Chapter 14: Access Control Mechanisms

- Mechanism: a method, tool, or procedure for enforcing *a security policy*
  - Access control lists
  - Capabilities
  - Locks and keys
  - Ring-based access control
  - Propagated access control lists

Access Control Lists

- Columns of access control matrix

	file1	*file2*	*file3*
Andy	rx	r	rwo
Betty	rwxo	r	
Charlie	rx	rwo	w

ACLs:

- file1: { (Andy, rx) (Betty, rwxo) (Charlie, rx) }
- file2: { (Andy, r) (Betty, r) (Charlie, rwo) }
- file3: { (Andy, rwo) (Charlie, w) }

ACL Design Considerations

- Representation of ACLs, groups, wildcards
- Modifications of ACLs
- Privileged user

Access Control Entries and Lists

- An Access Control List (ACL) is a sorted list of zero or more Access Control Entries (ACEs)
- An ACE refers specifies that a certain set of accesses (e.g., read, execute and write) to the resources is allowed or denied for a user or group
- Examples of ACEs for folder "Bob's ITIS 6200 Grades"
  - Bob; Read; Allow
  - TAs; Read; Allow
  - TWD; Read, Write; Allow
  - Bob; Write; Deny
  - TAs; Write; Allow

Acknowledgement: Matt Bishop

Linux vs. Windows

- Linux
  - Allow-only ACEs
  - Access to file depends on ACL of file and of all its ancestor folders
  - Start at root of file system
  - Traverse path of folders
  - Each folder must have execute (cd) permission
  - Different paths to same file not equivalent
  - File's ACL must allow requested access
- Windows
  - Allow and deny ACEs
  - By default, deny ACEs precede allow ones
  - Access to file depends only on file's ACL (explicit + inherited)
  - ACLs of ancestors ignored when access is requested
  - Permissions set on a folder usually propagated to descendants (inheritance)
  - System keeps track of inherited ACE's

Linux File Access Control

- File Access Control for:
  - Files
  - Directories
  - Therefore…
    - \dev\ : *devices*
    - \mnt\ : *mounted file systems*
    - What else? *Sockets*, *pipes*, *symbolic links*…

Linux File System

- Tree of directories (folders)
- Each directory has links to zero or more files or directories
- Hard link
  - From a directory to a file
  - The same file can have hard links from multiple directories, each with its own filename, but all sharing owner, group, and permissions
  - File deleted when no more hard links to it
- Symbolic link (symlink)
  - From a directory to a target file or directory
  - Stores path to target, which is traversed for each access
  - The same file or directory can have multiple symlinks to it
  - Removal of symlink does not affect target
  - Removal of target invalidates (but not removes) symlinks to it
  - Analogue of Windows shortcut or Mac OS alias

Acknowledgement: Matt Bishop

Unix Permissions

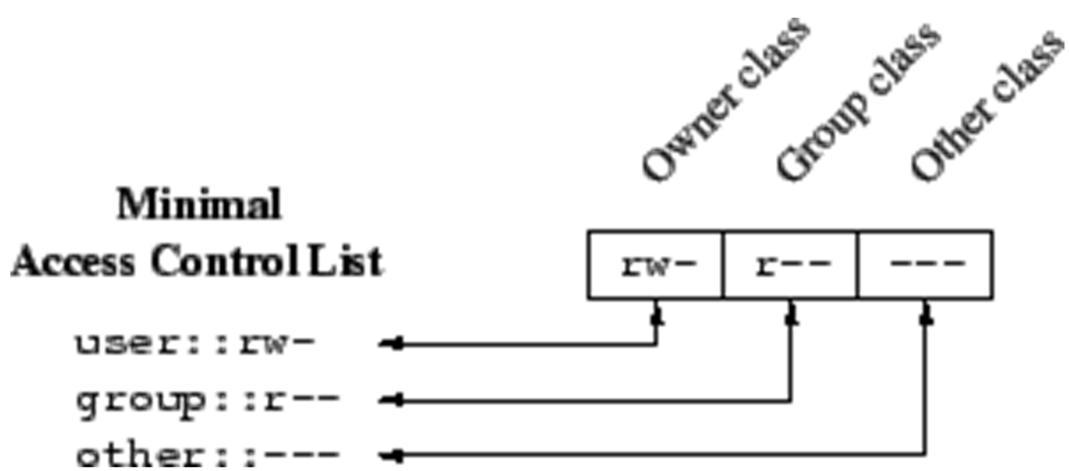
- Standard for all UNIXes
- Every file is owned by a user and has an associated group
- Permissions often displayed in compact 10-character notation
- To see permissions, use `ls -l`

```
jk@sphere:~/test$ ls -l
total 0
-rw-r-----  1 jk ugrad 0 2005-10-13 07:18 file1
-rwxrwxrwx  1 jk ugrad 0 2005-10-13 07:18 file2
```

Acknowledgement: Matt Bishop

Minimal ACLs

- In a file with minimal ACLs, *name* does not appear, and the ACLs with *type* “user” and “group” correspond to Unix user and group permissions, respectively.
  – When name is omitted from a “user” type ACL entry, it applies to the file owner.



Acknowledgement: Matt Bishop

Permissions Examples (Regular Files)

-rw-r--r--	read/write for owner, read-only for everyone else
-rw-r-----	read/write for owner, read-only for group, forbidden to others
-rwx------	read/write/execute for owner, forbidden to everyone else
-r--r--r--	read-only to everyone, including owner
-rwxrwxrwx	read/write/execute to everyone

Acknowledgement: Matt Bishop

Permissions for Directories

- Permissions bits interpreted differently for directories
- *Read* bit allows listing names of files in directory, but not their properties like size and permissions
- *Write* bit allows creating and deleting files within the directory
- *Execute* bit allows entering the directory and getting properties of files in the directory
- Lines for directories in `ls -l` output begin with `d`, as below:

```
jk@sphere:~/test$ ls -l
Total 4
drwxr-xr-x  2 jk ugrad 4096 2005-10-13 07:37 dir1
-rw-r--r--  1 jk ugrad    0 2005-10-13 07:18 file1
```

Acknowledgement: Matt Bishop

Permissions Examples (Directories)

drwxr-xr-x	all can enter and list the directory, only owner can add/delete files
drwxrwx---	full access to owner and group, forbidden to others
drwx--x---	full access to owner, group can access known filenames in directory, forbidden to others
-rwxrwxrwx	full access to everyone

File Sharing Challenge

- Creating and modifying groups requires root
- Given a directory with permissions drwx------x and a file in it
  - Give permission to write the file to user1, user2, user3, … without creating a new group
  - Selectively revoke a user
- One possible solution
  - Give file write permission for everyone
  - Create different random hard links: user1-23421, user2-56784, …
- Problem! Selectively removing access: hard link can be copied

Acknowledgement: Matt Bishop

Root

- “root” account is a super-user account, like Administrator on Windows
- Multiple roots possible
- File permissions do not restrict root
- This is *dangerous*, but necessary, and OK with good practices

Becoming Root

- su
  - Changes home directory, PATH, and shell to that of root, but doesn’t touch most of environment and doesn’t run login scripts
- su -
  - Logs in as root just as if root had done so normally
- sudo <command>
  - Run just one command as root
- su [-] <user>
  - Become another non-root user
  - Root does not require to enter password

Acknowledgement: Matt Bishop

Changing Permissions

- Permissions are changed with chmod or through a GUI like Konqueror
- Only the file owner or root can change permissions
- If a user owns a file, the user can use chgrp to set its group to any group of which the user is a member
- root can change file ownership with chown (and can optionally change group in the same command)
- chown, chmod, and chgrp can take the -R option to recur through subdirectories

Acknowledgement: Matt Bishop

Examples of Changing Permissions

chown -R root dir1	Changes ownership of dir1 and everything within it to root
chmod g+w,o-rwx file1 file2	Adds group write permission to file1 and file2, denying all access to others
chmod -R g=rwX dir1	Adds group read/write permission to dir1 and everything within it, and group execute permission on files or directories where someone has execute permission
chgrp testgrp file1	Sets file1’s group to testgrp, if the user is a member of that group
chmod u+s file1	Sets the setuid bit on file1. (Doesn’t change execute bit.)

Limitations of Unix Permissions

- Unix permissions are not perfect
  - Groups are restrictive
  - Limitations on file creation
- Linux optionally uses POSIX ACLs
  - Builds on top of traditional Unix permissions
  - Several users and groups can be named in ACLs, each with different permissions
  - Allows for finer-grained access control
- Each ACL is of the form *type*:[*name*]:*rwx*

Extended ACLs

- ACLs that say more than Unix permissions are extended ACLs
  - Specific users and groups can be named and given permissions via ACLs, which fall under the group class (even for for ACLs naming users and not groups)
- With extended ACLs, mapping to and from Unix permissions is a bit complicated.
- User and other classes map directly to the corresponding Unix permission bits
- Group class contains named users and groups as well as owning group permissions.

Extended ACL Example

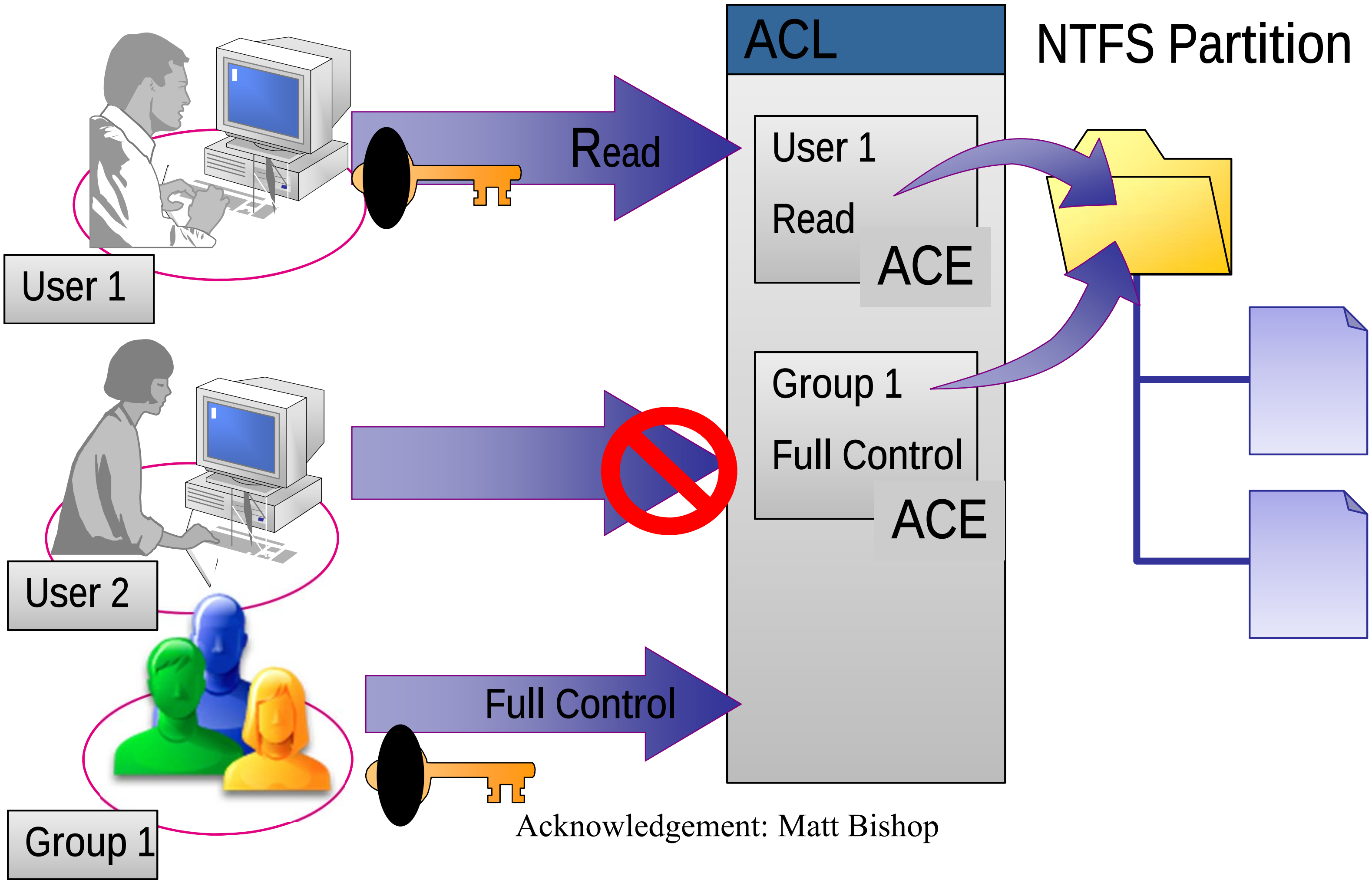
```
jimmy@techhouse:~/test$ ls -l
total 4
drwxr-xr-x  2 jimmy jimmy 4096 2005-12-02 04:13 dir
jimmy@techhouse:~/test$ setfacl -m user:joe:rwx dir
jimmy@techhouse:~/test$ getfacl dir
# file: dir
# owner: jimmy
# group: jimmy
user::rwx
user:joe:rwx
group::r-x
mask::rwx
other::r-x

jimmy@techhouse:~/test$ ls -l
total 8
drwxrwxr-x+ 2 jimmy jimmy 4096 2005-12-02 04:13 dir
```

Extended ACL Example Explained

- The preceding slide grants the named user `joe` read, write, and execute access to `dir`.
  - `dir` now has extended rather than minimal ACLs.
- The mask is set to rwx, the union of the two group class ACLs (named user `joe` and the owning group).
- In `ls -l` output, the group permission bits show the mask, not the owning group ACL
  - Effective owning group permissions are the logical and of the owning group ACL and the mask, which still equals r-x.
  - This could reduce the effective owning group permissions if the mask is changed to be more restrictive.
- The `+` in the `ls -l` output after the permission bits indicates that there are extended ACLs, which can be viewed with `getfacl`.

NTFS Permissions



Acknowledgement: Matt Bishop

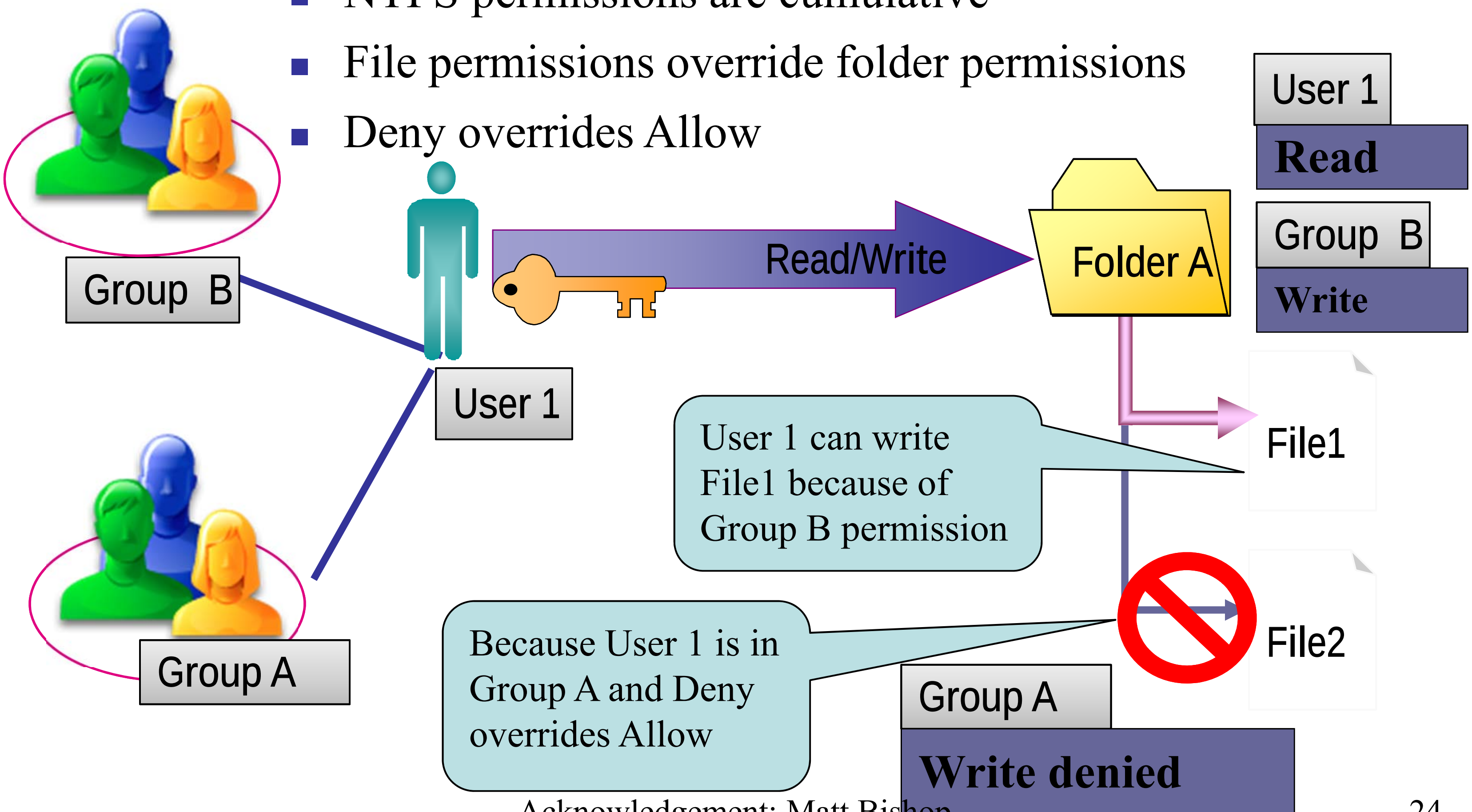
Basic NTFS Permissions

NTFS Permission	Folders	Files
Read	Open files and subfolders	Open files
List Folder Contents	List contents of folder, traverse folder to open subfolders	Not applicable
Read and Execute	Not applicable	Open files, execute programs
Write	Create subfolders and add files	Modify files
Modify	All the above + delete	All the above
Full Control	All the above + change permissions and take ownership, delete subfolders	All the above + change permissions and take ownership

Acknowledgement: Matt Bishop

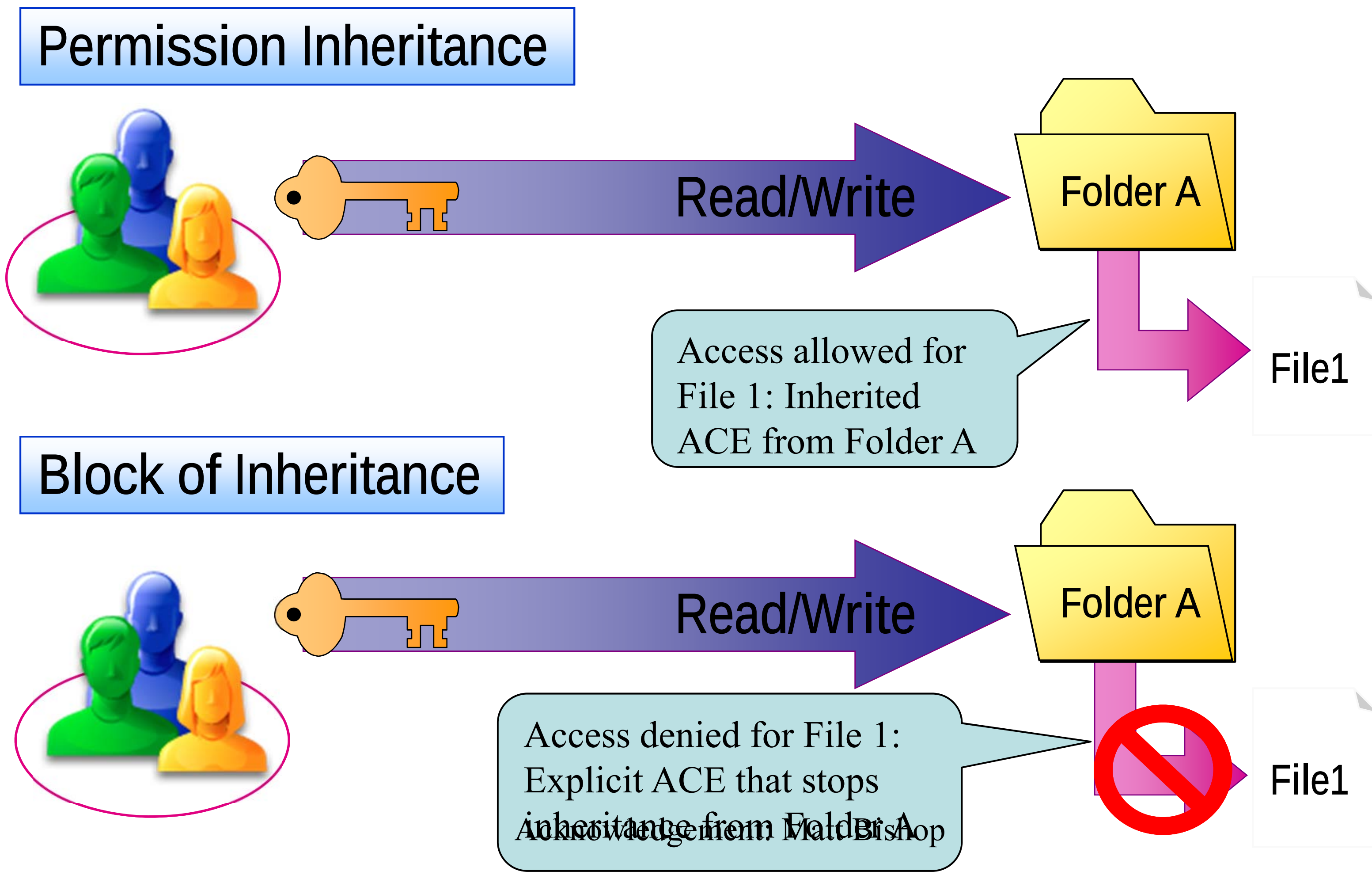
Multiple NTFS permissions

- NTFS permissions are cumulative
- File permissions override folder permissions
- Deny overrides Allow



Acknowledgement: Matt Bishop

NTFS: permission inheritance

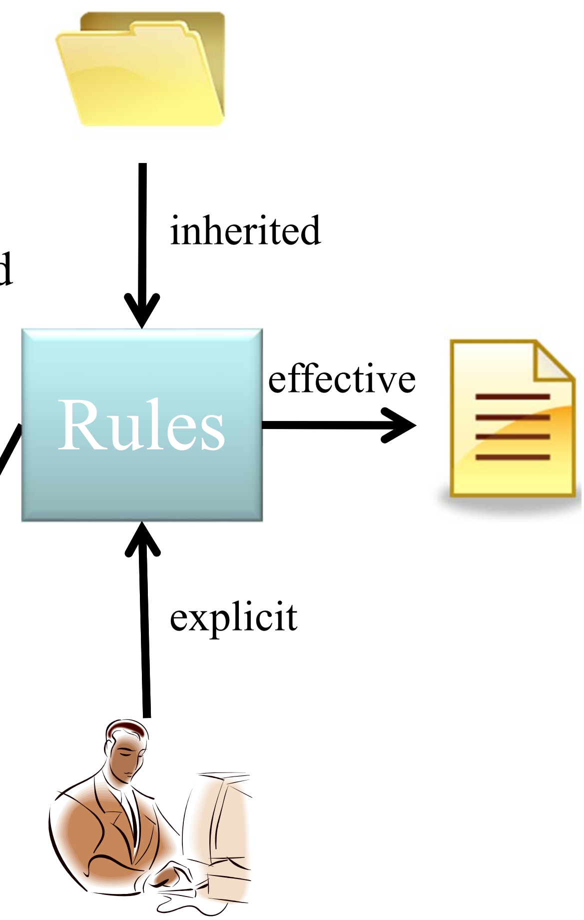


NTFS File Permissions

- **Explicit**: set by the *owner* for each user/group.
- **Inherited**: dynamically inherited from the explicit permissions of ancestor folders.
- **Effective**: obtained by combining the explicit and inherited permission.

Determining effective permissions:

- By default, a user/group has no privileges.
- Explicit permissions override conflicting inherited permissions.
- Denied permissions override conflicting allowed permissions.



Access Control Algorithm

- The DACL of a file or folder is a sorted list of ACEs
  - Local ACEs precede inherited ACEs
  - ACEs inherited from folder F precede those inherited from parent of F
  - Among those with same source, Deny ACEs precede Allow ACEs
- Algorithm for granting access request (e.g., read and execute):
  - ACEs in the DACL are examined in order
  - Does the ACE refer to the user or a group containing the user?
  - If so, do any of the accesses in the ACE match those of the request?
  - If so, what type of ACE is it?
    - **Deny**: return ACCESS_DENIED
    - **Allow**: grant the specified accesses and if there are no remaining accesses to grant, return ACCESS_ALLOWED
  - If we reach the end of the DACL and there are remaining requested accesses that have not been granted yet, return ACCESS_DENIED

Acknowledgement: Matt Bishop

Linux vs. Windows

- Linux
  - Allow-only ACEs
  - Access to file depends on ACL of file and of all its ancestor folders
  - Start at root of file system
  - Traverse path of folders
  - Each folder must have execute (cd) permission
  - Different paths to same file not equivalent
  - File's ACL must allow requested access
- Windows
  - Allow and deny ACEs
  - By default, deny ACEs precede allow ones
  - Access to file depends only on file's ACL (explicit + inherited)
  - ACLs of ancestors ignored when access is requested
  - Permissions set on a folder usually propagated to descendants (inheritance)
  - System keeps track of inherited ACE's

Capability Lists

- Rows of access control matrix

	file1	*file2*	*file3*
Andy	rx	r	rwo
Betty	rwxo	r	
Charlie	rx	rwo	w

C-Lists:

- Andy: { (file1, rx) (file2, r) (file3, rwo) }
- Betty: { (file1, rwxo) (file2, r) }
- Charlie: { (file1, rx) (file2, rwo) (file3, w) }

Acknowledgement: Matt Bishop

Semantics

- Like a bus ticket
  - Mere possession indicates rights that subject has over object
  - Object identified by capability (as part of the token)
    - Name may be a reference, location, or something else
- Must prevent process from altering (forging) capabilities
  - Otherwise subject could change rights encoded in capability or object to which they refer

Example: File Handle in Linux

- Open a file to get a handle, which serves as a capability for follow-up operations on that file
  - fd = open (“homework.txt”, RW)
- Use it by presenting the capability (handle)
  - read(fd, buffer, 100)
  - write(fd, “This is a test”)
- When it is done, release the capability
  - close(fd)

Brief History of Capability-based Systems

- HYDRA (Carnegie-Mellon, William Wulf, 1971)
  - Targeting early MIMD architecture (16 PDP-11 computing nodes)
- GNOSIS, KeyKOS (Tymshare Inc., Key Logic, 1979)
  - IBM S/370, emulating VM, MVS and UNIX environments
- EROS, CapROS, Coyotos (Johns Hopinks University, University of Pennsylvania, Jonathan Shapiro, 1991)
  - 386, 486
- seL4 (NICTA, General Dynamics, 2006)
  - Capability-based mechanisms inspired by Jonathan Shapiro

Implementation

- Tagged architecture
  - Bits protect individual words
    - B5700: tag was 3 bits and indicated how word was to be treated (pointer, type, descriptor, *etc*.)
- Paging/segmentation protections
  - Like tags, but put capabilities in a read-only segment or page
    - CAP system did this
  - Programs must refer to them by pointers
    - Otherwise, program could use a copy of the capability—which it could modify
- Cryptography
  - Associate with each capability a cryptographic checksum enciphered using a key known to OS
  - When process presents capability, OS validates checksum

Amplifying

- Allows *temporary* increase of privileges
- Needed for modular programming
  - Module `passwordDB`

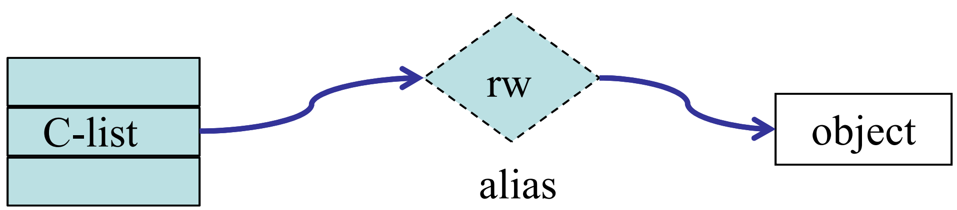
    `module passwordDB ... endmodule.`
  - It saves all passwords in a file (e.g., /etc/passwd)
  - It has method `change_password`, uses can call it
  - *Only* `passwordDB` module can read and alter /etc/passwd
    - So normal user process doesn't get capability, but needs it when calling `change_password`
  - Solution: give process the required capabilities while it is running in the module

An Example

- HYDRA: amplification templates
  – Associated with each procedure and function in a module
  – Adds rights to process capability *while the procedure or function is being executed*
  – Rights are deleted on exit

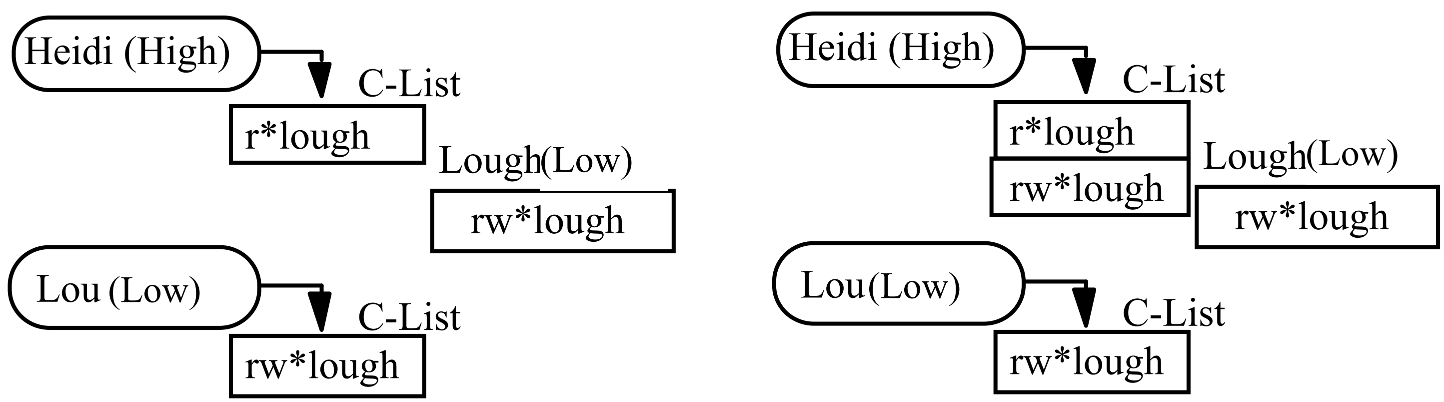
Revocation

- Scan all C-lists, remove relevant capabilities
  – Far too expensive!
- Use indirection
  – Each object has entry in a global object table
  – Names in capabilities name the entry, not the object
    - To revoke, delete the entry in the table
    - Can have multiple entries for a single object to allow control of different sets of rights and/or groups of users for each object
  – Example: HYDRA



Limits

- Problems if you don't control copying of capabilities



The capability to write file *lough* is Low, and Heidi is High so she reads (copies) the capability; now she can write to a Low file, violating the *-property!

Remedies

- Label capability itself
  - Rights in capability depends on relation between its compartment and that of object to which it refers
    - In example, as capability copied to High, and High dominates object compartment (Low), write right removed
- Distinguish between “read” and “copy capability”
  - Take-Grant Protection Model does this (“read”, “take”)

ACLs vs. Capabilities

- Both theoretically equivalent; consider 2 questions
  1. Given a subject, what objects can it access, and how?
  2. Given an object, what subjects can access it, and how?
  – ACLs answer second easily; C-Lists, first
- Suggested that the second question, which in the past has been of most interest, is the reason ACL-based systems more common than capability-based systems
  – As first question becomes more important (in incident response, for example), this may change

Locks and Keys

- Associate information (*lock*) with object, information (*key*) with subject
  - Latter controls what the subject can access and how
  - Subject presents key; if it corresponds to any of the locks on the object, access granted
- This can be dynamic
  - ACLs, C-Lists static and must be manually changed
  - Locks and keys can change based on system constraints or other factors (not necessarily manual)

Cryptographic Implementation

- Enciphered object is lock; deciphering key is key
  - Encipher object o; store $E_k(o)$
  - Use subject's key k' to compute $D_{k'}(E_k(o))$
  - Any of n subjects can access o: store
    $$o' = (E_1(o), \ldots, E_n(o))$$
  - Requires consent of all n subjects to access o: store
    $$o' = (E_1(E_2(\ldots(E_n(o))\ldots))$$

Acknowledgement: Matt Bishop

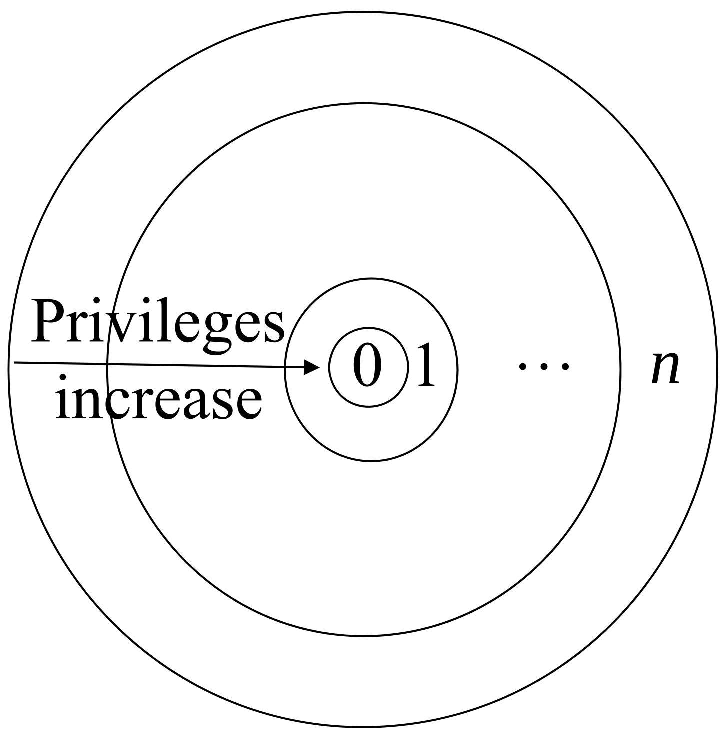
Example: IBM

- IBM 370: process gets access key; pages get storage key and fetch bit
  - Fetch bit clear: read access only
  - Fetch bit set, access key 0: process can write to (any) page
  - Fetch bit set, access key non-zero and matches storage key: process can write to page
  - Fetch bit set, access key non-zero and does not match storage key: no access allowed

Type Checking

- Lock is type, key is operation
  - Example: UNIX system call *write* can't work on directory object but does work on file
    - To avoid corruption of directory entries
  - Example: LOCK (Logical Coprocessor Kernel) system:
    - Compiler produces "data"
    - Trusted process must change this type to "executable" before program can be executed

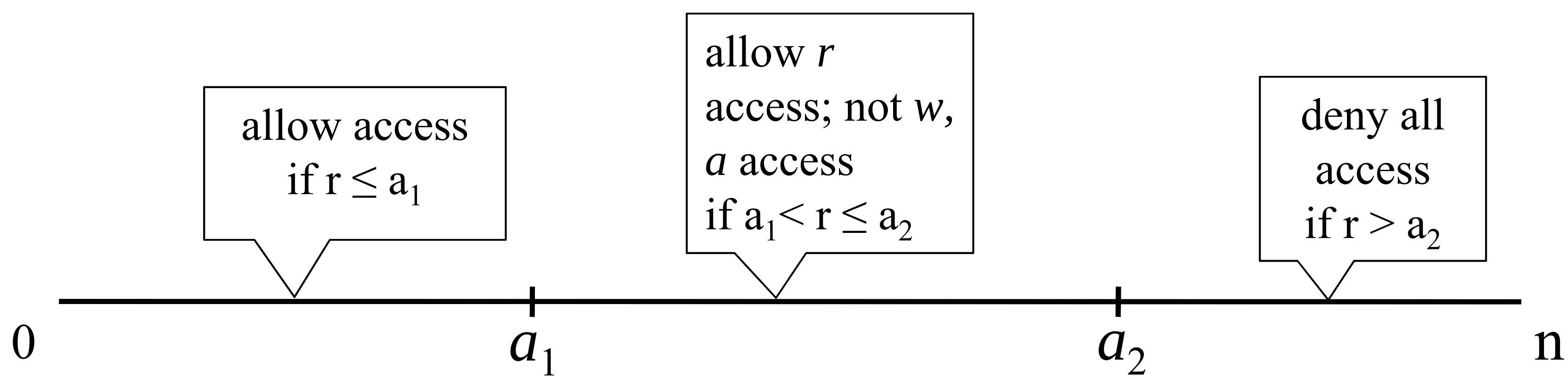
Ring-Based Access Control



- Process (segment) accesses another segment
  - Read
  - Execute
- *Gate* is an entry point for calling segment
- Rights:
  - *r* read
  - *w* write
  - *a* append
  - *e* execute

Reading/Writing/Appending

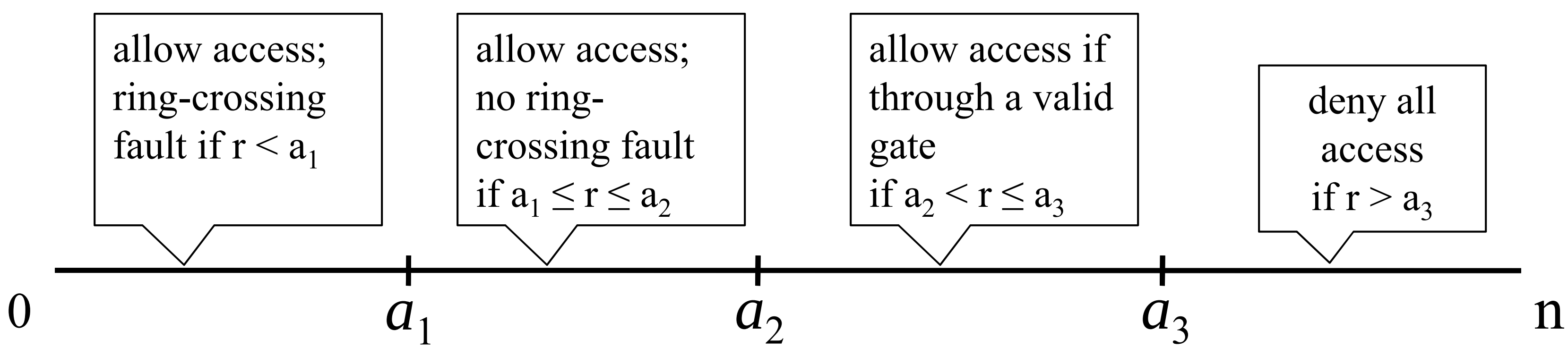
- Procedure executing in ring *r*
- Data segment with *access bracket* (a_1, a_2)
- Mandatory access rule



Acknowledgement: Matt Bishop

Executing

- Procedure executing in ring *r*
- Call procedure in segment with *access bracket* (a_1, a_2) and *call bracket* (a_2, a_3)
  - Often written (a_1, a_2, a_3)
- Mandatory access rule



Versions

- Multics (https://www.multicians.org/multics.html)
  - 8 rings (from 0 to 7)
- Digital Equipment's VAX (https://en.wikipedia.org/wiki/VAX)
  - 4 levels of privilege: user, monitor, executive, kernel
- Older systems
  - 2 levels of privilege: user, supervisor

Relevance of Ring-Based Access Control

- Many modern CPU architectures (including the Intel x86 architecture) include some form of ring protection
  - Intel CPU has 4 rings
- Windows and Linux: OS runs in ring 0, user applications run in ring 3
- Xen: hypervisor runs in ring 0, OS runs in ring 1, user applications run in ring 3

PACLs

- Propagated Access Control List
  - Implements ORCON (ORiginator CONtrolled)
- Creator is kept with PACL and its copies
  - Only creator can change PACL
  - Subject reads object: object's PACL associated with subject
  - Subject writes object: subject's PACL associated with object
- Notation: PACL_s means *s* created object; PACL(*e*) is PACL associated with entity *e*

Acknowledgement: Matt Bishop

Multiple Creators

- Betty reads Ann's file *dates* (PACL(*dates*) = PACL_{Ann})

  PACL(Betty) = $\text{PACL}_{\text{Betty}} \cap$ PACL(*dates*)

  $= \text{PACL}_{\text{Betty}} \cap \text{PACL}_{\text{Ann}}$
- Betty creates file *dc*

  PACL(*dc*) = PACL(Betty) = $\text{PACL}_{\text{Betty}} \cap \text{PACL}_{\text{Ann}}$
- If $\text{PACL}_{\text{Betty}}$ allows Char to access objects, but PACL_{Ann} does not; and both allow June to access objects
  - June can read *dc*
  - Char cannot read *dc*

Key Points

- Access control mechanisms provide controls for users accessing files
- Many different forms
  - ACLs, capabilities, locks and keys
  - Ring-based mechanisms (Mandatory)
  - PACLs (ORCON)