Administrivia

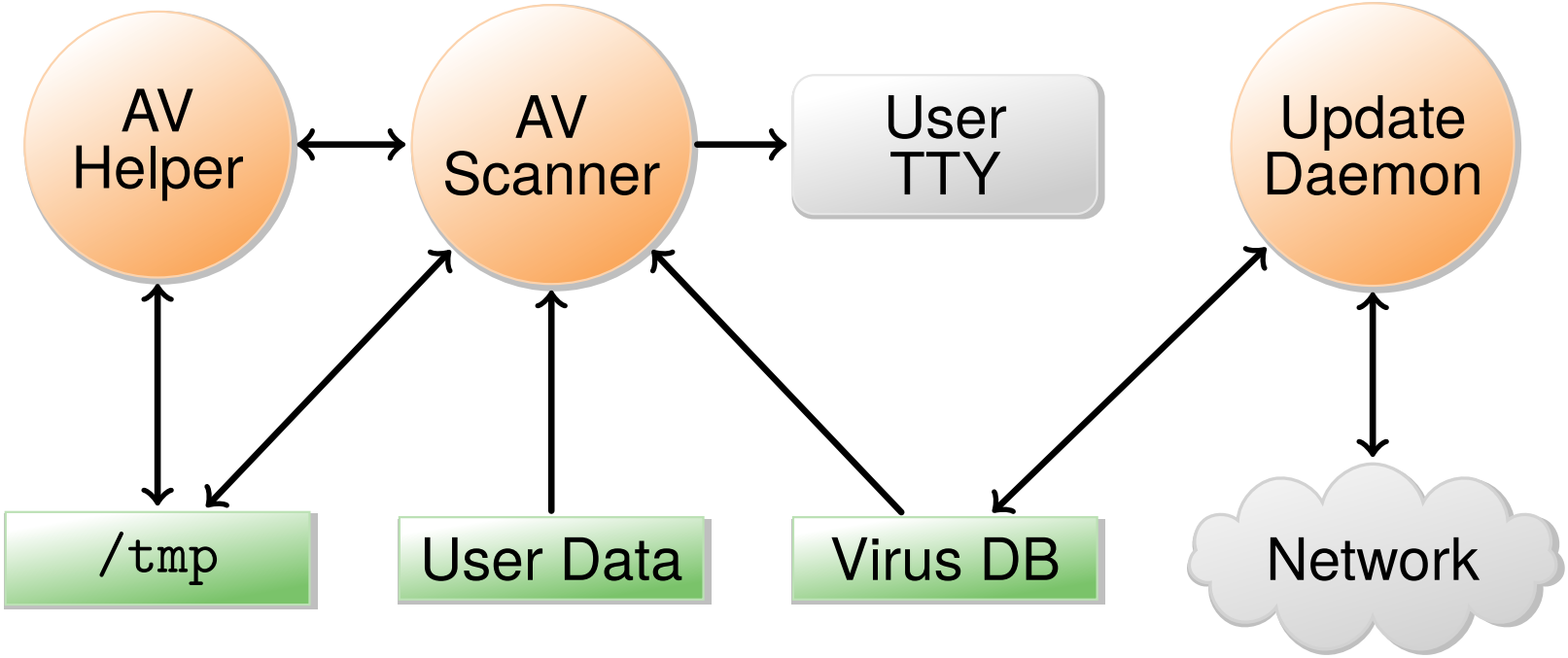
- Last project due Friday noon
- Final Exam
  - Wednesday December 9, 3:30pm-6:30pm in Gates B-03
  - Open notes, closed textbook, just like midterm
  - Covers material from all 19 lectures (including today)
- SCPD students please take in person if possible
  - Otherwise, give SCPD distribution your monitor's email
  - In case of emergency (don't get exam), mail `cs140-staff`
- I'll have extra office hours next week, check web site
  - I also plan to be around most of Monday/Tuesday, so feel free to drop by if you have questions for exam
- Televised final review session Friday 12:30pm

Outline

Software vulnerabilities are everywhere

- High-profile software (nginx, Synamtec)
- But also web applications (Paymaxx)
  - One-off designs receive little outside scrutiny
  - See a wide range of programmer abilities (unlike core components like kernels)
- Also embedded systems (fridge, TV) [BBC]
- "Internet of things" $\overset{?}{\approx}$ remote exploit of things
- Fewer and fewer settings where security doesn't matter

Recall virus scanner



- Even Symantec could not get it right (from lecture 17)
- AntiVirus product remote exploit [Symantec]
  - Software deployed on 200,000,000 systems
- What hope is there for non-security companies?

The only solution

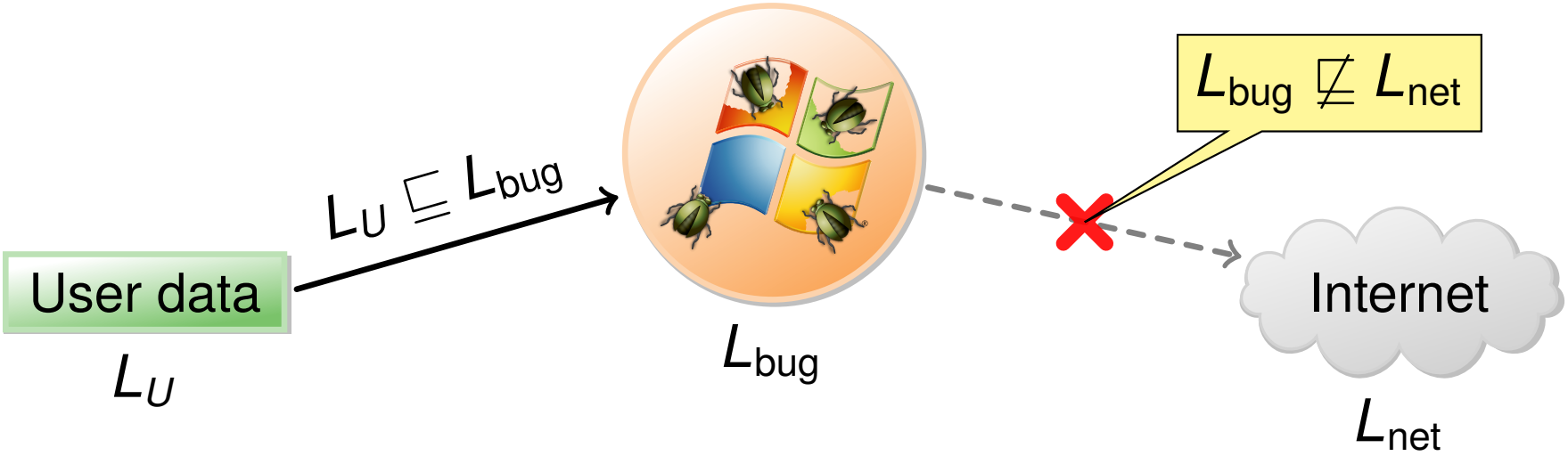
The median programmer must build secure systems.

- Sadly, far from a solved problem
- I will discuss research that brought us closer to the goal
- Deian Stefan will provide an industrial perspective

Steps towards the goal

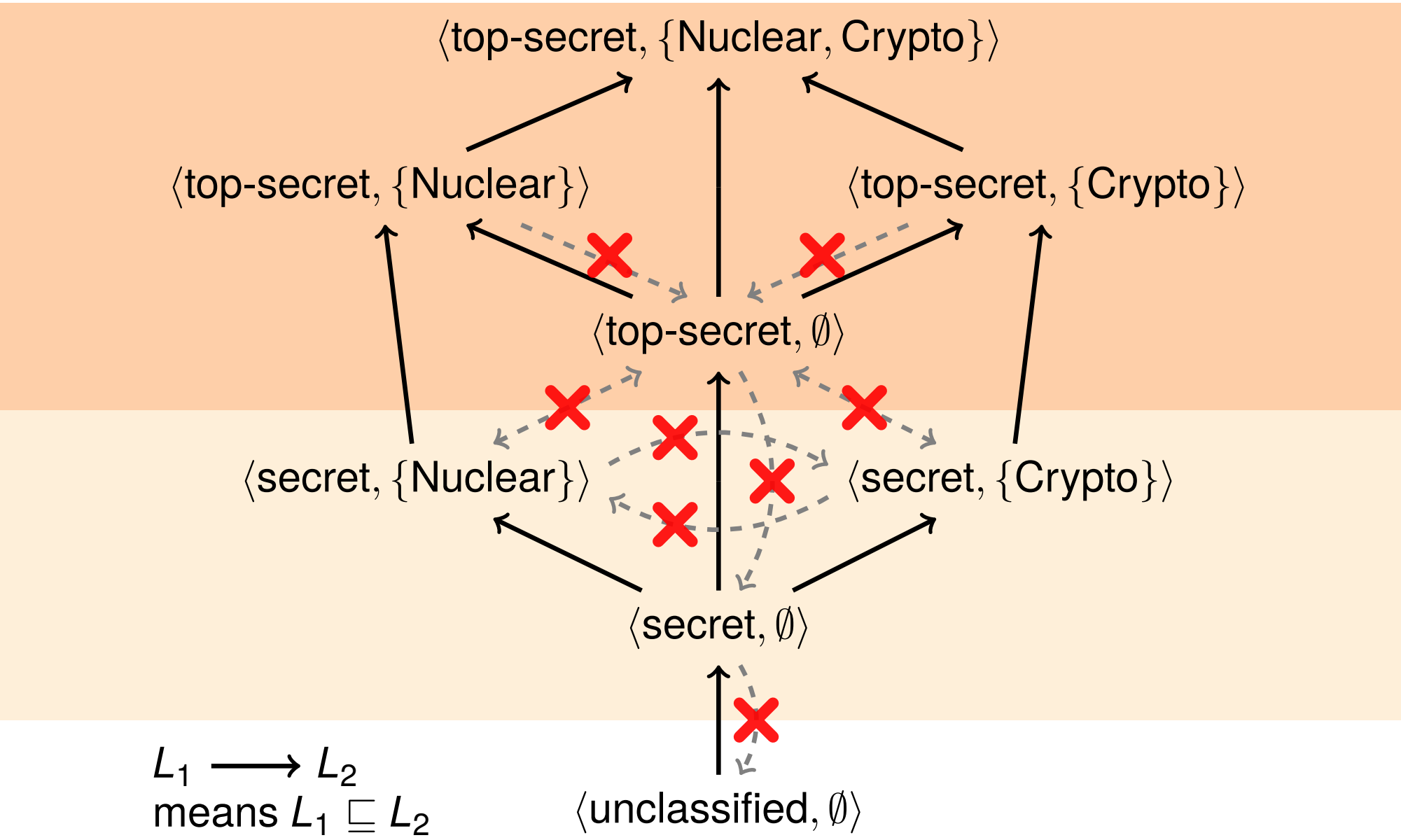
1. Allow experts to incorporate third-party code into secure systems
   - Achievable if you are willing to use a new operating system (HiStar)
   - Brings compatibility, deployment issues
2. Allow experts to manage non-experts building secure systems
   - Possible if you apply OS ideas at language level
   - Research required teaching programmers a new language (Haskell)
   - Much easier than adopting a new OS
3. Allow *anyone* to hire non-experts to build secure systems
   - This is *the* big open problem
   - Deian will talk about how this may be achievable leveraging JavaScript instead of Haskell

Recall labels from lecture 17



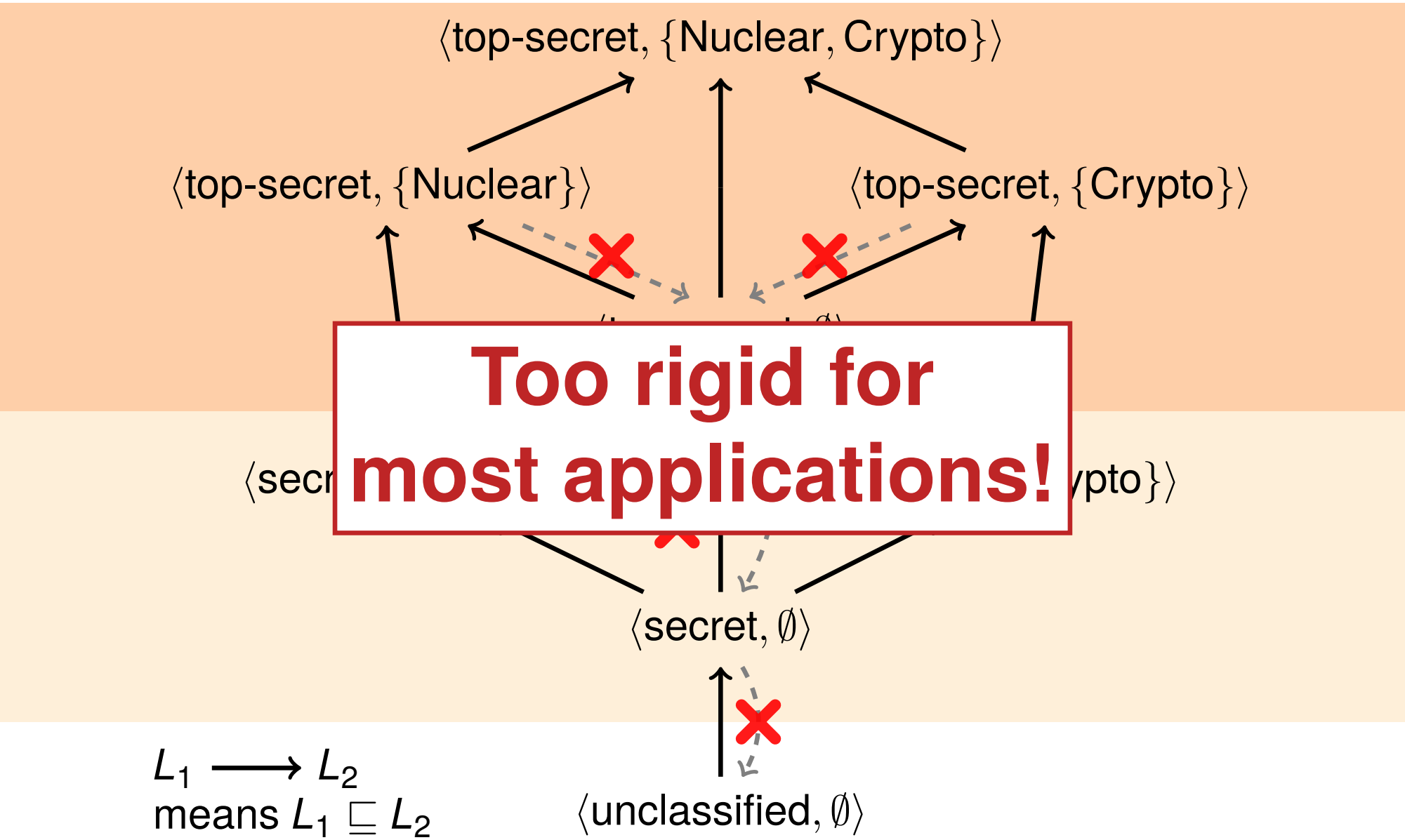
- Transitivity of labels makes it easier to reason about data security
  - Don't need to understand what code does to secure data
  - User data won't flow to network no matter what
- But how do you set the labels?
- If code uses labels to protect its data from other code …will the first code lose network access?

Recall standard military lattice



$L_1 \longrightarrow L_2$
means $L_1 \sqsubseteq L_2$

Recall standard military lattice



Too rigid for most applications!

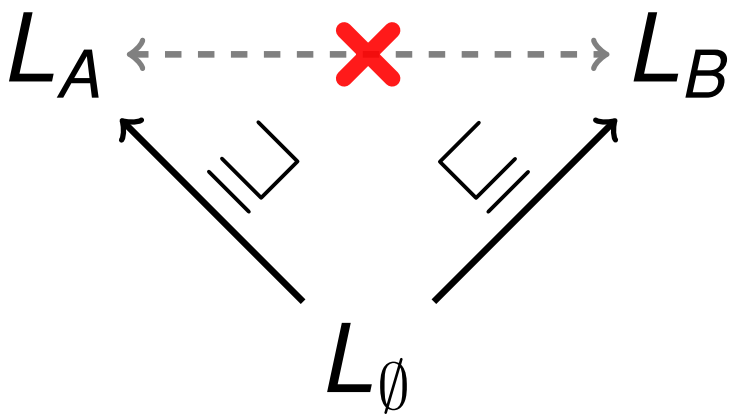
$L_1 \longrightarrow L_2$
means $L_1 \sqsubseteq L_2$

Dynamic labels more flexible

$$L_A \sqsupseteq L_\emptyset$$

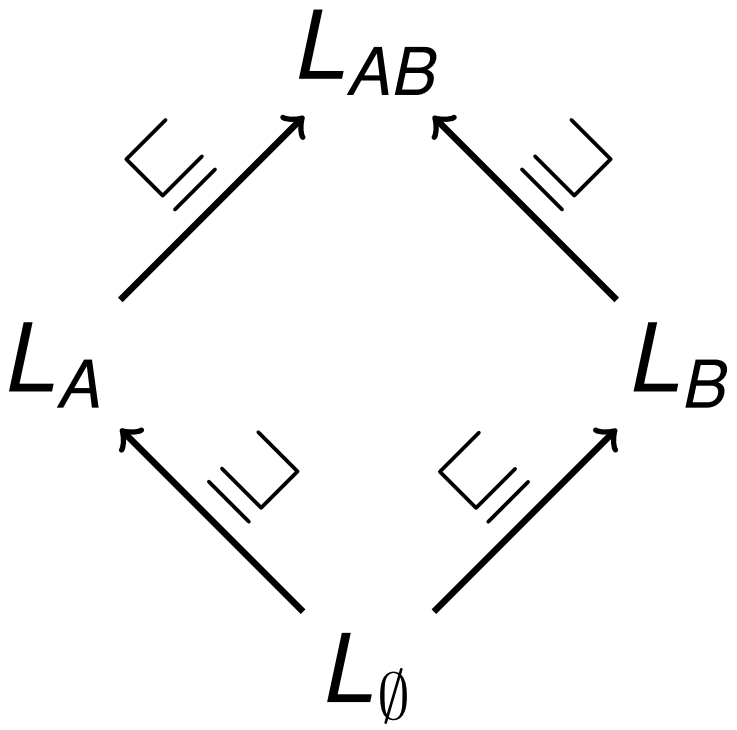
- E.g., label public data $L_\emptyset$, label user A's private data L_A

Dynamic labels more flexible



- E.g., label public data $L_\emptyset$, label user A's private data L_A
- New user B joins web site... label his data L_B
  - A and B cannot read each other's private data

Dynamic labels more flexible

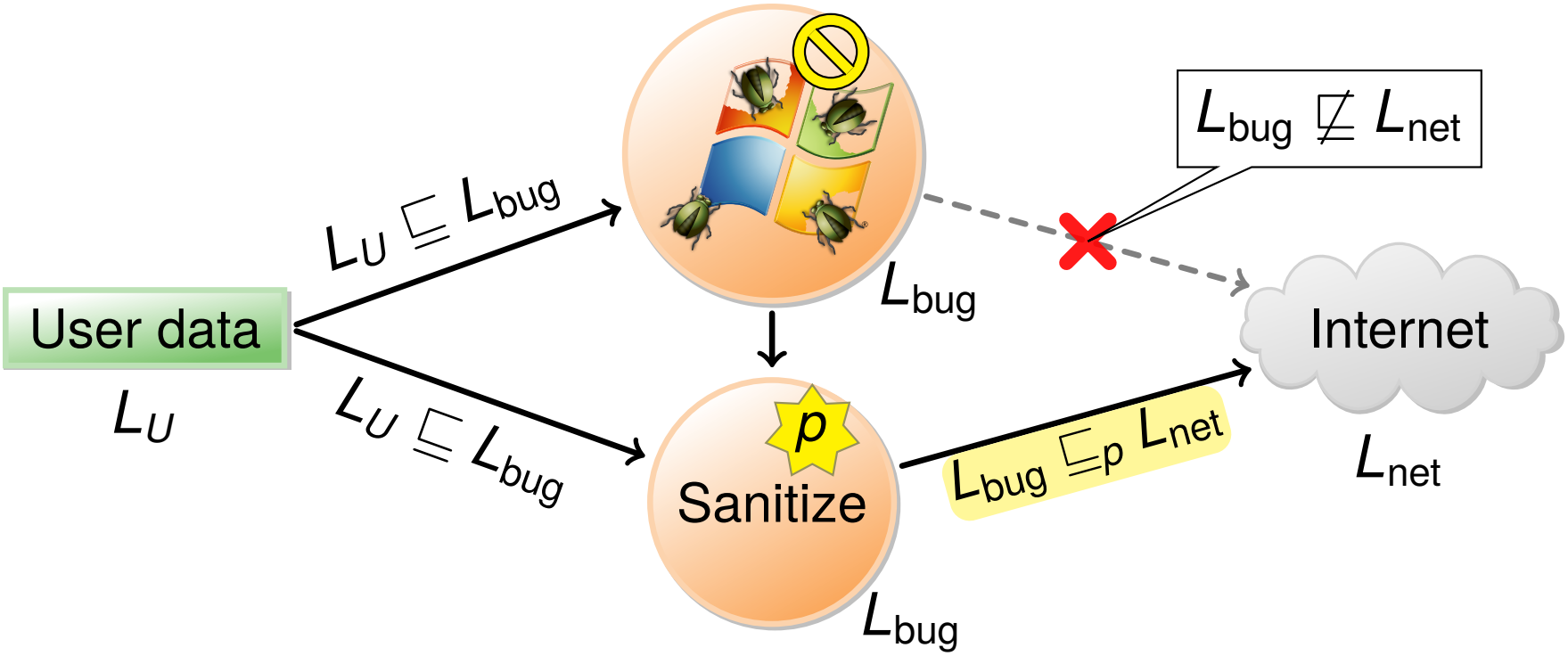


- E.g., label public data $L_\emptyset$, label user A's private data L_A
- New user B joins web site... label his data L_B
  - A and B cannot read each other's private data
- Mix A's and B's private data? Need new label $L_{AB} = L_A \sqcup L_B$

Implications of dynamic labels

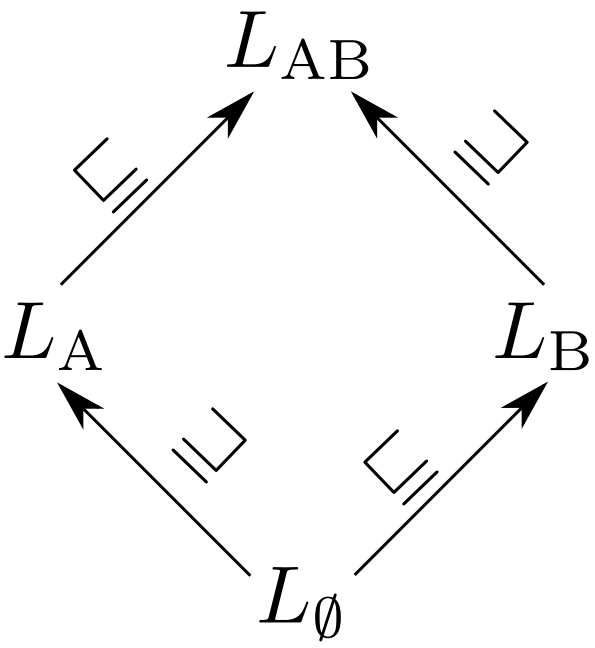
- Need many more labels than traditional systems
  - E.g., Orange book required 16+ hierarchical classifications and 64+ non-hierarchical categories
  - Effectively need an unbounded number of categories (e.g., 2^{64})
  - Each visitor of a web site should likely have unique category
  - Software may freely allocate single-use throw-away categories
- Enforcement must be dynamic
  - At compile time might not know all code (e.g., OS loads new program, browser loading Javascript, database with stored procedures)
  - At compile time, certainly won't what labels will be created
- Need a principled way to bypass information flow restrictions
  - Otherwise each new label would only further restrict communication—who could ever write to network?

Decentralized information flow control [Myers]



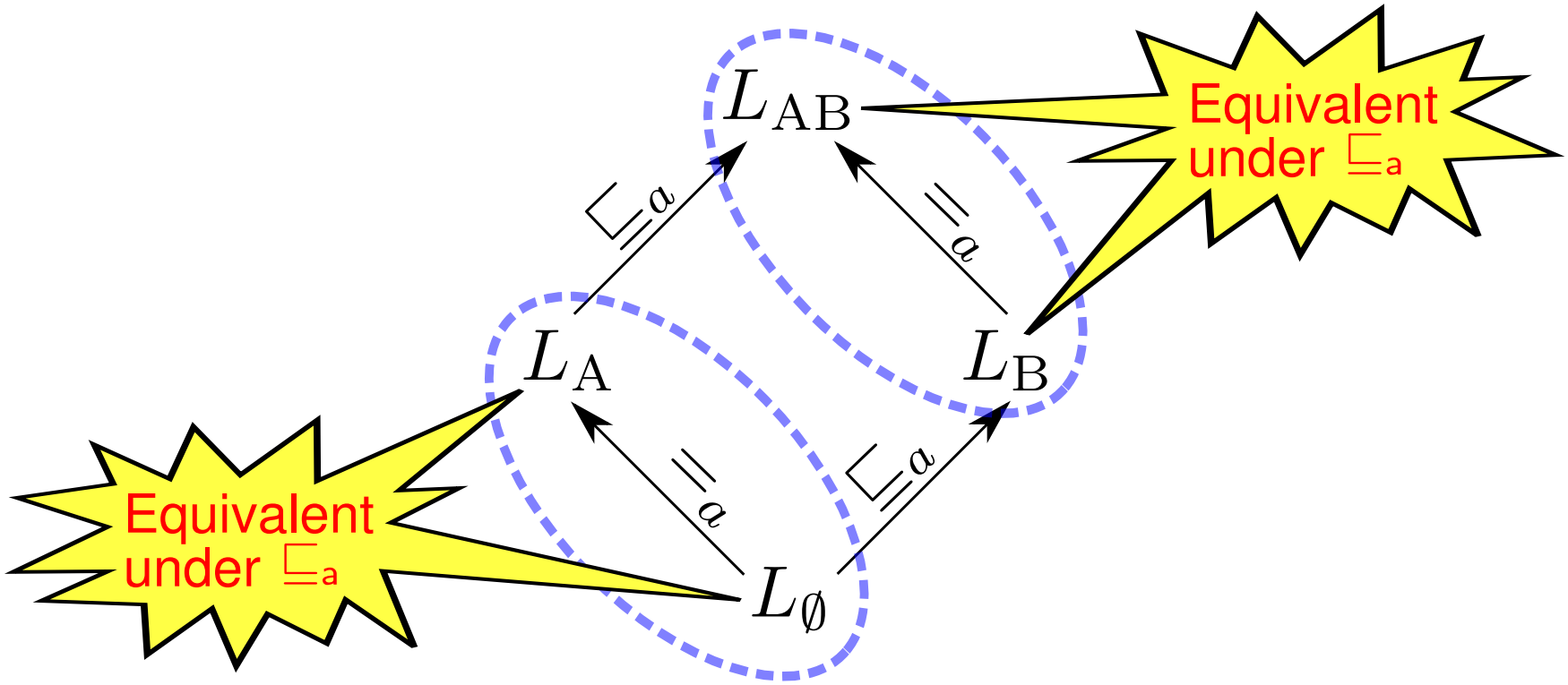
- Different processes have different *declassification privileges*
- E.g, privilege p can bypass restrictions of L_{bug} (represented)
- Exercising p loosens label requirements to a pre-order, $\sqsubseteq_p$
  - $L_F \sqsubseteq_p L_{proc}$ to read, and additionally $L_{proc} \sqsubseteq_p L_F$ to write file
- Idea: Set labels so you know who has relevant privs

Example privileges



- Consider again the simple two user lattice
- Let a be user A's privileges, b be user B's privileges
- Users should be able to *declassify* their own private data
  - So clearly $L_A \sqsubseteq_a L_\emptyset$ and $L_B \sqsubseteq_b L_\emptyset$
  - At API level, often creator of label L_A is given privileges a

Example privileges

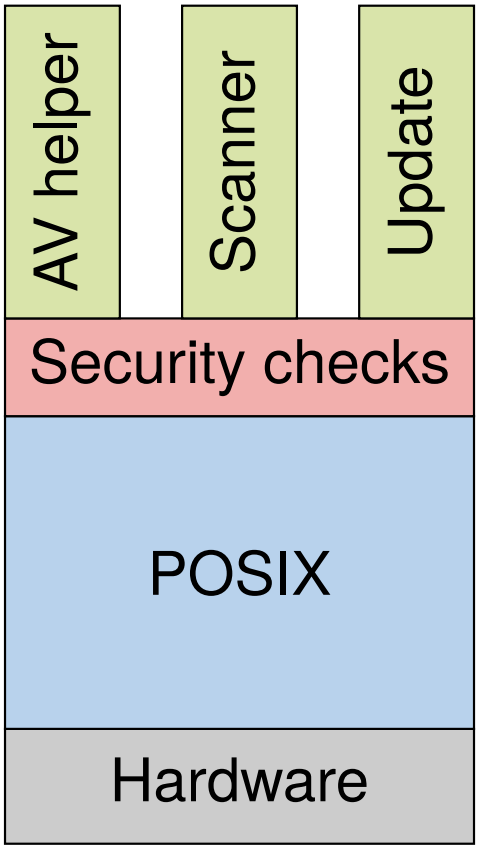


- Consider again the simple two user lattice
- Let *a* be user *A*'s privileges, *b* be user *B*'s privileges
- Users should be able to *declassify* their own private data
- Users should also be able to *partially declassify* data
  - I.e., $L_{AB} \sqsubseteq_a L_B$ and $L_{AB} \sqsubseteq_b L_A$

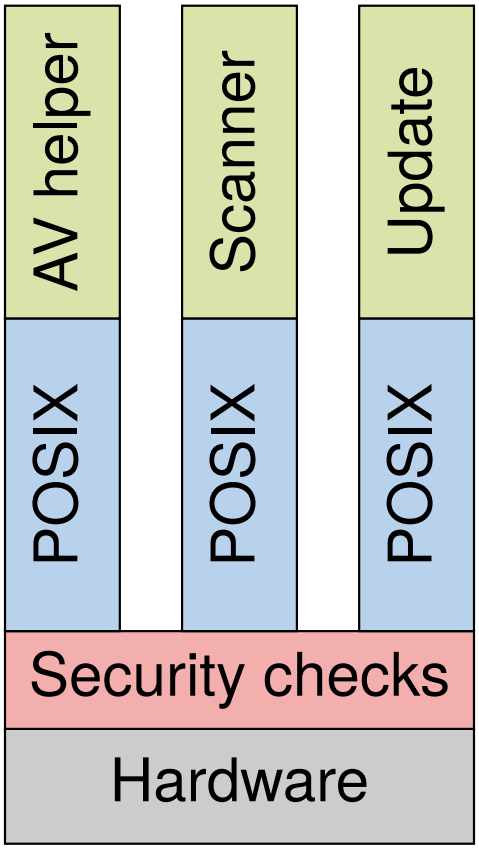
HiStar OS [Zeldovich]

- Clean-slate OS that made all information flow explicit
  - Idea: be a pure label system the way KeyKOS is pure capability
- Key feature: partial declassification privileges
  - All other security features built on partial declassification
- Example: user IDs
  - Each uid corresponds to two privileges, for reading and writing user's files
  - User's login shell receives privileges after authentication
- Example: web security
  - Each web user is associated with unique declassification privileges
  - Ensures Paymaxx-style dump-the-database attacks not possible

HiStar architecture



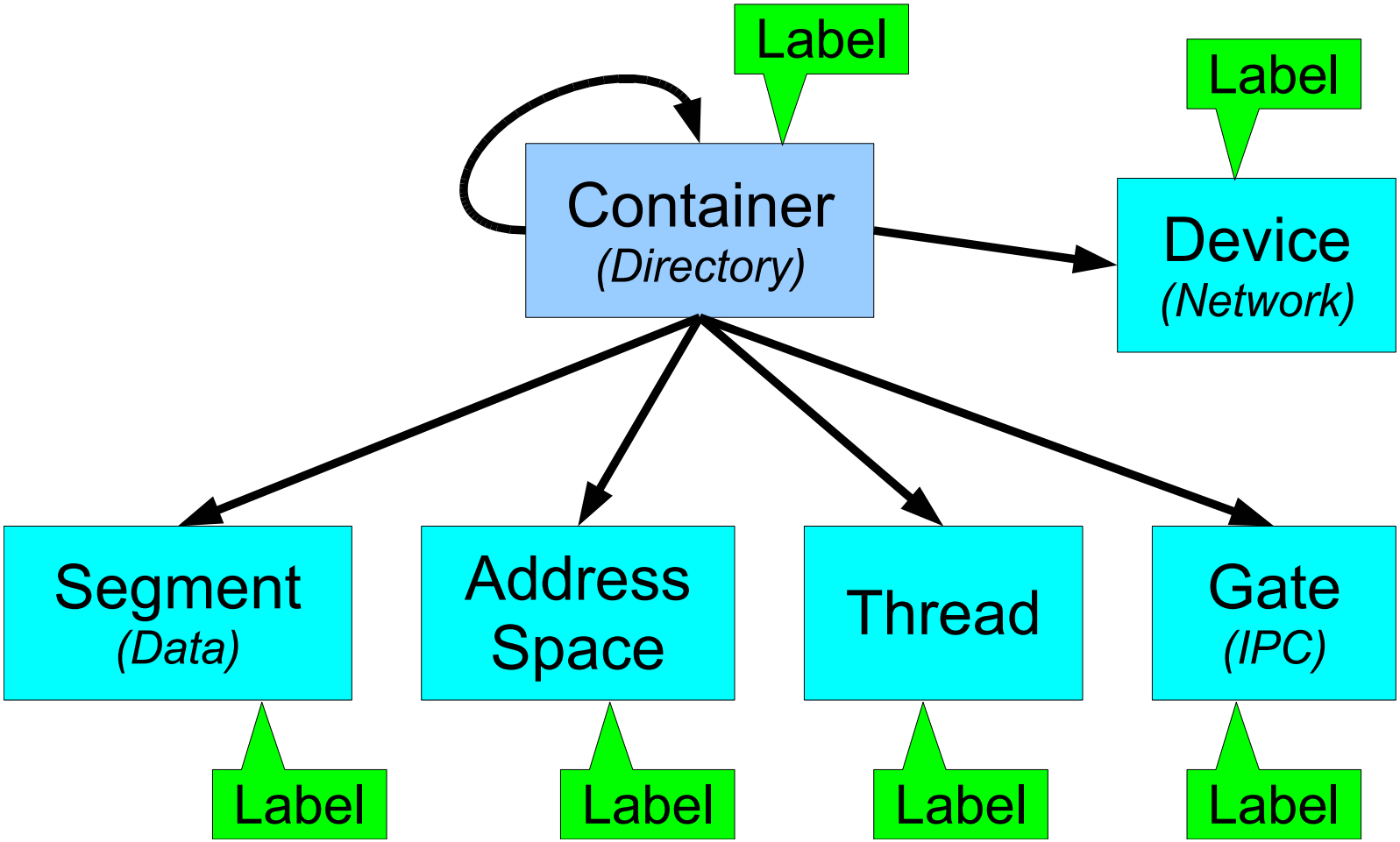
Linux



HiStar

- Kernel provides small number of simple objects
  - Simple enough that information flow is unambiguous
- Layer POSIX API as untrusted library on top of kernel

HiStar kernel objects

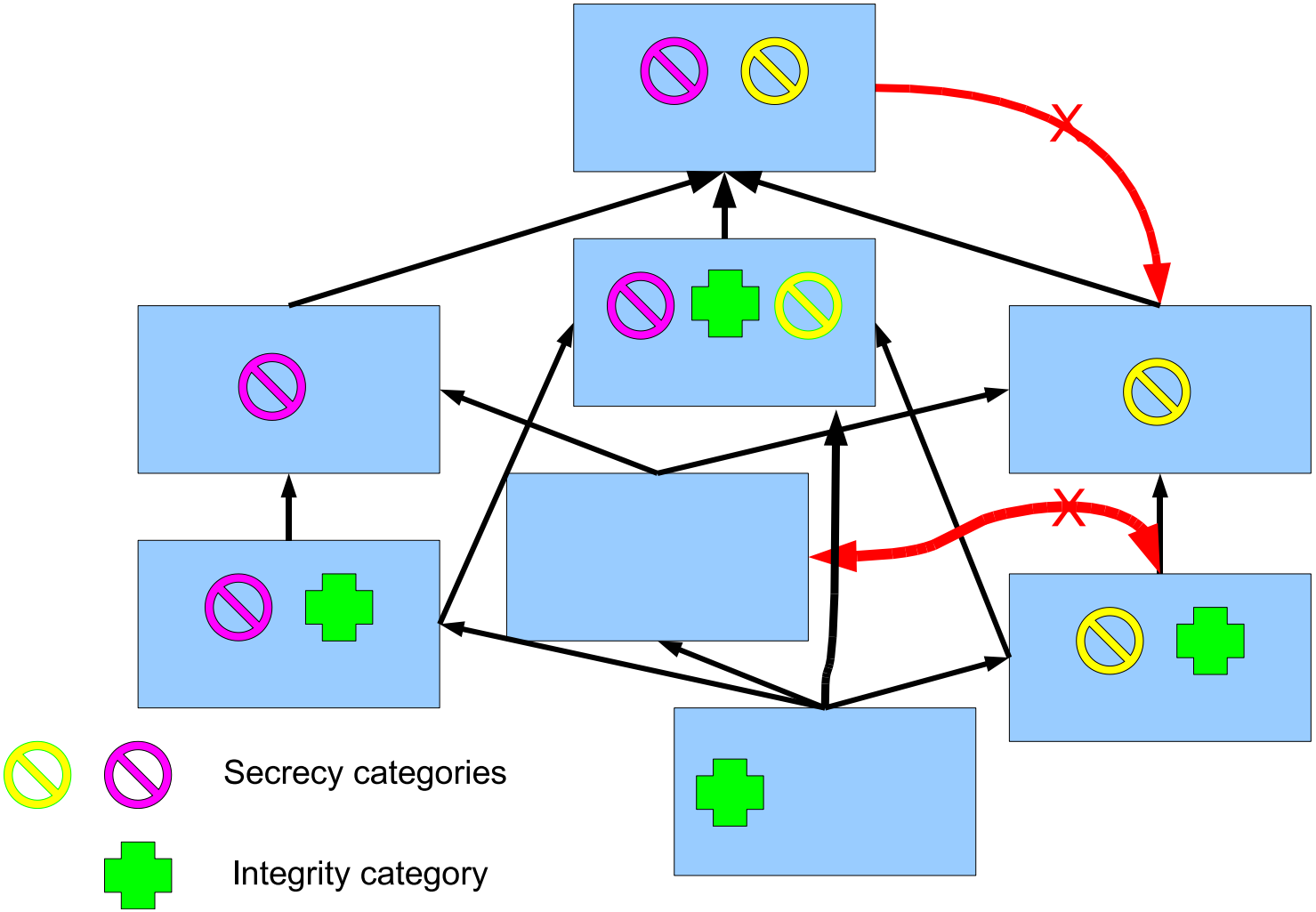


- Only six types of object exposed by kernel interface
- *Labels* specify who can observe/modify each object

What's in a label?

- Labels represent 2 types of information flow concern
  - secrecy – preventing people from observing information
  - integrity – preventing people from modifying information
- Each concern is represented by a *category* of taint
- Secrecy categories shown as
  - Different colors represent different secrecy categories
  - Secrecy category in label $\Rightarrow$ object *tainted* in that category
- Integrity categories shown as
  - Integrity category in label $\Rightarrow$ object *less* tainted in that category
- $L_A \sqsubseteq L_B$ ("L_A can flow to L_B") when:
  - L_A is no more tainted than L_B in every category
  - I.e., L_A has all integrity categories in L_B and L_B has all secrecy categories in L_A

HiStar labels

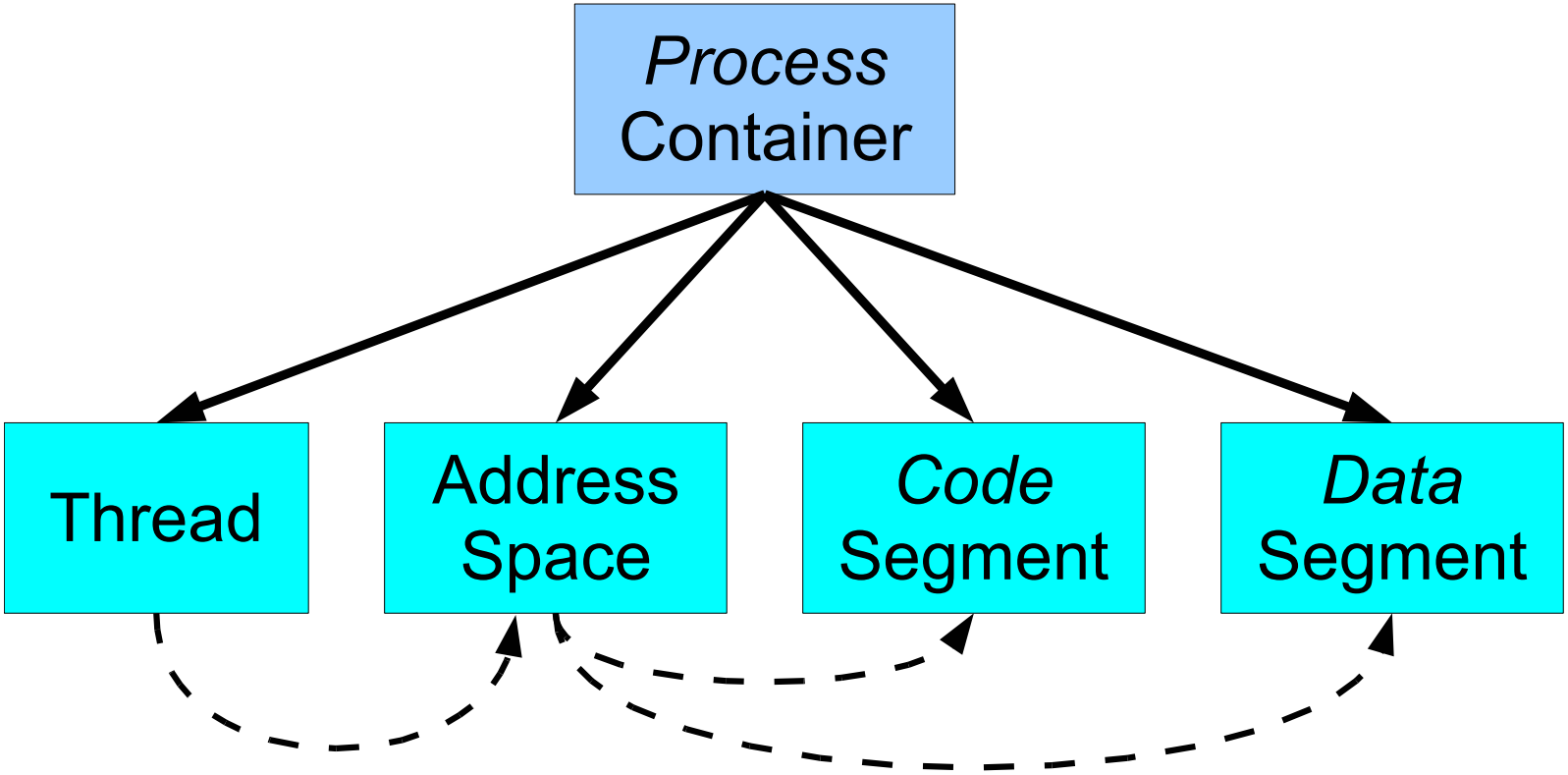


- Can add but not remove secrecy categories
- Can remove but not add integrity categories

Downgrading privileges

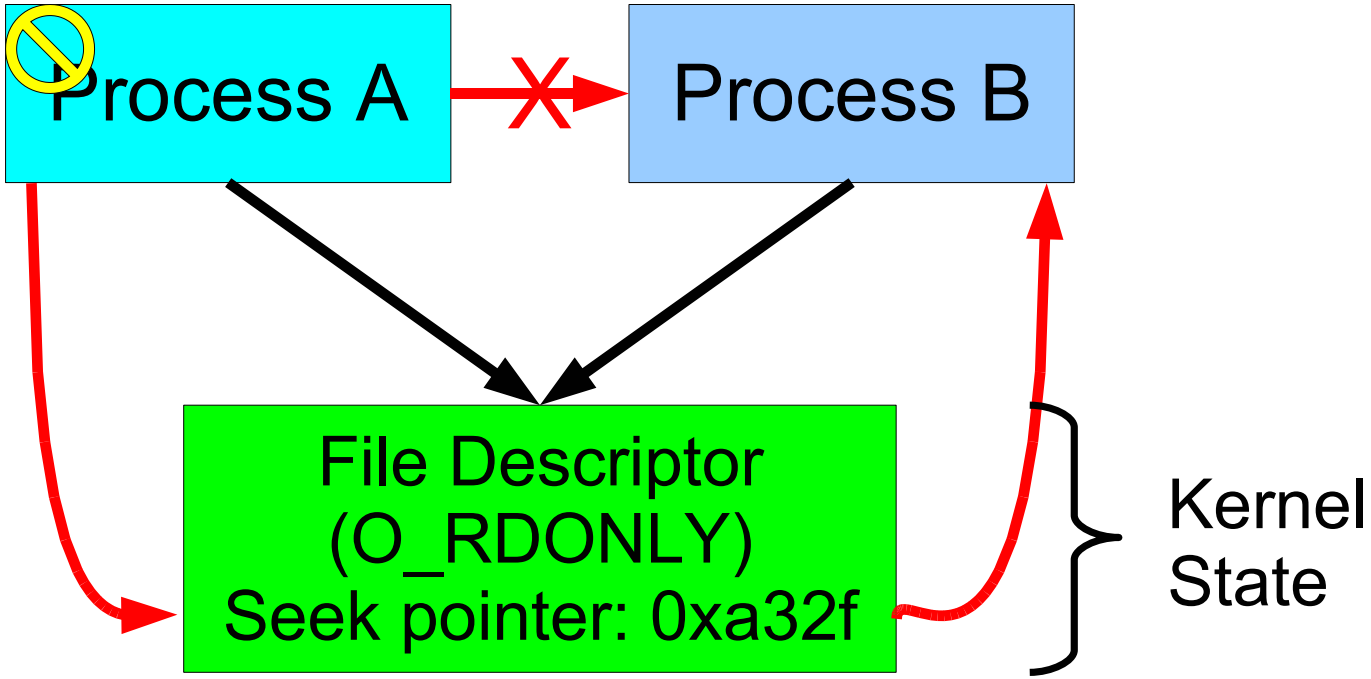
- Downgrading privileges are *decentralized*
  - Represented by per-category stars (★) in threads' labels
  - Means thread can ignore taint in that category (⊘)
- Any thread can create a new category
  - You get the stars for the categories you create
- Can implement Unix UIDs using categories
  - Each UID u corresponds to *two* categories:
    An *integrity* category, u_i and, a *secrecy* category, u_s
  - u's shell gets those stars whenever s/he logs in
  - File with 0644 permissions (`-rw-r-r-`) is labeled w. u_i
  - File with 0600 permissions (`-rw----`) labeled w. both u_s, u_i
- Can implement other policies alongside UIDs

Unix processes



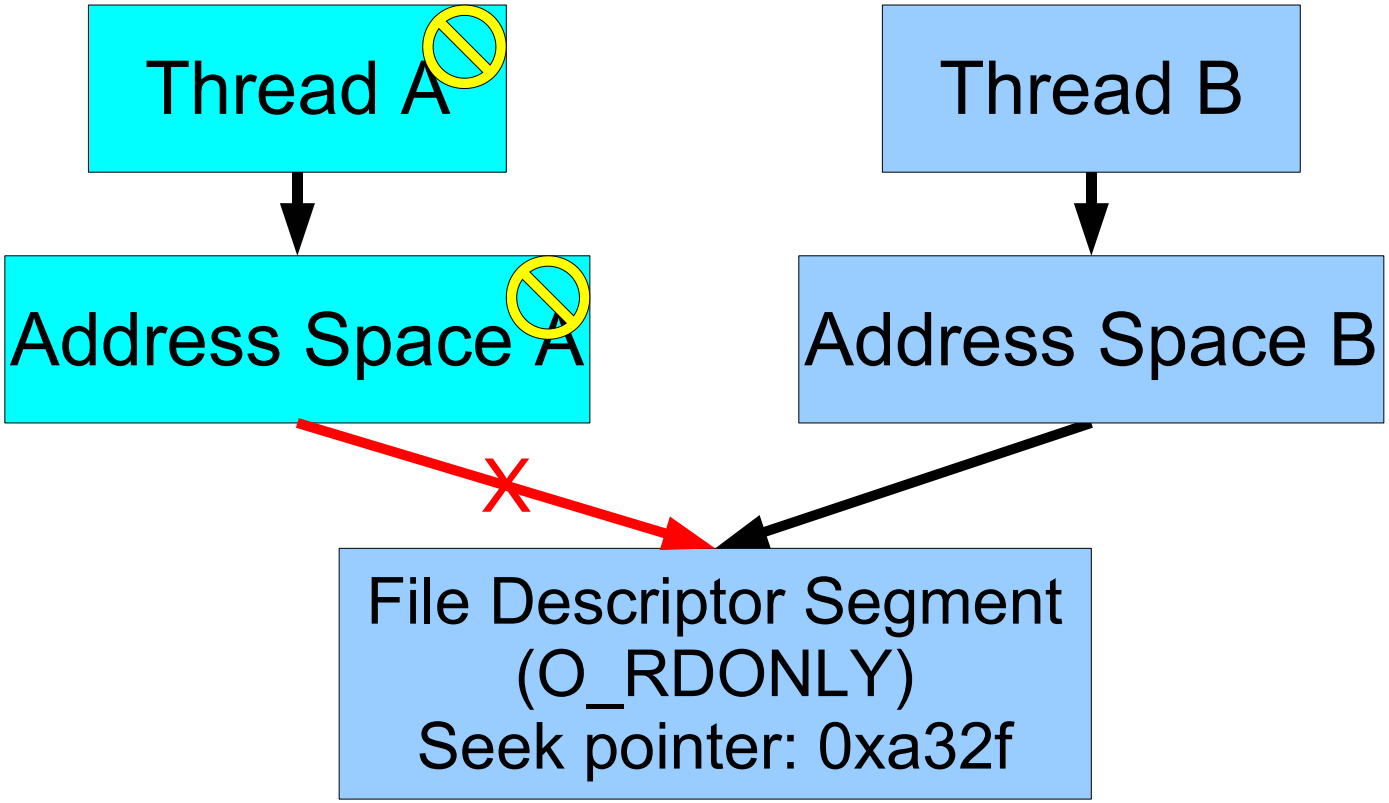
- Unix abstractions built from HiStar objects
- But implemented in untrusted library code
  - E.g., a bug in the code implementing processes won't violate information flow restrictions

Example: File descriptors in Unix



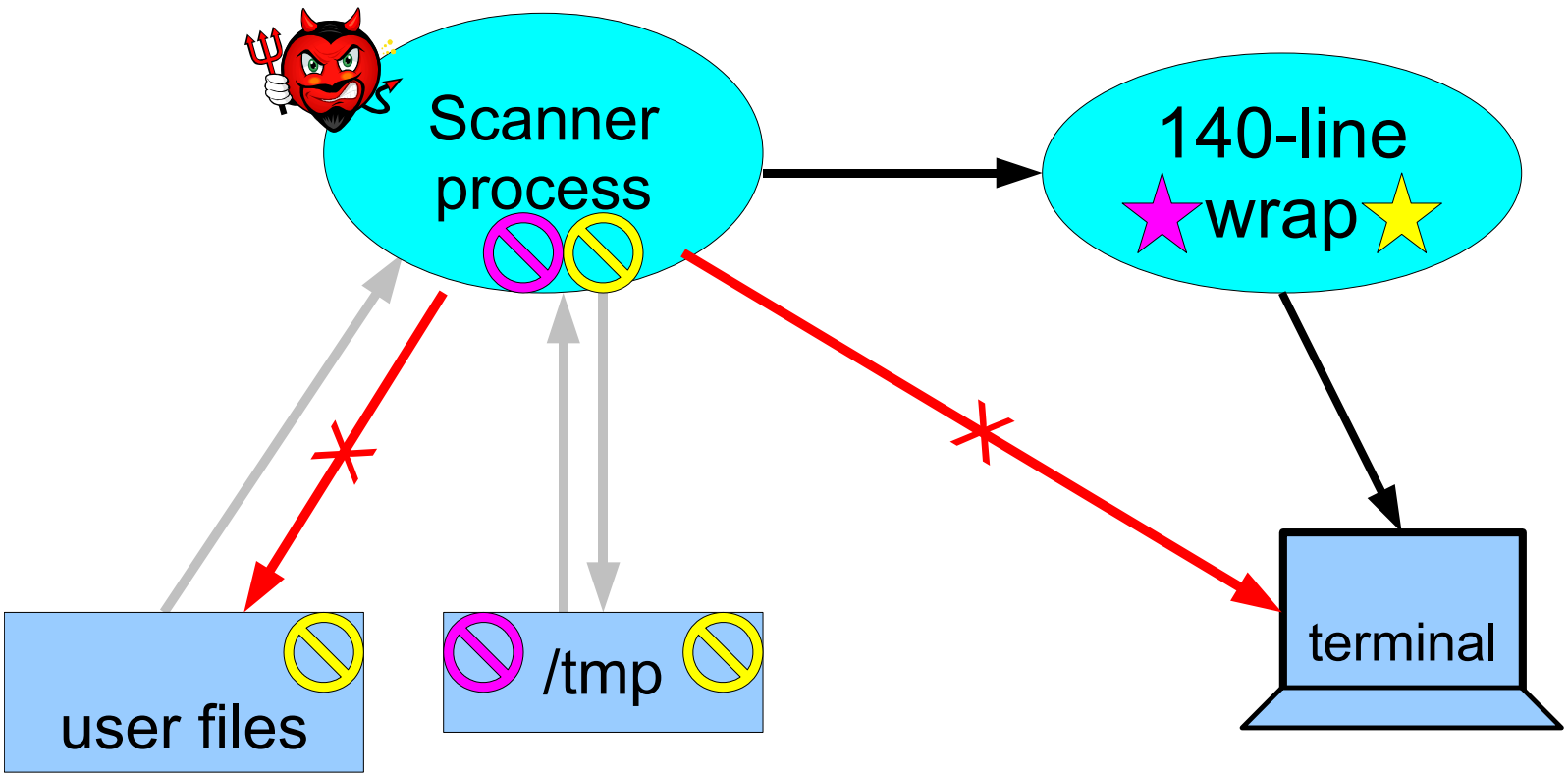
- Suppose *A* can't flow to *B*, but they share a file desc.
  - Restriction represented as ⊘ on *A* but not *B*
- Typical mistake: Consider read-only fd to be okay
  - But even a read-only fd has mutable state, such as offset pointer
  - Hard to catch all such shared state

Example: File descriptors in HiStar



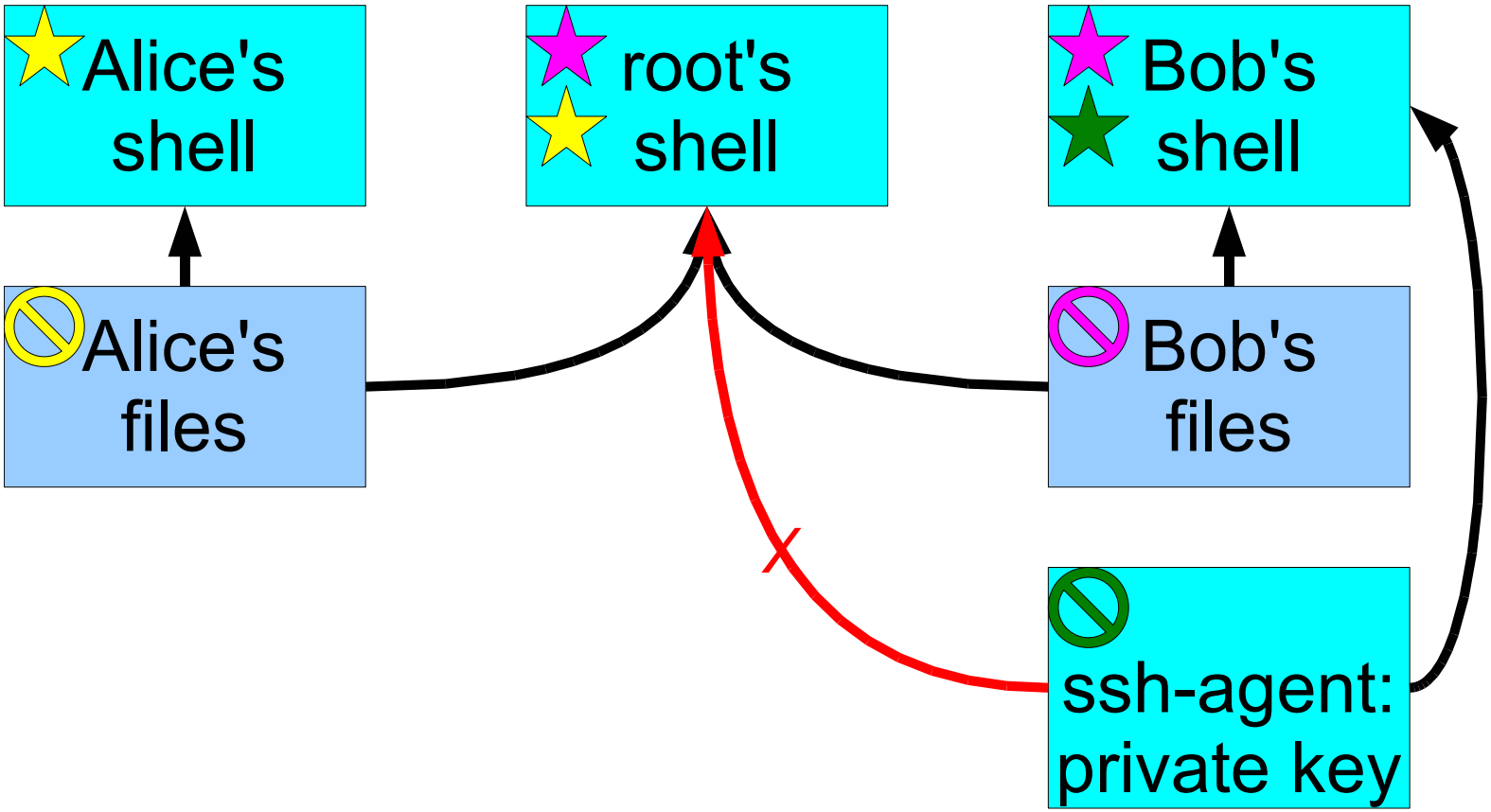
- Implemented in terms of lower-level HiStar objects
  - Seek pointer must be stored in labeled segment
  - Ensures it can't be used to leak information

Example: virus scanner



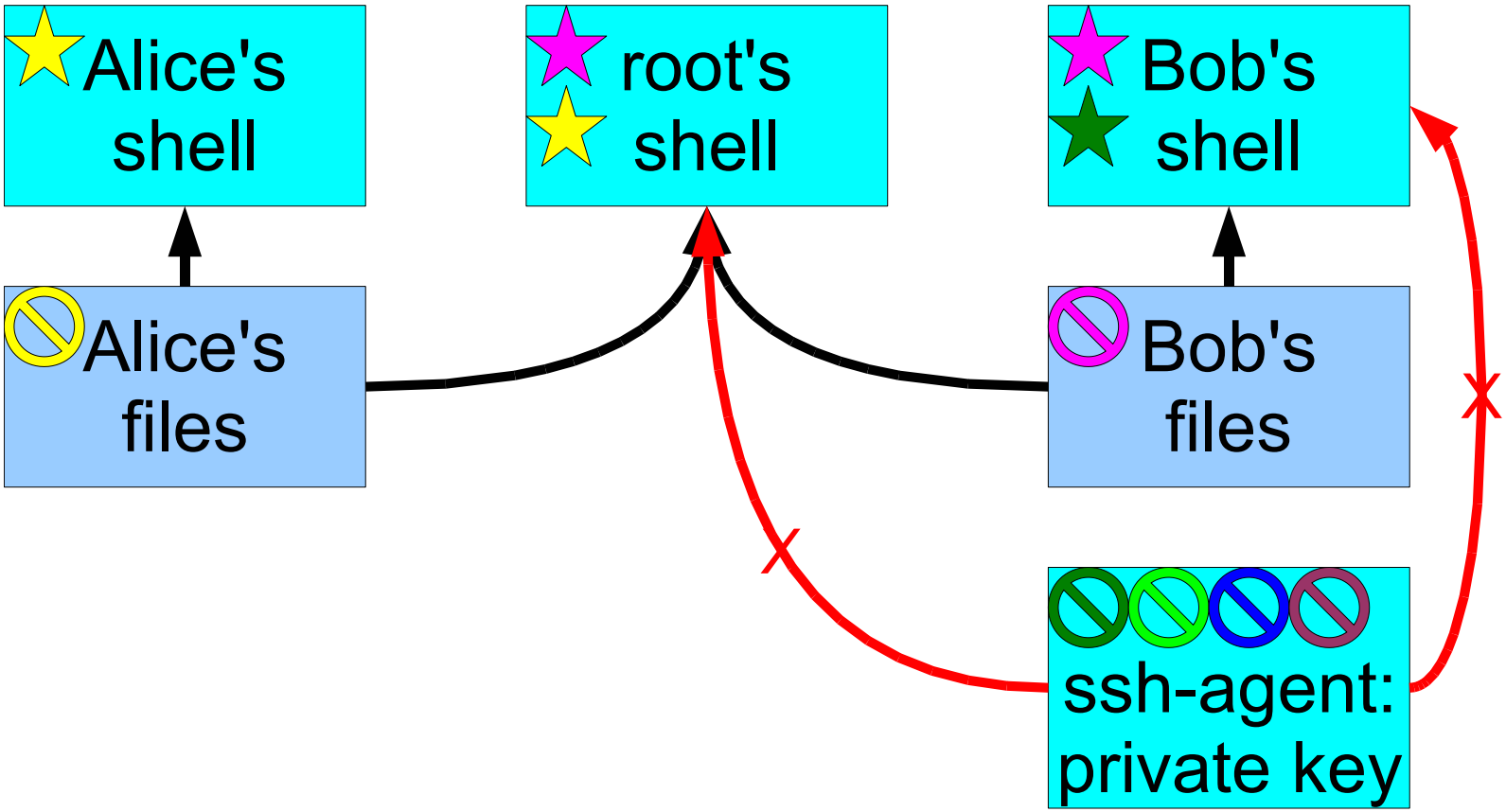
- Scanner tainted, so cannot write to network
- Dynamically allocated ⊘ also prevents scanner from corrupting files

Example: ssh-agent



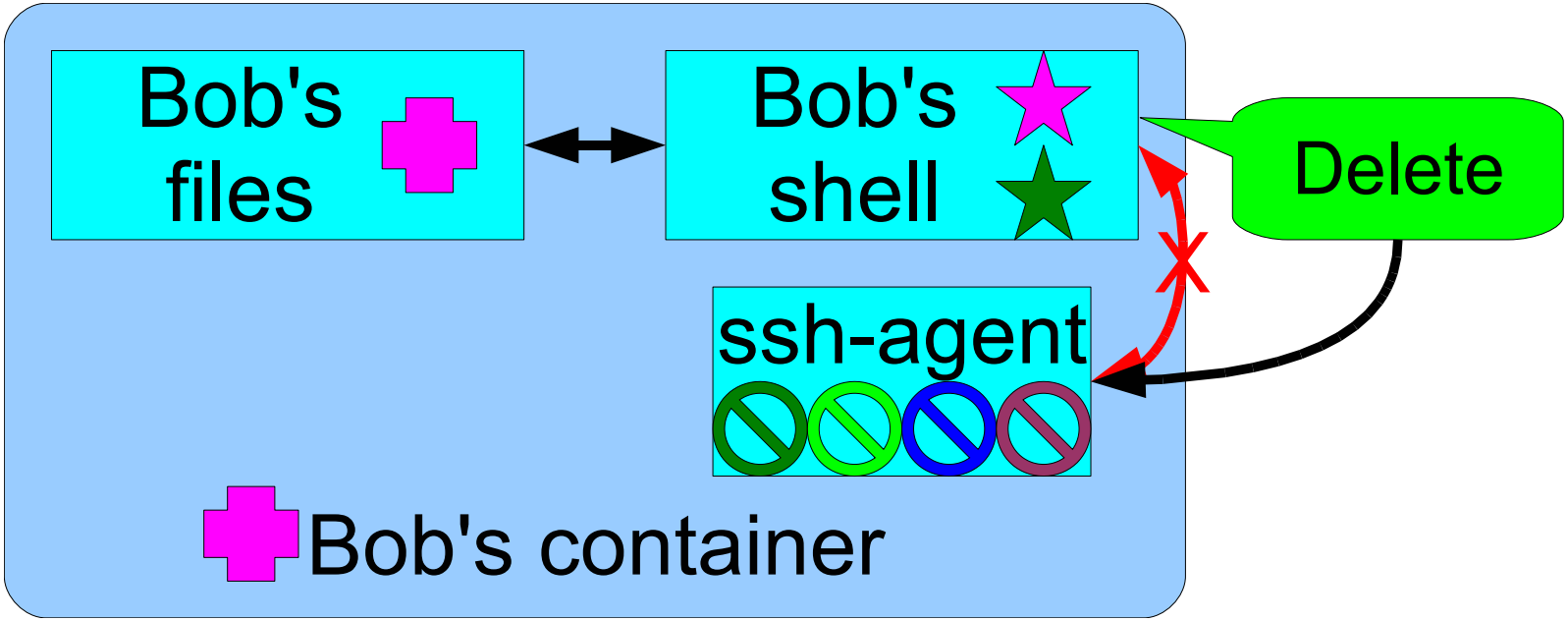
- ssh-agent stores your private key
  - Want to prevent even root from getting the key
  - ssh-agent can just allocate its own secrecy category

Runaway processes



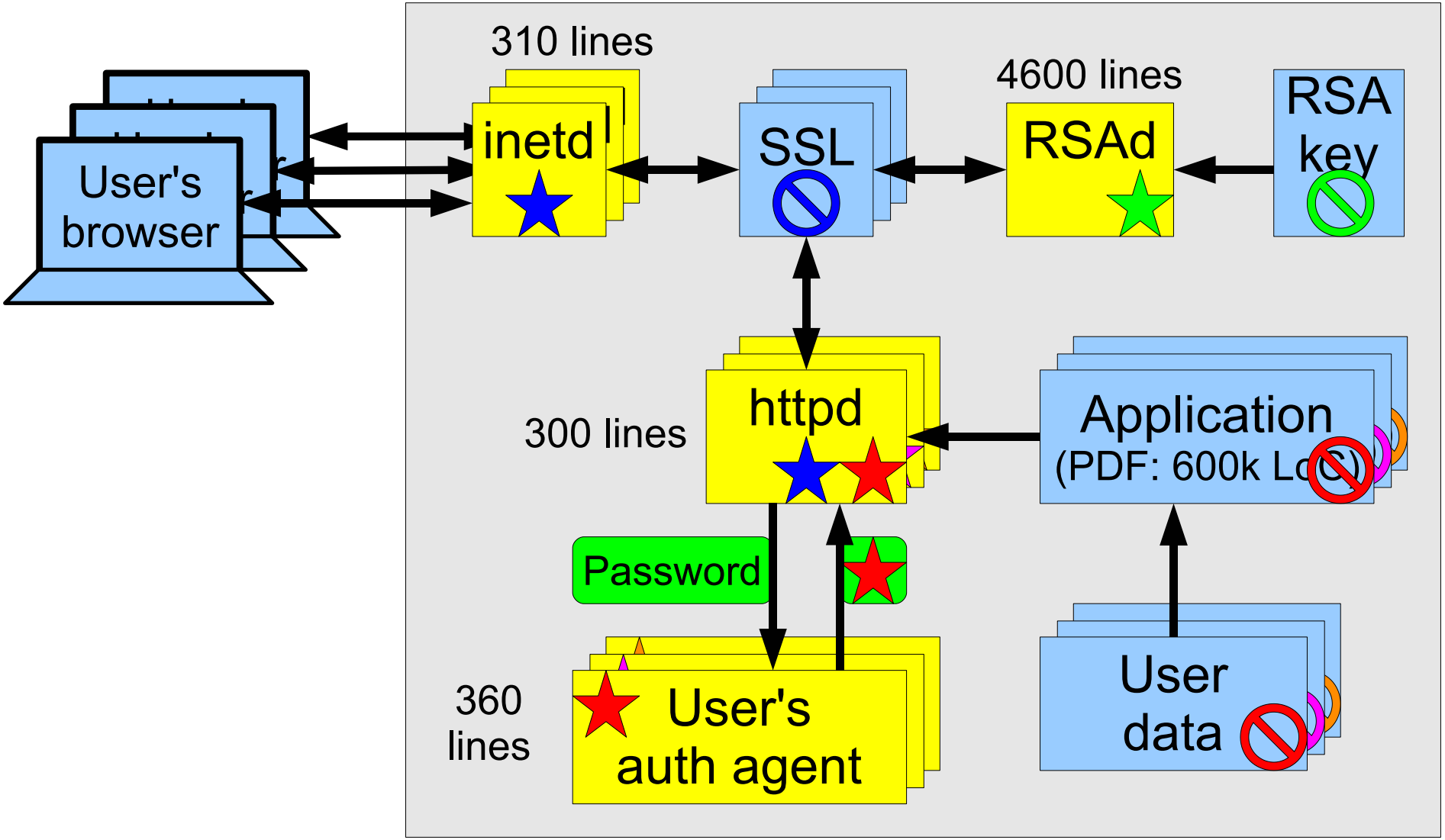
- What if ssh-agent goes nuts?
  - Can allocate lots of categories, cut itself off from the world!

Containers



- Separate resource allocation from access control
- Can reclaim resources if you can write container
  - Even if you can't read or write the objects you are deleting

Web server


310 lines
4600 lines
User's browser
inetd
SSL
RSAd
RSA key
300 lines
httpd
Application
(PDF: 600k LoC)
Password
360 lines
User's auth agent
User data

What we learned from HiStar

- Nickolai Zeldovich can secure 1,000,000+ lines of third-party code
  - But he is *not* the median programmer to say the least
- System-wide egalitarian access control is practical
  - All users/code get same interfaces regardless of privilege
  - No all-powerful superuser who can read/write all data
- Every single bit must conceptually be associated with a label
  - Must be able to point to any bit and say what the label is (program counter, registers, labels themselves, even the fact that a process still exists)
  - Need a covert channel? Implement with privileges outside kernel

Outline

1. HiStar OS
2. Closing remarks
3. Deian Stefan – the industry perspective

System design issues

- Systems often have many goals:
  - Performance, reliability, availability, consistency, scalability, security, versatility, modularity/simplicity
- Designers face trade-offs:
  - Availability vs. consistency
  - Scalability vs. reliability
  - Reliability vs. performance
  - Performance vs. modularity
  - Modularity vs. versatility
  - Latency vs. throughput

Engineering vs. research

- Engineering:
  - Find the right design point in the trade-off
  - Minimize cost/benefit, etc.
  - It's important to know how to do this (even/especially in research)
- Research:
  - Fundamentally alter the trade-offs
  - Ideally get "best of both worlds"

Example: Scheduler activations

- Problem: Kernel-level threads suck
  - Many expensive context switches
  - Kernel doesn't know about application-specific priorities
- Problem: User-level threads suck
  - Scheduler doesn't know which system calls block
- Solution: New kernel interface
  - Expose information needed by user-level scheduler: preemption, blocking system calls, I/O completion, ...
  - Provides the best of both worlds
  - Facilitates other abastractions, too! (async I/O)

Example: Receive livelock

- Problem: Interrupt-driven network stack bad
  - Causes livelock under heavy load
- Problem: Polling adds too much latency
- Solution: Re-architect kernel for best of both worlds
  - Mogul and Ramakrishnan show several techniques
  - Hybrid interrupt/polling, queue-length feedback, CPU quotas
  - Could maybe unify some with BVT to reduce need for tuning

Placing functionality

The end-to-end argument

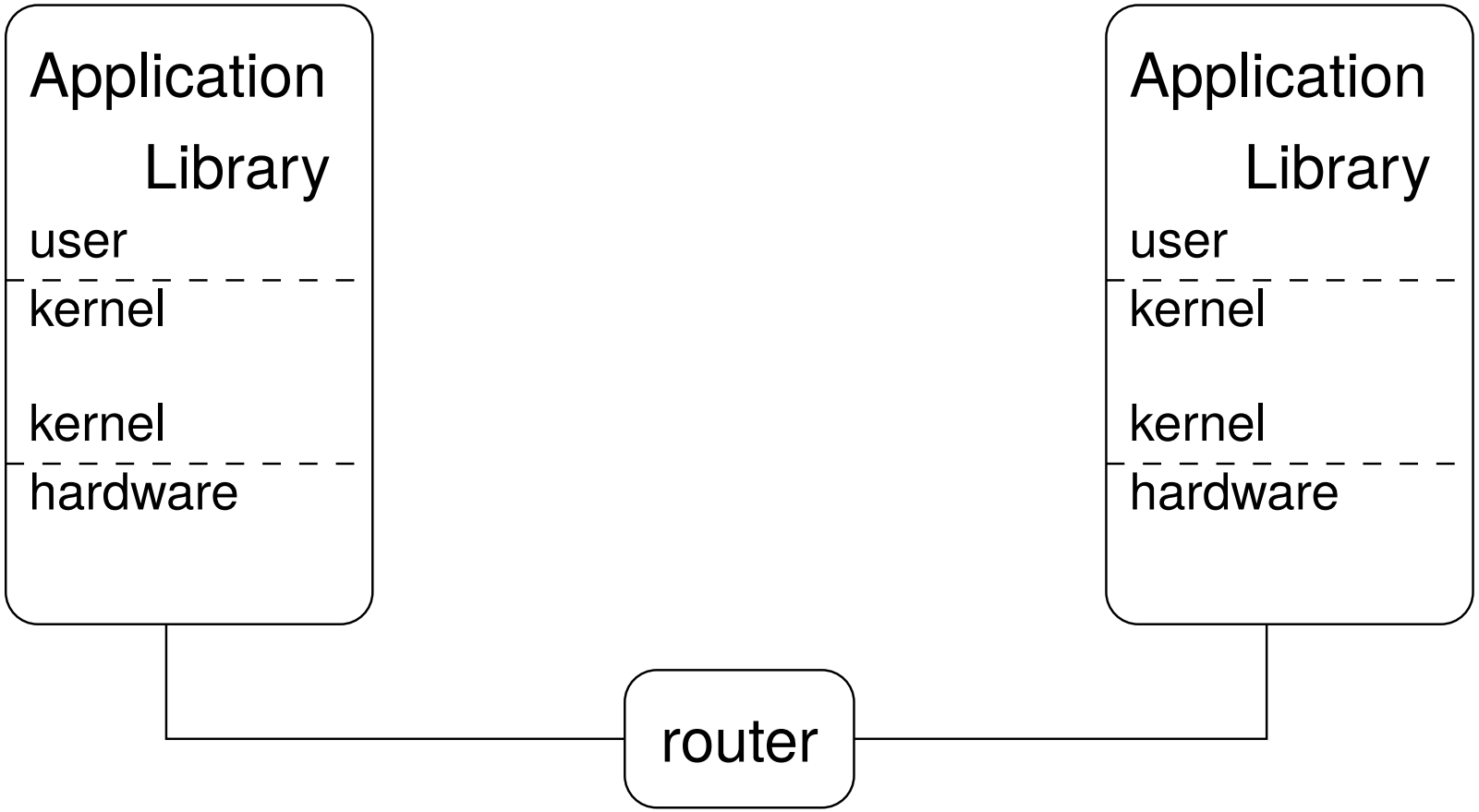
The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the end points of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible. (Somtimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.)

- Seminal paper by Saltzer, Reed, and Clark
- Articulated principle for where to place functionality
- Profoundly influenced the design of TCP/IP
  - Allowed Internet to evolve so much faster than phone network

Example applications of principle

- Link-by-link reliable message delivery
  - Often ensured by application (higher-level reply)
  - Can't trust every component of network
  - Inappropriate for many applications (e.g., voice over IP)
- FIFO message delivery, duplicate suppression
  - Redundant, just slows down two-phase commit, etc.
- Security and data integrity checks
  - Only make sense end-to-end

End-to-end principle generalized



- Place functionality closer to the endpoints
- Often leads to insight that yields best of both worlds!

Applying the end-to-end argument

- Keep lower-level functionality only for performance
  - E.g., Ethernet tries several times after a collision
  - Avoids unnecessarily triggering TCP retransmits
- Provide “least common denominator” abstractions
  - Can implement threads on events, but not vice versa
  - Can implement threads or events on sched. activations
  - Can implement many tricks on top of good VM primitives

Hints for low-level abstraction design

- Expose information
  - Lets applications/libraries make intelligent decisions (Is thread runnable? How much memory is available?)
- Expose hardware and other low-level functionality
  - Appel & Li: Exposing VM helps applications
  - Frangipani: Exploits low-level block protocol, locks
- Avoid "outsmarting" higher-level software
  - We still sometimes see papers on buffer cache management
  - Databases routinely bypass the buffer cache
  - Maybe OS shouldn't dictate the policy
- Good interfaces give $\sqrt{\text{complexity}}$
- Bad interfaces increase complexity

Things to think about

- System designers face many trade-offs
- When possible, gain the best of both choices
  - Rethink layer interfaces and abstractions
  - Push functionality upwards (end-to-end priciple)
- High-performance servers particularly demanding
  - Often uncomfortable fit on traditional OS abstractions
- Use “OS techniques” at application level

Final thoughts

- You are all now operating systems experts
- Use this knowledge to build better applications
  - Sometimes need to coax right behavior out of kernel
  - Should be much easier now that you know what's going on
- Syscall interface can be an *innovation* barrier
  - Much harder to change kernel than user code
  - Other examples include standardized network protocols, servers
  - Get these wrong and many people will suffer
- Some of you will go on to design interfaces that many people are later subjected to
  - Strive to achieve both simplicity and flexibility for users
  - Good rule of thumb: don't finalize an interface until you have at least three very different applications using it

How to learn more about OSes

- Take CS240 – Advanced Topics in Operating Systems
  - Class will bring you up to speed on OS research
  - Read & discuss 18–25 research papers
  - By the end, should be ready to do OS research
- Get involved in research!
- Lot's of interesting OS work at Stanford
  - Rosenblum – launched the virtual machine resurgence
  - Ousterhout – RAMcloud involves low-latency networking
  - Lam – collective system, software for mobile devices
  - Levis – seminal work on sensor nets, power management, now IoT
  - Winstein – pushing the limits of network protocol design
  - Engler – tools to find OS bugs automatically
  - Boneh/Mitchell – lots of practical OS security work
  - Mazières – OS and language-level work, emphasis on security

Outline