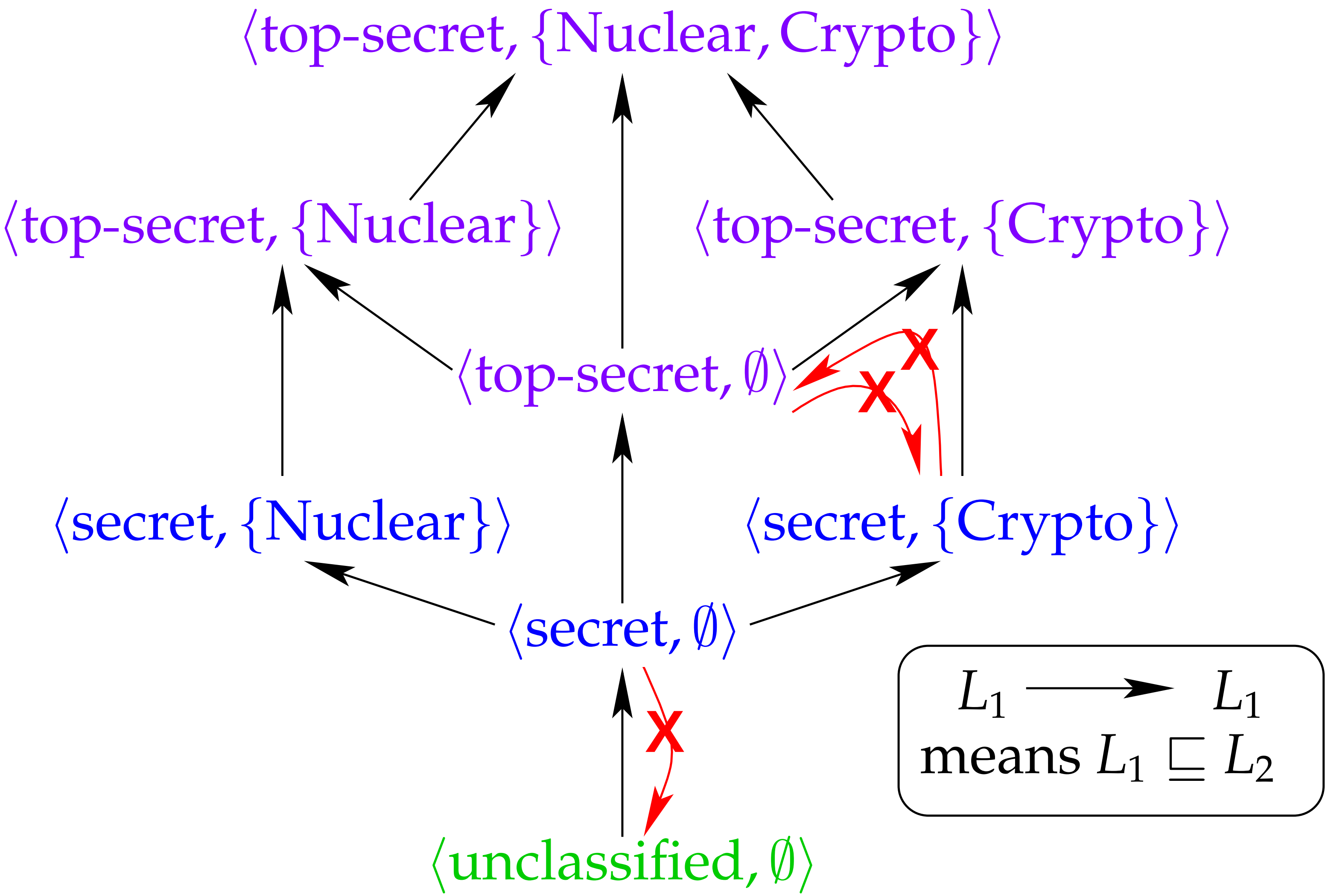
The short-term plan

- **Last time we talked about OS protection**
  - Unix permissions, Capabilities
  - TOCTTOU bugs, the confused deputy problem
  - Mandatory Access Control (MAC)
- **Today's related topic: confining untrusted code**
- **We will consider issue from the OS level on up**
  - Continued discussion of MAC & how it applies
  - Other OS extensions
  - System call interposition
  - User-level sandboxing

Recall Bell-La Padula's labels



- **Information can only flow up the lattice**
  - "No read up, no write down"

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 Assumes 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 Assumes 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 $\rightarrow$ 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

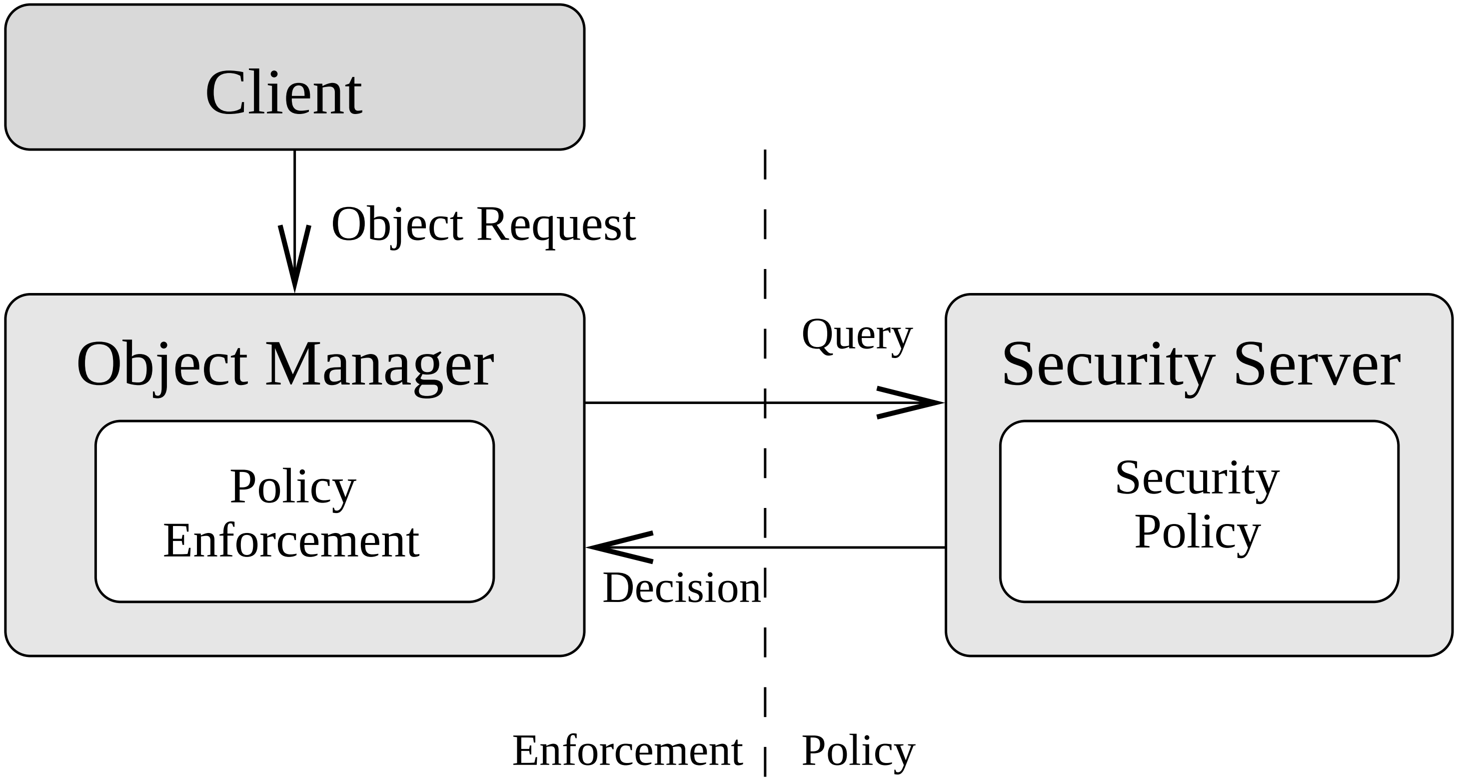
- **Don't consider pipes to be real objects**
- **Join multiple processes together in a "job"**
  - Pipe ties processes together in job
  - Any processes tied to job when they read or write to pipe
  - So will lower integrity of both ps and grep
- **Similar idea applies to shared memory and IPC**
- **LOMAC 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**

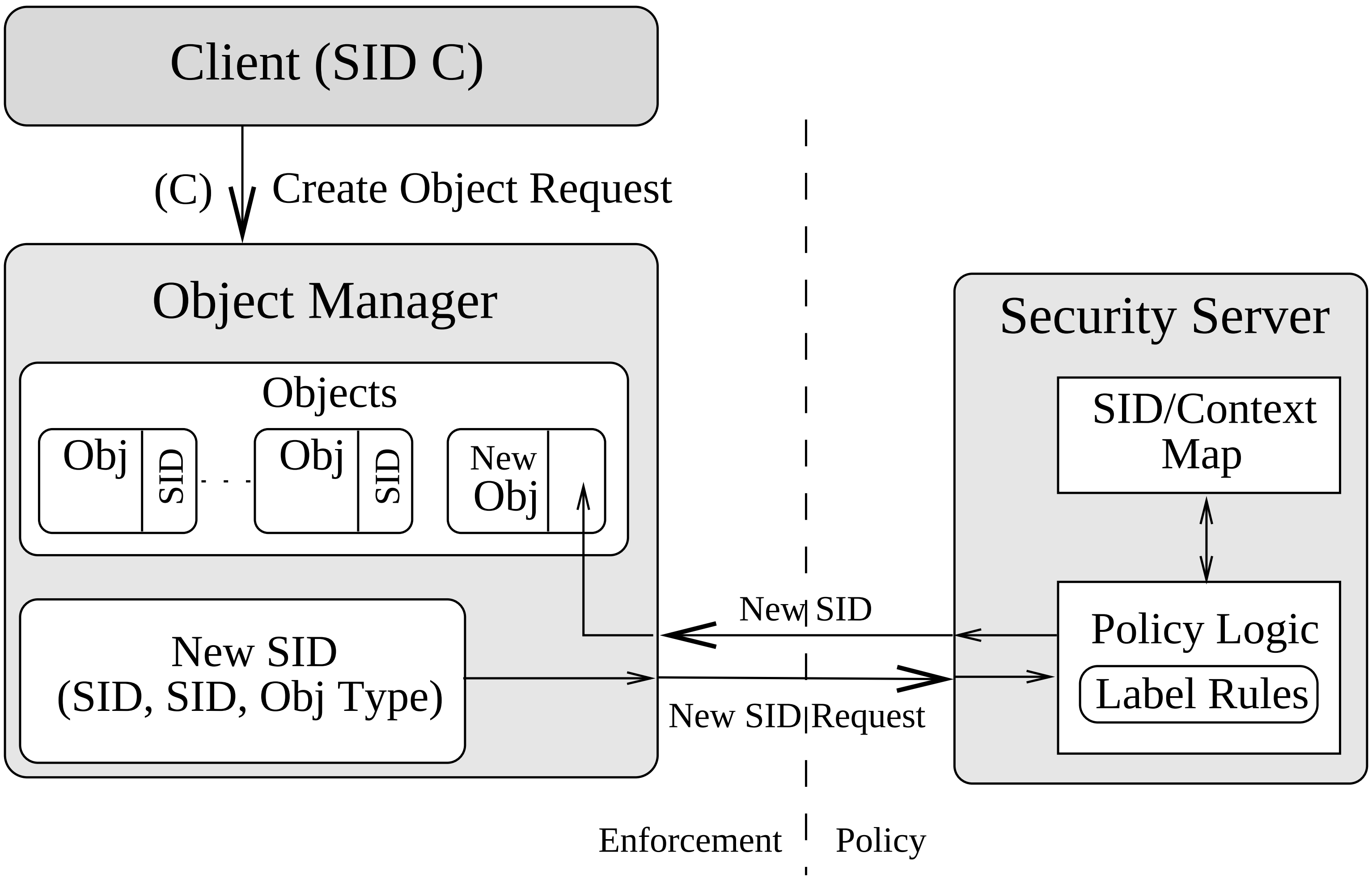
Challenges

- **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: {invoice [(Andy, 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)? → SID
  - Allow (client SID, obj SID, perms)? → {yes, 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);
```

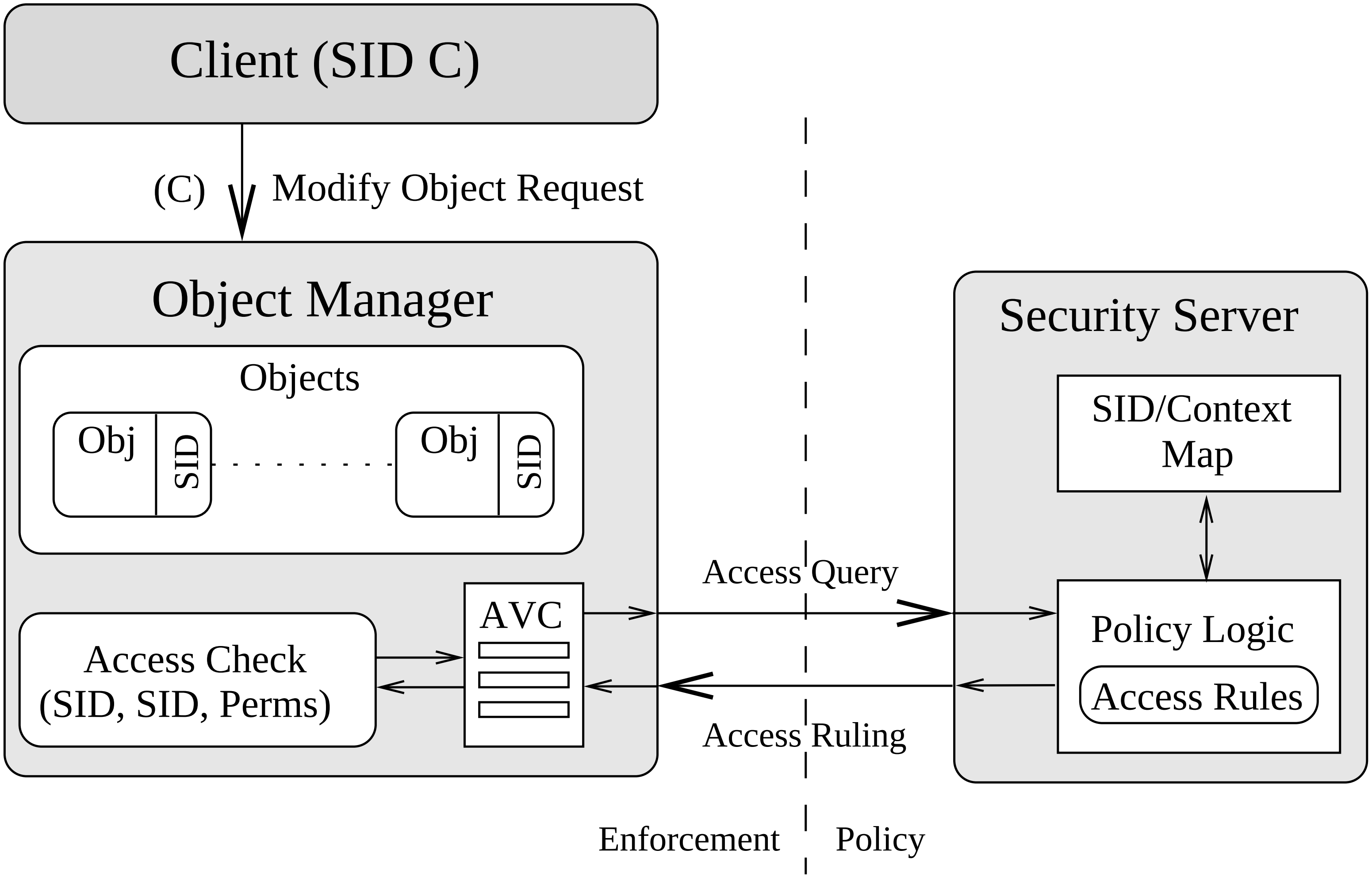
- **Server can decide more than it is asked for**
  - `decided` will contain more than `requested`
  - Effectively implements decision prefetching

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);
```

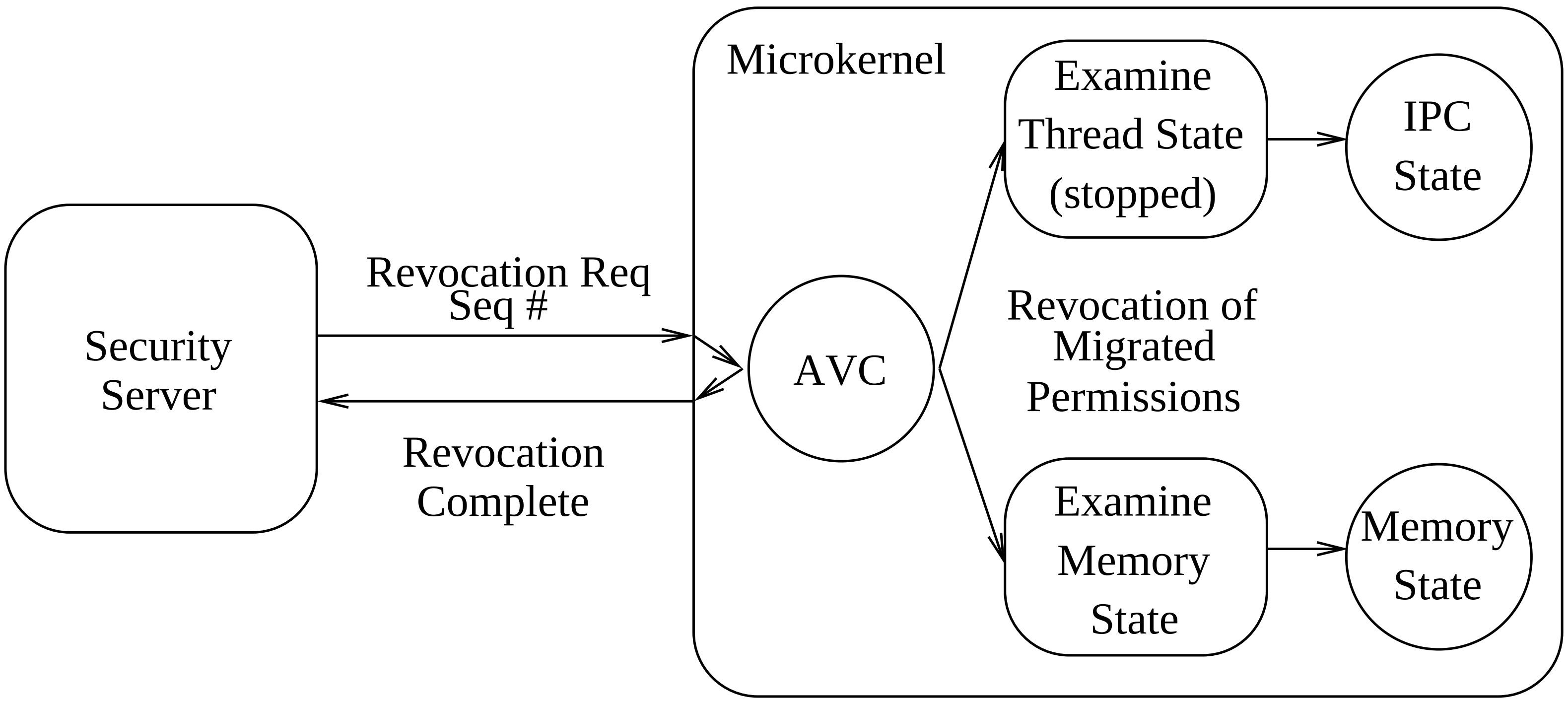
- `access_vector_t` **is bitmap of permissions to check**
- `aeref` **argument is hint**
  - On first call, will be set to relevent AVC entry
  - On subsequent calls speeds up lookup
- **Example: New kernel check when binding a socket:**

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

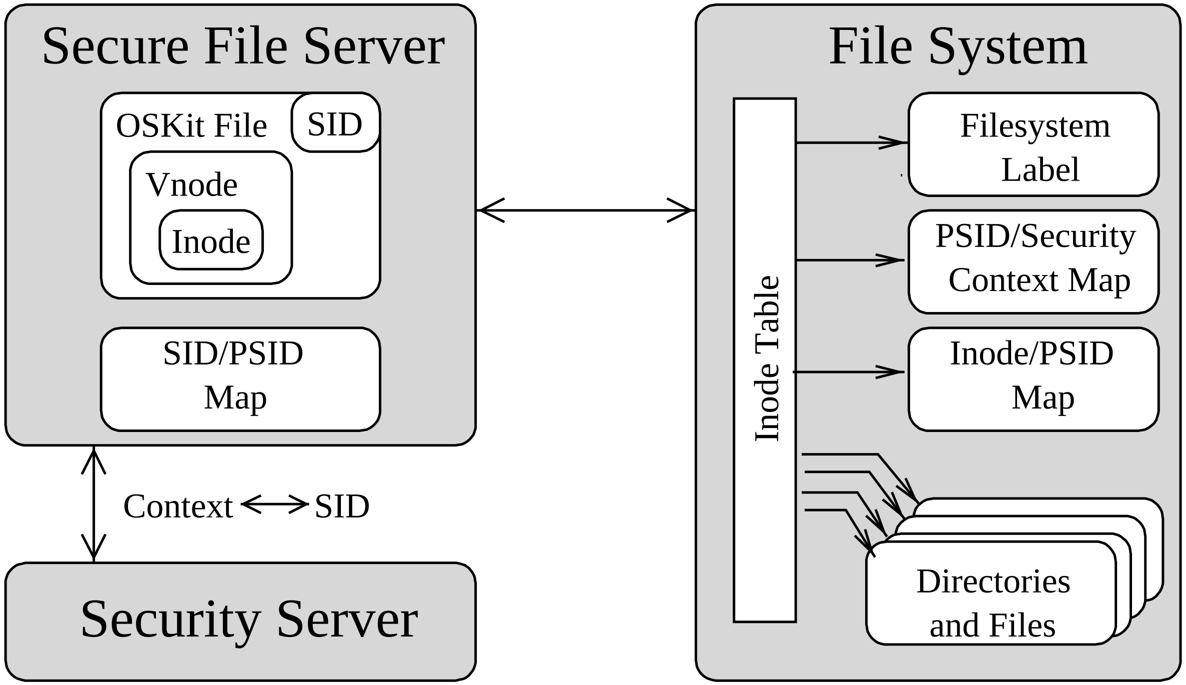
Revocation support

- **Decisions may be cached in in AVCs**
- **Decisions may implicitly be cached in migrated permissions**
  - Unix file descriptors obtained after a file open
  - Memory mapped pages
  - Open sockets/pipes
- **AVC contains hooks for callbacks**
  - After revoking in AVC, AVC makes callbacks to revoke migrated permissions

Revocation protocol



Persistence



- **Track "persistent SIDs" (PSIDs), specific to each file system**

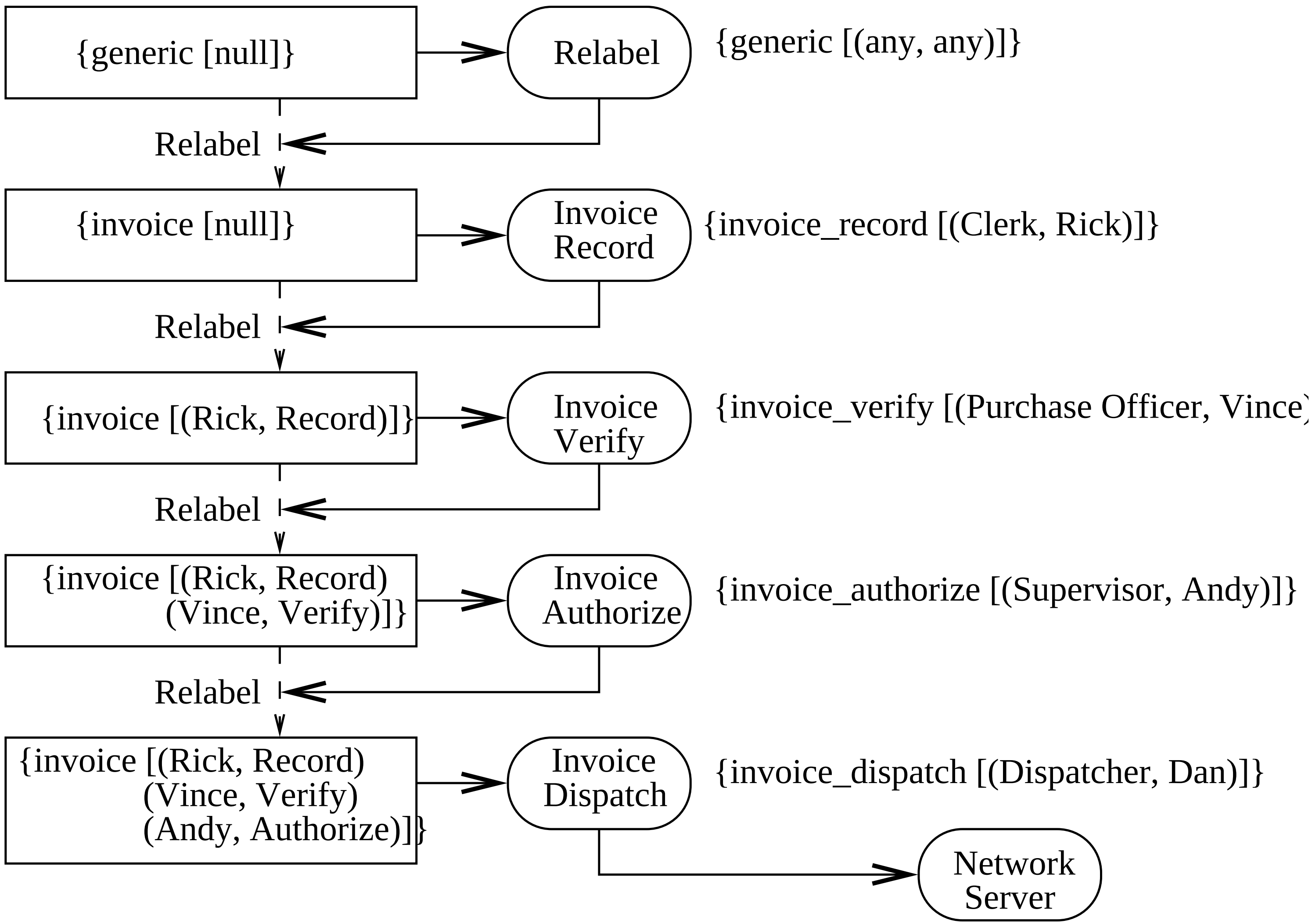
Transitioning SIDs

- **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, one 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

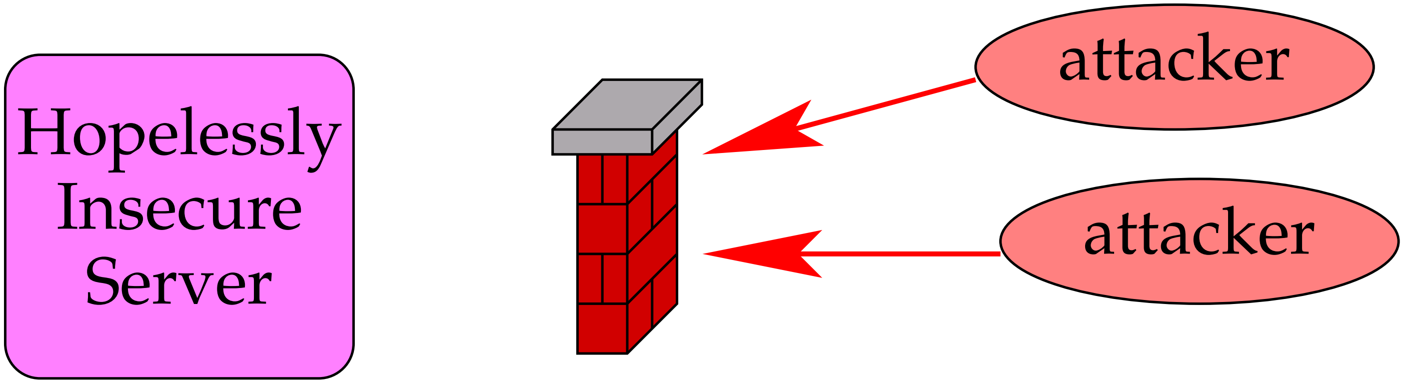
4: Let insmod inherit file descriptors from sysadm

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

6: Let insmod signal sysadm with SIGCHLD when done

Confining code with legacy OSes

- **Often want to confine code on legacy OSes**
- **Analogy: Firewalls**



  - Your machine runs hopelessly insecure software
  - Can't fix it—no source or too complicated
  - *Can* reason about network traffic
- **Similarly block unrusted code within a machine**
  - By limiting what it can interact with

Using chroot

- `chroot (char *dir)` **"changes root directory"**
  - Kernel stores root directory of each process
  - File name "`/`" now refers to `dir`
  - Accessing "`..`" in `dir` now returns `dir`
- **Need root privs to call chroot**
  - But subsequently can drop privileges
- **"Chrooted process" can't affect system outside of** `dir`
  - Even process still running as root cannot escape chroot

Using chroot

- `chroot (char *dir)` **"changes root directory"**
  - Kernel stores root directory of each process
  - File name "/" now refers to `dir`
  - Accessing "`..`" in `dir` now returns `dir`
- **Need root privs to call chroot**
  - But subsequently can drop privileges
- **"Chrooted process" can't affect system outside of** `dir`
  - Even process still running as root cannot escape chroot
- **Wrong: Many ways to create damage outside of** `dir`

Escaping chroot

- **Re-chroot to a lower directory, then chroot ..**
  - Each process has one root directory, so chrooting to a new directory can put you above your new root
- **Create devices that let you access raw disk**
- **Send signals to or ptrace non-chrooted processes**
- **Create setuid program for non-chrooted proc. to run**
- **Bind privileged ports, mess with clock, reboot, etc.**
- **Problem: chroot was not originally intended for security**
  - FreeBSD jail, Linux vserver have tried to address problems

System call interposition

- **Why not use *ptrace* or other debugging facilities to control untrusted programs?**
- **Almost any "damage" must result from system call**
  - delete files → unlink
  - overwrite files → open/write
  - attack over network → socket/bind/connect/send/recv
  - leak private data → open/read/socket/connect/write …
- **So enforce policy by allowing/disallowing each syscall**
  - Theoretically much more fine-grained than chroot
  - Plus don't need to be root to do it
- **Q: Why is this not a panacea?**

Limitations of syscall interposition

- **Hard to know exact implications of a system call**
  - Too much context not available outside of kernel (e.g., what's does this file descriptor number mean?)
  - Context-dependent (e.g., `/proc/self/cwd`)
- **Indirect paths to resources**
  - File descriptor passing, core dumps, "unhelpful processes"
- **Race conditions**
  - Remember difficulty of eliminating TOCCTOU bugs?
  - Now imagine malicious application deliberately doing this
  - Symlinks, directory renames (so "`..`" changes), ...

Sandboxing code

- **What about protecting code *within* an application?**
- **Often security ends up restricting functionality**
  - Take insecure system, add restrictions,
  - Hope result is more secure
- **Sometimes can actually *enhance* functionality**
  - What if you could safely use "unsafe" code?
  - Could allow previously impractical enhancements

Uses of unsafe code

- **Extensible applications**
  - E.g., browser, photoshop, etc., plug-ins
  - Wouldn’t it be nice if they couldn’t crash application?
- **Saving kernel/user crossings**
  - Packet filters (e.g., bpf for tcpdump)
  - Applications-specific virtual memory management
  - Active messages (application-specific msg. handlers)
- **Could just run in separate process, but…**

Cross-address-space calls expensive

- **System call overhead much higher than procedure**
  - Requires trapping into the kernel
  - Often requires draining the processor pipeline
- **Switching address spaces increasingly expensive**
  - On some architectures requires flushing the TLB
  - Increases cache pressure
  - Cache/TLB miss service times increasingly expensive compared to faster and faster cycle times
- **Kernel must copy arguments back and forth between address spaces**
  - Change page mappings, etc.

Sandboxing also gives *control*

- **Example: Exokernel OS**
  - Goal: Let applications manage resources as much as possible
- **Don't hardcode TCP/IP or other protocols**
- **Instead, download packet filters into kernel**
  - Express which packets an application wants to see
  - By downloading filters, kernel can ensure no conflicts
  - Also ensures apps don't leak information on other's pkts
- **DPF (dynamic packet filter) created code on the fly**

Exokernel disk abstraction

- **How to multiplex disk with untrusted apps?**
  - Need metadata–i.e., for a file, what blocks to use
  - Don't want to hard-code metadata formats
- **Solution: UDFs (untrusted deterministic functions)**
  - Download metadata interpretation code
  - UDF takes metadata, outputs list of blocks
  - Kernel checks metadata updates by output of UDF
  - Downloading ensures that UDFs are deterministic
- **Determinism useful in less esoteric settings**
  - Ensure code you sign will keep behaving same way

Challenges of untrusted code

- ***Fault domain*—logically separate portion of A.S.**
  - Each untrusted component runs in its own fault domain
- **Prevent FDs from trashing each other's memory**
- **Prevent FDs from jumping to arbitrary locations**
- **Prevent code from accessing operating system**
  - Otherwise, e.g., could execute arbitrary programs
- **Other possible goals:**
  - Prevent FDs from *reading* each other's memory
  - Prevent infinite loops
  - Bound physical memory utilization

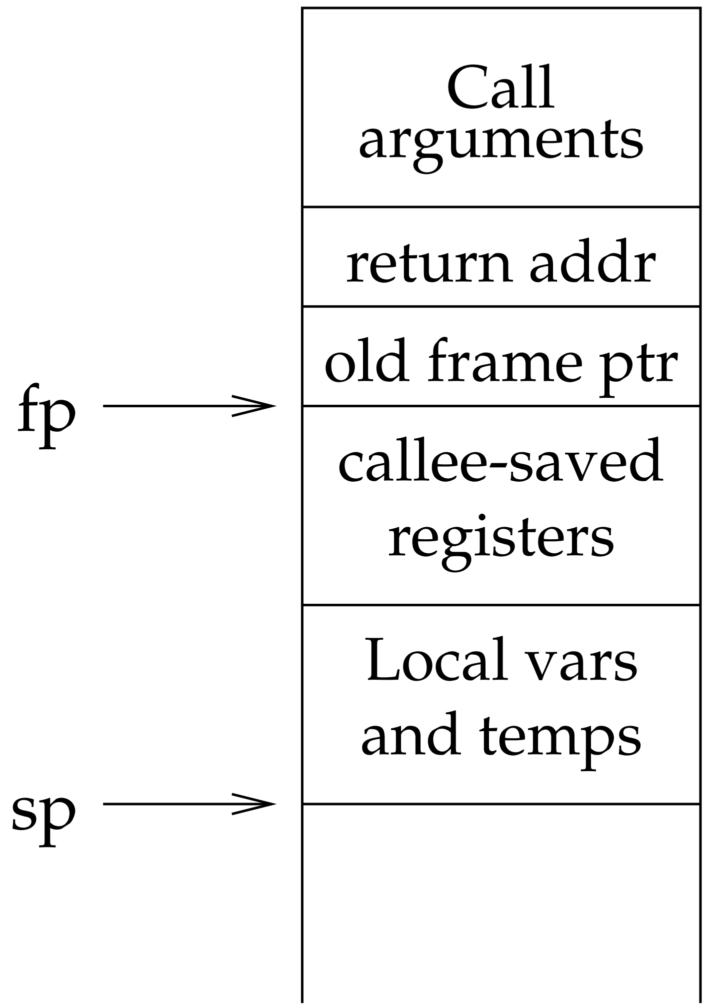
Software fault isolation

- **Goal: Make fault isolation cheap enough that developers can ignore performance impact**
- **General approach:**
  - Modify compiler to generate "safe" code
  - Verifier can check code is safe before loading/running it
- **Alternate approach: *binary patching***
  - Rewrite unsafe binaries to be safe
  - Doesn't tie system to one compiler/language
  - Unfortunately, binary rewriting hard to do

Review: Typical RISC instruction sets

- **Have 31 general-purpose integer registers**
  - Instruction set treats all registers identically
  - Convention dictates certain uses (e.g., stack ptr, ...)
  - Across calls, some regs caller-saved, some callee-
  - All ALU operations occur on registers
- **Memory accessed w. load/store instructions only**
  - `LD rd, offset(rp)`   `ST rs, offset(rp)`
- **All instructions 32 bits (and must be aligned)**
  - Makes it easy to check each instruction in code

MIPS calling conventions



- **Like x86; should be very familiar from project 1**

SFI implementation

- **Divide virtual address space into segments**
  - All addresses in a segment share same prefix
  - Not all virtual addresses in segment need to be valid
- **Each fault domain has two segments**
  - Code segment and separate data segment
  - Q: Why not use one combined segment?
- **Go over code identifying *unsafe instructions***
  - Any store or jump that can't be statically verified
  - PC-relative branches OK, stores to static vars often OK
  - Insert checking code before instructions that are not OK

Segment matching

- **Use dedicated registers to hold addresses**
- **Always check segment ID of target address of store**

```
dedicated-reg <= target address
scratch-reg <= (dedicated-reg >> shift-reg)
compare scratch-reg segment-reg
trap if not equal
store value dedicated-reg
```

- **Adds 4 instructions to every store**
- **Q: Why use dedicated register for store address?**

Address sandboxing

- **Segment matching good for debugging, but slow**
- **Instead of checking segment IDs, can just set them:**

```
dedicated-reg <= target-reg & and-mask-reg
dedicated-reg <= dedicated-reg | segment-reg
store value dedicated-reg
```

- **Now requires only 2 extra instructions per store**
- **Again, dedicated register prevents harm if code jumps to middle of store sequence**

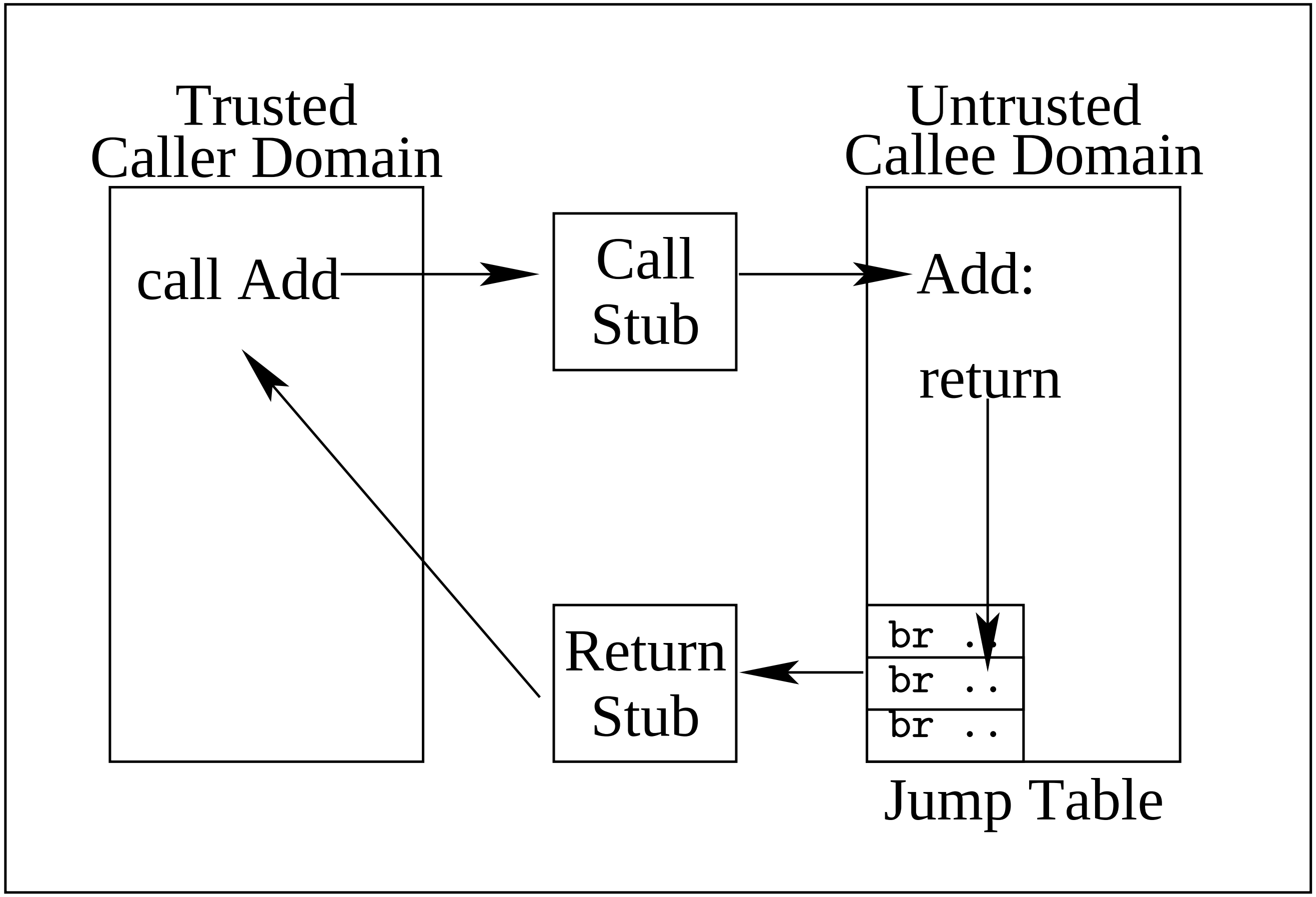
Optimizations

- **Traditional compiler optimizations**
  - E.g., might move sandboxing out of a loop
- **Guard zones at each end of data segment**
  - Load/store instructions tage address reg. & offset
  - Unmapped zones larger than maximum ld/st offset
  - Means only register need be sandboxed, not full addr
  - Sandbox the stack pointer only when it is set
  - Avoid sandboxing SP if adjusted by small amount and used before next control transfer

Cross-domain calls

- ***Jump table*** **contains allowed exit points from FD**
  - Each jump table entry is a control transfer instruction (address hard-coded into instruction, so no register use)
  - Explicitly enumerates allowed calls between each 2 FDs
  - Jump table trusted, and in read-only code segment
- **Jump table entries transfer control to** ***stubs***
  - Must save any caller-saved registers (can't trust target)
  - Copy arguments of call from caller's segment to target's

Fig 4



- **Q: Why not embed stubs directly in segment?**

Sharing memory accross domains

- **Read sharing is not a problem**
- **If we need write sharing, use VM hardware**
  - Just map the same page into multiple segments in same A.S.
- **Slight trickiness: pointer comparisons**
  - Don't compare aliased ptrs w. different segment IDs
  - Give shared region canonical address
  - Fix pointer for write access (automatic w. sandboxing)

Limitations of SFI

- **Performance**
  - Usually good, but slowdown bad for packet filters, ...
- **Harder to implement on some architectures**
  - E.g., x86 has variable-length, unaligned instructions (would have to do more expensive checks on jumps)
  - x86 has fewer registers (can't dedicate 5 of them)
  - Most x86 instructions affect memory (more sandboxing)
- **Compiler and verifier tightly bound**
  - Once verifier deployed, might be hard to make further improvements in compiler