*. Since a file's name is not part of the file itself, distinct directory entries (here, now two) may name the file differently. Also, this same command syntax may allow a directory-file to be hardlinked (i.e., *existing* may be a directory), although for technical reasons, hardlinking a directory is usually discouraged or disallowed.

Example *(Symlink).* For a symlink the `-s` option is used: `ln -s existing new2` results in a `dir-entry` for an item assigned the name *new2* but in this case it references a new inode, of file type `symlink`, whose datablock provides a symbolic name representing the object *existing*, e.g., its ASCII pathname string. When *new2* is referenced, the filesystem uses this symbolic name to find the inode for *existing*. If the file is no longer reachable by pathname *existing* (e.g., its path or that directory entry itself is changed due to renaming; or removed because the file is moved; or deleted from that directory), the symbolic link will thus fail while a hardlink still works.

DELETING LINKS. Deleting a `symlink` file (e.g., *new2*) does not delete the file it retrieves (e.g., *existing*). Deleting a hardlinked file by specifying a particular `dir-entry` eliminates that directory entry, but the file is deleted only when its link-count (the number of hardlinks referencing it) drops to zero. Confused? Recall the design. An inode itself does not "live" in any directory but rather exists independently; directory entries simply organize inodes into a structure. Figure 5.7 and Table 5.8 summarize how links work.

WORLD-WRITABLE FILES. Some files are writable by all users (world-writable). In a 10-character permission string like `-rwxr-xrwx`, the `w` in the second-last character indicates world-writable; the leading dash means it is a non-directory (regular) file. Many executables, including editors, mail, shells, and login programs, invoke startup files. You would not want, e.g., startup files to be world-writable, as then any process could modify them, and become a recurring visitor. A root-owned setuid world-writable executable is a

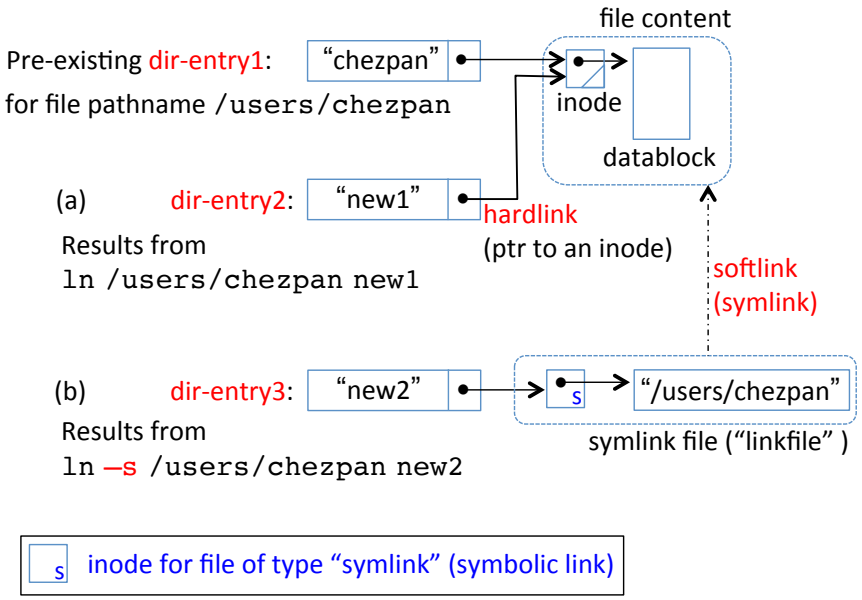

Figure 5.7: Comparison of hardlink (a) and symbolic link (b).

	What happens to ...	
... if	hard-link file	symbolic-link file
target file deleted (e.g., dir-entry1 of Fig. 5.7 removed)	dir-entry1 removed, but file content does not disappear since its linkcount is above 0	softlink will fail (dir-entry3 of symlink remains but is stale, i.e., can't be resolved)
target file renamed or moved to a new pathname	hardlink remains intact, as inode has not changed (nor moved); only the dir-entry1 changes	softlink will fail as target pathname can't be resolved
rm of symbolic-link file (e.g., new2 of Fig. 5.7)	—	linkfile inode and its dir-entry3 disappear, but target file is unchanged

Figure 5.8: Results of deleting, renaming/moving, and removing linked files. "Deleting" a file removes its dir-entry, but does not always result in the file content objects being deleted (see table). As an example target file, consider `/Users/chezpan` in Fig. 5.7.

particularly bad idea—any process could replace the contents with a malicious file, which would run with root privileges regardless of who invoked it.

Exercise (finding world-writable files). The Unix **find** command can search directories for files with specific properties, including permissions. Explore your own system for world-writable files using the command: `find` */users/yourhome* `-perm -2 -print` (replace */users/yourhome* by the directory you wish to explore; start at a directory with a relatively small subtree, as the output will generally be extensive). The search will be denied access to directories you do not have X permission for; the command will continue with the next directory in its recursive tree search.

Exercise (access control outside of filesystem). Suppose a copy of a filesystem's data (cf. detailed example above) is backed up on secondary storage or copied to a new machine. You build customized software tools to explore this data. Are your tools constrained by the permission bits in the relevant inode structures? Explain.

5.6 ‡Protection rings: isolation meets finer-grained sharing

Practical requirements call for an efficient middleground between the security of full isolation (complete containment, no sharing) and the convenience of shared objects. Protection rings generalize the two-mode hardware model (supervisor, user mode) to multiple privilege levels and domains.

PROTECTION RINGS. Section 5.1 introduced isolation, and selective (basic) sharing across processes. A third desirable feature is layered protection within processes—affording user-space processes separation analogous to supervisor-user separation, and sharing of *protected subsystems*. This can be provided by *protection rings*, a major innovation in early operating systems; hardware support for eight rings was designed into early-1970s Multics. Rings overlayed additional access control on Multics segmented memory, and generalized privilege classes from 2 (supervisor, user) to n modes. Rings selectively allow complete isolation of processes, controlled sharing between programs (e.g., for reuse of common code and data), and layered protection for varying degrees of separation. The idea is that segments in stronger rings are protected from access by weaker rings; these conditions are then relaxed to provide greater flexibility, when authorized.

Consider a set of rings 0 to $n-1$ as a nested, ordered set of levels, as in Figure 5.9(a). Ring 0 is central and most privileged, with privilege decreasing moving outward—to aid memory, think "the core is strong". Lower numbered rings having higher privilege confuses some; we thus may say *stronger* (*inner*) for higher-privilege and *weaker* (*outer*) for lower-privilege. We add to every segment descriptor (Section 5.1) a new `ring-num` field for ring number, and to the CPU program counter (instruction address register) a `PCring-num` indicating the ring number of the executing process. See Figure 5.9(b).

For access control functionality, we associate to each segment an *access bracket* (n_1, n_2) denoting a range of consecutive rings, used as explained below.

PROCEDURE SEGMENT ACCESS BRACKETS. A user process may desire to transfer control to stronger rings (e.g., for privileged functions—such as input-output functions,

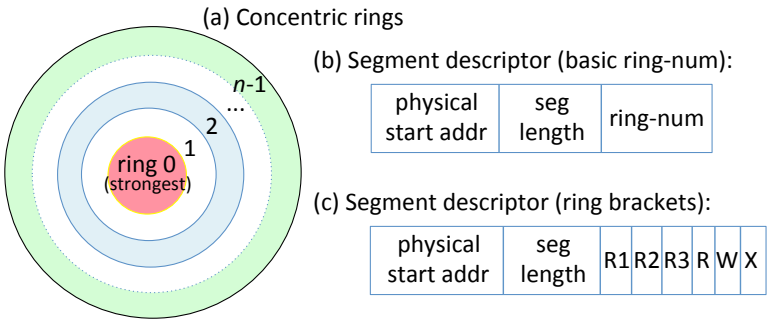

Figure 5.9: Protection rings and supporting descriptors. (a) Protection rings. (b) Descriptor including basic ring-number. (c) Descriptor with support for ring brackets.

or to change permissions to a segment's access control list in supervisor memory); or to weaker rings (e.g., to call simple shared services). A user process P_1 may wish to allow another user's process P_2, operating in a weaker ring, access to data memory in P_1's ring, but only under condition that such access is through a program (segment) provided by P_1, and verified to have been accessed at a pre-authorized *entry point*, specified by a memory address. Rings allow this, but now transfers that change rings (*cross-ring transfers*) will require extra checks. In the simple "within-bracket" case, a calling process P_1 executing in ring i requests a transfer to procedure segment P_2 with *access bracket* (n_1, n_2) and $n_1 \leq i \leq n_2$. Transfer is allowed, without any change of control ring (`PCring-num` remains i).[11] The harder "out-of-bracket" case is when i is outside P_2's access bracket. Such transfer requests trigger a fault; control goes to the supervisor to sort out, as now discussed.

PROCEDURE SEGMENT GATE EXTENSION. Suppose a process in ring $i > n_2$ attempts transfer to a stronger ring bracketed (n_1, n_2). Processes are not generally permitted to call stronger-ring programs, but to allow flexibility, the design includes a parameter, n_3, designating a *gate extension* for a triple (n_1, n_2, n_3). For case $n_2 < i \leq n_3$, a transfer request is now allowed, but only to pre-specified entry points. A list (*gate list*) of such entry points is specified by any procedure segment to be reachable by this means. So $i > n_2$ triggers a fault and a software fault handler handles case $n_2 < i \leq n_3$. The imagery is that gates are needed to cross rings, mediated by *gatekeeper* software as summarized next.

RING GATEKEEPER MEDIATION. When a ring-i process P_1 requests transfer to procedure segment P_2 having ring bracket (n_1, n_2, n_3), the following mediation occurs.

- $n_1 \leq i \leq n_2$ (within access bracket): allow transfer†
- $i < n_1$ or $n_2 < i$: triggers fault to gatekeeper for resolution:
  - Case $i < n_1$ (calling weaker ring): allow transfer†

[11] It would be unclear which value in the access bracket (n_1, n_2) to change the `PCring-num` to, as the bracket declares the program suitable in the full range. This suggests that many programs will have a single-ring access bracket, namely the ring best suiting the program.

- Case $n_2 < i \leq n_3$ (within gate extension): allow if transfer address on gatelist
- Case $i > n_3$ (above gate extension): error, unauthorized transfer

†On transfer to a weaker ring P_2, the gatekeeper should check all arguments passed will be accessible (P_2 might not have access to higher-privilege memory). Possibilities include copying arguments into accessible memory.

CROSS-RING RETURNS. Cross-ring returns (e.g., P_2 returning to P_1) likewise trigger mediation, and may involve *return gates*, which we do not discuss further. The gatekeeper enforces that returns match stack expectations, using details stored on the standard call stack such as segment descriptors of return segments (including previous `ring-num`).

RING NUMBER AFTER TRANSFER FROM OUTSIDE-BRACKET. After an outside-bracket transfer into bracket (n_1, n_2), what value x should be assigned to `PCring-num`? The easy case is $n_2 < i \leq n_3$: least-privilege suggests $x = n_2$, temporarily increasing privileges by the least necessary. For $i < n_1$, privileges should be reduced, with strict least-privilege dictating $x = n_2$, while the "fewest-hops" choice $x = n_1$ may be slightly more compelling for overall simplicity, given that an "in-bracket" transfer from ring n_2 would leave the ring of execution at n_2. Thus "fewest-hops" provides a reasonable choice for x in both cases. Another choice would be for the segment, including possibly its gate entry, to specify a new ring of execution.

Exercise (Address arguments passed to stronger segment). Suppose weaker program segment P_w calls stronger program segment P_s, passing an argument involving a memory address A which P_s is sufficiently privileged to access, but which P_w is not. P_w does not itself try to access A (so there is no access fault). Is it possible that P_s could, as a result, disrupt the integrity of its own data segment, or damage other segments in its ring? Are additional gatekeeper actions, therefore, necessary for inward calls? If so, explain.

EXECUTE, READ AND WRITE BRACKETS. We now introduce *read access brackets* and *write access brackets*, with semantics as explained below. Our discussion of cross-ring execution privileges resulted in defining access bracket and gate extension parameters (n_1, n_2, n_3). These values would typically be stored in hardware registers R_1, R_2, R_3 respectively, populated from corresponding values in Multics segment descriptors, with *execute bracket* (R_1, R_2) and gate extension $(R_2 + 1, R_3)$. To define corresponding read and write access brackets respectively delimited by integer pairs (d_1, d_2) and (w_1, w_2), it would be convenient to re-use the same registers. Consider the choices: read bracket $(0, R_2)$, write bracket $(0, R_1)$, with $R_1 \leq R_2 \leq R_3$. See Figure 5.10.

The read and write lower bounds stem from reasoning that ring 0 processes should have access if any lower privileged rings do. Equating the write bracket's top and execute bracket's bottom allows a single ring in which a segment can be both written and executed (arguably, ruling out some possible errors while leaving flexibility). Setting equal the tops of the read and execute brackets appears reasonable (to deny R access, turn the R bit off).

This provides a complete ruleset for R, W and X access to a segment. Note that access requires both (1) the requesting process' ring be within the associated bracket, and (2) the segment descriptor's relevent RWX flag be on. See Figure 5.9(c).

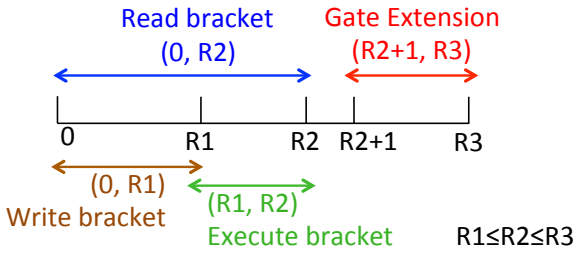

Figure 5.10: Illustrative read, write and execute brackets.

ALTERNATE BRACKET RULESET. For a data segment D, suppose parameters (n_1, n_2) are used to define an alternate ruleset as follows, for a process running in ring-i:

- write bracket $(0, n_1)$: can write into D if $i \leq n_1$ (and D's descriptor allows W)
- read bracket $(n_1 + 1, n_2)$: can read D if $n_1 < i \leq n_2$ (and D's descriptor allows R)

For example then, for $(n_1, n_2) = (2, 3)$, a ring-i process has access as follows: for $i = 1, 2$, can only write to D; for $i = 3$, can only read from D; and for $i = 4$, can neither read nor write D. By this ruleset, in no case can the ring-i process both read and write into D; to do both, a process would have to alternate between rings.

Exercise (Ruleset discussion). Is the alternate bracket ruleset useful, i.e., does it offer advantages over other possible rulesets? (Note that a ruleset has implications on subsystem design, e.g., in which rings to locate program functionality. Different permissions can be specified in different segment descriptors, but a single ruleset is fixed for all processes.)

5.7 ‡Protection domains, subjects and processes

Protections rings are an example of protection *domains*, a term we seek now to define more precisely, often associated with the terms protection *contexts* and protection *environments*. We also refine our definition of *subject*, mindful that we used this term as row index of the access matrix in the subject-object permission model (Section 5.2).

Access control is about controlling access to objects. A *subject* requests access to objects, and thus is also an entity whose access to those objects must be controlled. A subject involves a process, but that is only part of the story—more precisely, we define a subject as $S = (\mathcal{P}, \mathcal{D})$. Here $\mathcal{P}$ denotes a process and $\mathcal{D}$ a domain as explained next.

The term *domain* was introduced (Section 5.1) as the permission set associated with the objects a process can access. Over time, the domain of a process can change—when a Unix process calls a root-owned setuid root process it retains its process identifier, but temporarily runs with a different `eUID` yielding different access privileges. Viewing the domain as a room, the objects in the room are accessible to the subject; when the subject changes rooms, the accessible objects typically change. To define *protection domain* more

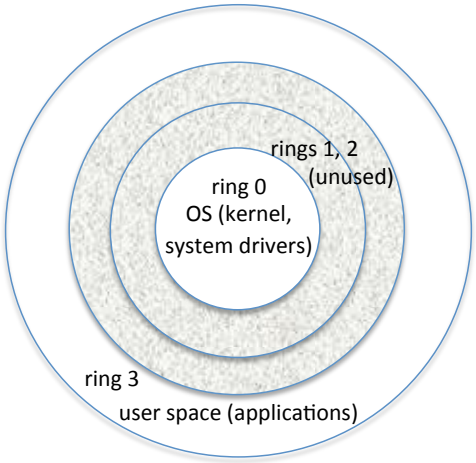

Figure 5.11: Underuse of hardware support for protection rings.

precisely, consider as a specific example the ring system and segmented virtual addressing of Multics. For our subject $S = (\mathcal{P}, \mathcal{D})$, its domain is defined by $\mathcal{D} = (r, T)$. Here r is the execution ring of the segment g that $\mathcal{P}$ is running in, and T is $\mathcal{P}$'s descriptor segment table containing segment descriptors (including for g) each including access indicators.

The subject associated with process $\mathcal{P}$ now becomes $S = (\mathcal{P}, r, T)$, or more concretely, `(processID, ring-number, descriptor-seg-ptr)`. Some observations follow.

1. A change of execution ring changes the domain, and thus the privileges associated with a process as well as the subject S. At any specific execution point, a process operates in one domain or context; privileges change with context (mode or ring).

2. A transfer of control to a different segment, but within the same ring (same process, same descriptor segment), changes neither the domain nor subject.

3. Access bracket entry points specify allowed, gatekeeper-enforced domain changes.

4. Virtual address translation maps constrain physical memory accessible to a domain.

5. A system with n protection rings defines a strictly ordered set of n protection domains, $\mathcal{D}^{(i)} = (i, T)$, $0 \leq i \leq n-1$. Associating these with a process $\mathcal{P}$ and its fixed descriptor segment defined by T, defines a set of subjects $(\mathcal{P}, 0, T), ..., (\mathcal{P}, n-1, T)$.

6. Informally, C-lists define the environment of a process. If we equate C-lists with domains, then they can be substituted in the definition $S = (\mathcal{P}, \mathcal{D})$.

Exercise (Ring changes vs. switching userid). It is recommended that separate accounts be set up on personal computers for regular user activities and administration activities. If a user needs to carry out a task requiring superuser privileges, they then log into

the administrative account, or switch to that account. Discuss how this compares, from an operating system viewpoint, to a process changing domains by changing rings.

CPU MODES SUPPORTED BY HARDWARE, UNUSED BY SOFTWARE. Many computers in use run operating systems supporting only two CPU modes (supervisor, user) despite hardware support for more. For example, widely deployed Intel x86 hardware supports four modes (rings), but Windows systems running thereon typically use only rings 0 and 3. See Figure 5.11. The following observation explains this. If a major operating system vendor seeks maximum market share through deployment on all plausible hardware platforms, the lowest-functionality hardware (here, in terms of CPU modes) constrains all others. The choices are to abandon deployment on the low-functioning hardware (giving up market share), incur major costs of redesign and support for multiple software streams across hardware platforms, or reduce software functionality across all platforms. Operating systems custom-built for dedicated hardware can offer richer features, but a consequent disadvantage is fewer hardware platforms suitable for deployment.

Exercise (Plaforms supporting more than two modes). Discuss, with technical details, how many CPU modes, privilege levels, or rings are supported by the ARMv7 processor architecture. Likewise for operating systems OpenVMS and OS/2. Do any versions of Windows support more than two modes?

SEGMENT DESCRIPTORS FROM ACCESS CONTROL ENTRIES. To complete the Multics story of access control through segment descriptors, consider how these themselves originate. When a user logs in, a new process P is created for the user activity. The virtual memory that P can see–its universe of addressable memory—is limited to memory reachable by the segment descriptors in P's descriptor segment. The supervisor creates P's virtual memory space by populating this descriptor segment. Recall the descriptor base register/DBR points to the base of P's descriptor segment table. How does the supervisor select which segments to give P descriptors for? Segments (as objects) have corresponding access control list entries in the system, specifying which subjects may access them. From such entries, the supervisor constructs P's descriptor segment.

Exercise (Supervisor creation of descriptor segment). A user logs in to their account. The supervisor creates a new process P for the user activity, and sets out to create the descriptor segment for P. Abstractly, there is an access control matrix with subjects as rows, and objects (including segments) as columns. Discuss which implementation would more efficiently support the supervisor task of creating P's descriptor segment—a matrix stored by row in the form of capabilities, or by column in the form of ACLs. Also, where does the supervisor find information from which to appropriately populate the access control indicators necessary in segment descriptors?

5.8 ‡Endnotes and further reading

REFERENCES AND FURTHER READING. Graham [10] gives an early, authoritative view of protection rings (including early identification of race condition issues); see also Graham-Denning [9] (which Section 5.7 follows), and Schroeder and Saltzer [21] for a Multics-specific discussion of hardware-supported rings and related software issues. Lampson's 1971 conference paper [14] unified early access control mechanisms under the access matrix model. Section 5.1 draws from Graham [10] and Saltzer and Schroeder [20] (see also for capabilities). Saltzer [18] gives an insightful Multics survey including security features omitted herein. Dennis [5] is credited for segmented addressing.

Jaeger [12] is recommended for operating system security, SELinux, security kernels, and Multics including rings directly supporting multi-level security (MLS) policies. Curry [4] addresses Unix security. Ritchie and Thompson [17] give an early overview of Unix. Bishop and Dilger [2] discuss TOCTOU problems. Recommended books on operating systems include Saltzer and Kaashoek [19] (for security design principles), Silberschatz et al. [22] (chapters 14-15), and Tanenbaum [23] for conciseness and clarity. Gruenbacher [11] summarizes ACL support in Unix-like systems. Dittmer and Tripunitara [6], and earlier Chen et al. [3], explore inconsistent implementations of setuid() and related system calls. Loscocco et al. [16] argue for renewed interest in secure operating systems, as a necessary support for computer security in general.

The 1970 Ware report [25] explored security controls in resource-sharing computer systems. The 1972 Anderson report [1, pp.8-14] lays out the central ideas of the reference monitor concept, access matrix, and security kernel; it expressed early concerns about requiring trust in the entire computer manufacturing supply chain, and the ability to determine that *"compiler and linkage editors are either certified free from 'trap-doors', or that their output can be checked and verified independently"*—attacks later more fully explained in Thompson's Turing-award paper [24] detailing C-code for a trojan-horse compiler. The 1976 RISOS report [15, p.57] defined a security kernel as *"that portion of an operating system whose operation must be correct in order to ensure the security of the operating system"* (cf. Chapter 1, principle of SMALL-TRUSTED-BASES P8). Their small-size requirement, orginally specified as part of the validation mechanism for the reference monitor, has made *microkernels* a focus for security kernels (cf. Jaeger above). These 1970-era reports indicate long-standing awareness of computer security challenges.

Lampson's 1973 note on the confinement problem [13] raises the subject of untrusted programs leaking data over covert channels. Gasser's book [8] gives an early integrated treatment of topics including the reference monitor and security kernels, segmented virtual memory, MLS/mandatory access control, and covert channels.

References

[1] J. P. Anderson. Computer Security Technology Planning Study (Vol. I and II, "Anderson report"), Oct 1972. James P. Anderson and Co., Box 42, Fort Washington, PA, 19034 USA.

[2] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, 1996.

[3] H. Chen, D. Wagner, and D. Dean. Setuid demystified. In *USENIX Security Symp.*, 2002.

[4] D. A. Curry. *UNIX System Security: A Guide for Users and System Administrators*. Addison-Wesley, 1992.

[5] J. B. Dennis. Segmentation and the design of multiprogrammed computer systems. *Journal of the ACM*, 12(4):589–602, 1965.

[6] M. S. Dittmer and M. V. Tripunitara. The UNIX process identity crisis: A standards-driven approach to setuid. In *ACM Conf. Computer and Comm. Security (CCS)*, pages 1391–1402, 2014.

[7] W. Ford and M. J. Wiener. A key distribution method for object-based protection. In *ACM Conf. Computer and Comm. Security (CCS)*, pages 193–197, 1994.

[8] M. Gasser. *Building a Secure Computer System*. Van Nostrand Reinhold, 1988. Free online, `http://cs2.ist.unomaha.edu/~stanw/gasserbook.pdf`.

[9] G. S. Graham and P. J. Denning. Protection—principles and practice. In *AFIPS Spring Joint Computer Conf.*, pages 417–429, May 1972.

[10] R. M. Graham. Protection in an information processing utility. *Commun. ACM*, 11(5):365–369, 1968. Appeared as the first paper, pp.1-5, first ACM Symposium on Operating System Principles, 1967.

[11] A. Gruenbacher. POSIX access control lists on LINUX. In *USENIX Annual Technical Conf.*, pages 259–272, 2003.

[12] T. Jaeger. *Operating System Security*. Morgan and Claypool Publishers, 2008.

[13] B. W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973.

[14] B. W. Lampson. Protection. *ACM Operating Systems Review*, 8(1):18–24, 1974. Originally published 1971 in Proc. 5th Princeton Conf. on Information Sciences and Systems.

[15] T. Linden. Security Analysis and Enhancements of Computer Operating Systems ("RISOS report"), Apr 1976. NBSIR 76-1041, The RISOS Project, Lawrence Livermore Laboratory, Livermore, CA.

[16] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *National Info. Systems Security Conf. (NISSC)*, pages 303–314, 1998.

[17] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Commun. ACM*, 17(7):365–375, 1974.

[18] J. H. Saltzer. Protection and the control of information sharing in Multics. *Commun. ACM*, 17(7):388–402, 1974.

[19] J. H. Saltzer and M. F. Kaashoek. *Principles of Computer System Design*. Morgan Kaufmann, 2010.

[20] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[21] M. D. Schroeder and J. H. Saltzer. A hardware architecture for implementing protection rings. *Commun. ACM*, 15(3):157–170, 1972. Earlier version in ACM SOSP 1971, pp.42–54.

[22] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts (seventh edition)*. John Wiley and Sons, 2005.

[23] A. S. Tanenbaum. *Modern Operating Systems (third edition)*. Pearson Prentice Hall, 2008.

[24] K. Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8):761–763, 1984.

[25] Willis H. Ware (Chair). Security controls for computer systems: Report of Defense Science Board Task Force on Computer Security. RAND Report R-609-1 ("Ware report"), 11 Feb 1970. Office of Director of Defense Research and Engineering, Wash., D.C. Confidential; declassified 10 Oct 1975.