A Look at the EROS Operating System
Part I: A High Level View

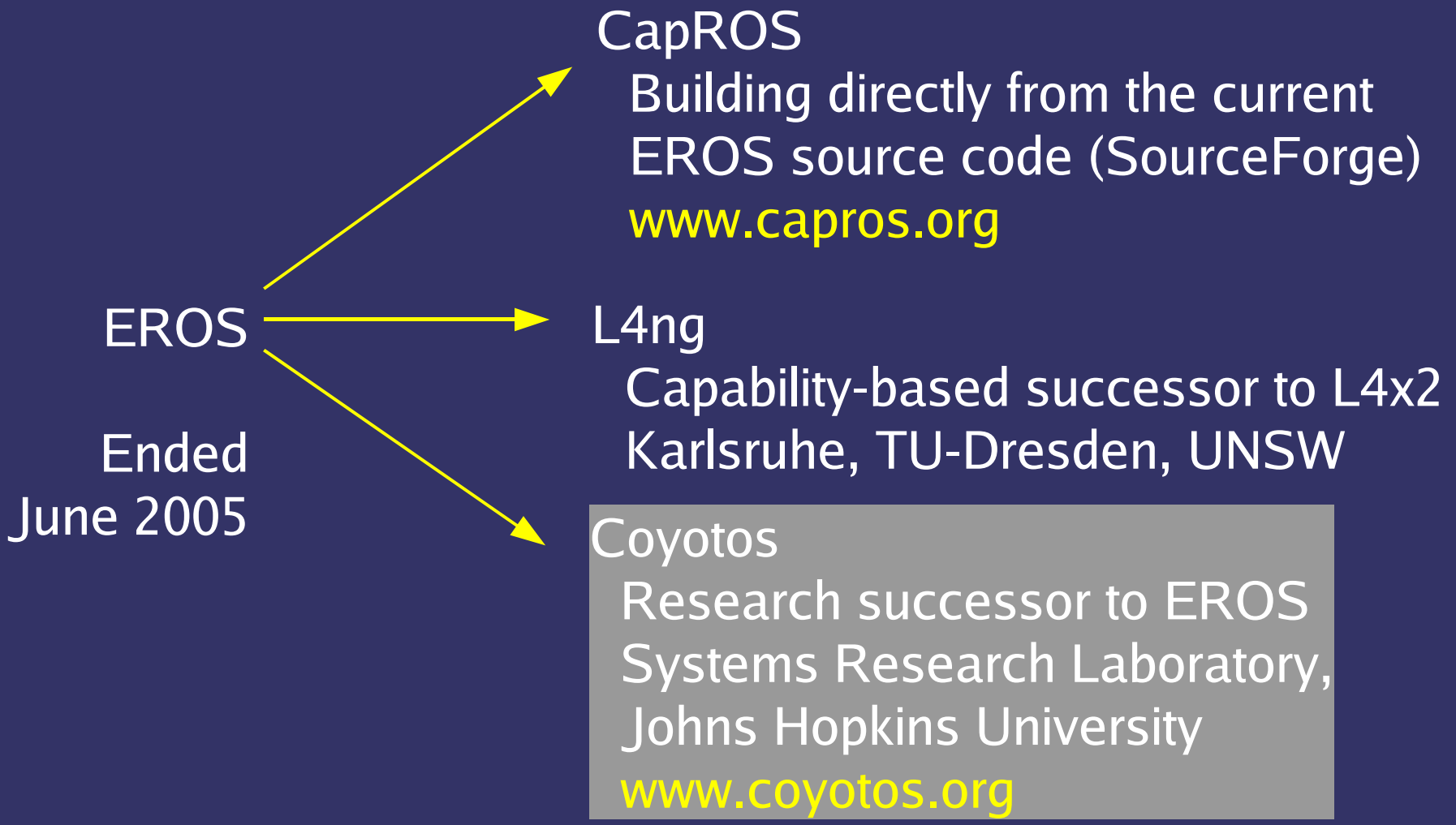
Jonathan S. Shapiro

Managing Director,
The EROS Group, LLC and

Assistant Professor
Department of Computer Science
Johns Hopkins University

These slides were first presented as part of the
Libre Software Meeting, Dijon France, July 2005.

EROS Derivatives



Goal: Defensible, Robust, Sound OS

- Robust: uptime measured in years
- Defensible:
  - Operates from a secure base
  - Provide mechanisms that allow applications to *build on* and *extend* this base.
  - Provide a system structure in which secure design is the default behavior, not an extended effort.
- Sound:
  - Based on a formally effective access control model
  - Ability to verify that security policy is enforceable
    - **Enforceable**, not enforced, because implementation is unverified.

Invert the Problem

How Would You Build a ***Vulnerable*** OS?

Suppose the goal were to *maximize* compatibility **for viruses and worms**.

1. Maximize Authority

- Give *every* program access to *everything*
  - Network
  - Shared file system
  - Awareness of other processes
- Do not let the user have any control
  - Make every process run with the entire authority of the user
- Use an access control system that mathematically doesn't work: ACLs [HRU 1976]
- Later, try to rescue it with one that isn't manageable and *still* doesn't work: SE-Linux (RBAC)

2. Impose No Resource Controls

- Make quotas hard to use
  - This way, they will be disabled.
  - Make them second-class (a quota should be a resource)
  - Have it default to “no limits.”
- Do not provide accounting for in-memory resources (residency, #processes, CPU schedule)
- Design resource controls so that they are *relative* (priority based), rather than *absolute*.
  - More processes => more resources
  - No limits on fork()

3. Put Policy in Kernel

- Standardizes methods for how to get the *OS* to *help* compromise other programs.
- When combined with relative resource controls, this is *extremely* helpful to the attacker.
- Allows each developer to invent their own, bad, inconsistent mechanism.

4. Have an “admin” or “root” role

- Concentrates total authority at a single point of vulnerability.
- Even better if *anybody* can become root because the protection system is flawed [HRU 1976]
- Make sure that root can violate all policies and all guarantees!
  - Supports administrator error
  - Maximizes employment for expensive consultants

5. Maximize Communication

- If two *arbitrary* processes are running on the same machine, make sure that it is possible for them to communicate without additional permissions
- For example:
  - Create named pipe
  - Grant world RW permission
  - Both open
  - Use read/write to send data
  - Use ioctl(..., I_SENDFD, ...) to send descriptors!
- Cumbersome, but effective

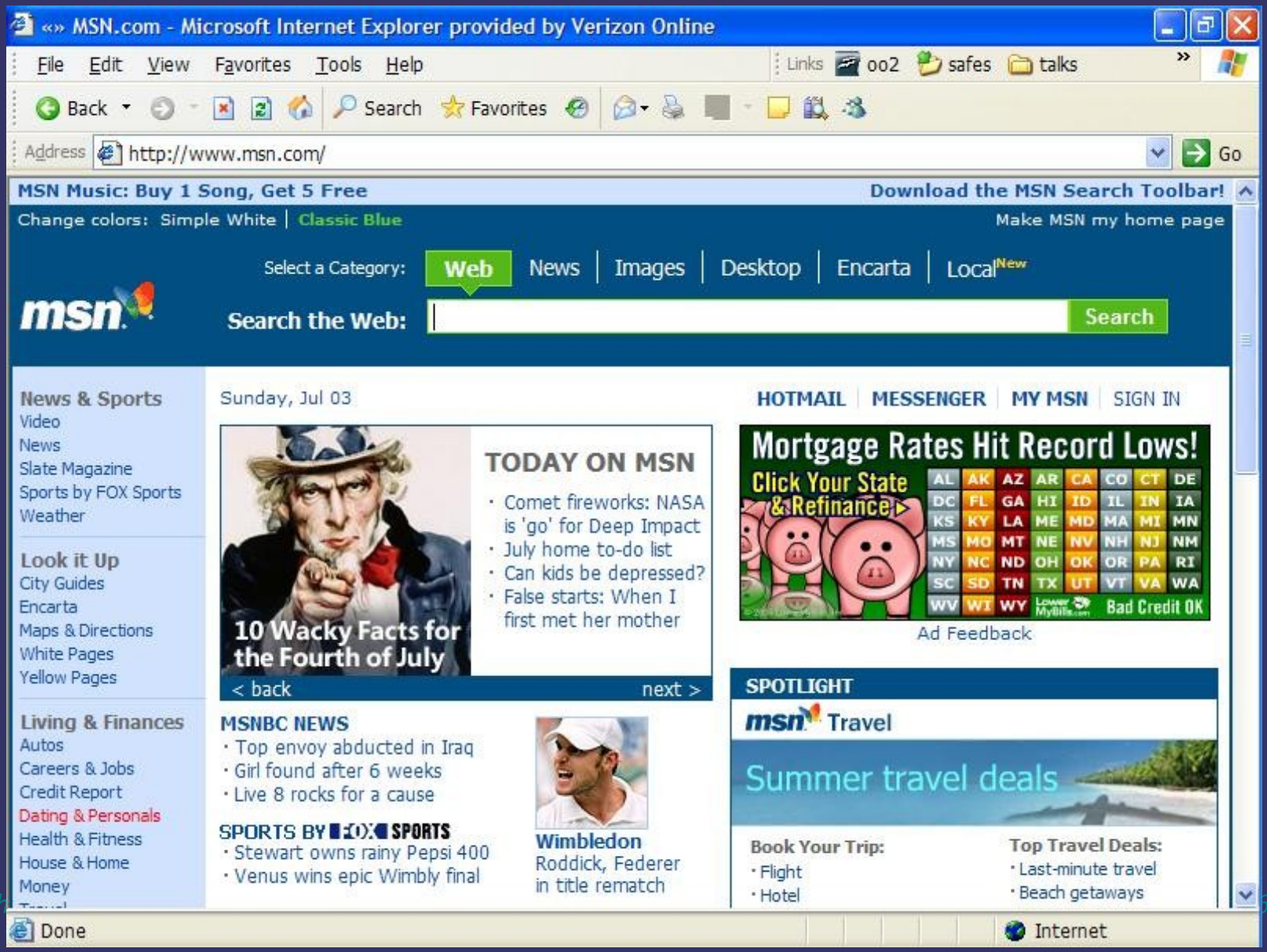
6. Use Scripting Everywhere

- Make sure that most programs run scripting or configuration code (java, VB, javascript, but also .exrc, /bin/sh, .emacs)
- Ensure that programs are written in unsafe languages.
- Provide no isolation.

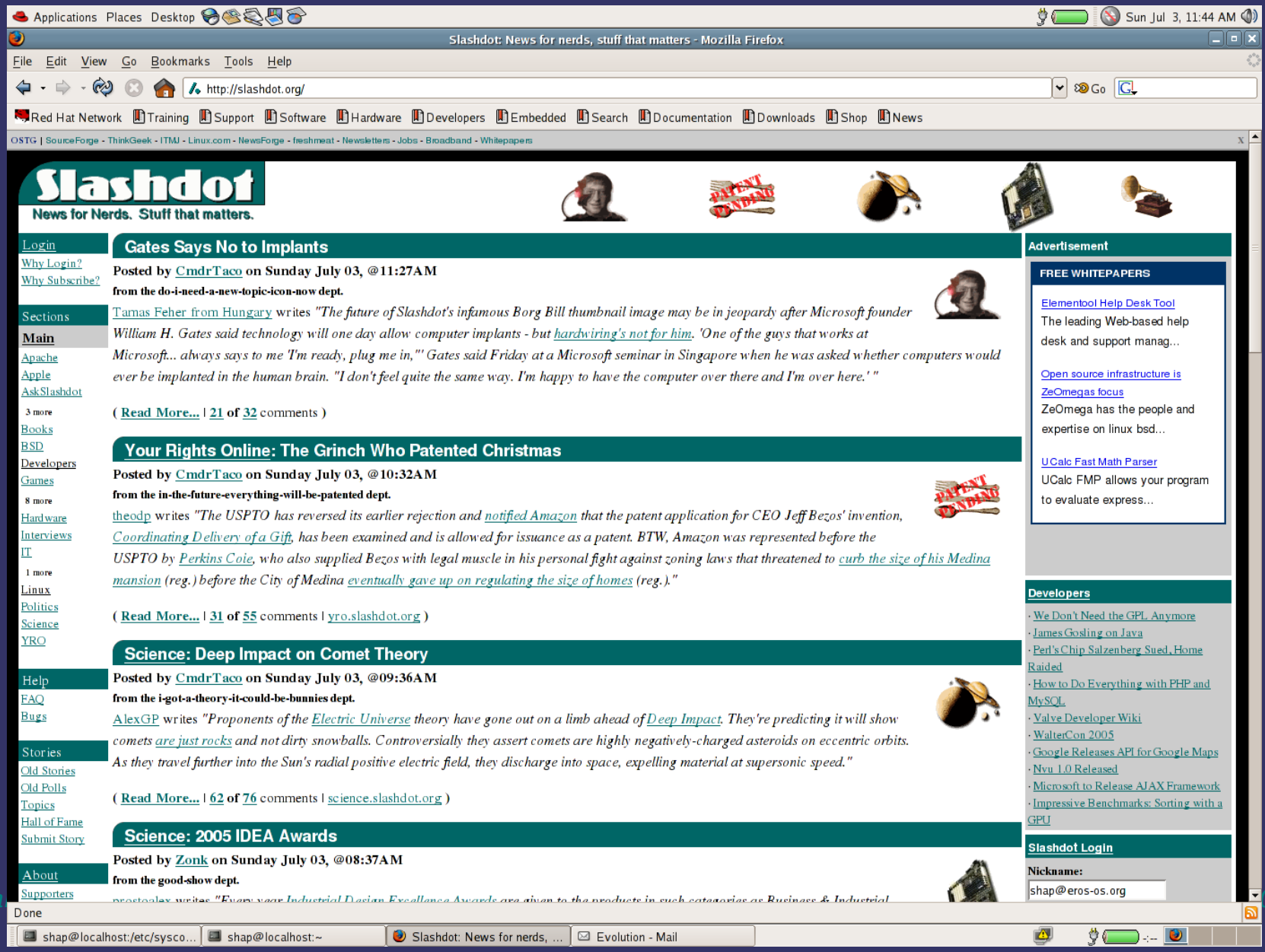
7. Disable Authentication

- Sometimes, my program *relies* on the fact that it is supported by an authentic implementation of some service.
- Example:
  - I cannot keep a secret unless I know that the containing pages are private to me
  - I cannot know that the pages are private unless I know that they are allocated from an authentic storage allocator (one which guarantees exclusivity)
- Not an issue in UNIX. *Critical* issue in a capability based multiserver system.
- Especially for confinement (as we will shortly see)

8. Now Run Internet Explorer



8(b) ... or FireFox



Why are IE/Firefox Dangerous?

- They obey hostile code
- Run with *total* authority
  - *Not* just the user's authority – UNIX and Windows are theoretically undefendable!!!
- Executes on a system with no structural defenses
  - Windows/UNIX defense model is boundary defense
  - No effective runtime resource controls
- No code safety, no verification...
- No principled security design
- Good programs cannot defend themselves

IE/FireFox, but also...

- Emacs
- GCC
- OpenOffice
- X-Windows
- Java
- Most Gnome/KDE Apps
- Nautilus
- Evolution
- CVS
- Java
- CGI Scripts
- Apache
- IMAPD

NONE of these programs needs to be vulnerable in order to function!!!

The (Sad) Objective Reality

- **UNIX-based operating systems are just as insecure as Windows**.
- Open source is no defense
  - **The “many eyeballs” theory (ESR) is wrong**
  - Not all bugs are shallow
  - When inspecting, malice != error
- Security is a problem of *architecture*
  - Neither Windows nor UNIX were architected to be secure.
  - You can have compatibility or security, but not both.

EROS

Essential Features

- A pure, capability-based operating system
  - It is an object-based, not a client-server architecture
- High performance invocation (includes IPC)
- Transparent persistence
- Built on a decidable access model
  - Questions of policy enforceability are decidable (and the outcome is good)
  - Confinement mechanism is verified
  - Implementation is not (and won't be)

Capability: Classical Definition

- Term “capability” is due to Dennis and van Horn, 1966, *Programming Semantics for Multiprogrammed Computations*
- A capability is an (object name, access rights pair)
- The term “object name,” in this context, has been commonly (mis)understood to mean “the global name of some system resource.”

- A **capability system is a pure object system**

History

GNOSIS start	1968	Hardy *et al.*	Tymshare, Inc.
GNOSIS	1972	Hardy *et al.*	Tymshare, Inc.
KeyKOS	1978	Hardy *et al.*	Key Logic, Inc.
EROS version 0	1991	Shapiro, Hardy	Synergistic Computing Associates
EROS research 1	1999	Shapiro	University of Pennsylvania
EROS research 2	2004	Shapiro *et al.*	Johns Hopkins University
EROS version 1	2005	Shapiro *et al.*	The EROS Group, LLC

Formal Models and Results

- Anita K. Jones, 1973
  - *Protection in Programmed Systems*
- Harrison, Ruzzo, Ullman, 1976
  - *Protection in Operating Systems*
- Jones, Lipton Snyder, 1976
  - *A Linear-Time Algorithm for Deciding Security*
- Neumann, Boyer, *et al.*, 1980
  - *A Provably Secure Operating System: The System, Its Applications, and Proofs*
- Shapiro, Weber, 2000
  - *Verifying the EROS Confinement Mechanism*
- Notably *not*:
  - Lampson, *Protection*
  - Static snapshots reveal very little about the evolution of dynamic systems

Higher-Level Features

- Resource pools are first class
- “Type-of” mechanism for capabilities
  - Supports capability identification
- Explicit mapping structures
  - Supports user-level memory management
  - Supports Copy-On-Write
- Hierarchical mechanism for resource allocation
  - User account gets a “space bank”
  - This is subdivided for programs that the user runs

Design Philosophy

- Authority is explicit in capabilities
  - Kernel protected (object ID, permission) pair
- All communication must occur over channels *that are described by capabilities*
  - *Authority propagation by introduction* ⇒ transitive closure
- Processes have *no* initial authority
  - All authority is explicitly granted.
  - No intrinsic access to file system, network, process list
  - No intrinsic right to memory pages, CPU, process creation
  - Applications divided into multiple processes
- No “logical” kernel policy (well, very very little)
- No root/admin concept

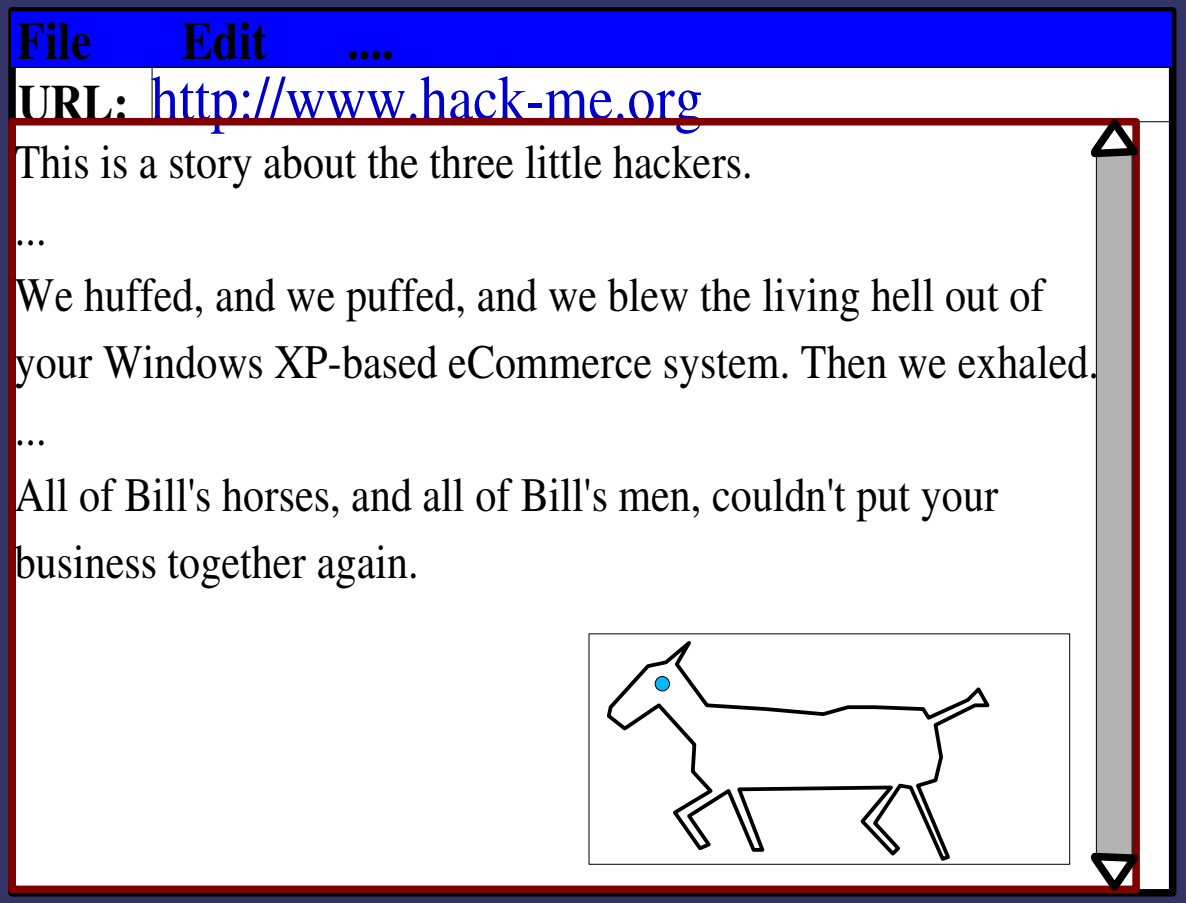
Key Ideas in EROS

- Capabilities
  - Can they be fast on commodity hardware?
  - Are they an effective unifying mechanism for resource management
- Persistence – is transparent persistence useful/good?
- Confinement
  - Is confinement an effective basic building block in robust/secure systems?
  - If so, can it be used pervasively enough
- Kernel as extended microprocessor (state machine)
- Design rests on an abstract formal access model and op. semantics

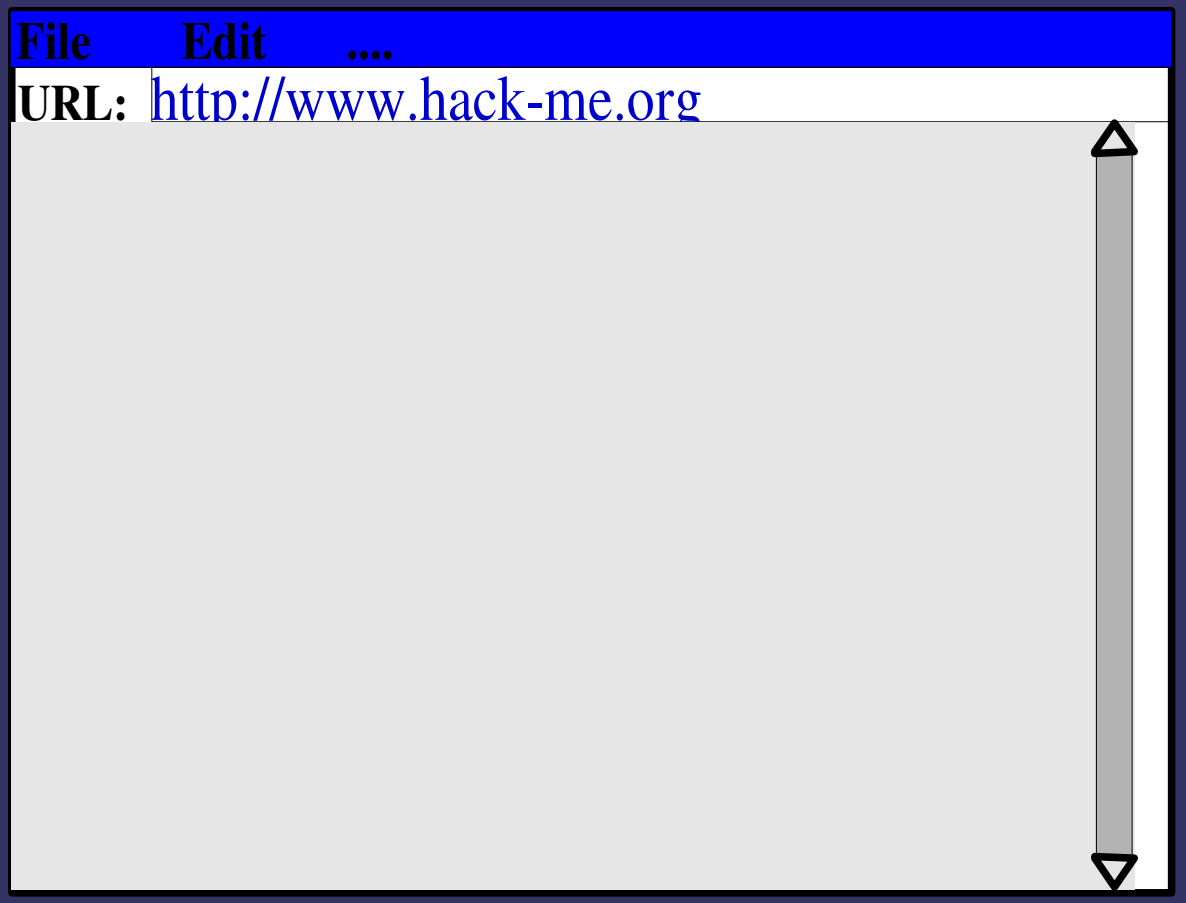
Structuring Application Security

- Divide Programs into two types
  - Programs that act (purely) for the user, and must be trusted
  - Applications (untrusted)
- Ensure that when an application does something risky, it must always act with user consent.
  - Really: user or reference monitor

Your Basic Browser Consists Of



An "Application Shell"



And a Content Renderer

This is a story about the three little hackers.
...
We huffed, and we puffed, and we blew the living hell out of your Windows XP-based eCommerce system. Then we exhaled.
...
All of Bill's horses, and all of Bill's men, couldn't put your business together again.

Which May Recurse

(Recursively)

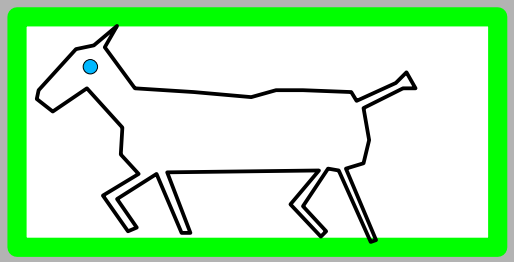
This is a story about the three little hackers.

...

We huffed, and we puffed, and we blew the living hell out of your Windows XP-based eCommerce system. Then we exhaled.

...

All of Bill's horses, and all of Bill's men, couldn't put your business together again.



Concept: Gate Keeper

- A small, trusted program (the gate keeper) stands between a hostile program and a precious resource. There is no way that the hostile program can bypass the gate keeper.
- A gate keeper can implement any test it wants:
  - Consult the user
  - Run a test case
  - Check for a virus
  - Verify that the regression suite has been run
  - Any decision whose answer is *yes* or *no*
- The tests don't need to be simple, but simple tests often have a lot of power.

Example: SaveAs Gate Keeper

Windows “Save As”

- Application says:

```
fileName := SaveFileDialog(...);
fd = open(fileName);
write(fd, ...);
close(fd);
```

- Maybe it's safe, maybe there is a virus: program *could* open *any* file.
- User has no way to know.

Challenge: try to design a virus that can corrupt my files when SaveAs is implemented this way.

- EROS “Save As”
- Application says:

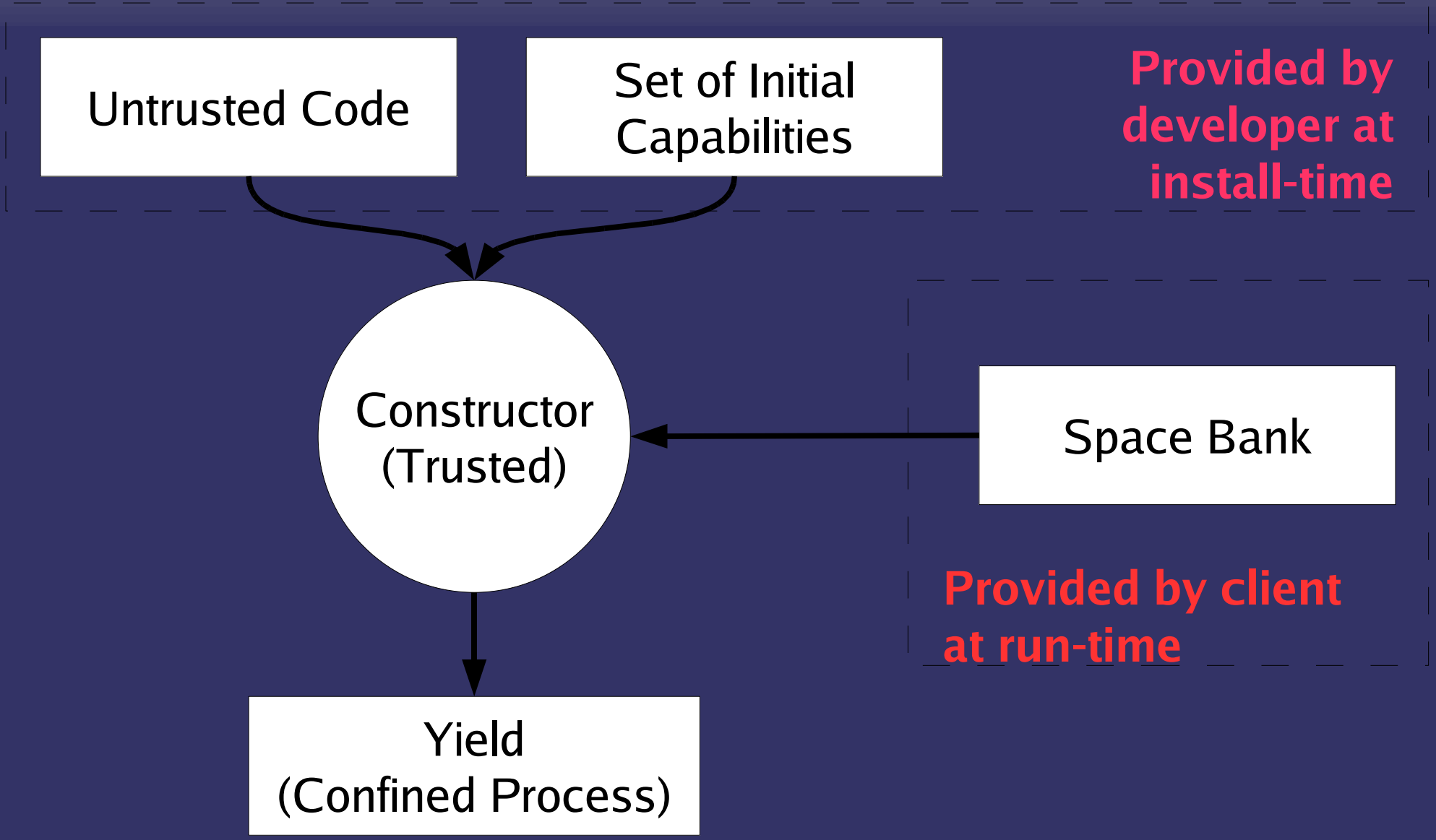
```
int fd = SaveFileDialog(...);
write(fd, ...);
close(fd);
```

- “SaveFile” ***is a separate program***, SaveFileDialog() is a wrapper that invokes it
- SaveFile has:
  - Access to my file system
  - Access to me (so I can click)
- Application has access to SaveFile
- **There is no way** the application can write a file without talking to SaveAs.

Virus/Worm Compatibility

System	*Compatibility*
Windows	**Excellent**
Linux	**Excellent**
OS-X	**Excellent**
BSD	**Excellent**
EROS	**Poor**

EROS Constructor



Why is this "Defense in Depth"

- Suppose I hack your web server. What can I do? Successful attacks do not grant the attacker a rich platform from which to expand their control of the machine.
- The attack that compromises component X does not generalize to component X+1: need sequenced, specialized attacks to gain any substantive control.
- Attacks must proceed through a narrowly specified, type-checked channel (the capability dispatch loop). This is automatically generated.
  - Deals with "well formedness" bugs, not concept bugs.
  - Facilitates tracing at the level of application semantics

Fastest, but also Oldest

- This morning, many “design pattern” questions were presented
  - Some: “How do I use capabilities effectively to do X?”
  - Others: “How do I re-think my design assumptions and restrictions so that capabilities can be exploited effectively
- EROS captures 38 years of uninterrupted experience with high-performance capability-based design
- We host an open mailing list for discussions about capability systems in general (not just EROS):
  - cap-talk@eros-os.org

Summary of EROS Results

EROS Accomplishments

- First fast capability system on commodity hardware
  - Shapiro, Smith, Farber. “EROS: A Fast Capability System.” *SOSP 1999*
- First system to demonstrate performance and defense in depth simultaneously
  - Sinha, Sarat, Shapiro. “Network Subsystems Reloaded.” *USENIX 2004*
  - Shapiro, Vanderburgh, Northup, Chizmadia “Design of the EROS Trusted Window System.” *USENIX Security, 2004*
- First verification of (overt) confinement
  - Shapiro, Webber, “Verifying the EROS Confinement Mechanism”, *IEEE Symposium on Security and Privacy, 2000*
- Microkernel vulnerabilities analysis
  - Shapiro, “Vulnerabilities in Synchronous IPC”, *IEEE Symposium on Security and Privacy, 2004*

In English:

- It's as fast or faster than conventional operating systems (comparable to L4)
- The primitives provide engineering options we haven't seen before that provide protection *with performance*.
- The resulting system is both empirically and formally securable in practical terms.
  - We are primarily concerned with day-to-day users.
  - We aren't very concerned with covert channels
  - We are *very* concerned about robustness

(Biased) Comparison to L4

- ✓ It's as fast or faster than conventional operating systems (comparable to L4)
- The primitives provide engineering options we haven't seen before that provide protection *with performance*.
- The resulting system is both empirically and formally securable in practical terms.
  - We are primarily concerned with day-to-day users.
  - We aren't very concerned with covert channels
  - We are *very* concerned about robustness

L4ng addresses most of these issues

What You Should Infer

- Microkernel architecture is bloody hard.
  - There is a *reason* that there were 15 active research microkernels in 1990 and only two today.
- There has been a convergence in microkernel design. The fundamentals of this area are now mature and reasonably well understood.
- L4ng and Coyotos *may* be the last fundamentally new microkernel architectures.

Break!

A Look at the EROS Operating System
Part II: EROS Architecture

Jonathan S. Shapiro

Managing Director,
The EROS Group, LLC and

Assistant Professor
Department of Computer Science
Johns Hopkins University

Everything is an Object

- Kernel Implemented
  - Pages (hold data)
  - Nodes (hold capabilities)
  - Wrappers
  - Processes
  - Void object
- User Implemented
  - Implemented by some user process
  - One process can implement multiple objects, multiple interfaces, or mutiple facets on a single object

(type, object-id, permissions)

(type, object-id, facet-id)

For the L4 Fans

- EROS is an object-based system, L4 is a server-based system.
  - In EROS, the idiom is that programs invoke objects.
  - In L4, the idiom is that clients send messages to servers.
- EROS does not have anything comparable to map/grant/unmap.
- EROS has a different primary objective
  - L4: *Performance Uber Alles*
  - EROS: *Security Uber Alles*
  - Or as Jochen once put it: "*Fast, ya. But correct? Eh.*"

Persistence

- Entire system is periodically checkpointed in the background
- Motivation: simplest path to secure bootstrap
  - Do not need to argue successful reduction of authority
  - Argue instead that saved state is successfully resumed
  - Argue that any saved state resulted from a correctness-preserving sequence of operations proceeding from an initially safe state
  - Check the base case separately
    - Via assurance (trusted components)
    - Via reachability (initial capabilities)

Capability Rescind

- Allocation Count
  - Most capability types carry a version number: the **allocation count**.
  - Every object likewise carries a version number.
  - Version is incremented on object rescind.
  - No match => capability is void.
- Call Count
  - Special mechanism for call/return. Similar to allocation count
  - Every node has a call count. Incremented by every call.
  - Call generates a resume key that contains call count for node.
  - No match => capability is void

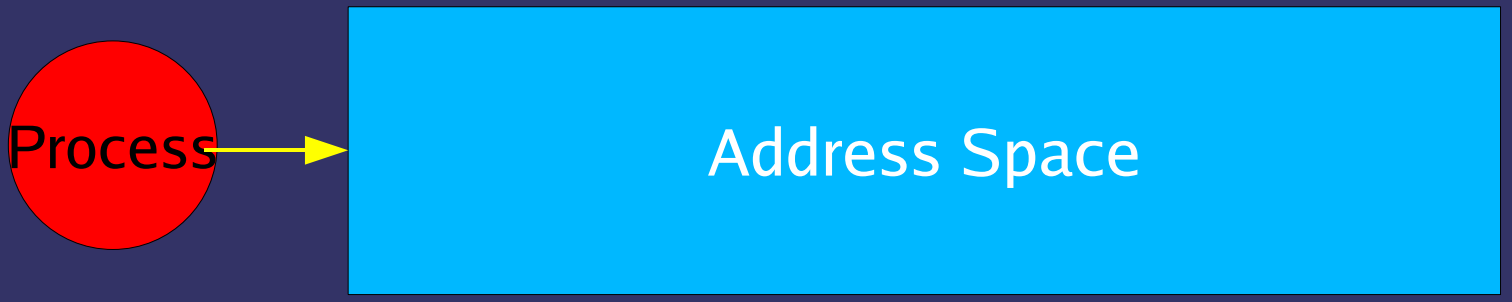
Application View of Process State

- Process State (capabilities)
  - Address space capability
  - Scheduling Capability
  - Register Set
  - Keeper (fault handler) capability
- Operating Environment
  - Constituents node
  - Space bank (storage allocator)
  - Process creator
- Capability Register Set
  - CR0 Void Capability
  - CR1..CR31 application defined
  - Certain conventions imposed by RT library

What an App Can Do

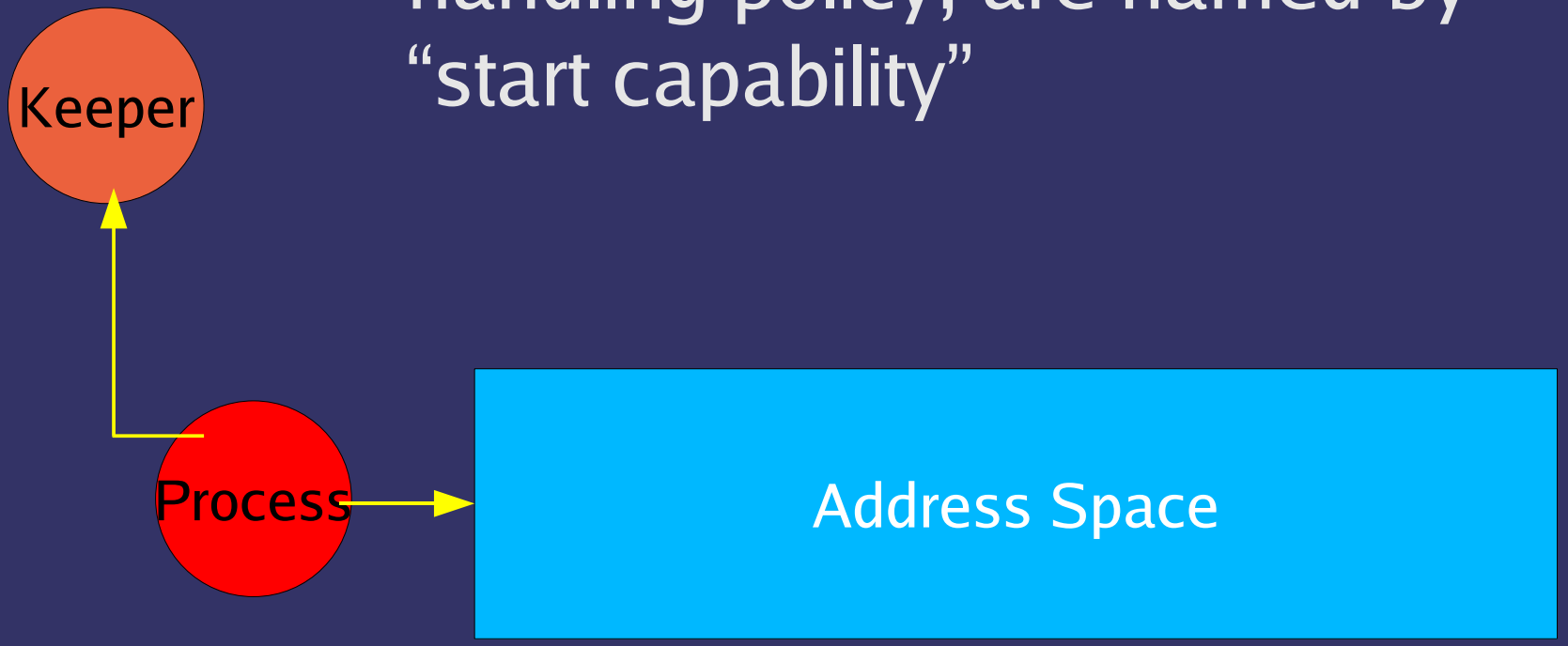
- Only one system call: invoke capability
  - Three variants: call, return, send
- Consequence: operating environment of a process is entirely defined by the capabilities it can invoke.
- *There is no canonical set of system calls. It is all object interfaces.*
- Capabilities are kernel-protected. Applications cannot “invent” them.

Application Environment



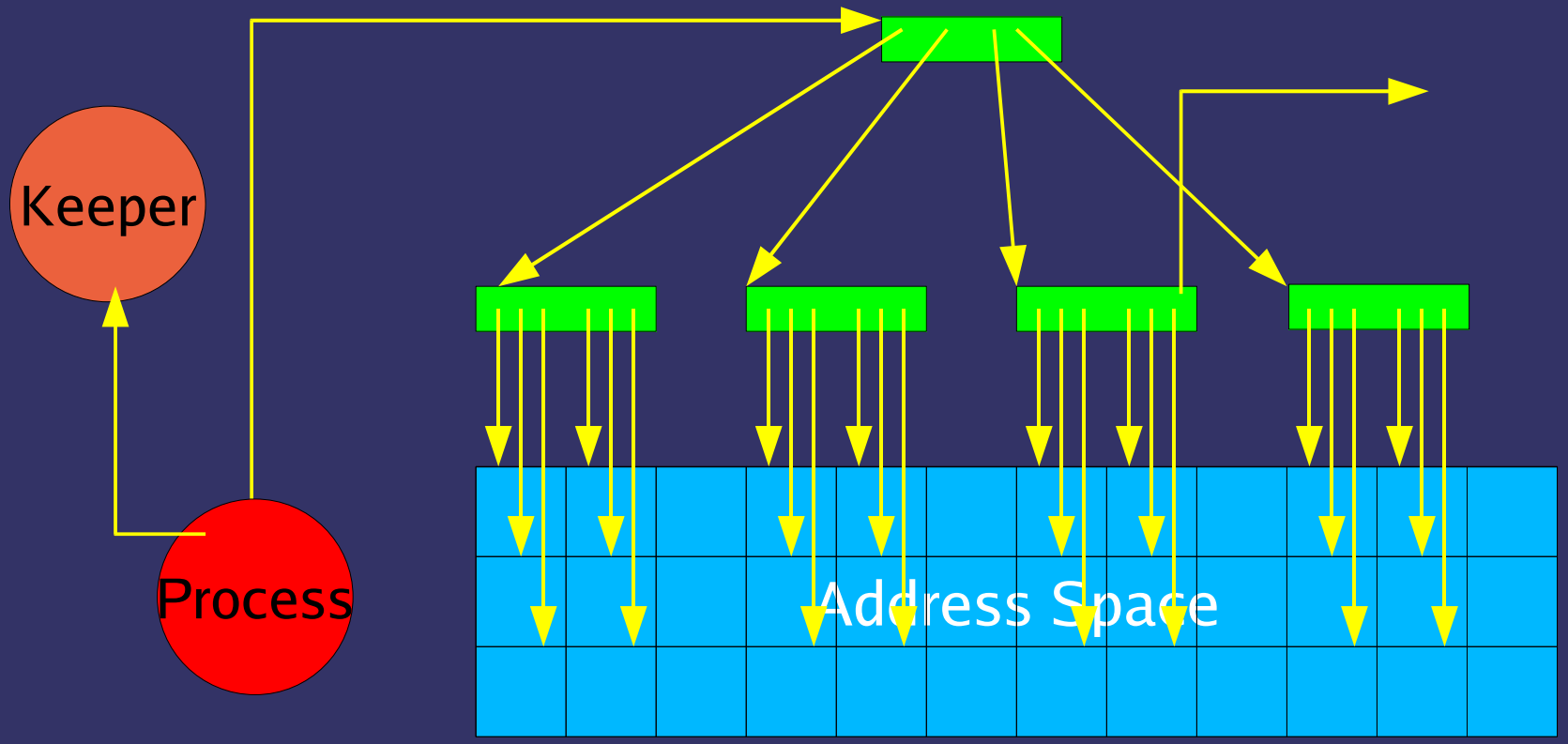
Application Environment

Keepers implement exception handling policy, are named by "start capability"



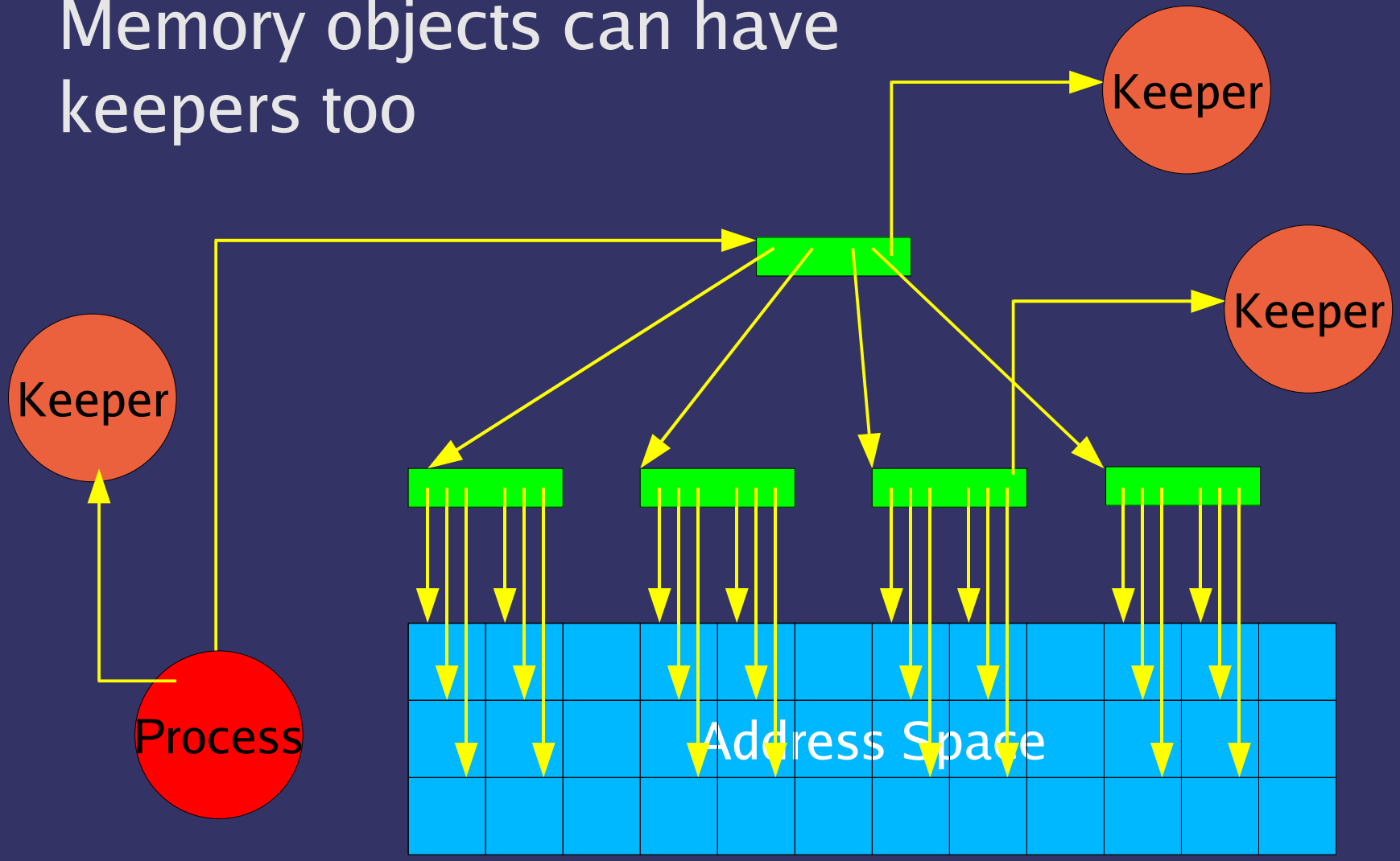
Application Environment

Address space metadata (mapping structures) is “first class.” **SW-defined mapping structure.**

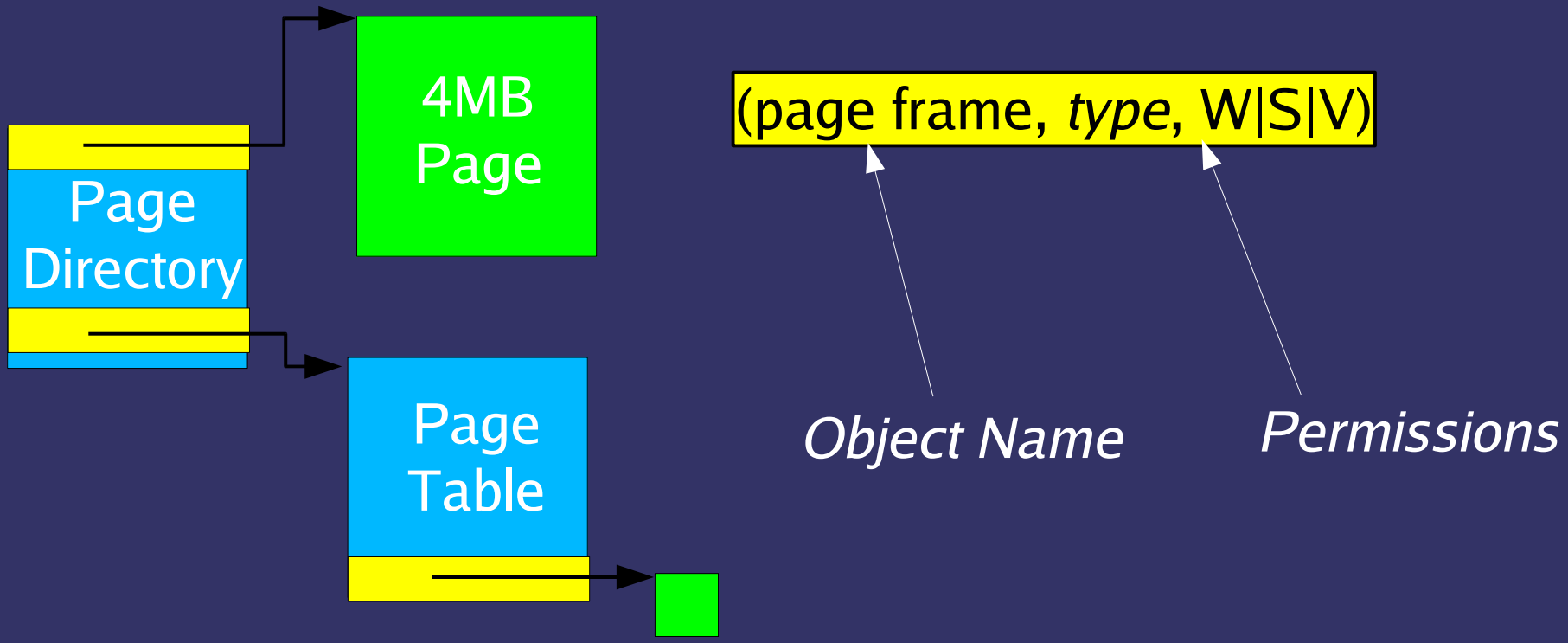


Application Environment

Memory objects can have keepers too



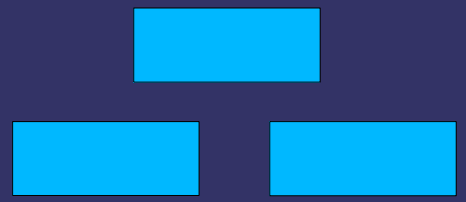
Modern Page Tables



Conventional page table entries are capabilities!

It works, Use It

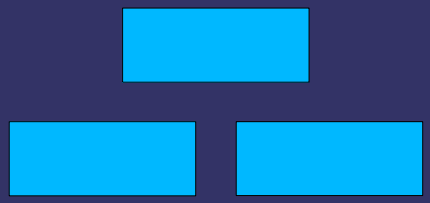
Node Tree
(Software Page Table)

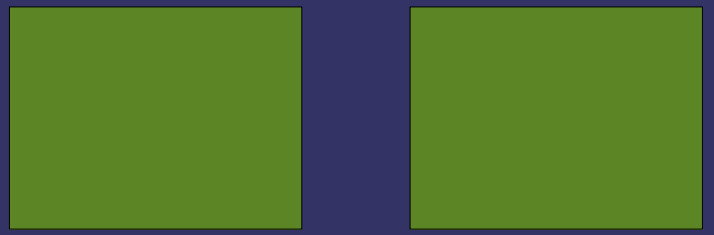


Demand Translation

Page Table Tree
(Hardware Page Table)

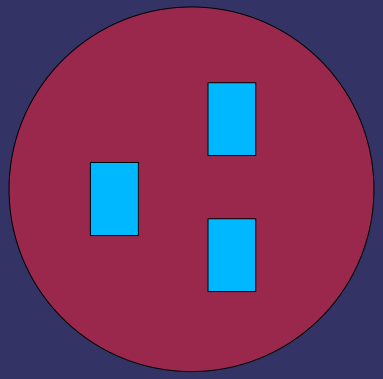
Reverse Dependency Tracking





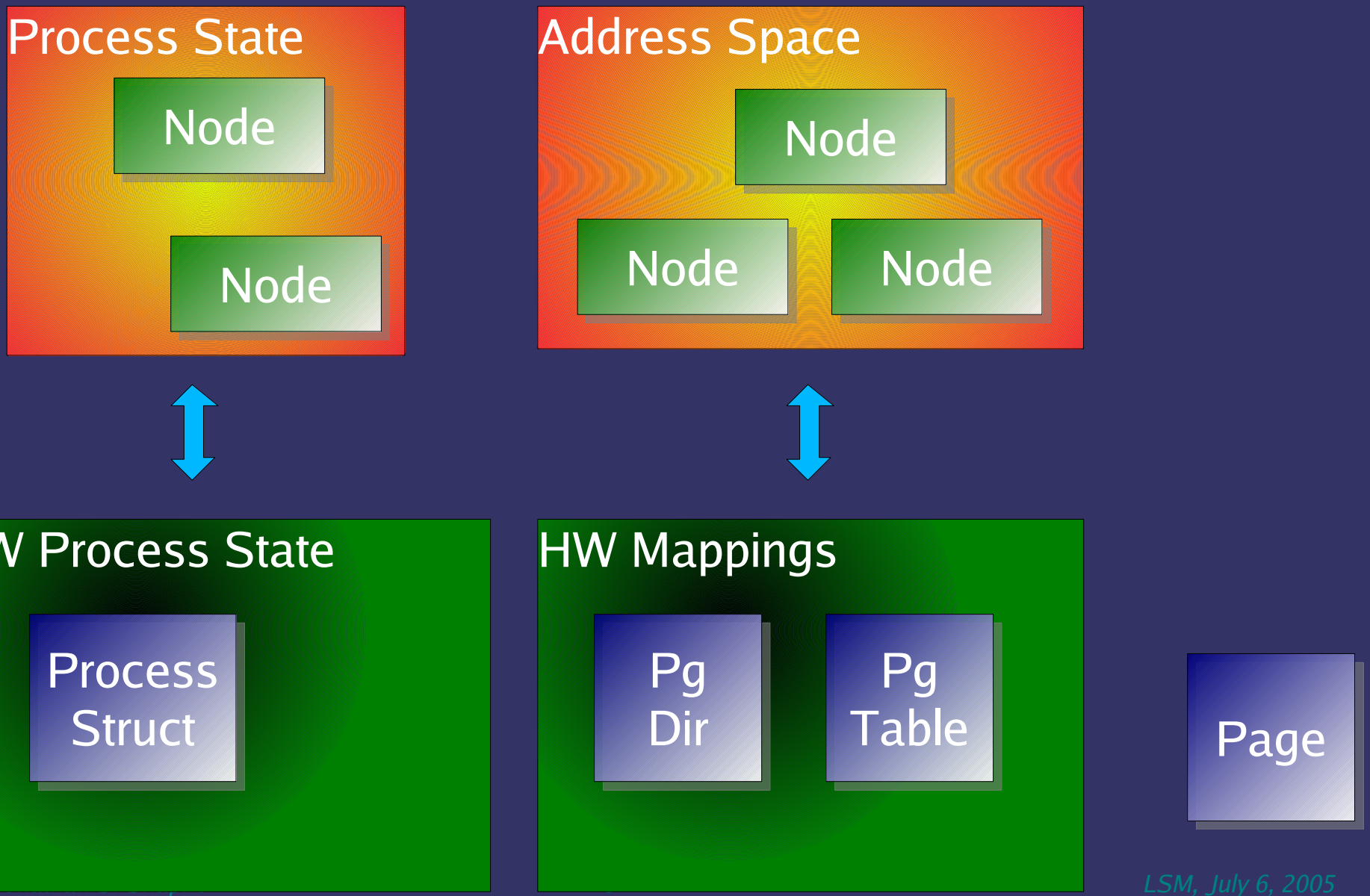
Processes

- From the kernel perspective, interesting process state is capability state. For this reason, process state is represented as an arrangement of Nodes
  - New capability type “number capability” to hold the register bits.

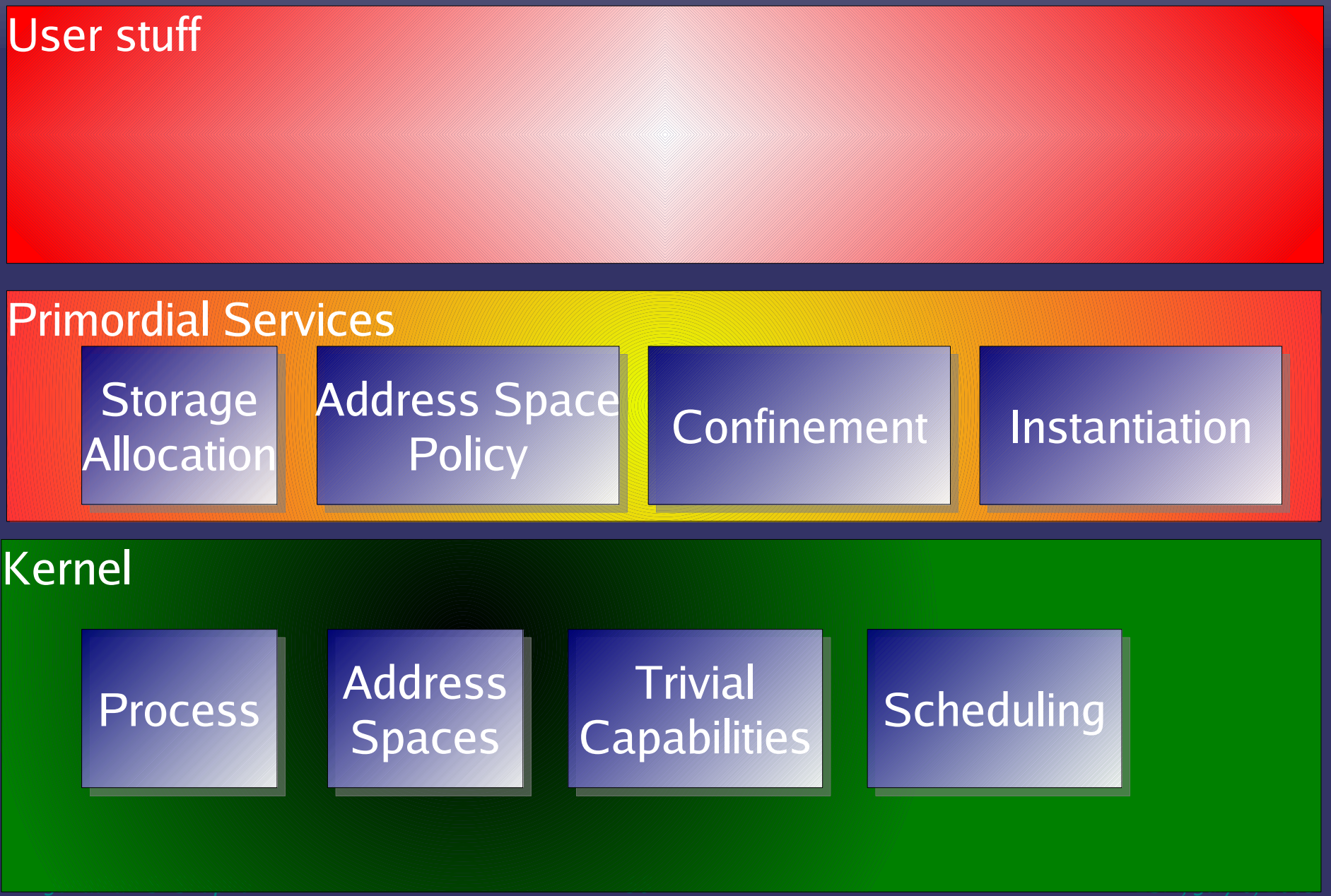


- Each process has “capability registers”

EROS Data Structures



EROS System Structure

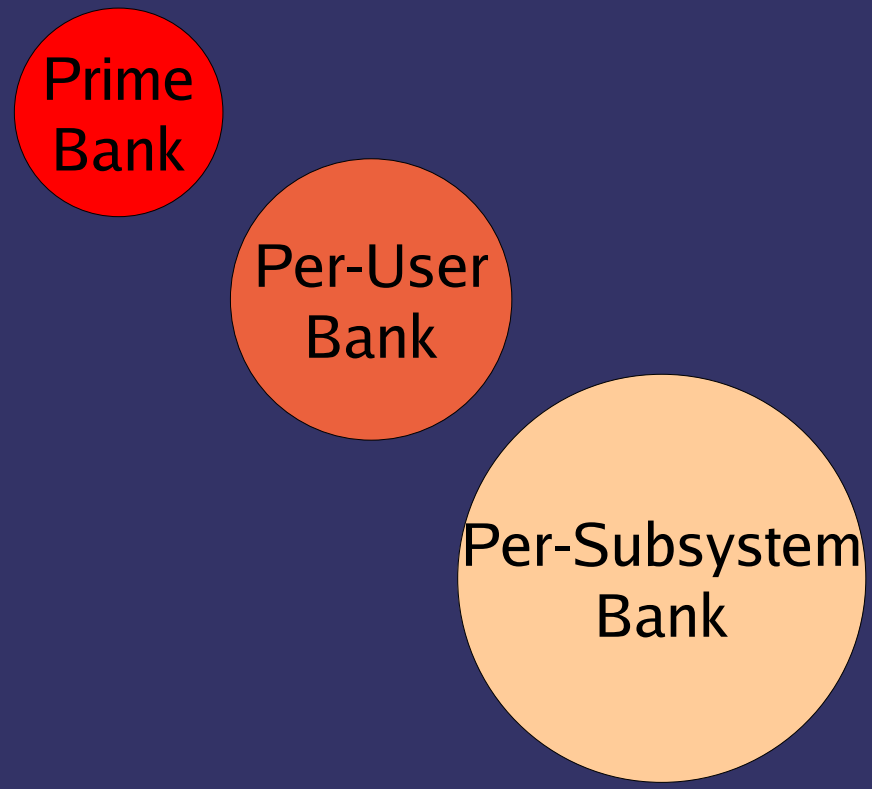


Primordial Services

- Space Bank
- Process Creator
- Constructor
- Virtual Copy Space Keeper (VCSK)
  - Implements Copy-On-Write spaces

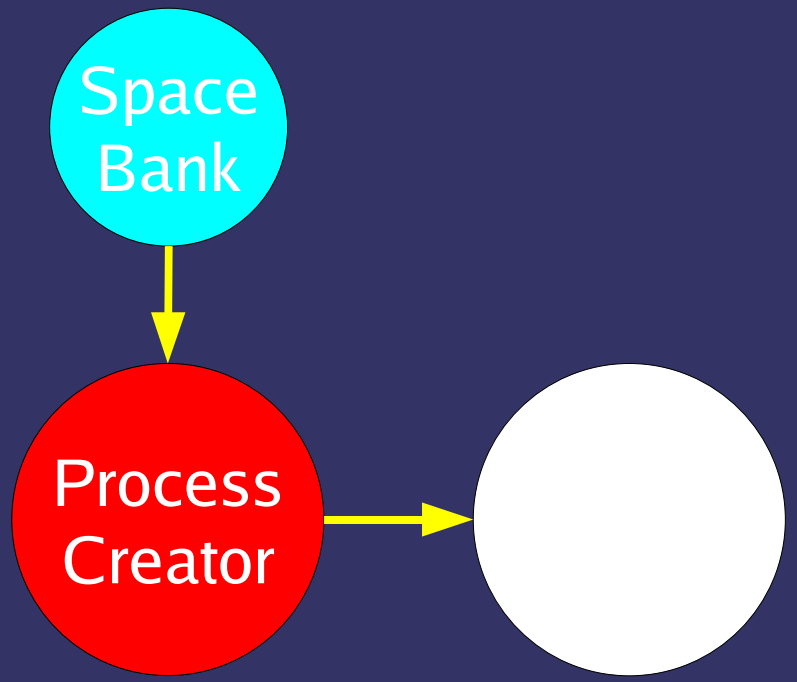
Space Bank

- Allocates nodes, pages
- Arranged in hierarchy
  - Rooted at “prime bank”
- All banks implemented by a single server
- Managed storage: bank destroy reclaims all allocated space
- Contract: storage allocated by a bank is exclusively held by requester. Will not be given to anyone else.



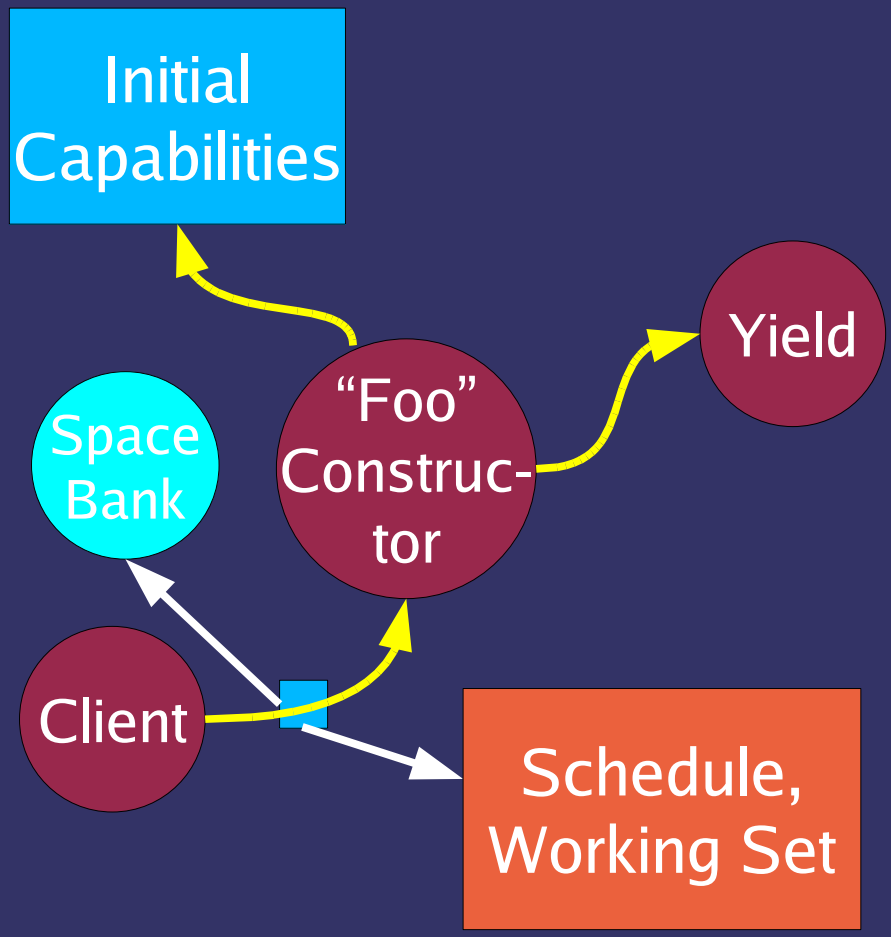
Process Creator

- Given a bank, returns a process allocated from that bank's storage
- A Process Creator can identify the processes that it creates, because they are “branded.”
- This enables us to do *server authentication*:
  - “Is this an *official* space bank?”
  - “Is this the object server I know I can rely on?”
  - Not just the desired interface, but the desired *implementation*.



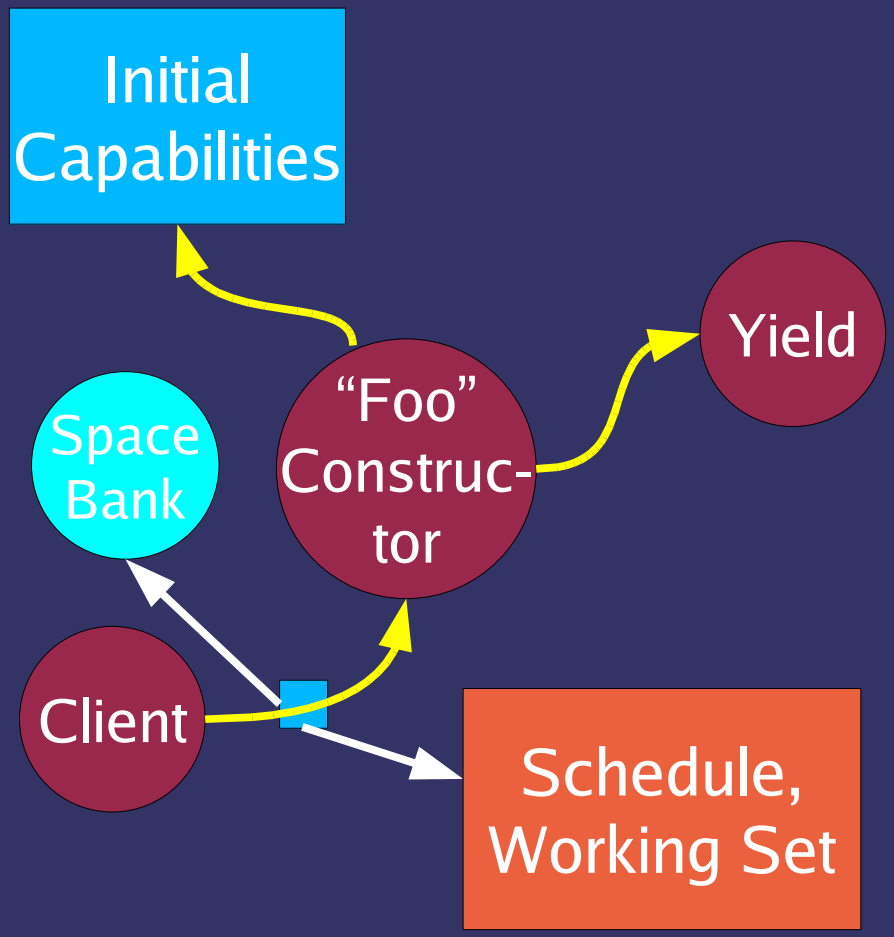
Constructor (Process Instantiation)

- Constructs instances of some program
- Tests for confinement
  - By testing initial capabilities
  - New instance can *only* write to client at creation time.
  - Any further permission must come from client
- Definition is recursive
  - Capability to constructor of confined thing is considered safe
- **Constructor can hold caps that client doesn't**



Constructor: Things to Notice

- Resources are supplied explicitly
  - Concrete resources (pages, nodes)
  - Multiplexing authority (frames, CPU)
- Resource pools are subdivisible, first-class
- Confinement is recursive
- Kill -9 ⇒ destroy subspace



Constructor: Other Operations

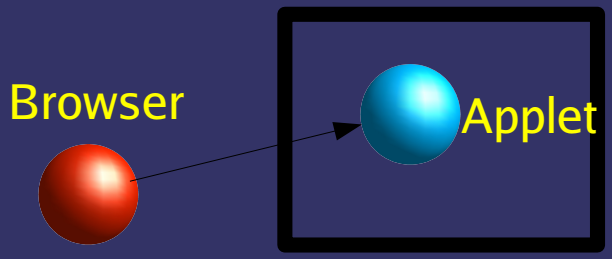
- Authenticate: “Did you create this process?”
  - This can also be used for “identify”
- Design pattern:
  - Your app must invoke a client-supplied capability that is *supposed* to be an X object.
  - Problem: client may not have provided an X object. Safety relies on knowing the *implementation* (not just the interface) of the object
- Solution: arrange that the app already holds (by prior construction) a constructor to X
- Observation: most of these checks turn out to be unnecessary.

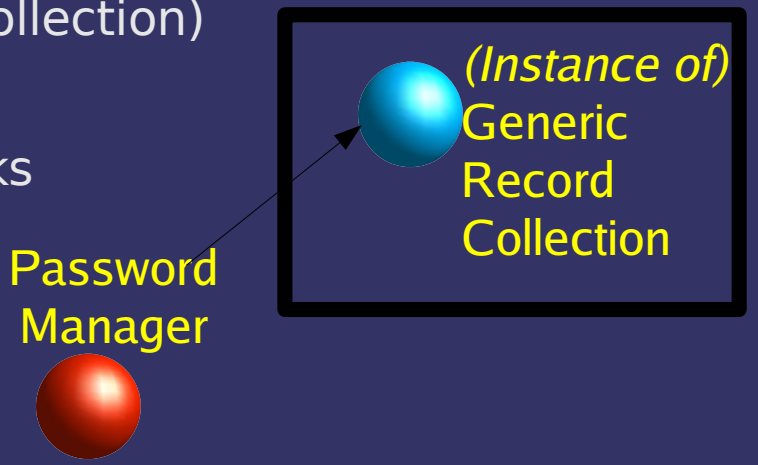
Security Underpinnings

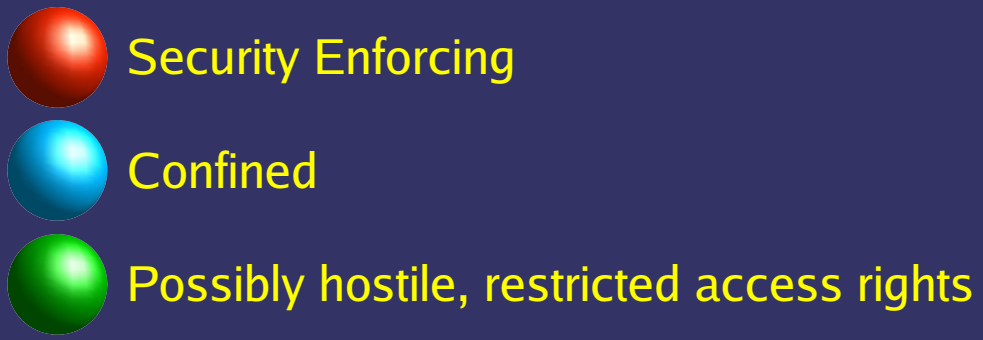
- Technical Features
  - **Confinement**. Applications start with essentially *no* authority and must be given the authority they require. This means that applications cannot disclose *anything* unless you let them. Even a word processor needs to be granted the authority to store files.
  - **Capabilities**: a token of authority. If you want a program to have some authority, you grant a capability to it. No capability, no authority. *There is no "back door" around this mechanism.*
  - **Persistence**: programs run forever, and can therefore implement application-aware security policies (guards)
  - **Fault factoring**: needed to allow guarded sharing of state
- Extensibility
  - Applications can use the same mechanisms that the core system components use.
  - In addition to security, "guards" provide a basis for integrity management
  - Mechanisms for componentwise test and upgrade management

Applications: Confinement

- Running an untrusted applet
  - Don't know what it will do.
  - Need to keep it away from rest of system
- Guard sensitive data
  - Standard “KSAM” component (record collection)
  - Used to store the password database
  - Want to make sure their aren't any leaks

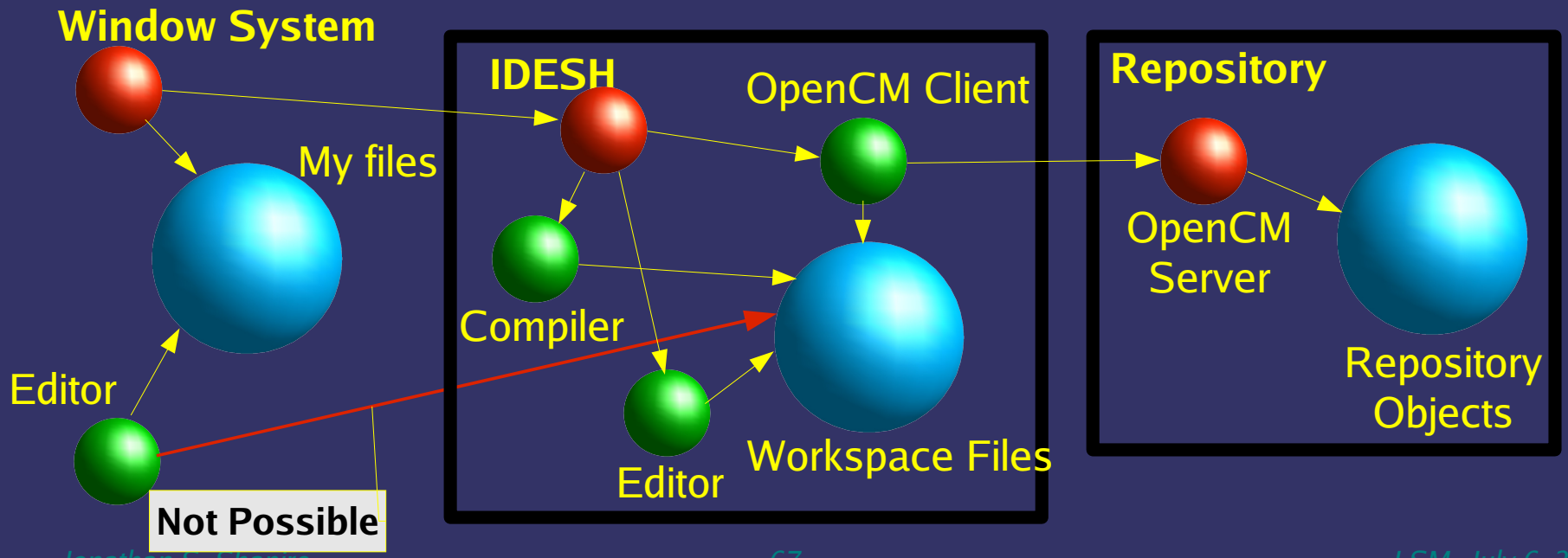






Private File Systems

- EROS does not have a primitive concept of a file system. ***There is no kernel-implemented file system at all.***
- The normal functions of a file system are implemented by application code. *Anybody* can start one of these applications and create a ***private*** file system. Sharing of this file system is subject to the control of policy enforcement tools such as gate keepers.
- Properly used, this means that files in my development workspace cannot be revealed to applications outside the development environment:



Example: SaveAs Guard

Windows “Save As”

- Application says:

```
fileName := SaveFileDialog(...);
fd = open(fileName);
write(fd, ...);
close(fd);
```

- Maybe it's safe, maybe there is a virus: program *could* open *any* file.
- User has no way to know.

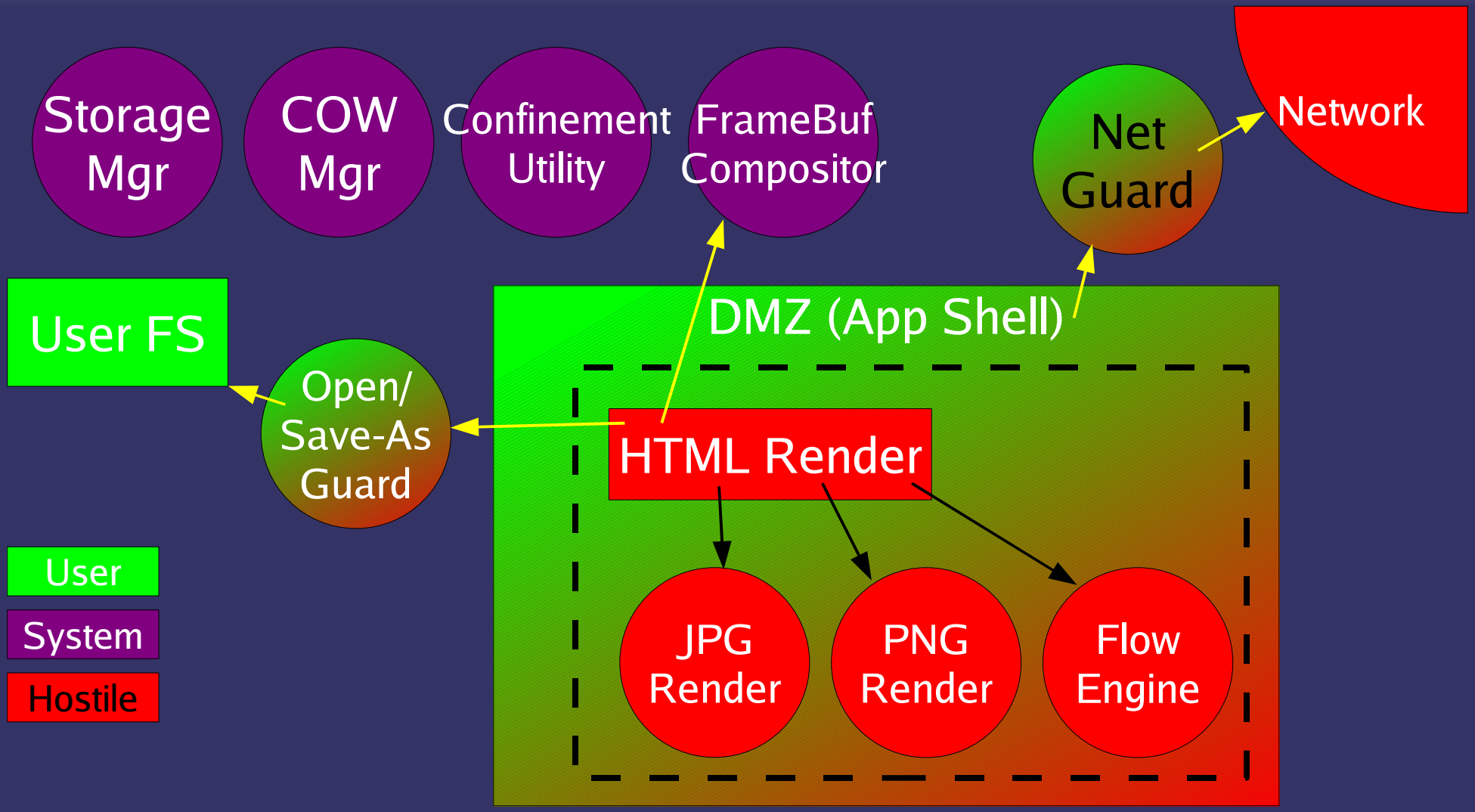
EROS “Save As”

- Application says:

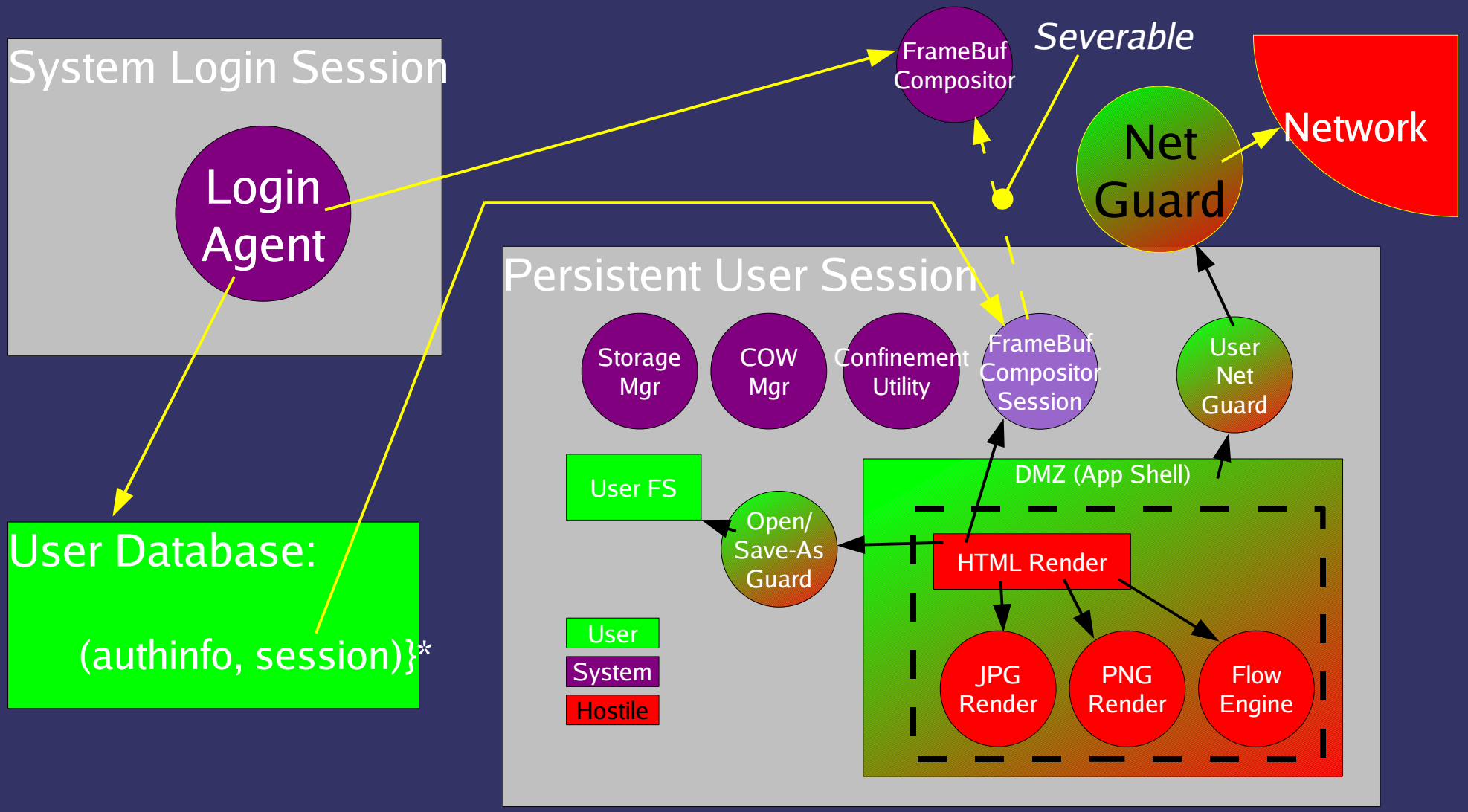
```
int fd = SaveFileDialog(...);
write(fd, ...);
close(fd);
```

- “SaveFile” ***is a separate program***, SaveFileDialog() is a wrapper that invokes it
- SaveFile has:
  - Access to my file system
  - Access to me (so I can click)
- Application has:
  - Access to SaveFile
- **There is no way** the application can write a file without talking to SaveAs.

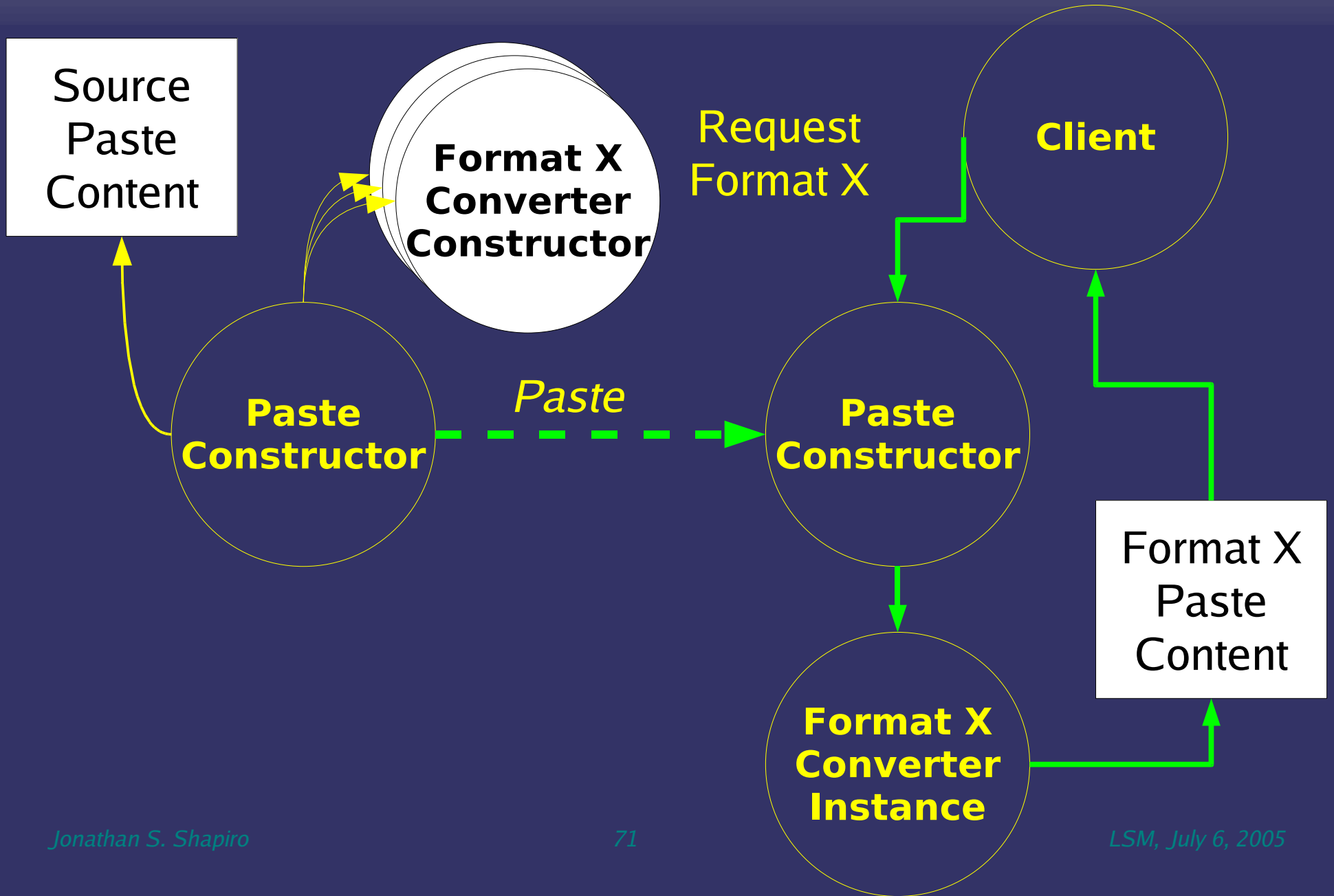
Browser Defenses



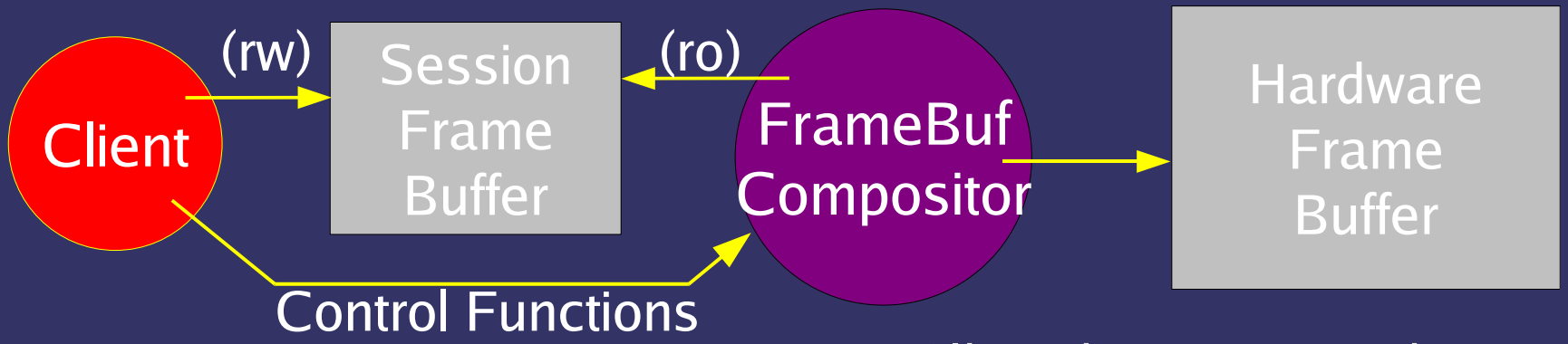
Login/Session Relationships



Unidirectional Cut&Paste Channel



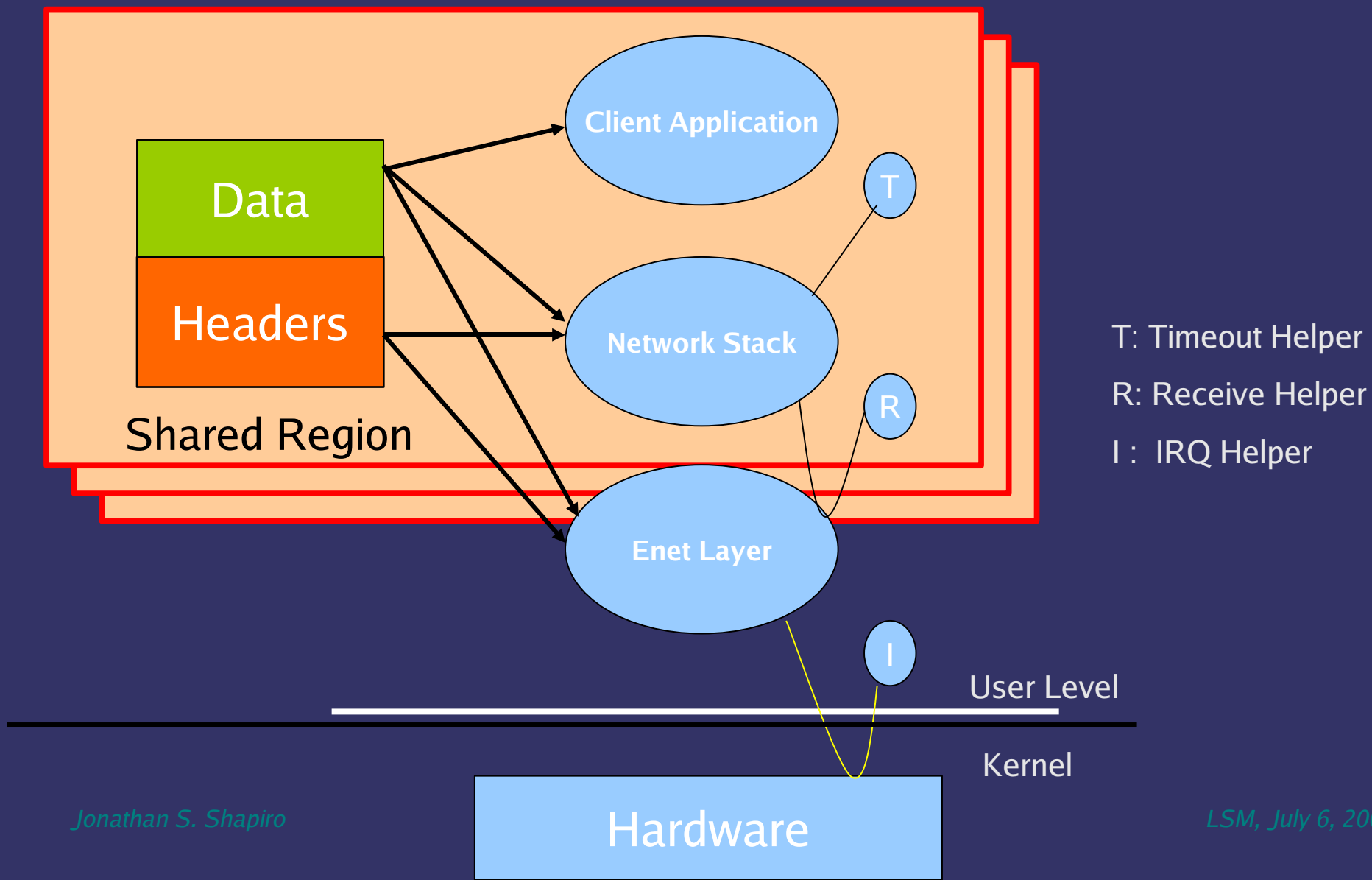
Window System Structure

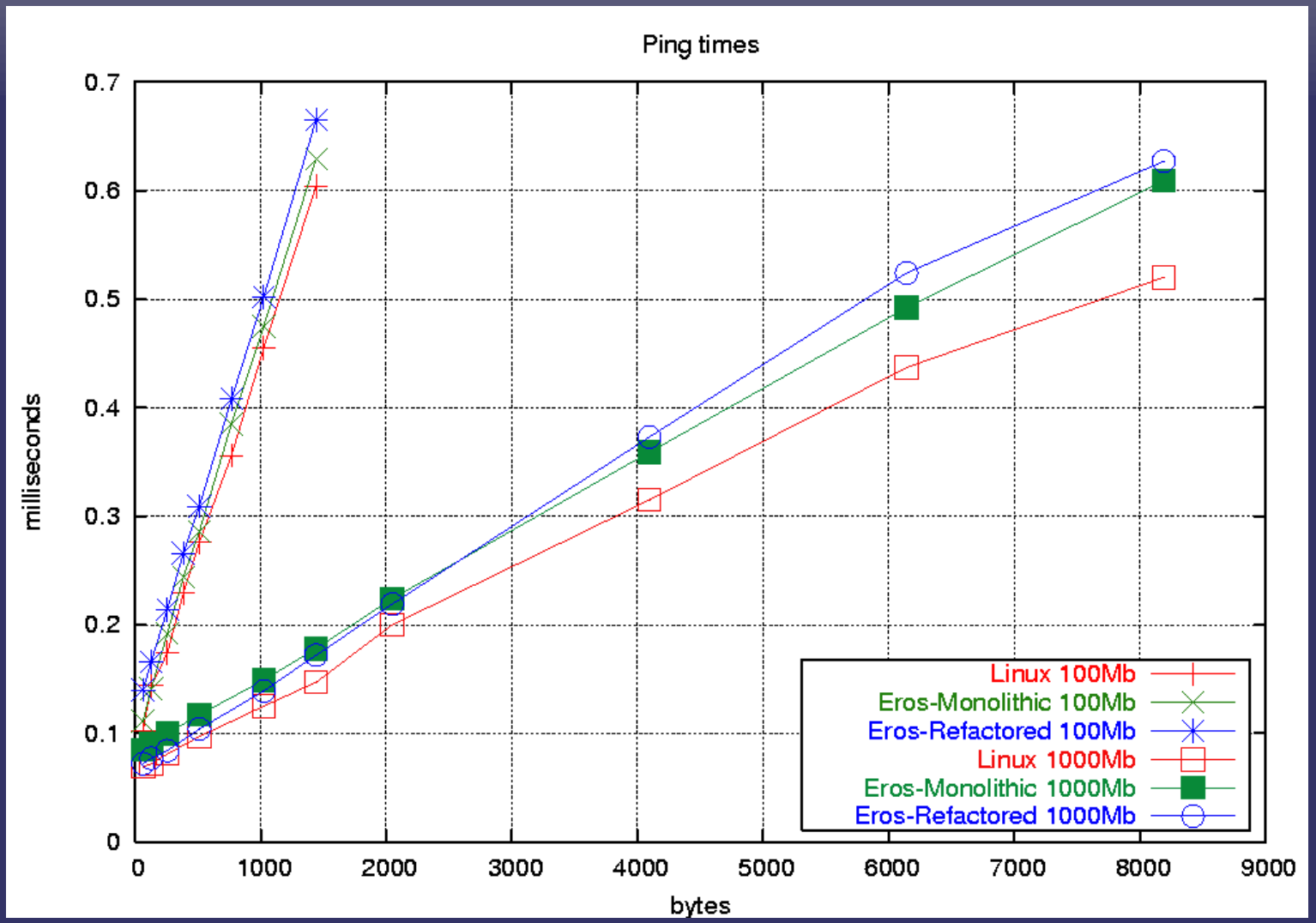


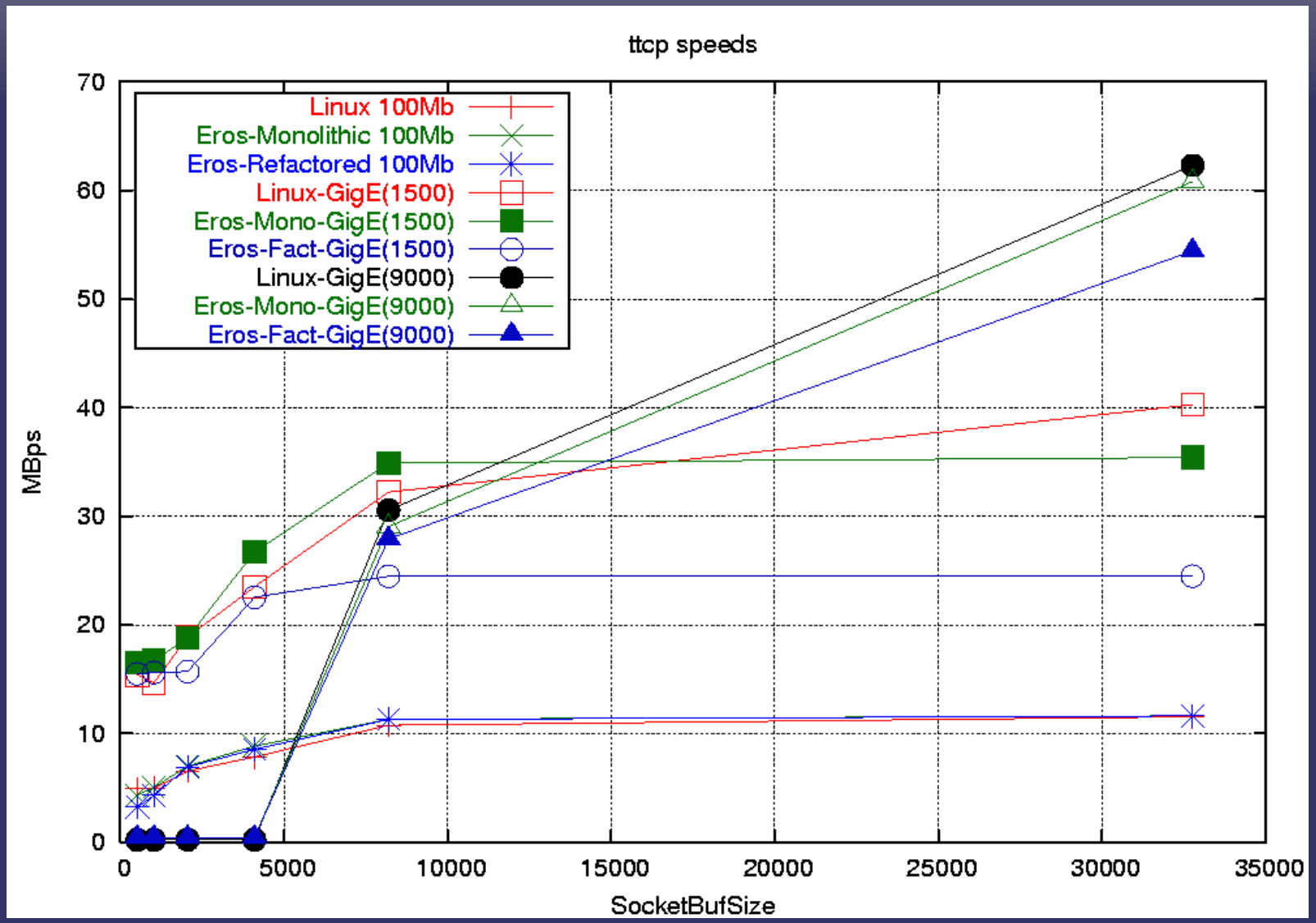
- Compositor can always read window frame buffer, else client has defected
- Compositor maintains per-client session
- Main compositor operation is BitBlt

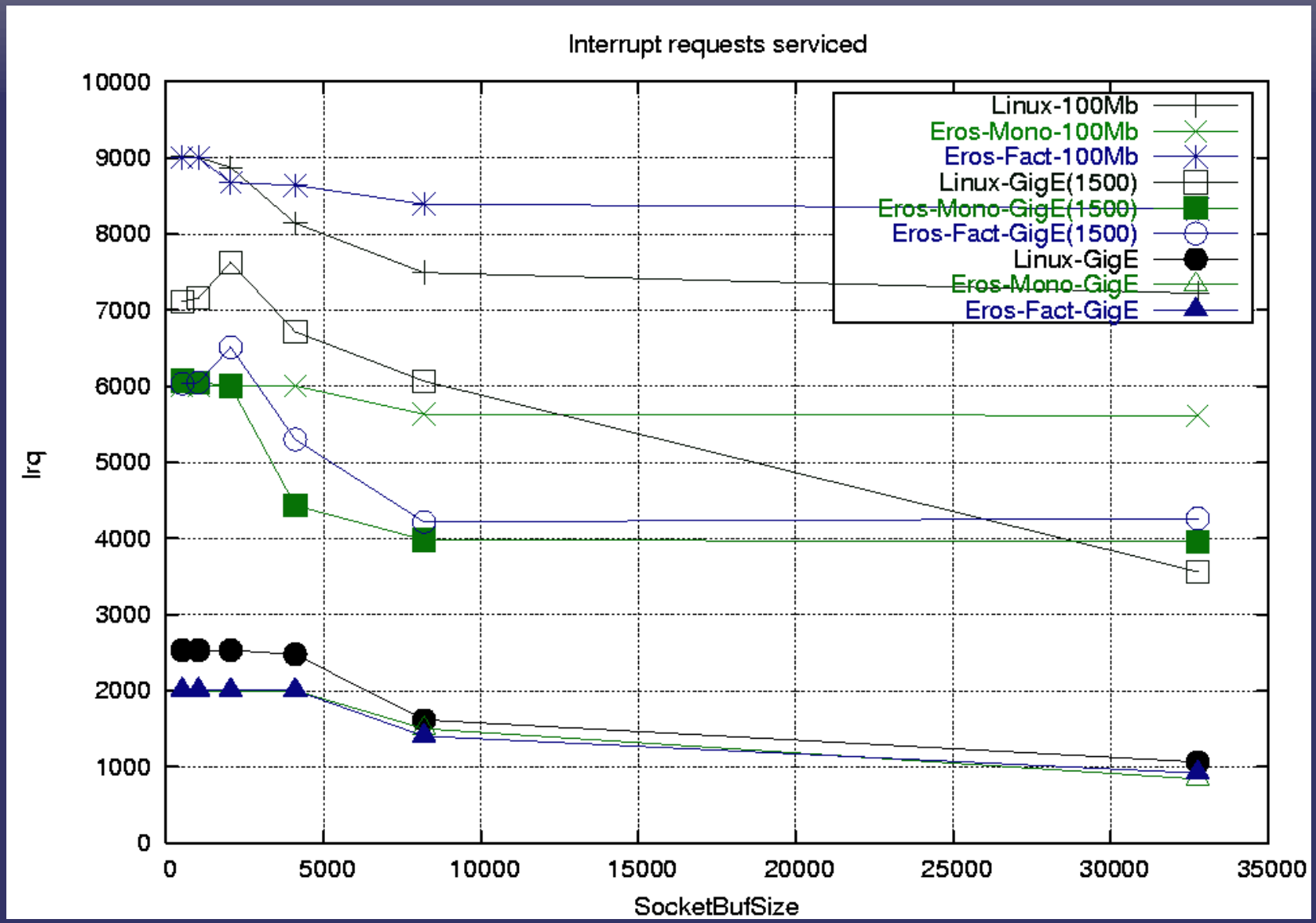
- Client has per-session top level window
- Client does all rendering, notifies compositor of “update region”
- Second client thread performs “WaitNextEvent()
- X11 can be done as application

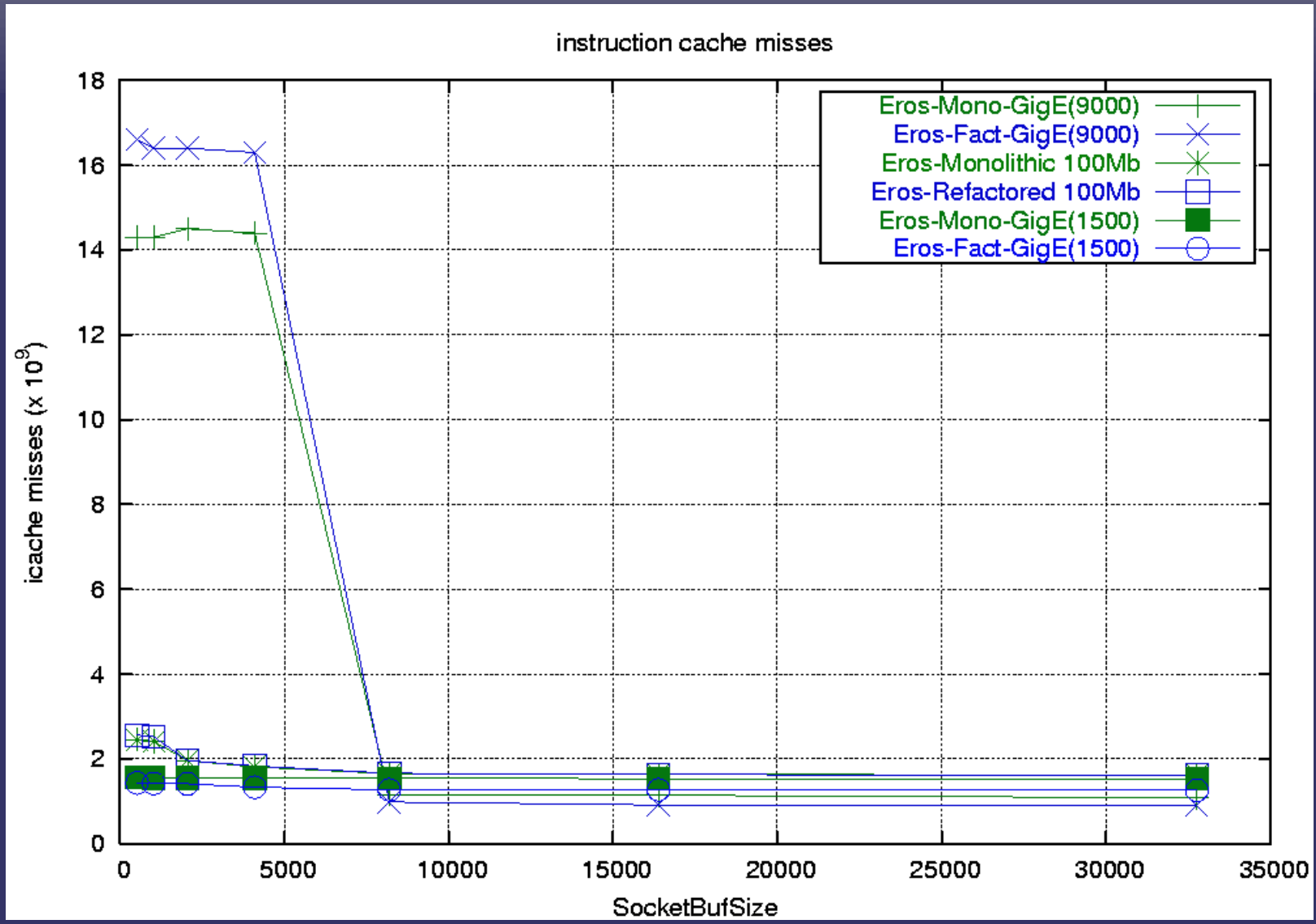
EROS Domain Factored Network

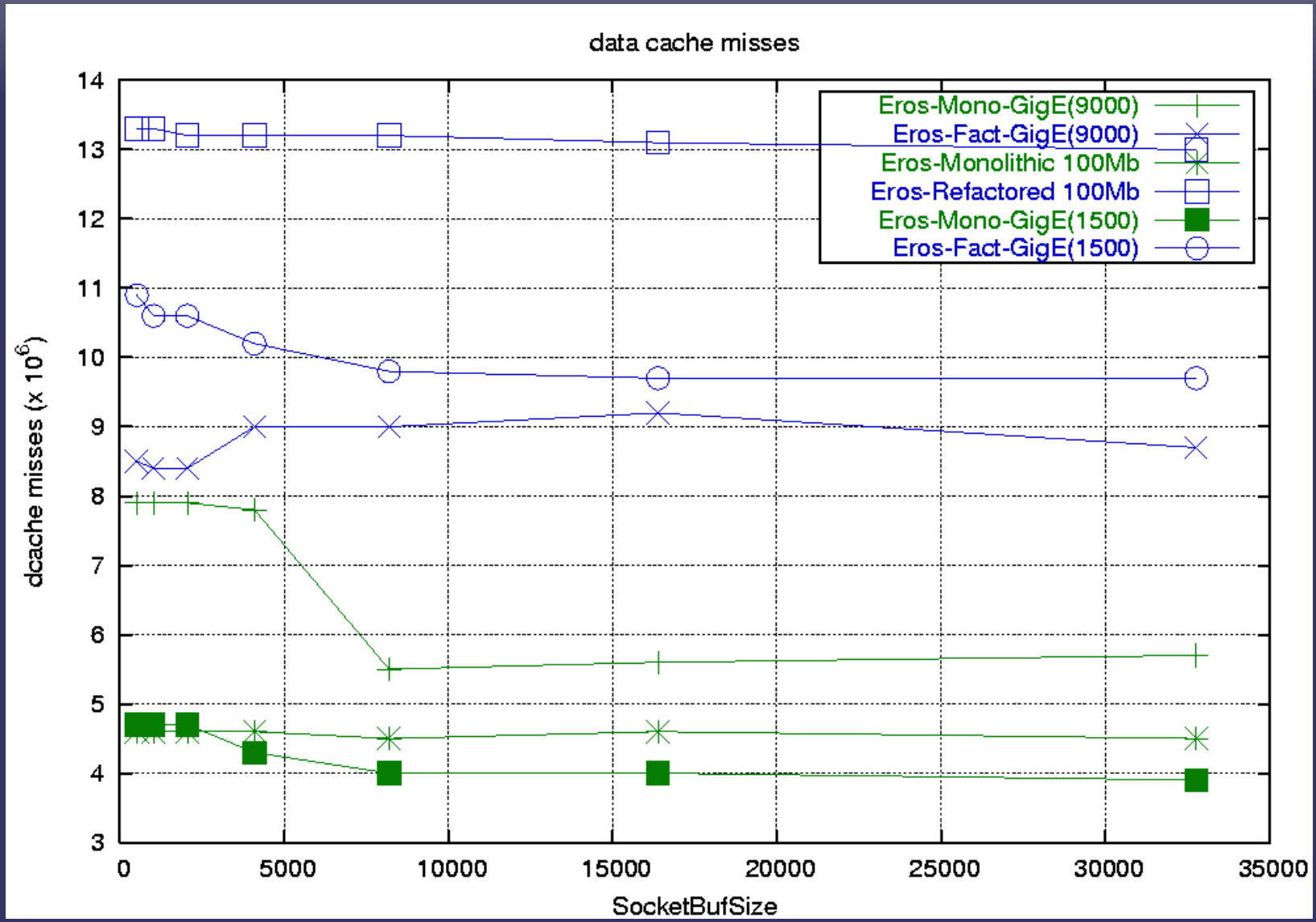



Ping times
milliseconds
bytes
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
Linux 100Mb
Eros-Monolithic 100Mb
Eros-Refactored 100Mb
Linux 1000Mb
Eros-Monolithic 1000Mb
Eros-Refactored 1000Mb

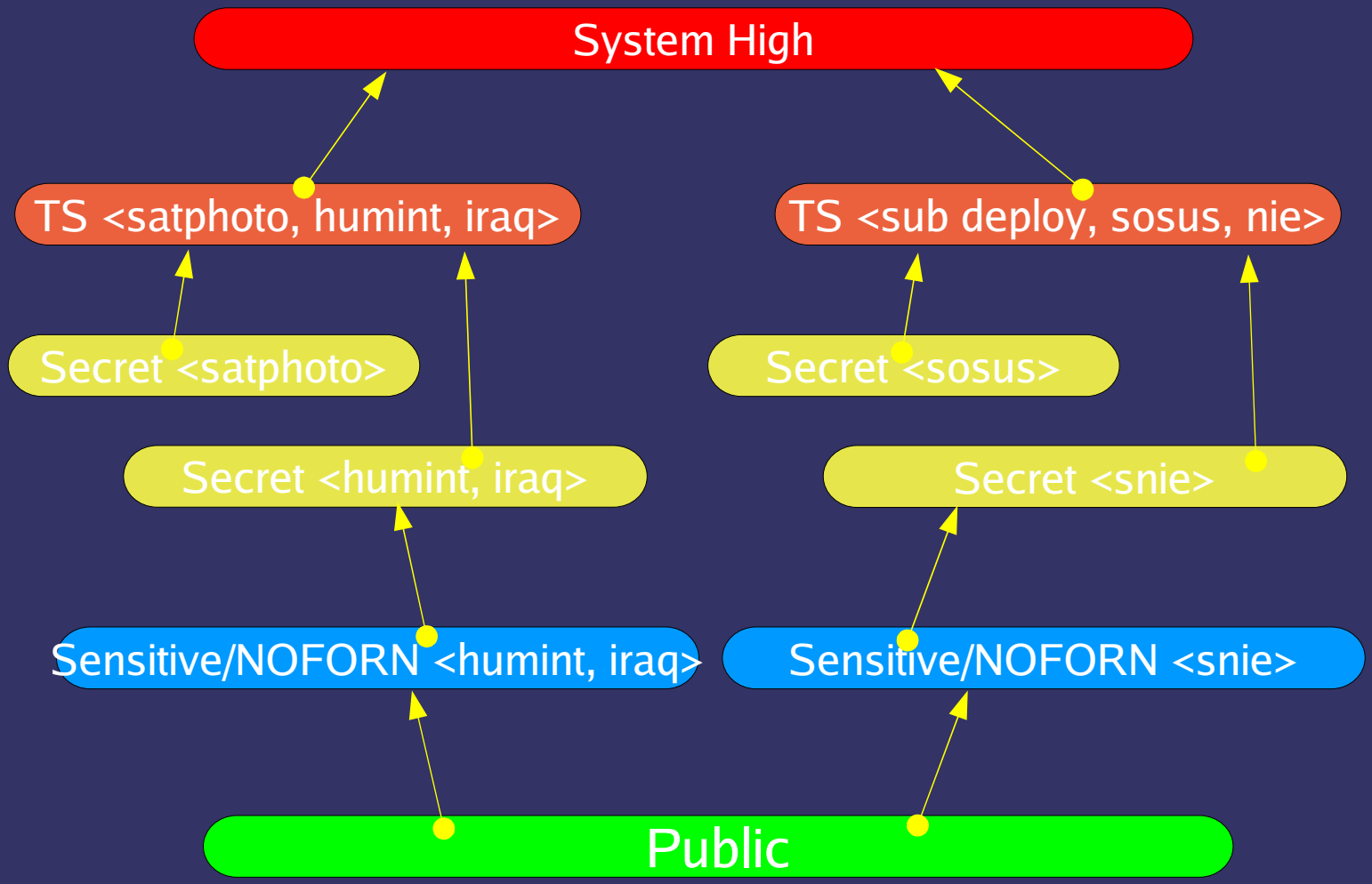

ttcp speeds
Linux 100Mb
Eros-Monolithic 100Mb
Eros-Refactored 100Mb
Linux-GigE(1500)
Eros-Mono-GigE(1500)
Eros-Fact-GigE(1500)
Linux-GigE(9000)
Eros-Mono-GigE(9000)
Eros-Fact-GigE(9000)
MBps
SocketBufSize
0
10
20
30
40
50
60
70
0
5000
10000
15000
20000
25000
30000
35000


Interrupt requests serviced
Irq
SocketBufSize
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
5000
10000
15000
20000
25000
30000
35000
Linux-100Mb
Eros-Mono-100Mb
Eros-Fact-100Mb
Linux-GigE(1500)
Eros-Mono-GigE(1500)
Eros-Fact-GigE(1500)
Linux-GigE
Eros-Mono-GigE
Eros-Fact-GigE


instruction cache misses
Eros-Mono-GigE(9000)
Eros-Fact-GigE(9000)
Eros-Monolithic 100Mb
Eros-Refactored 100Mb
Eros-Mono-GigE(1500)
Eros-Fact-GigE(1500)
icache misses (x 10^9)
SocketBufSize
0
2
4
6
8
10
12
14
16
18
0
5000
10000
15000
20000
25000
30000
35000


data cache misses
dcache misses (x 10^6)
SocketBufSize
Eros-Mono-GigE(9000)
Eros-Fact-GigE(9000)
Eros-Monolithic 100Mb
Eros-Refactored 100Mb
Eros-Mono-GigE(1500)
Eros-Fact-GigE(1500)
14
13
12
11
10
9
8
7
6
5
4
3
0
5000
10000
15000
20000
25000
30000
35000

Mandatory Controls in EROS



Application-Level Reference Monitor (Trusted)

What EROS Got Wrong

Architectural Issues

- Weak support for multithreading
  - Needed first-class communication endpoints
- No non-blocking event mechanism
  - Cost 15% of gigabit networking performance
- Memory mapping structure could have been better
  - Coyotos: Nodes ⇒ PATTs

Planning/Acceptance/Security Issues

- Underestimated the need for a UNIX layer
  - Did not want to repeat the Mach mistake
  - Failed to understand just how badly designed the *autoconf* system really is
- Didn’t start porting applications early enough!
- Source verification wasn’t feasible in 1991

Future Directions

2004 L4 Summit Meeting

- January 2004
- L4ng will be a capability system
  - Now provides descriptors for all system resources
- L4ng is borrowing substantially all of the things that worked in EROS, independently arrived at some of the same problems and solutions that we did.
- Strong collaboration between EROS and L4 groups
- Extended “team” includes several groups interested in formal verification.

L4ng/Coyotos Status

- L4ng/Coyotos largely have a common resource model
- One remaining fundamental disagreement: *Should capabilities be value types*
  - Impacts the security model (issue in capability authentication)
  - Impacts the revocation model (issue with untrusted intermediary)
  - Impacts resource allocation and resource faults
  - Impacts time to completion (big change relative to L4)
- Coyotos team has decided to stick with capabilities as value types in this round
  - We are engaged in kernel refinement, not kernel rearchitecture

Questions