DAC vs. MAC

- **Most people familiar with discretionary access control (DAC)**
  - Unix permission bits are an example
  - Might set a file `private` so only group `friends` can read it
- **Discretionary means anyone with access can propagate information:**
  - `Mail sigint@enemy.gov < private`
- **Mandatory access control**
  - Security administrator can restrict propagation
  - Abbreviated MAC (NOT a message authentication code)

Bell-Lapadula model

- **View the system as subjects accessing objects**
  - The system input is requests, the output is decisions
  - Objects can be organized in one or more hierarchies, H (a tree enforcing the type of decendents)
- **Four modes of access are possible:**
  - execute – no observation or alteration
  - read – observation
  - append – alteration
  - write – both observation and modification
- **The current access set, b, is (subj, obj, attr) tripples**
- **An access matrix M encodes permissible access types (as before, subjects are rows, objects columns)**

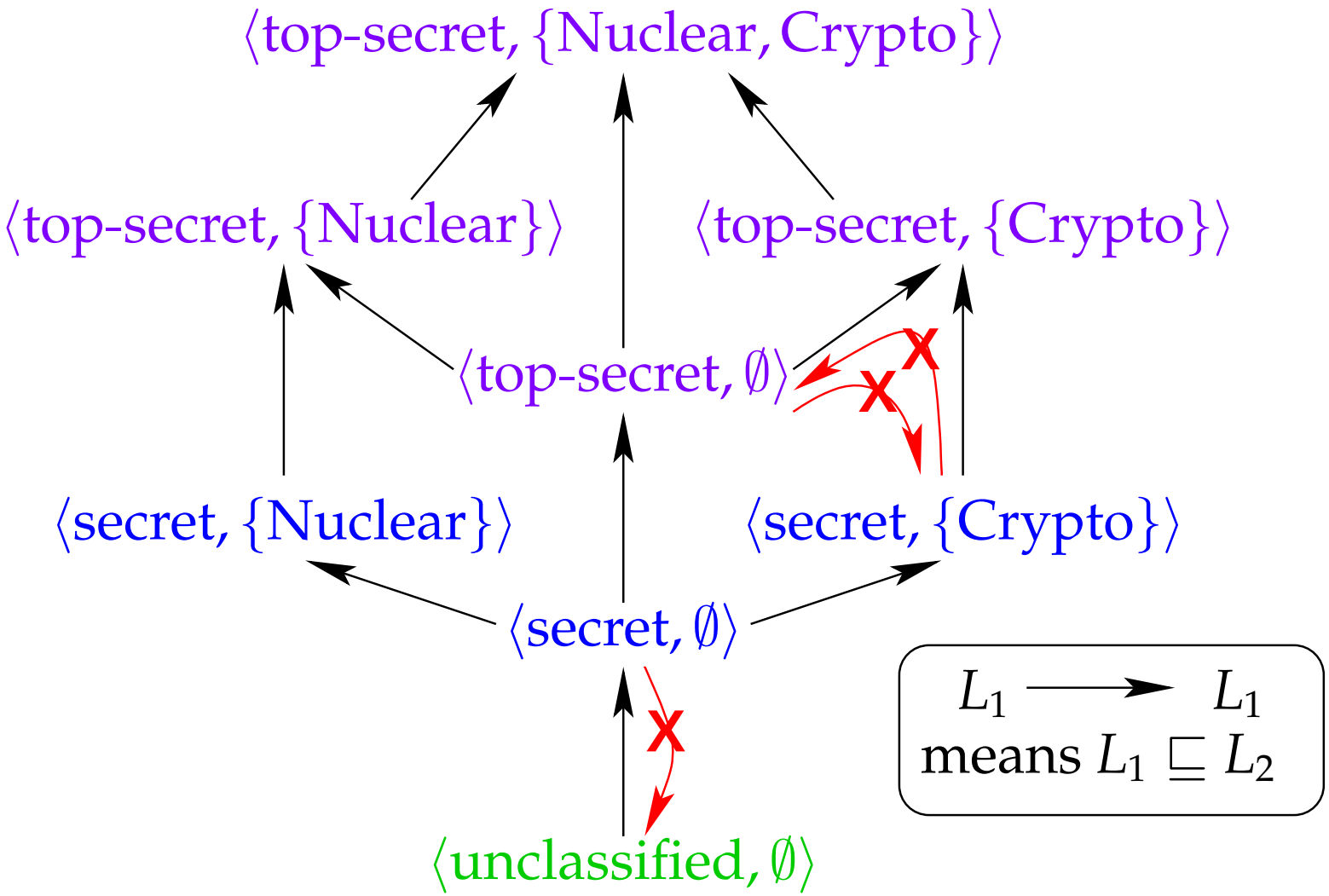
Security levels

- **A *security level* is a (c, s) pair:**
  - $c =$ classification – E.g., unclassified, secret, top secret
  - $s =$ category-set – E.g., Nuclear, Crypto
- (c_1, s_1) ***dominates*** (c_2, s_2) **iff** $c_1 \geq c_2$ **and** $s_2 \subseteq s_1$
  - L_1 *dominates* L_2 sometimes written $L_1 \sqsupseteq L_2$ or $L_2 \sqsubseteq L_1$
  - levels then form a *lattice* (partial order w. lub & glb)
- **Subjects and objects are assigned security levels**
  - level(S), level(O) – security level of subject/object
  - current-level(S) – subject may operate at lower level
  - level(S) bounds current-level(S) (current-level(S) $\sqsubseteq$ level(S))
  - Since level(S) is max, sometimes called S's *clearance*

Security properties

- **The simple security or *ss-property*:**
  - For any $(S, O, A) \in b$, if A includes observation, then level(S) must dominate level(O)
  - E.g., an unclassified user cannot read a top-secret document
- **The star security or *-*property*:**
  - If a subject can observe O_1 and modify O_2, then level(O_2) dominates level(O_1)
  - E.g., cannot copy top secret file into secret file
  - More precisely, given $(S, O, A) \in b$:
    if $A = r$ then current-level(S) $\sqsupseteq$ level(O) ("no read up")
    if $A = a$ then current-level(S) $\sqsubseteq$ level(O) ("no write down")
    if $A = w$ then current-level(S) $=$ level (O)

The lattice model



- **Information can only flow up the lattice**
  - System enforces "No read up, no write down"
  - Think of $\sqsubseteq$ as "can flow to" relation

Straw man MAC implementation

- **Take an ordinary Unix system**
- **Put labels on all files and directories to track levels**
- **Each user U has a security clearance (level(U))**
- **Determine current security level dynamically**
  - When U logs in, start with lowest curent-level
  - Increase current-level as higher-level files are observed (sometimes called a *floating label* system)
  - If U's level does not dominate current, kill program
  - If program writes to file it doesn't dominate, kill it
- **Is this secure?**

No: Covert channels

- **System rife with *storage channels***
  - Low current-level process executes another program
  - New program reads sensitive file, gets high current-level
  - High program exploits covert channels to pass data to low
- **E.g., High program inherits file descriptor**
  - Can pass 4-bytes of information to low prog. in file offset
- **Other storage channels:**
  - Exit value, signals, file locks, terminal escape codes, …
- **If we eliminate storage channels, is system secure?**

No: Timing channels

- **Example: CPU utilization**
  - To send a 0 bit, use 100% of CPU is busy-loop
  - To send a 1 bit, sleep and relinquish CPU
  - Repeat to transfer more bits
- **Example: Resource exhaustion**
  - High prog. allocate all physical memory if bit is 1
  - If low prog. slow from paging, knows less memory available
- **More examples: Disk head position, processor cache/TLB polution, …**

Reducing covert channels

- **Observation: Covert channels come from sharing**
  - If you have no shared resources, no covert channels
  - Extreme example: Just use two computers
- **Problem: Sharing needed**
  - E.g., read unclassified data when preparing classified
- **Approach: Strict partitioning of resources**
  - Strictly partition and schedule resources between levels
  - Occasionally reapportion resources based on usage
  - Do so infrequently to bound leaked information
  - In general, only hope to bound bandwidth of covert channels
  - Approach still not so good if many security levels possible

Declassification

- **Sometimes need to prepare unclassified report from classified data**
- **Declassification happens outside of system**
  - Present file to security officer for downgrade
- **Job of declassification often not trivial**
  - E.g., Microsoft word saves a lot of undo information
  - This might be all the secret stuff you cut from document

Biba integrity model

- **Problem: How to protect integrity**
  - Suppose text editor gets trojaned, subtly modifies files, might mess up attack plans
- **Observation: Integrity is the converse of secrecy**
  - In secrecy, want to avoid writing less secret files
  - In integrity, want to avoid writing higher-integrity files
- **Use integrity hierarchy parallel to secrecy one**
  - Now *security level* is a $\langle c, i, s \rangle$ triple, i =integrity
  - Only trusted users can operate at low integrity levels
  - If you read less authentic data, your current integrity level gets raised, and you can no longer write low files

DoD Orange book

- **DoD requirements for certification of secure systems**
- **4 Divisions:**
  - D – been through certification and not secure
  - C – discretionary access control
  - B – mandatory access control
  - A – like B, but better verified design
  - Classes within divisions increasing level of security

Divisions C and D

- **Level D: Certifiably insecure**
- **Level C1: Discretionary security protection**
  - Need some DAC mechanism (user/group/other, ACLs, etc.)
  - TCB needs protection (e.g., virtual memory protection)
- **Level C2: Controlled access protection**
  - Finer-graunlarity access control
  - Need to clear memory/storage before reuse
  - Need audit facilities
- **Many OSes have C2-security packages**
  - Is, e.g., C2 Solaris "more secure" than normal Solaris?

Division B

- **B1 - Labeled Security Protection**
  - Every object and subject has a label
  - Some form of reference monitor
  - Use Bell-LaPadula model and some form of DAC
- **B2 - Structured Protection**
  - More testing, review, and validation
  - OS not just one big program (least priv. within OS)
  - Requires covert channel analysis
- **B3 - Security Domains**
  - More stringent design, w. small ref monitor
  - Audit required to detect imminent violations
  - requires security kernel + 1 or more levels *within* the OS

Division A

- **A1 – Verified Design**
  - Design must be formally verified
  - Formal model of protection system
  - Proof of its consistency
  - Formal top-level specification
  - Demonstration that the specification matches the model
  - Implementation shown informally to match specification

Limitations of Orange book

- **How to deal with floppy disks?**
- **How to deal with networking?**
- **Takes too long to certify a system**
  - People don't want to run n-year-old software
- **Doesn't fit non-military models very well**
- **What if you want high assurance & DAC?**

Today: Common Criteria

- **Replaced orange book around 1998**
- **Three parts to CC:**
  - CC Documents, including protection profiles w. both functional and assurance requirements
  - CC Evaluation Methodology
  - National Schemes (local ways of doing evaluation)

Protection Profiles

- **Requirements for categories of systems**
  - Subject to review and certified
- **Example: Controlled Access PP (CAPP_V1.d)**
  - **Security functional requirements:** Authentication, User Data Protection, Prevent Audit Loss
  - **Security assurance requirements:** Security testing, Admin guidance, Life-cycle support, …
  - Assumes non-hostile and well-managed users
  - Does not consider malicious system developers

Evaluation Assurance Levels 1-4

- **EAL 1: Functionally Tested**
  - Review of functional and interface specifications
  - Some independent testing
- **EAL 2: Structurally Tested**
  - Analysis of security functions, incl high-level design
  - Independent testing, review of developer testing
- **EAL 3: Methodically Tested and Checked**
  - Development environment controls; config mgmt
- **EAL 4: Methodically Designed, Tested, Reviewed**
  - Informal spec of security policy, Independent testing

Evaluation Assurance Levels 5-7

- **EAL 5: Semi-formally designed and tested**
  - Formal model, modular design
  - Vulnerability search, covert channel analysis
- **EAL 6: Semi-formally verified design and tested**
  - Structured development process
- **EAL 7: Formally verified design and tested**
  - Formal presentation of functional specification
  - Product or system design must be simple
  - Independent confirmation of developer tests

LOMAC

- **Problem: MAC not widely accepted outside military**
- **LOMAC's goal is to make MAC more palatable**
  - Stands for Low water Mark Access Control
- **Concentrates on Integrity**
  - More important goal for many settings
  - E.g., don't want viruses tampering with all your files
  - Also don't have to worry as much about covert channels
- **Provides reasonable defaults (minimally obtrusive)**
- **Has actually had some impact**
  - Available for Linux
  - Integrated in FreeBSD-current source tree
  - Probably inspired Vista's Mandatory Integrity Control (MIC)

LOMAC overview

- **Subjects are *jobs* (essentially processes)**
  - Each subject has an integrity number (e.g., 1, 2)
  - Higher numbers mean more integrity
    (so unfortunately $2 \sqsubseteq 1$ by earlier notation)
  - Subjects can be reclassified on observation of low-integrity data
- **Objects are files, pipes, etc.**
  - Objects have fixed integrity level; cannot change
- **Security: Low-integrity subjects cannot write to high integrity objects**
- **New objects have level of the creator**

LOMAC defaults

- **By default two levels, 1 and 2**
- **Level 2 (high-integrity) contains:**
  - All the FreeBSD/Linux files intact from software distribution
  - The console and trusted terminals
- **Level 1 (low-integrity) contains**
  - Network devices, untrusted terminals, etc.
- **Idea: Suppose worm compromises your web server**
  - Worm comes from network → level 1
  - Won't be able to muck with system files

The self-revocation problem

- **Want to integrate with Unix unobtrusively**
- **Problem: Application expectations**
  - Kernel access checks usually done at file open time
  - Legacy applications don't pre-declare they will observe low-integrity data
  - An application can "taint" itself unexpectedly, revoking its own permission to access an object it created
- **Example:** `ps | grep user`
  - Pipe created before `ps` reads low-integrity data
  - `ps` becomes tainted, can no longer write to `grep`

Solution

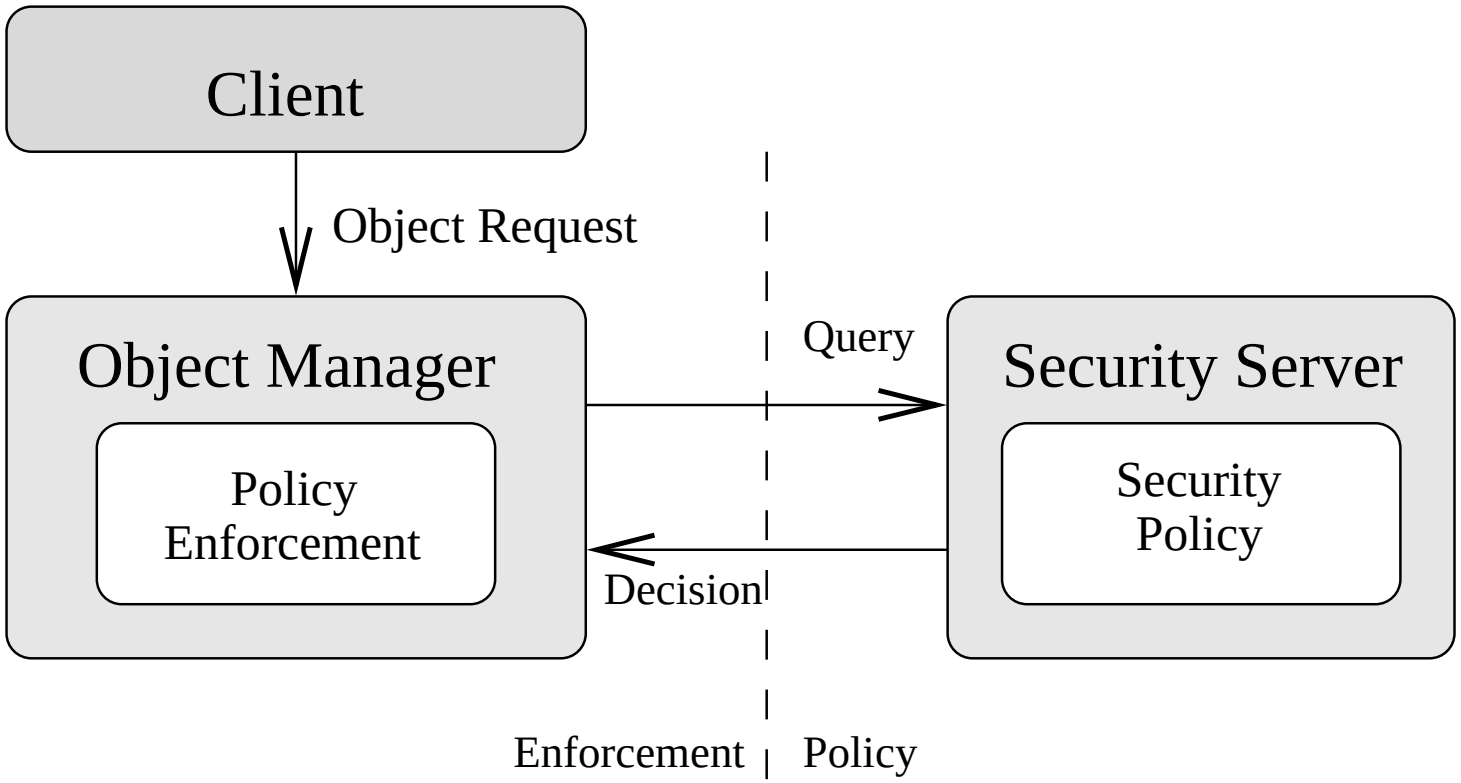
- **Don't consider pipes to be real objects**
- **Join multiple processes together in a "job"**
  - Pipe ties processes together in job
  - Any processes tied to job when they read or write to pipe
  - So will lower integrity of both ps and grep
- **Similar idea applies to shared memory and IPC**
- **LOMAC applies MAC to non-military systems**
  - But doesn't allow military-style security policies (i.e., with secrecy, various categories, etc.)

The flask security architecture

- **Problem: Military needs adequate secure systems**
  - How to create civilian demand for systems military can use?
- **Idea: Separate policy from enforcement mechanism**
  - Most people will plug in simple DAC policies
  - Military can take system off-the-shelf, plug in new policy
- **Requires putting adequate hooks in the system**
  - Each object has manager that guards access to the object
  - Conceptually, manager consults security server on each access
- **Flask security architecture prototyped in fluke**
  - Now part of SElinux, which NSA hopes to see accepted

[following figures from Spencer et al.]

Architecture



- **Separating enforcement from policy**

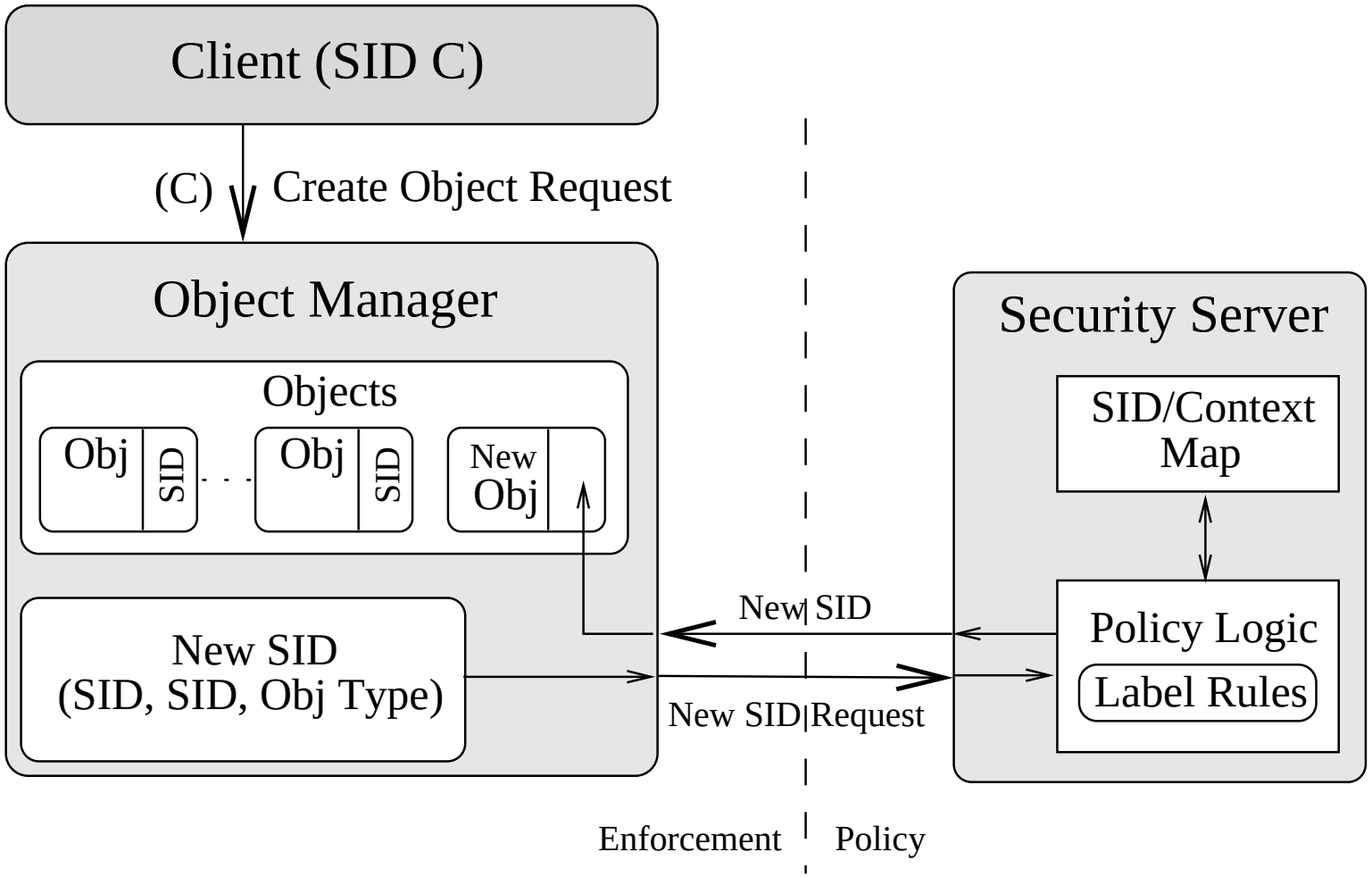
Challenges

- **Performance**
  - Adding hooks on every operation
  - People who don't need security don't want slowdown
- **Using generic enough data structures**
  - Object managers independent of policy still need to associate data structures (e.g., labels) with objects
- **Revocation**
  - May interact in a complicated way with any access caching
  - Once revocation completes, new policy must be in effect
  - Bad guy cannot be allowed to delay revocation completion indefinitely

Basic flask concepts

- **All objects are labeled with a *security context***
  - Security context is an arbitrary string—opaque to obj mgr
  - Example: $\{\text{invoice } [(\text{Andy}, \text{Authorize})]\}$
- **Labels abbreviated with security IDs (SIDs)**
  - 32-bit integer, interpretable only by security server
  - Not valid across reboots (can't store in file system)
  - Fixed size makes it easier for obj mgr to handle
- **Queries to server done in terms of SIDs**
  - Create (client SID, old obj SID, obj type)? $\rightarrow$ SID
  - Allow (client SID, obj SID, perms)? $\rightarrow$ $\{\text{yes}, \text{no}\}$

Creating new object



Security server interface

```
int security_compute_av(
    security_id_t ssid, security_id_t tsid,
    security_class_t tclass, access_vector_t requested,
    access_vector_t *allowed, access_vector_t *decided,
    __u32 *seqno);
```

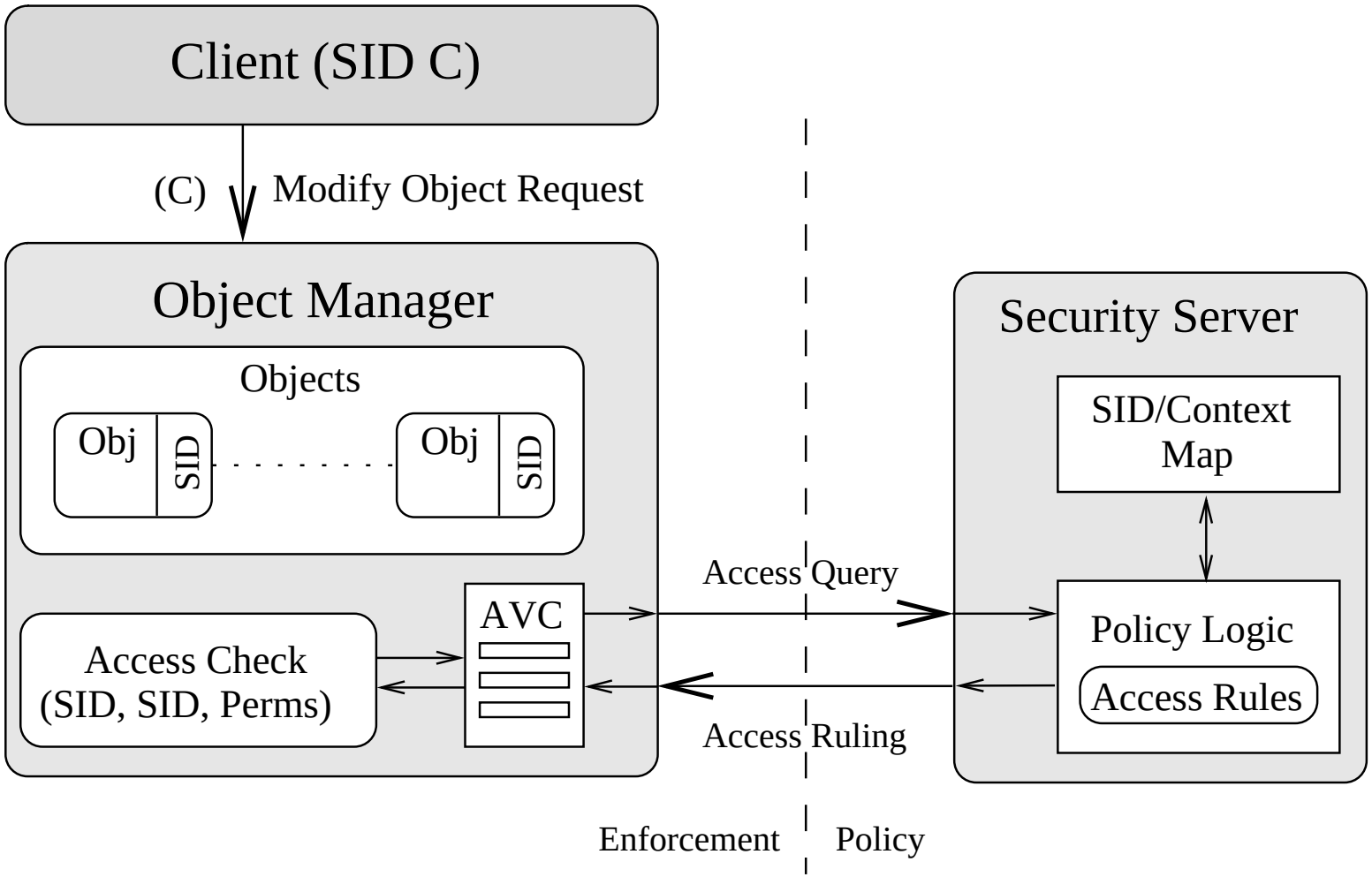
- `ssid`, `tsid` – **source and target SIDs**
- `tclass` – **type of target**
  - E.g., regular file, device, raw IP socket, TCP socket, ...
- **Server can decide more than it is asked for**
  - `access_vector_t` is a bitmask of permissions
  - `decided` can contain more than `requested`
  - Effectively implements decision prefetching
- `seqno` **used for revocation (in a few slides)**

Access vector cache (AVC)

- **Want to minimize calls into security server**
- **AVC caches results of previous decisions**
  - Note: Relies on simple enumerated permissions
- **Decisions therefore cannot depend on paremeters:**
  - Andy can authorize expenses up to $999.99
  - Bob can run processes at priority 10 or higher
- **Decisions also limited to two SIDs**
  - Complicates file relabeling, which requires 3 checks:

Source	**Target**	**Permission checked**
Subject SID	File SID	Relabel-From
Subject SID	New SID	Relabel-To
File SID	New SID	Transition-From

AVC in a query



AVC interface

```
int avc_has_perm_ref(
    security_id_t ssid, security_id_t tsid,
    security_class_t tclass, access_vector_t requested,
    avc_entry_ref_t *aeref);
```

- `avc_entry_ref_t` **points to cached decision**
  - Contains `ssid`, `tsid`, `tclass`, decision vec., & recently used info
- `aeref` **argument is hint**
  - On first call, will be set to relevent AVC entry
  - On subsequent calls speeds up lookup
- **Example: New kernel check when binding a socket:**

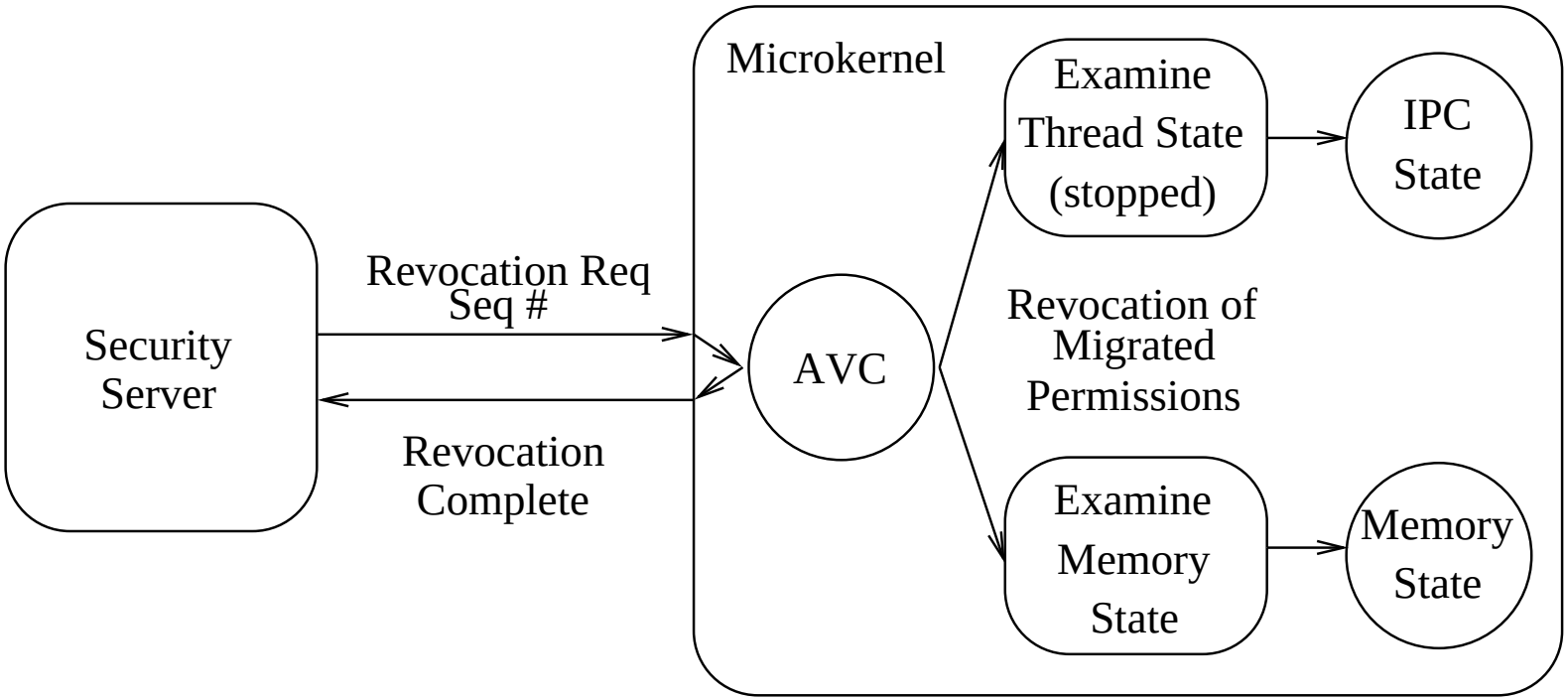
```
ret = avc_has_perm_ref(
    current->sid, sk->sid, sk->sclass,
    SOCKET__BIND, &sk->avcr);
```

  - Next socket op, `sk->avcr` likely points to appropriate entry

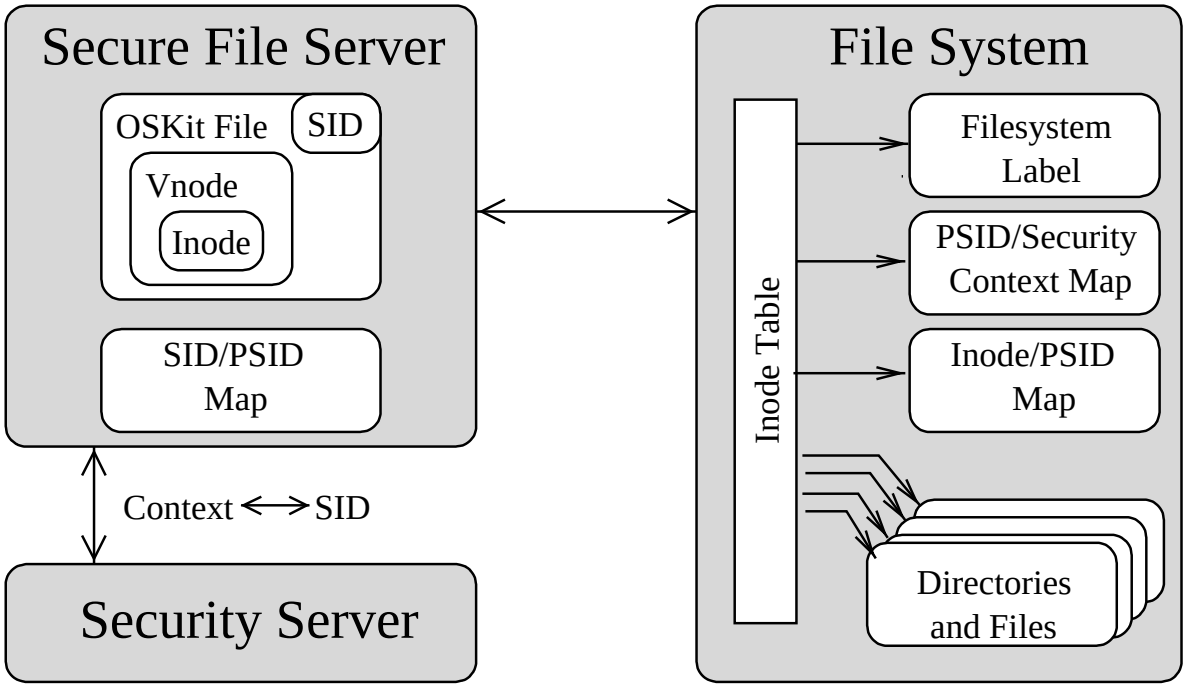
Revocation support

- **Decisions may be cached in in AVCs**
  - `seqno` prevents caching of decisions before revocation
- **Decisions may implicitly be cached in migrated permissions**
  - E.g., Unix checks file write permission on *open*
  - But may want to disallow future writes even on open file
  - Write permission migrated into file descriptor
  - May also migrate into page tables/TLB w. `mmap`
  - Also may migrate into open sockets/pipes, or operations in progress
- **AVC contains hooks for callbacks**
  - After revoking in AVC, AVC makes callbacks to revoke migrated permissions

Revocation protocol



Persistence



- **Must label persistent objects in file system**
  - Persistently map each file/directory to a security context
  - Security contexts are variable length, so add level of indirection
  - "Persistent SIDs" (PSIDs) – numbers local to each file system

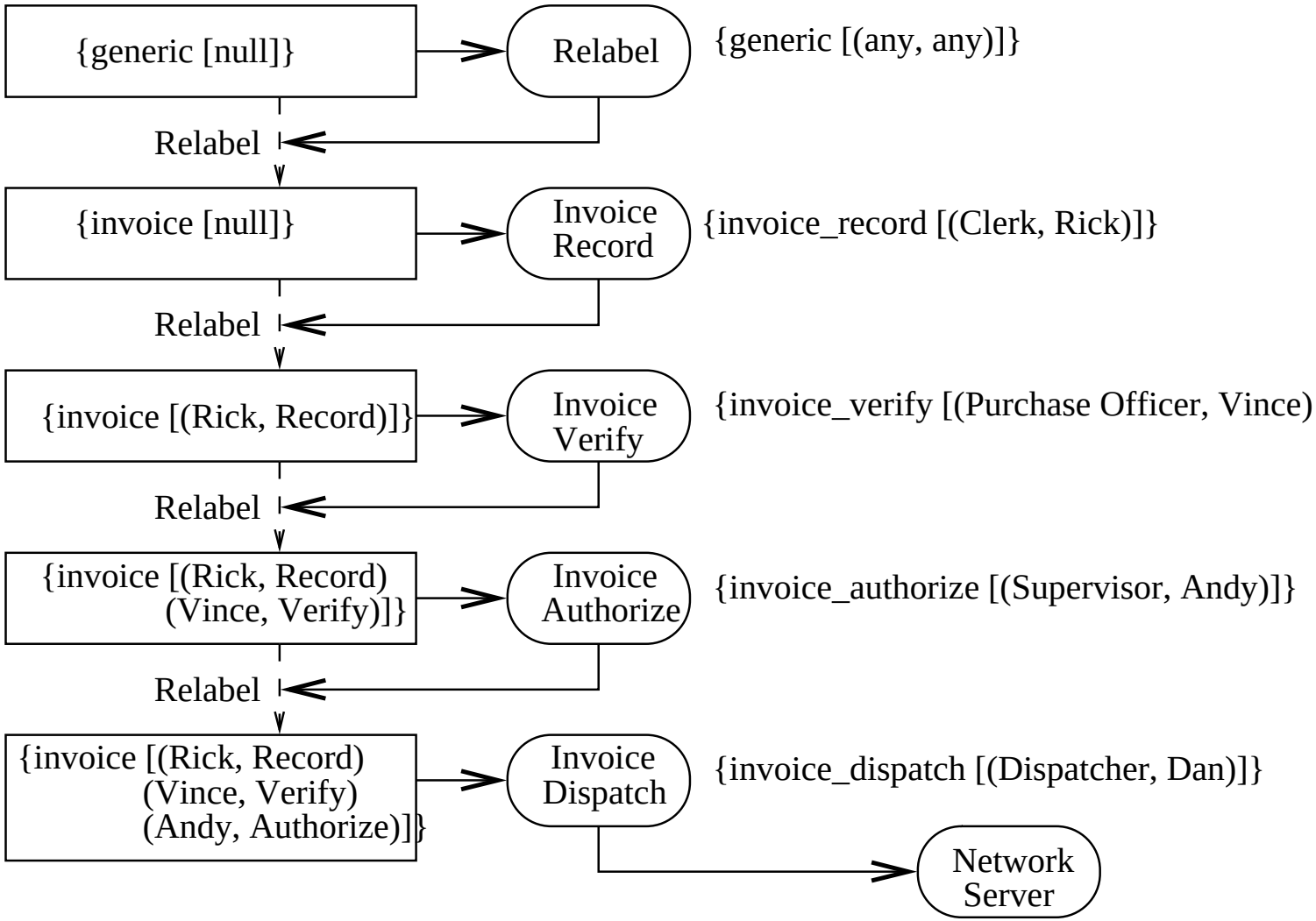
Transitioning SIDs

- **May need to relabel objects (e.g., files)**
  - E.g., in file system
- **Processes may also want to transition their SIDs**
  - Depends on existing permission, but also on program
  - SElinux allows programs to be defined as *entrypoints*
  - Thus, can restrict with which programs users enter a new SID

Example: Paying invoices

- **Invoices are special immutable files**
- **Each invoice must undergo the following processing:**
  - Receipt of the invoice recorded by a clerk
  - Receipt of of the merchandise verified by purchase officer
  - Payment of invoice approved by supervisor
- **Special programs allowed to record each of the above events**
  - E.g., force clerk to read invoice—cannot just write a batch script to relabel all files

Illustration



Example: Loading kernel modules

```
(1) allow sysadm_t insmod_exec_t:file x_file_perms;
(2) allow sysadm_t insmod_t:process transition;
(3) allow insmod_t insmod_exec_t:process { entrypoint execute };
(4) allow insmod_t sysadm_t:fd inherit_fd_perms;
(5) allow insmod_t self:capability sys_module;
(6) allow insmod_t sysadm_t:process sigchld;
```

1: Allow sysadm domain to run insmod

2: Allow sysadm domain to transition to insmod

3: Allow insmod program to be entrypoint for insmod domain

4: Let insmod inherit file descriptors from sysadm

5: Let insmod use CAP_SYS_MODULE (load a kernel module)

6: Let insmod signal sysadm with SIGCHLD when done