

Deep Specifications and Certified Abstraction Layers



Ronghui Gu Jérémie Koenig Tahina Ramananandro **Zhong Shao**
Newman Wu Shu-Chun Weng Haozhong Zhang¹ Yu Guo¹

Yale University

¹University of Science and Technology of China

January 17, 2015

<http://flint.cs.yale.edu>

Motivation

How to build reliable & secure **system software stacks**?

The image shows a Google search result for "system software stacks". The search results page includes a grid of various diagrams and articles related to system software stacks. The diagrams illustrate different layers and components of system software stacks, such as:

- Android Framework:** A diagram showing the layers of the Android operating system, including the Linux kernel, HAL, and various system services.
- Cray HPC Cluster Software Stack:** A diagram showing the software stack for a high-performance computing cluster, including the operating system, middleware, and applications.
- Software Stack:** A diagram showing the layers of a software stack, including the operating system, device drivers, and applications.
- TI Device:** A diagram showing the software stack for a Texas Instruments device, including the kernel, drivers, and applications.
- Application Framework:** A diagram showing the layers of an application framework, including the application framework, libraries, and applications.
- USB Device:** A diagram showing the software stack for a USB device, including the USB device driver, USB device controller, and applications.
- Computer:** A diagram showing the software stack for a computer, including the operating system, device drivers, and applications.
- TI Device (Kernel Mode):** A diagram showing the software stack for a TI device in kernel mode, including the kernel, drivers, and applications.
- TI Device (User Mode):** A diagram showing the software stack for a TI device in user mode, including the application framework, libraries, and applications.
- TI Device (Hardware):** A diagram showing the hardware components of a TI device, including the microcontroller unit (MCU), memory, and peripherals.

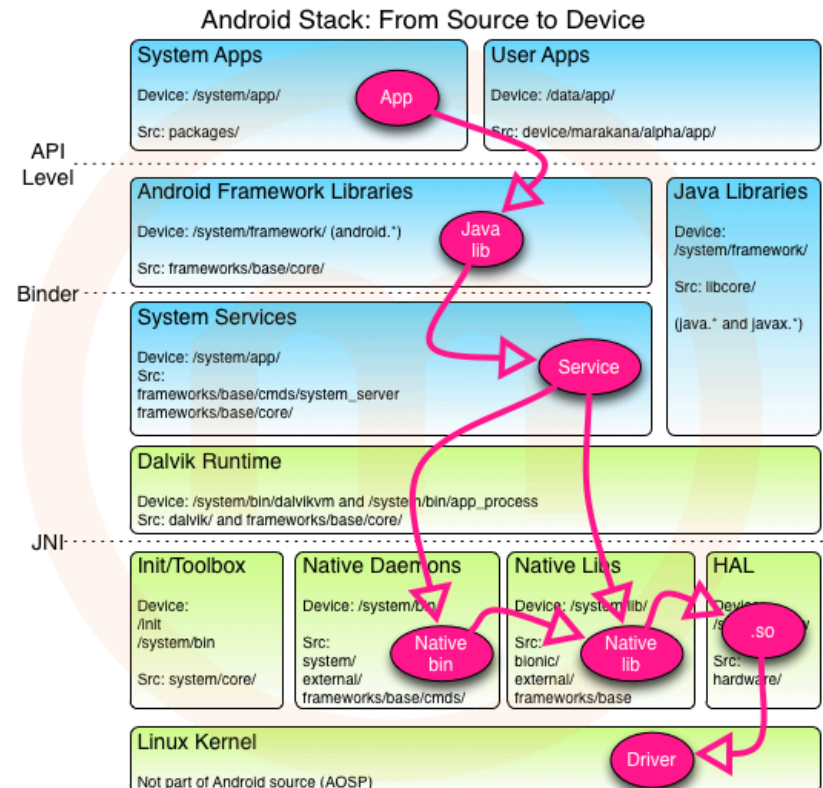
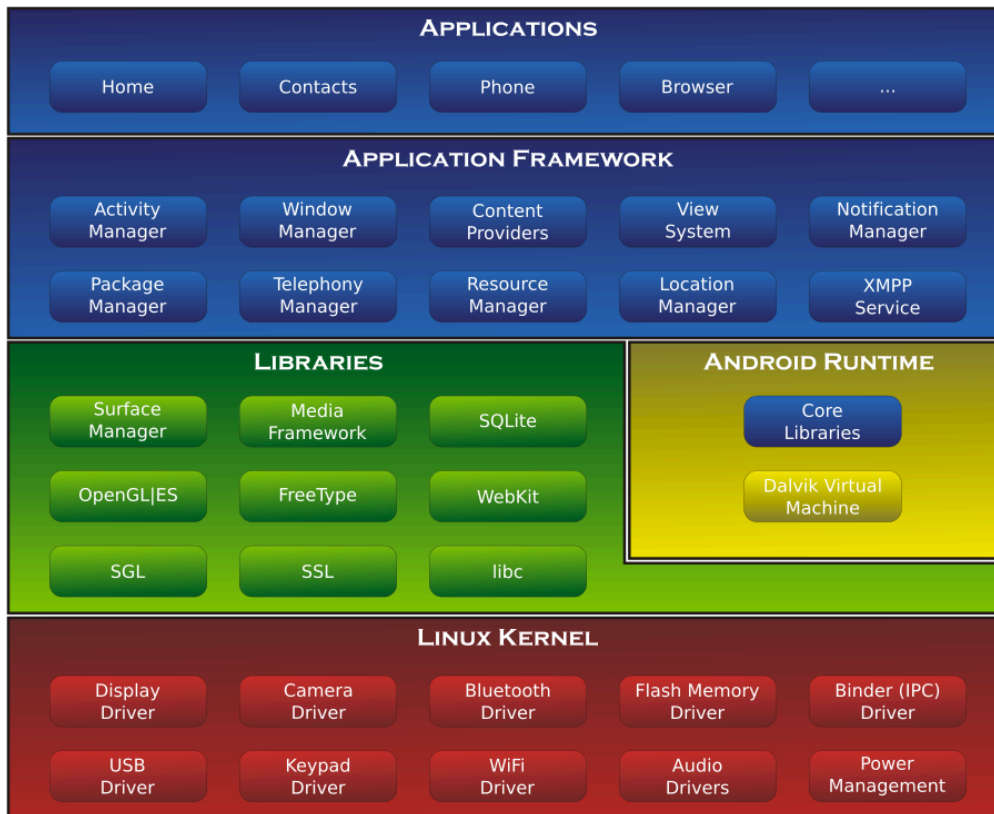
The search results also include several articles and documents, such as:

- "Meaning of 'Multicore-Ready'"
- "LabVIEW Support"
- "Software Stack Development tool"
- "Device drivers"
- "Operating system"
- "USB Device"
- "Computer"
- "TI Device"

Motivation

Android architecture & system stack

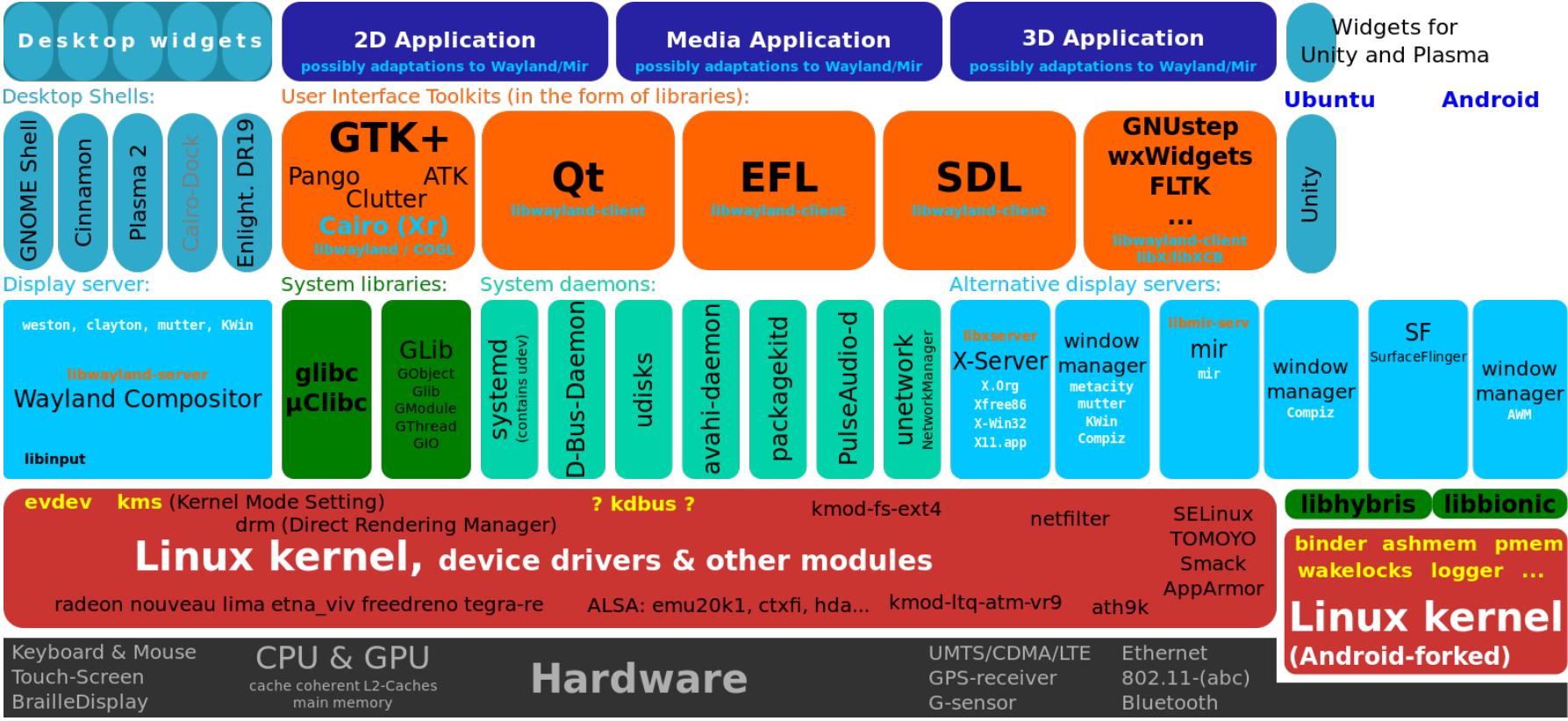
From https://thenewcircle.com/s/post/1031/android_stack_source_to_device & [http://en.wikipedia.org/wiki/Android_\(operating_system\)](http://en.wikipedia.org/wiki/Android_(operating_system))



Motivation

Visible software components of the Linux desktop stack

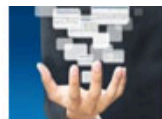
From <http://en.wikipedia.org/wiki/Linux>



Motivation

Software stack for HPC clusters

From <http://www.hpcwire.com/2014/02/24/comprehensive-flexible-software-stack-hpc-clusters/>



Essential Software and Management Tools Needed to Build a Powerful, Flexible, and Highly Available Supercomputer.

 **HPC Programming Tools**

Performance Monitoring	HPCC	Perfctr	IOR	PAPI/IPM	netperf
Development Tools	Cray® Compiler Environment (CCE)	Intel® Cluster Studio	PGI (PGI CDK)	GNU	
Application Libraries	Cray® LibSci, LibSci_ACC	MVAPICH2	OpenMPI	Intel® MPI- (Cluster Studio)	

 **Middleware Applications and Management**

Resource Management / Job Scheduling	SLURM	Grid Engine	MOAB	Altair PBS Pro	IBM Platform LSF	Torque/Maui
File System	NFS	Local FS (ext3, ext4, XFS)	PanFS	Lustre		
Provisioning	Cray® Advanced Cluster Engine (ACE) management software					
Cluster Monitoring	Cray ACE (iSCB and OpenIPMI)					
Remote Power Mgmt	Cray ACE					
Remote Console Mgmt	Cray ACE					

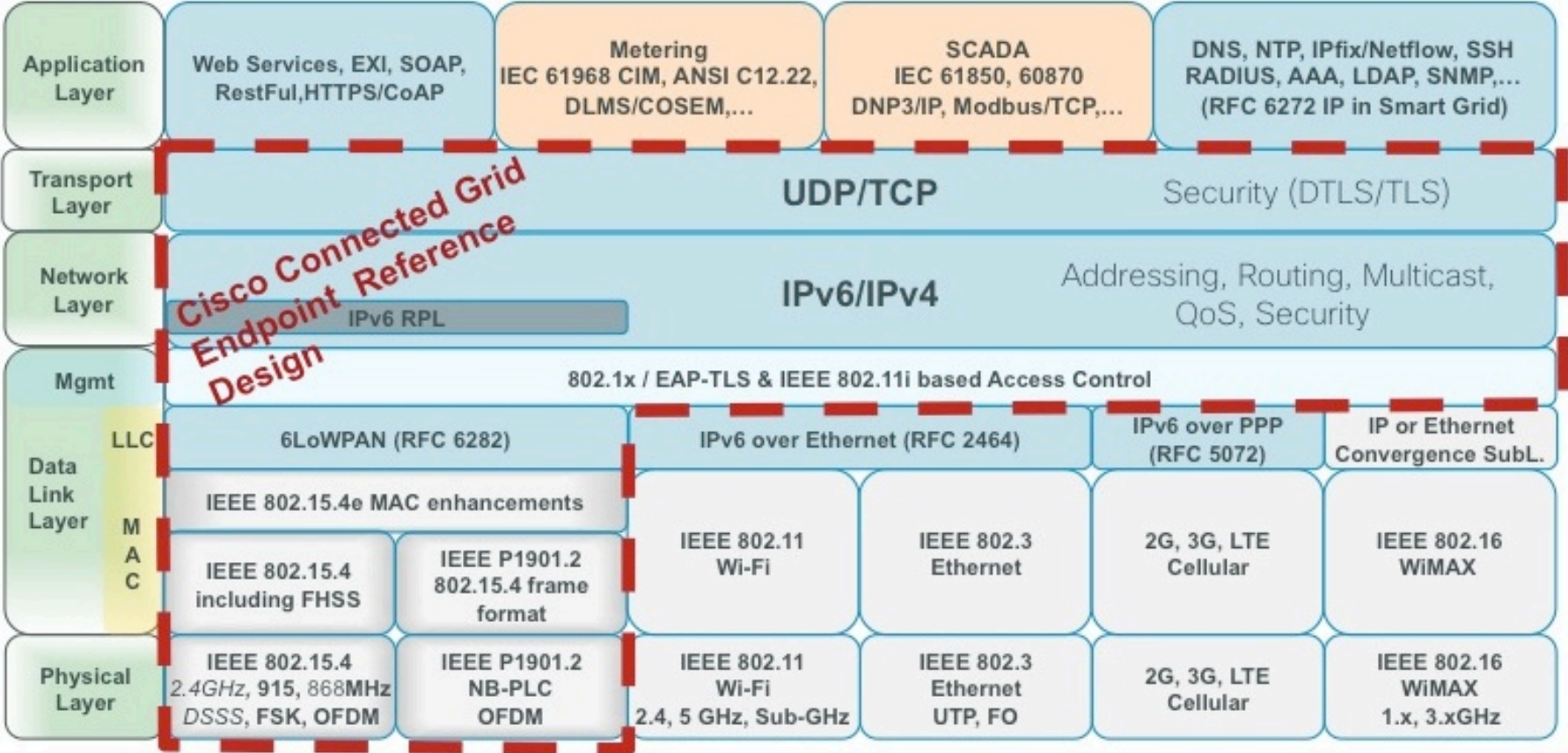
 **Operating Systems**

Operating System	Linux (Red Hat, CentOS, SUSE)
------------------	-------------------------------

Motivation

Cisco's FAN (Field-Area-Network) protocol layering

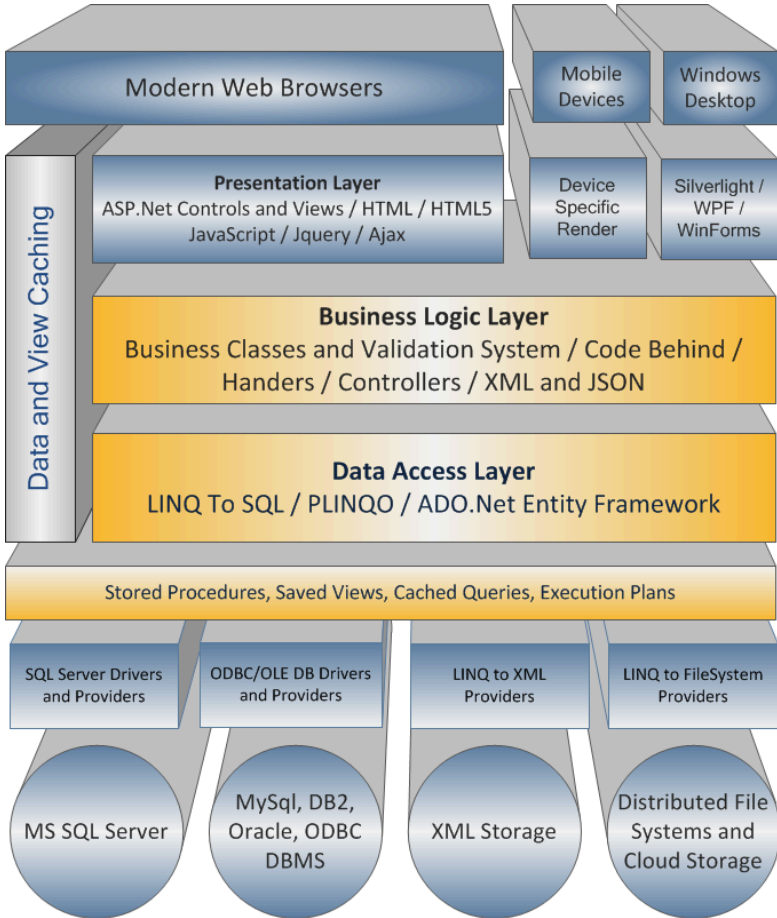
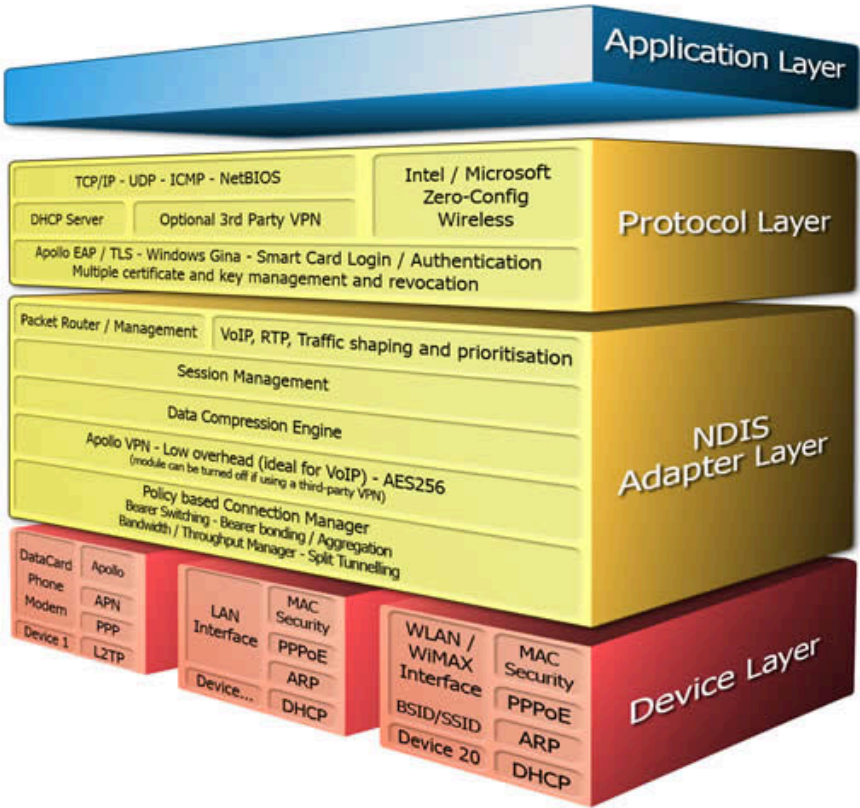
From <https://solutionpartner.cisco.com/web/cegd/overview>



Motivation

Apollo Mobile Communication Stack
http://www.layer2connections.com/apollo_clients.html

Web Application Development Stack
 From <http://www.brightware.co.uk/Technology.aspx>

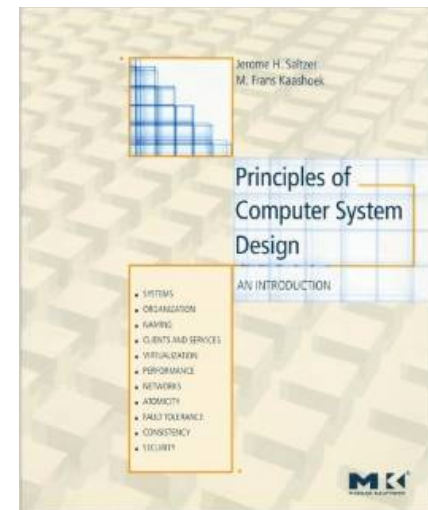
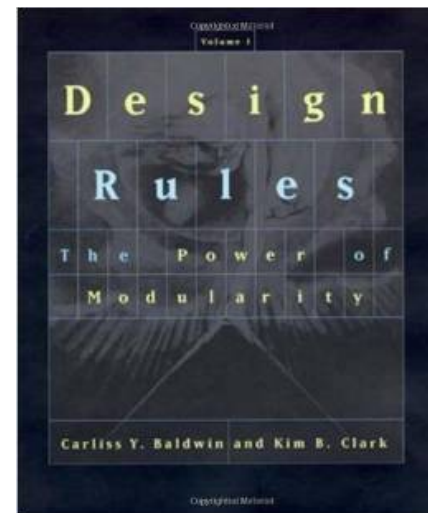


Motivation (cont'd)

- Common themes: all system stacks are built based on abstraction, modularity, and layering
- Abstraction layers are ubiquitous!

Such use of abstraction, modularity, and layering is “**the key factor that drove the computer industry toward today’s explosive levels of innovation and growth** because *complex products can be built from smaller subsystems that can be designed independently yet function together as a whole.*”

Baldwin & Clark “Design Rules: Volume 1, The Power of Modularity”, MIT Press, 2000



Do We Understand Abstraction?

In the PL community:

(abstraction in the small)

- Mostly formal but tailored within a single programming language (ADT, objects, existential types)
- Specification only describes type or simple pre- & post condition
- Hide concrete data representation (we get the nice *repr. independence* property)
- Well-formed *typing* or *Hoare-style judgment* between the impl. & the spec.

In the System world:

(abstraction in the large)

- Mostly informal & language-neutral (APIs, sys call libraries)
- Specification describes full functionality (but in English)
- Implementation is a black box (*in theory*); an *abstraction layer* hides all things below
- *The “implements” relation* between the impl. & the spec

Do We Understand Abstraction?

In the PL community:

(abstraction in the small)

- Mostly formal but tailored within a single programming language (ADT, objects, existential types)
- Specification only describes type or simple pre- & post condition
- Hide concrete data representation (we get the nice *repr. independence* property)
- Well-formed *typing* or *Hoare-style judgment* between the impl. & the spec.

In the System world:

(abstraction in the large)

- Mostly informal & language-neutral (APIs, sys call libraries)

Something
magical
going on ...
What is it?

between the impl. & the spec

Problems

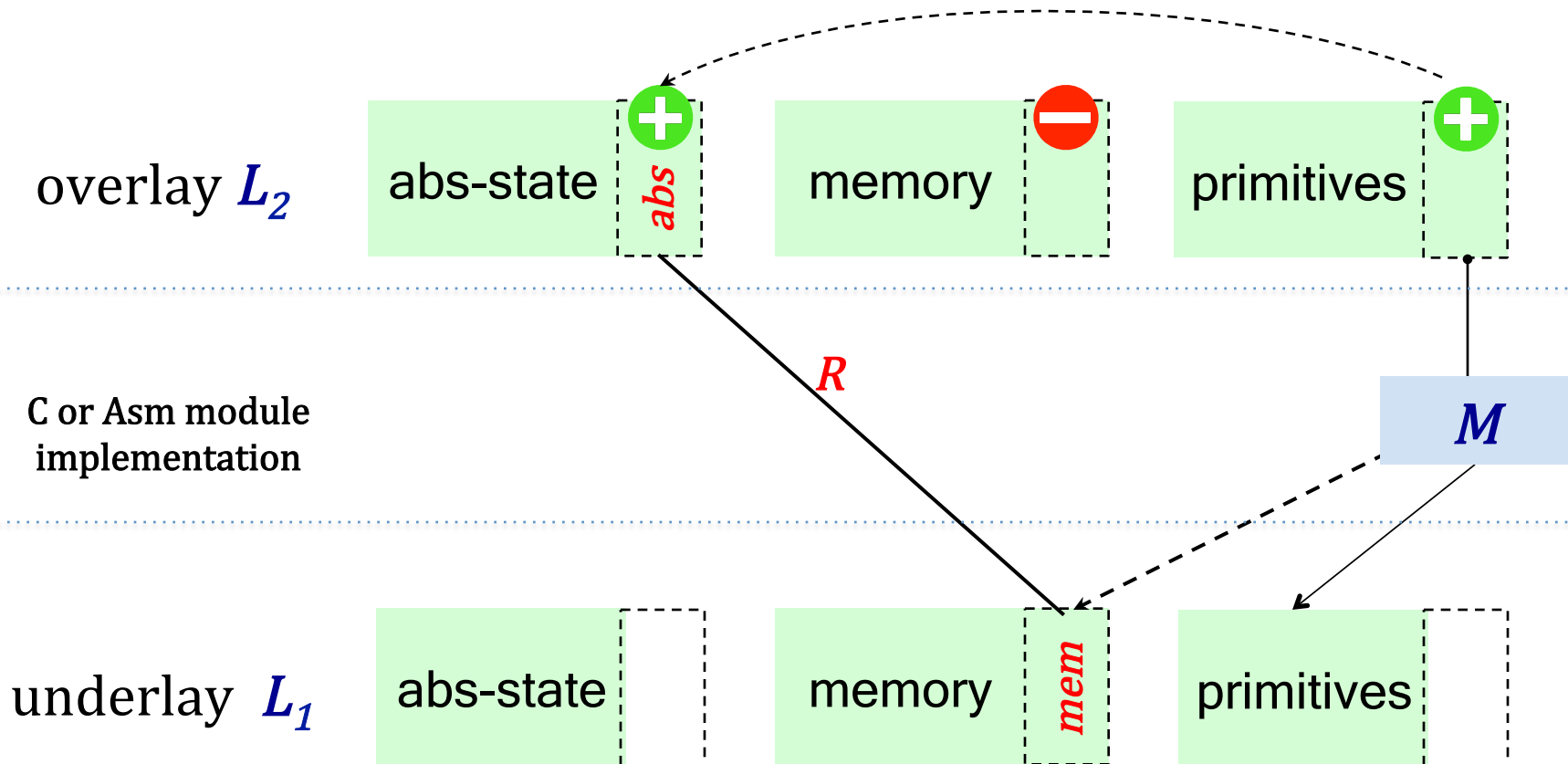
- What is an ***abstraction layer***?
- How to formally ***specify*** an abstraction layer?
- How to ***program***, ***verify***, and ***compile*** each layer?
- How to ***compose*** abstraction layers?
- How to apply ***certified abstraction layers*** to build *reliable* and *secure* system software?

Our Contributions



- We introduce **deep specification** and present a language-based formalization of **certified abstraction layer**
- We developed new languages & tools in Coq
 - **A formal layer calculus** for composing certified layers
 - **ClightX** for writing certified layers in a C-like language
 - **LAsm** for writing certified layers in assembly
 - **CompCertX** that compiles **ClightX** layers into **LAsm** layers
- We built multiple **certified OS kernels** in Coq
 - **mCertiKOS-hyper** consists of **37 layers**, took less than **one-person-year** to develop, and can boot **Linux** as a guest

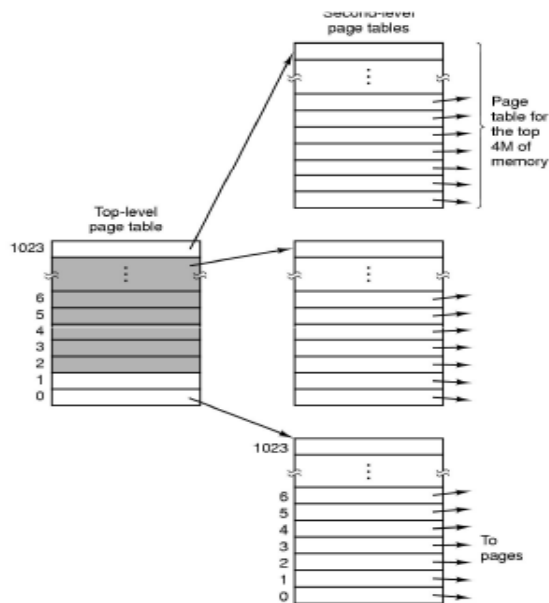
What is an Abstraction Layer?



Example: Page Tables

concrete C types

```
struct PMap {  
  char * page_dir[1024];  
  uint page_table[1024][1024];  
};
```



abstract Coq spec

Inductive **PTPerm**:Type :=

- | PTP
- | PTU
- | PTK.

Inductive **PTEInfo**:=

- | PTEValid (v : Z) (p : **PTPerm**)
- | PTEUnPresent.

Definition **PMap** := ZMap.t **PTEInfo**.

Example: Page Tables

abstract
layer spec

abstract state

$\text{PMap} := \text{ZMap.t PTEInfo}$
 $(* \text{vaddr} \rightarrow (\text{paddr}, \text{perm}) *)$

Invariants: kernel page table is
a direct map; user parts are isolated

abstract primitives

(Coq functions)

Function `page_table_init` = ...
Function `page_table_insert` = ...
Function `page_table_rmv` = ...
Function `page_table_read` = ...

concrete C
implementation

memory

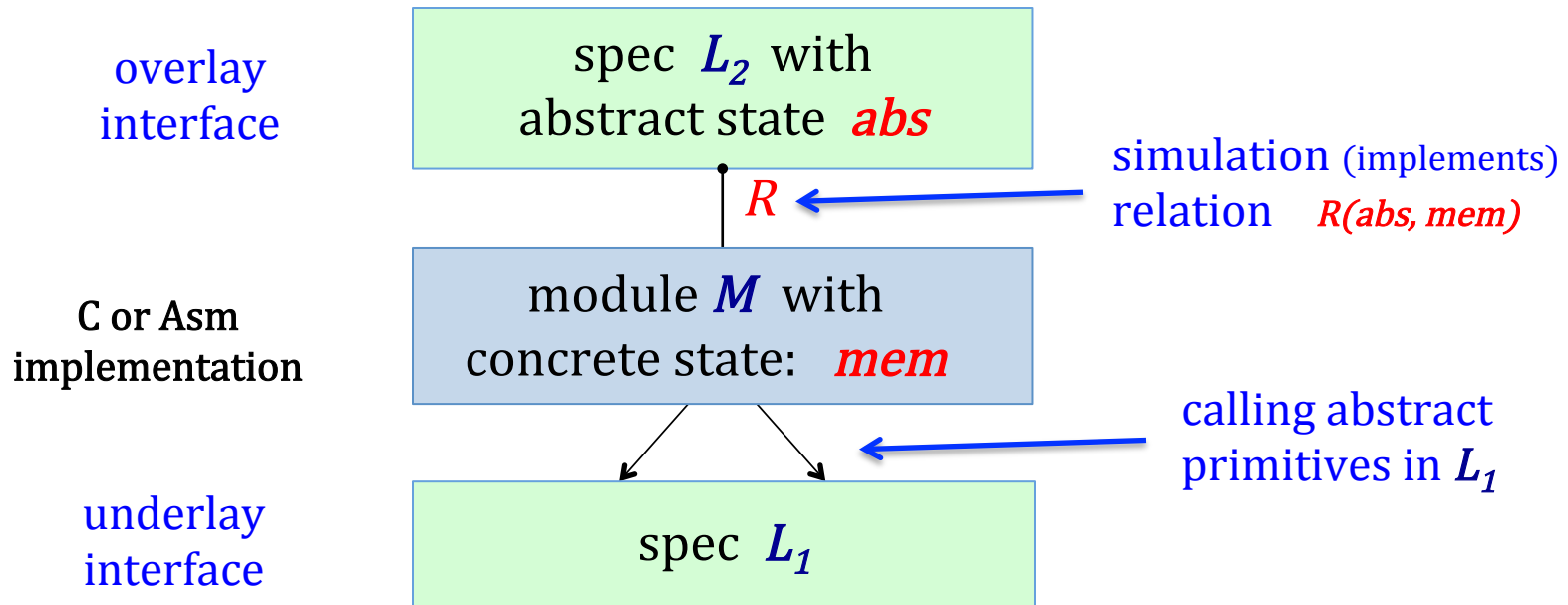
```
char * page_dir[1024];  
  
uint page_table[1024][1024];
```

C functions

```
int page_table_init() { ... }  
int page_table_insert { ... }  
int page_table_rmv() { ... }  
int page_table_read() { ... }
```

Formalizing Abstraction Layers

What is a *certified* abstraction layer (L_1, M, L_2) ?



Recorded as the *well-formed layer* judgment

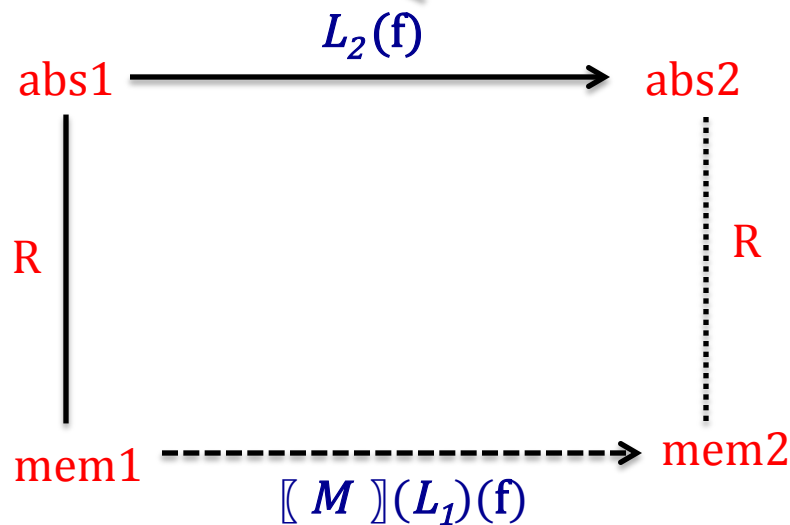
$$L_1 \vdash_R M : L_2$$

The Simulation Relation

compositional
per-module
semantics $\llbracket \bullet \rrbracket$

$$L_1 \vdash_R M : L_2 \quad \longrightarrow \quad L_2 \leq_R \llbracket M \rrbracket L_1$$

for each function f in $\text{Dom}(L_2)$



Forward Simulation:

- Whenever $L_2(f)$ takes abs1 to abs2 in one step, and $R(\text{abs1}, \text{mem1})$ holds,
- then there exists mem2 such that $\llbracket M \rrbracket(L_1)(f)$ takes mem1 to mem2 in zero or more steps, and $R(\text{abs2}, \text{mem2})$ also holds.

Reversing the Simulation Relation

$$L_1 \vdash_R M : L_2 \quad \longrightarrow \quad L_2 \leq_R \llbracket M \rrbracket L_1$$

If $\llbracket M \rrbracket (L_1)$ is
deterministic relative
to external events
(*a la CompCert*)

$$\llbracket M \rrbracket L_1 \leq_R L_2$$

$$\llbracket M \rrbracket L_1 \sim_R L_2$$

$\llbracket M \rrbracket (L_1)$ and L_2 are bisimilar!

L_2 captures everything about running M over L_1

Deep Specification

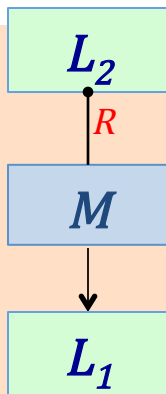
$$\llbracket M \rrbracket L_1 \sim_R L_2$$

$\llbracket M \rrbracket (L_1)$ and L_2 are bisimilar!

L_2 captures everything about running M over L_1

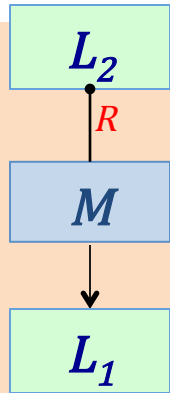


Making it “contextual” using
the whole-program semantics $\llbracket \bullet \rrbracket$



L_2 is a **deep specification** of M over L_1
if under any **valid** program context P of L_2 ,
 $\llbracket P \oplus M \rrbracket (L_1)$ and $\llbracket P \rrbracket (L_2)$ are
observationally equivalent

Why Deep Spec is Really Cool?



L_2 is a **deep specification** of M over L_1
if under any valid program context P of L_2 ,
 $\llbracket P \oplus M \rrbracket (L_1)$ and $\llbracket P \rrbracket (L_2)$ are
observationally equivalent

Deep spec L captures all we need to know about a layer M

- No need to ever look at M again!
- Any property about M can be proved using L alone.


Impl. Independence : any two implementations of the same deep spec are *contextually equivalent*

Is Deep Spec Too Tight?

- **Not really!** It still *abstracts* away:
 - the *efficient* concrete data repr & impl. algorithms & strategies
- It can still be **nondeterministic**:
 - **External nondeterminism** (e.g., I/O or scheduler events) modeled as a set of **deterministic traces** relative to external events (*a la CompCert*)
 - **Internal nondeterminism** (e.g., sqrt, rand, resource-limit) is also OK, but any *two* implementations must still be *observationally equivalent*
- It *adds* new logical info to make it *easier-to-reason-about*:
 - auxiliary **abstract states** to define the full functionality & invariants
 - accurate **precondition** under which each primitive is valid

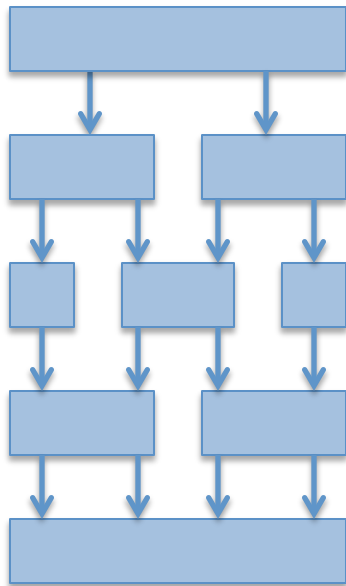
Problem w. Shallow Specs

 C or Asm module

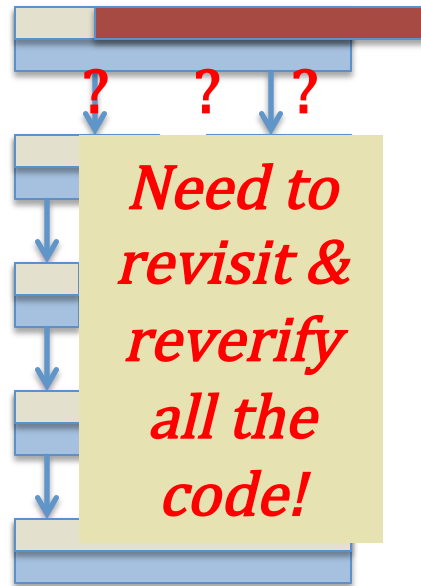
 shallow spec A

 shallow spec B

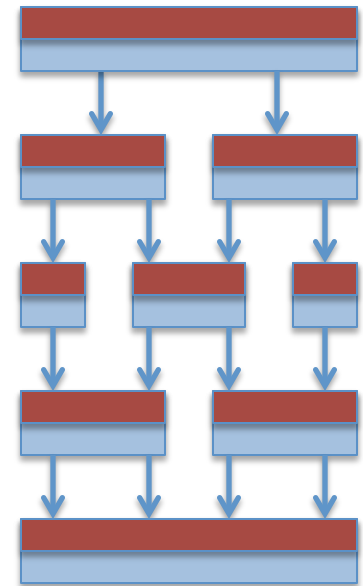
C & Asm Module
Implementation



C & Asm Modules
w. Shallow Spec A





*Want to prove
another spec B?*



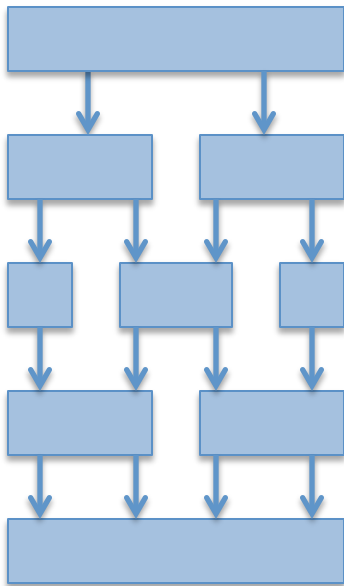
Shallow vs. Deep Specifications

 C or Asm module

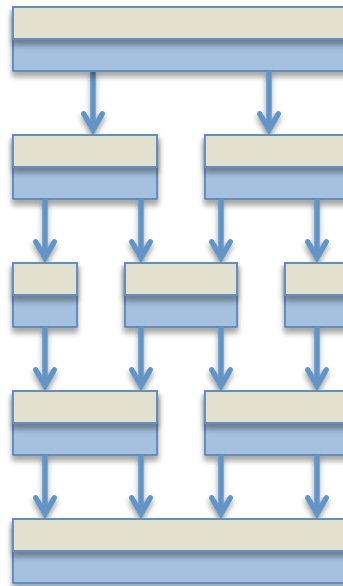
 shallow spec

 deep spec

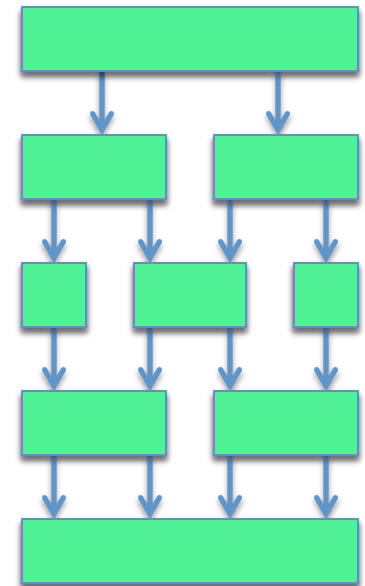
C & Asm Module Implementation



C & Asm Modules w. Shallow Specs



C & Asm Modules w. Deep Specs



How to Make Deep Spec Work?

No languages/tools today support deep spec & certified layered programming

Challenges:

- **Implementation** done in C or assembly or ...
- **Specification** done in richer logic (e.g., Coq)
- Need to mix **both** and also simulation proofs
- Need to compile C layers into assembly layers
- Need to compose different layers

Our Contributions



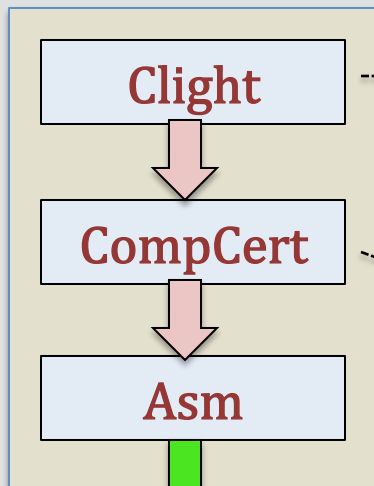
- We introduce **deep specification** and present a language-based formalization of **certified abstraction layer**
- We developed new languages & tools in Coq
 - **A formal layer calculus** for composing certified layers
 - **ClightX** for writing certified layers in a C-like language
 - **LAsm** for writing certified layers in assembly
 - **CompCertX** that compiles **ClightX** layers into **LAsm** layers
- We built multiple **certified OS kernels** in Coq
 - **mCertiKOS-hyper** consists of **37 layers**, took less than **one-person-year** to develop, and can boot **Linux** as a guest

What We Have Done

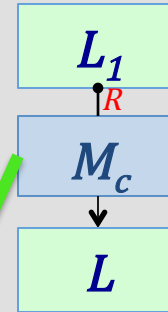
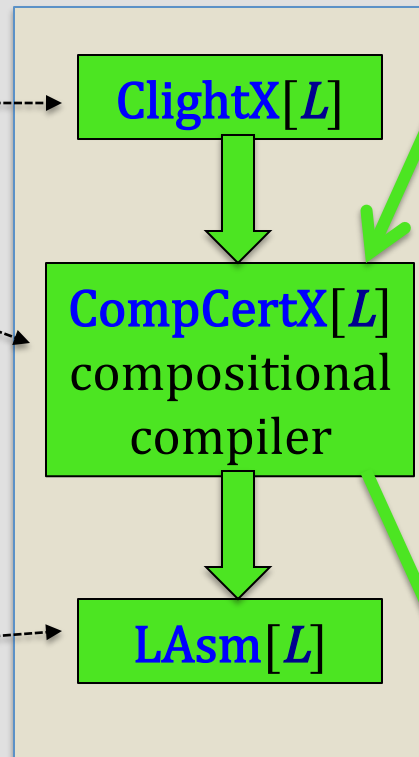
Coq

Layer Spec
 L

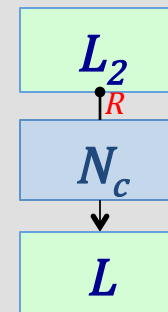
Parametrize it w.
abstract states &
primitives in L



Extended
Asm Language
 $L\text{Asm}$



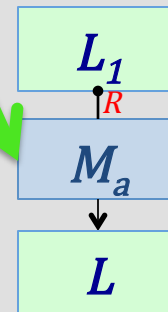
+



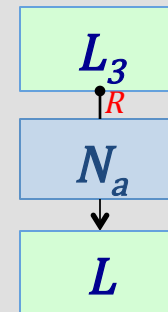
Layered
prog. in
ClightX

LayerLib
calculus

Link
everything
together

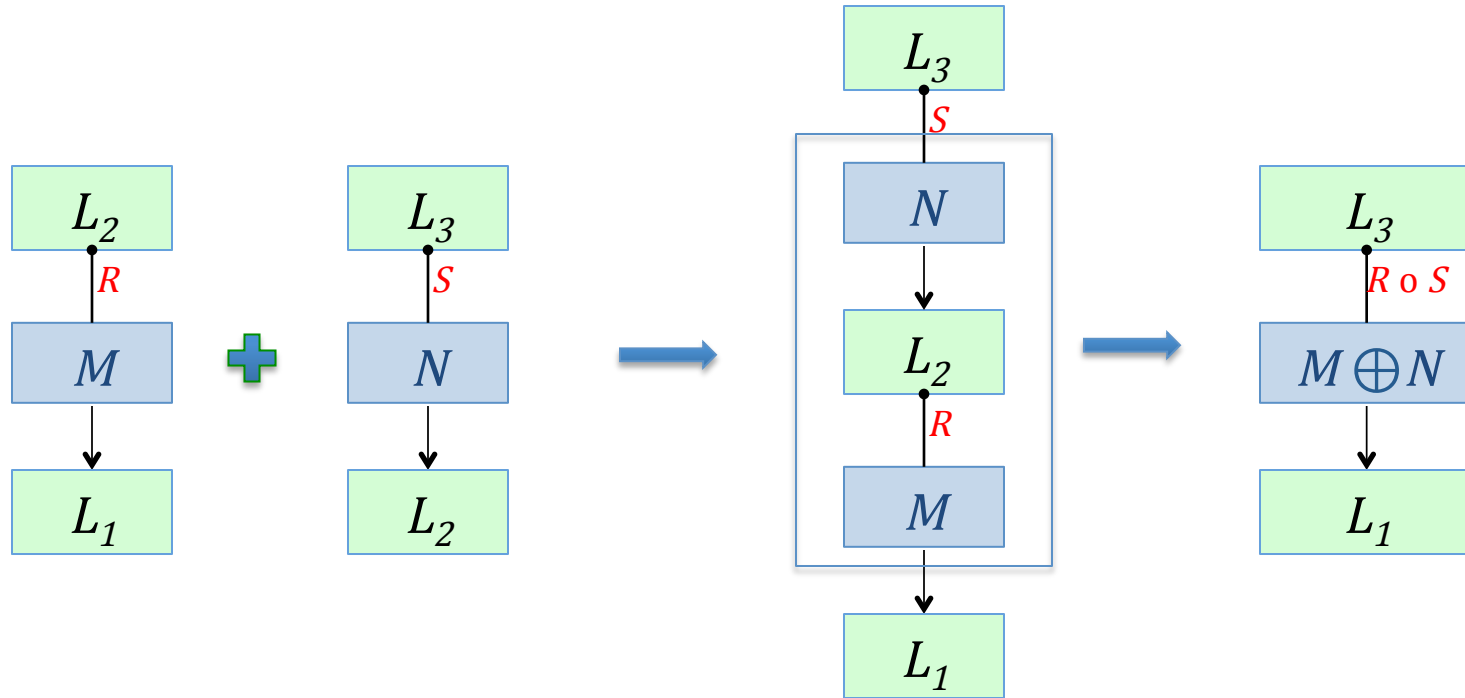


+



Layered
prog. in
 $L\text{Asm}$

LayerLib: Vertical Composition



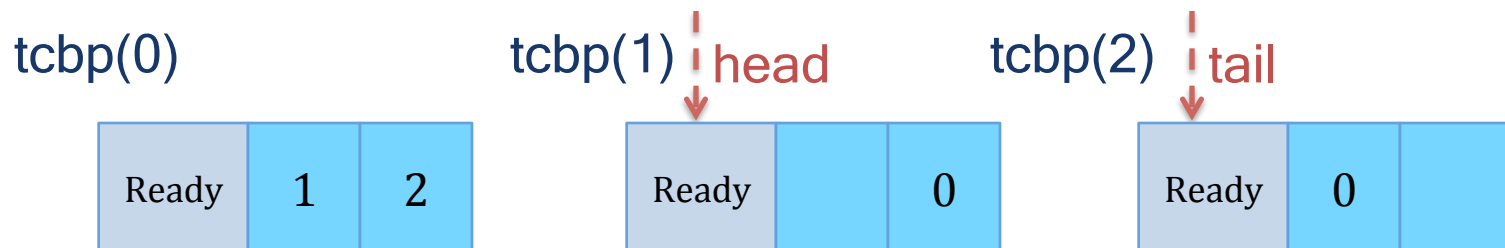
$$\frac{L_1 \vdash_R M : L_2 \quad L_2 \vdash_S N : L_3}{L_1 \vdash_{R \circ S} M \oplus N : L_3} \text{VCOMP}$$

Example: Thread Queues

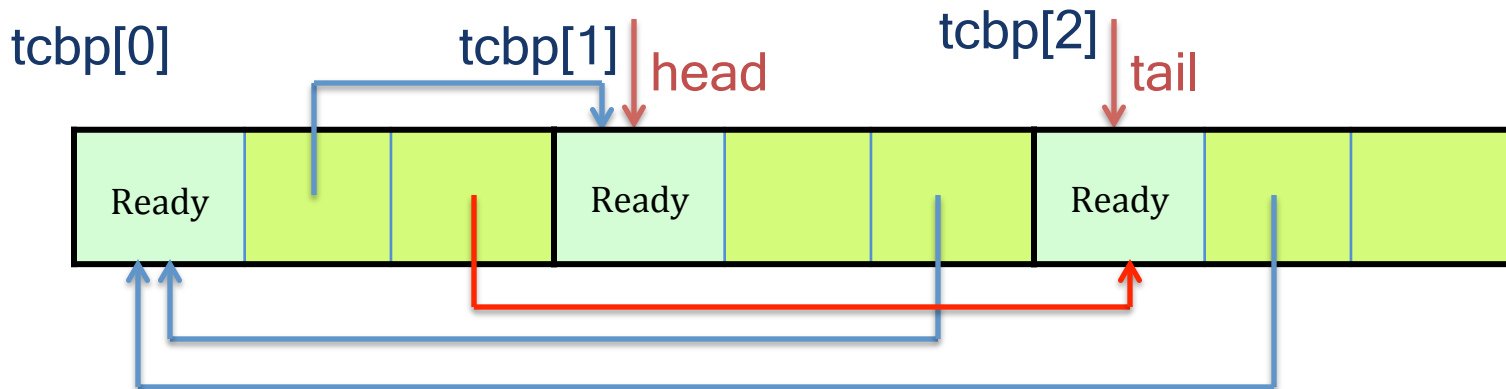
High
Abs-State



Low
Abs-State



Concrete
Memory



Example: Thread Queues

C Implementation

```
typedef enum {
    TD_READY, TD_RUN,
    TD_SLEEP, TD_DEAD
} td_state;

struct tcb {
    td_state tds;
    struct tcb *prev, *next;
};

struct tdq {
    struct tcb *head, *tail;
};

struct tcb tcbp[64];
struct tdq tdqp[64];

struct tcb * dequeue
    (struct tdq *q) {
    ..... }

```

Low Layer Spec in Coq

```
Inductive td_state :=
| TD_READY | TD_RUN
| TD_SLEEP | TD_DEAD.

Inductive tcb :=
| TCBV (tds : td_state)
      (prev next : Z)

Inductive tdq :=
| TDQV (head tail: Z)

Record abs := {
    tcbp : ZMap.t tcb;
    tdqp : ZMap.t tdq }

Function dequeue
    (d : abs) (i : Z) :=
.....

```

High Layer Spec in Coq

```
Inductive td_state :=
| TD_READY | TD_RUN
| TD_SLEEP | TD_DEAD.

Definition tcb := td_state.

Definition tdq := List Z.

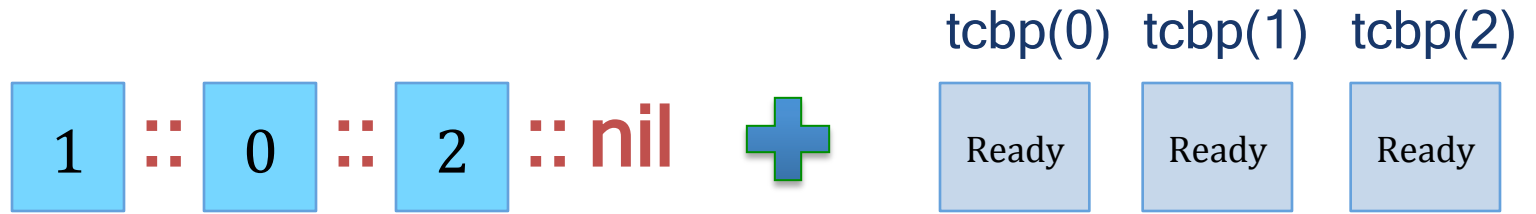
Record abs' := {
    tcbp : ZMap.t tcb;
    tdqp : ZMap.t tdq }

Function dequeue
    (d : abs') (i : Z) :=
match (d.tdqp i) with
| h :: q' =>
    Some(set_tdq d i q', h)
| nil => None
end

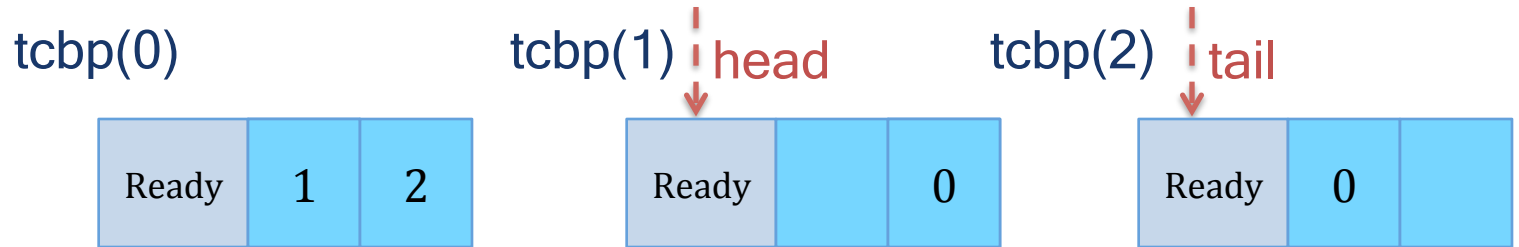
```

Example: Dequeue

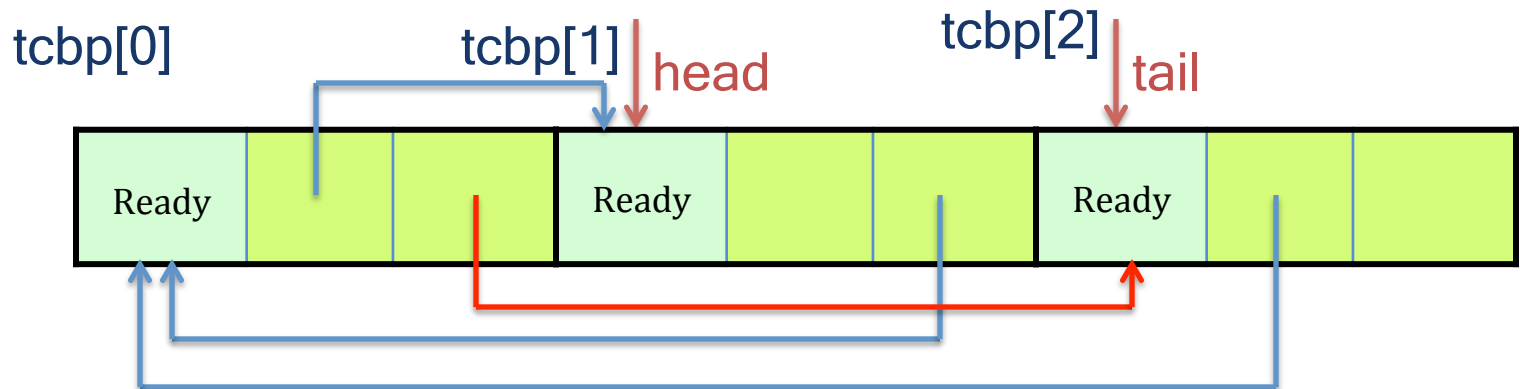
High
Abs-State



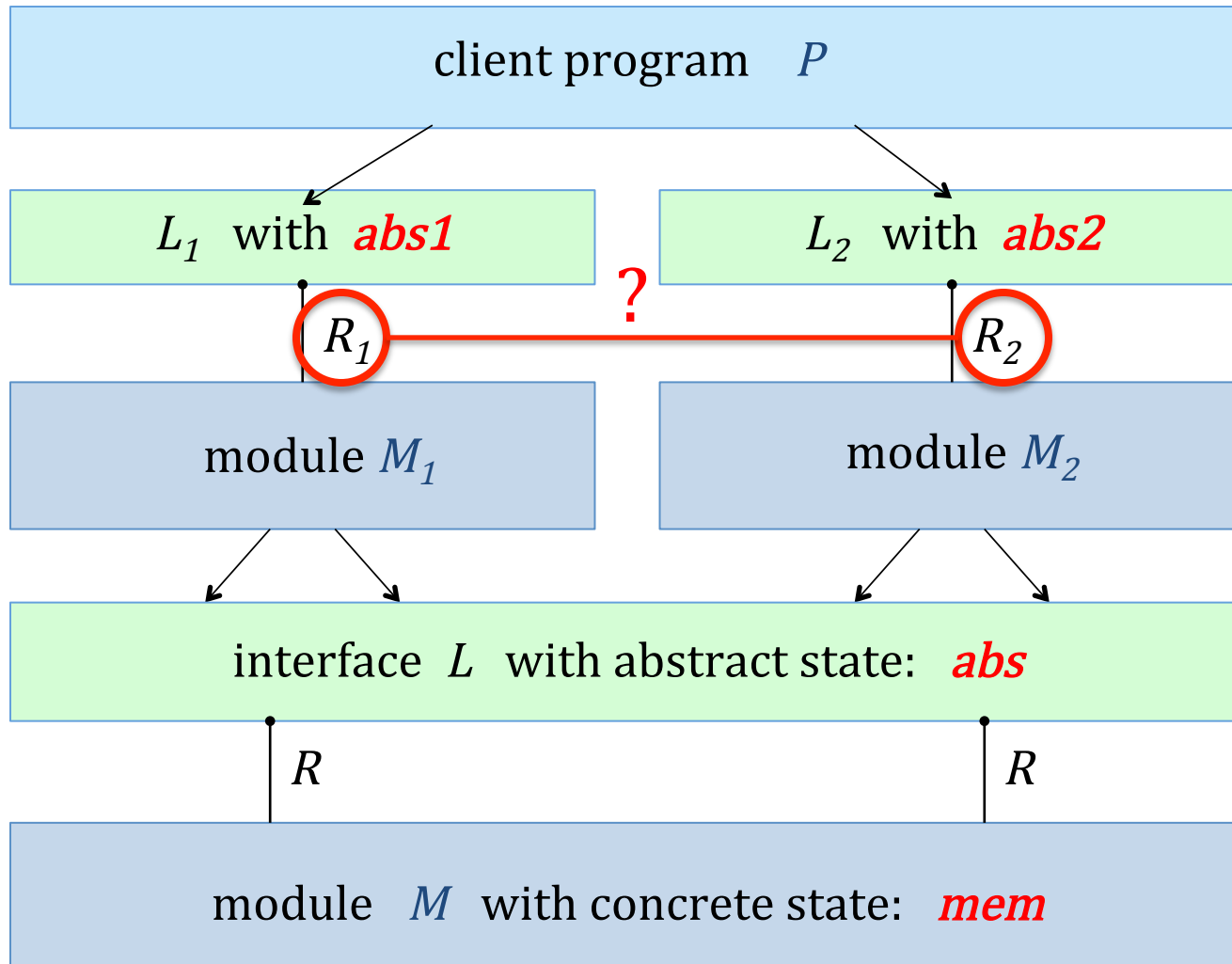
Low
Abs-State



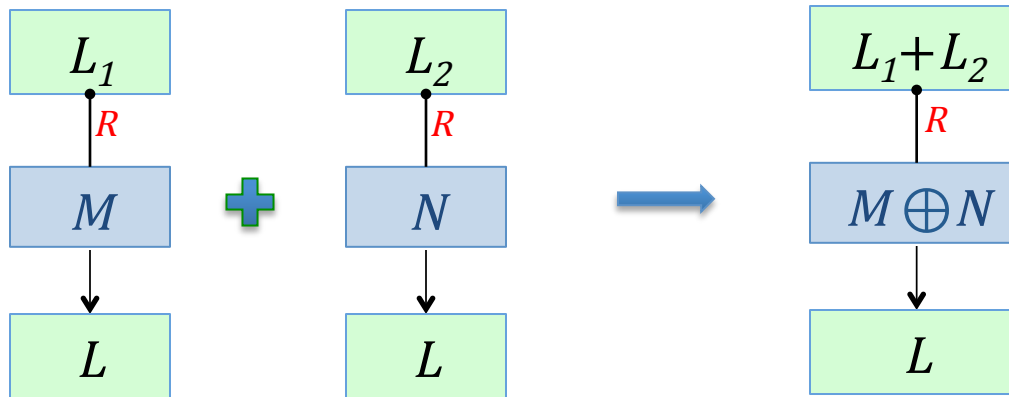
Concrete
Memory



Conflicting Abstract States?



LayerLib: Horizontal Composition



$$\frac{L \vdash_R M : L_1 \quad L \vdash_R N : L_2}{L \vdash_R M \oplus N : L_1 \oplus L_2} \text{HCOMP}$$

- L_1 and L_2 must have the same abstract state
- both layers must follow the same simulation relation R

Programming & Compiling Layers

ClightX

$$L \vdash_R M_c : L_1$$



$$L_1 \leq_R \llbracket M_c \rrbracket_{\text{ClightX}}(L)$$



CompCertX correctness theorem (where *minj* is a special kind of memory injection)

$$\llbracket M_c \rrbracket_{\text{ClightX}}(L) \leq_{\text{minj}} \llbracket \text{CompCertX}[L](M_c) \rrbracket_{\text{LAsm}}(L)$$



$$L_1 \leq_{R \circ \text{minj}} \llbracket \text{CompCertX}[L](M_c) \rrbracket_{\text{LAsm}}(L)$$



R must absorb such memory injection: $R \circ \text{minj} = R$ then we have:

$$L_1 \leq_R \llbracket \text{CompCertX}[L](M_c) \rrbracket_{\text{LAsm}}(L)$$



Let $M_a = \text{CompCertX}[L](M_c)$ then $L \vdash_R M_a : L_1$

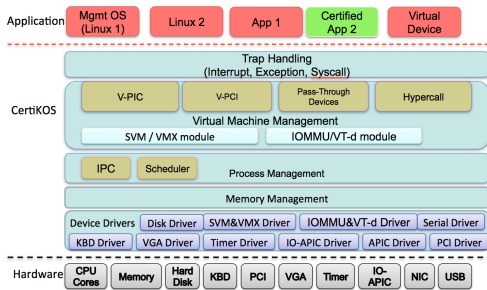
LAsm

Our Contributions



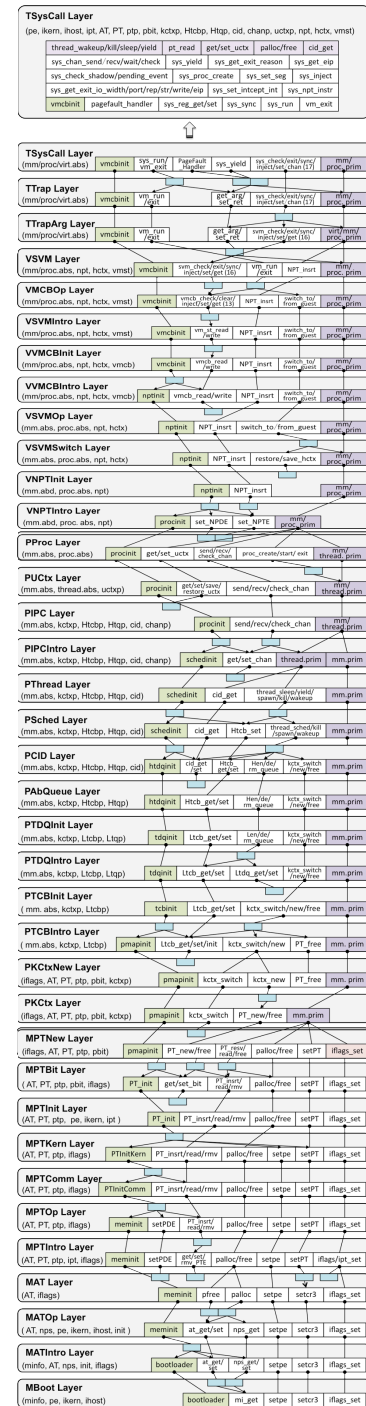
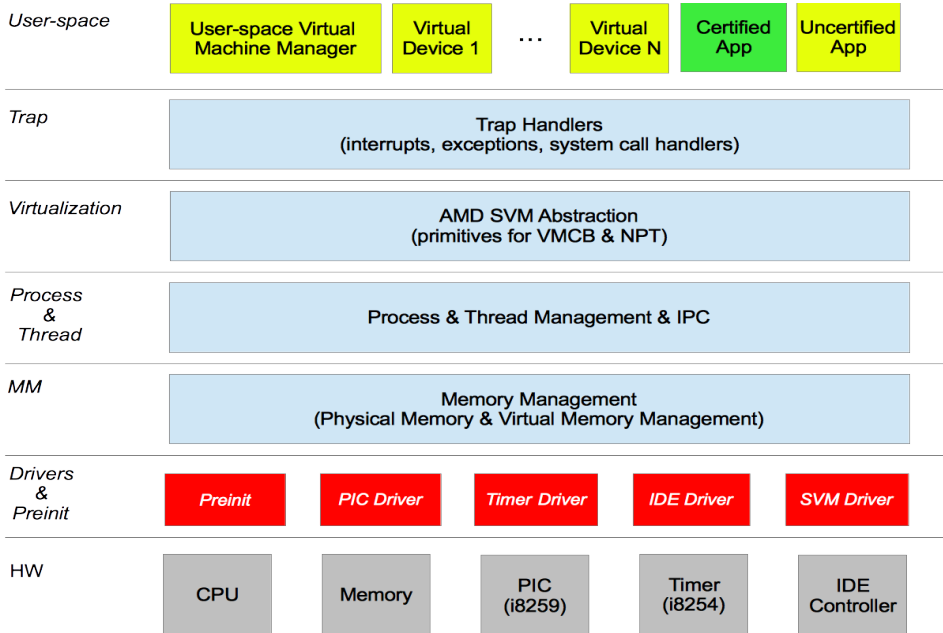
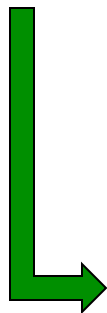
- We introduce **deep specification** and present a language-based formalization of **certified abstraction layer**
- We developed new languages & tools in Coq
 - **A formal layer calculus** for composing certified layers
 - **ClightX** for writing certified layers in a C-like language
 - **LAsm** for writing certified layers in assembly
 - **CompCertX** that compiles **ClightX** layers into **LAsm** layers
- We built multiple **certified OS kernels** in Coq
 - **mCertiKOS-hyper** consists of **37 layers**, took less than **one-person-year** to develop, and can boot **Linux** as a guest

Case Study: mCertikOS

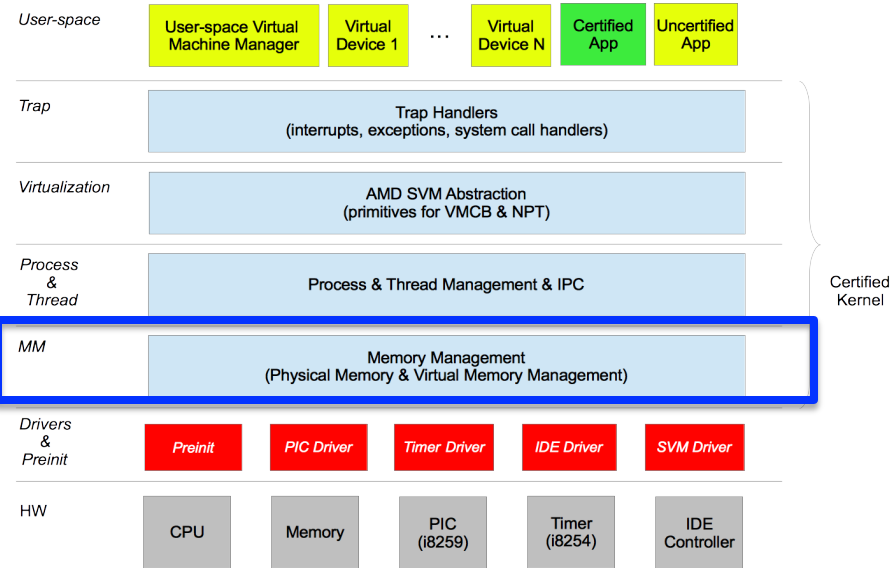


Single-core version of *CertiKOS* (developed under DARPA CRASH & HACMS programs), 3 kloc, can boot Linux

Aggressive use of abstraction over deep specs (37 layers in *ClightX* & *LAsm*)

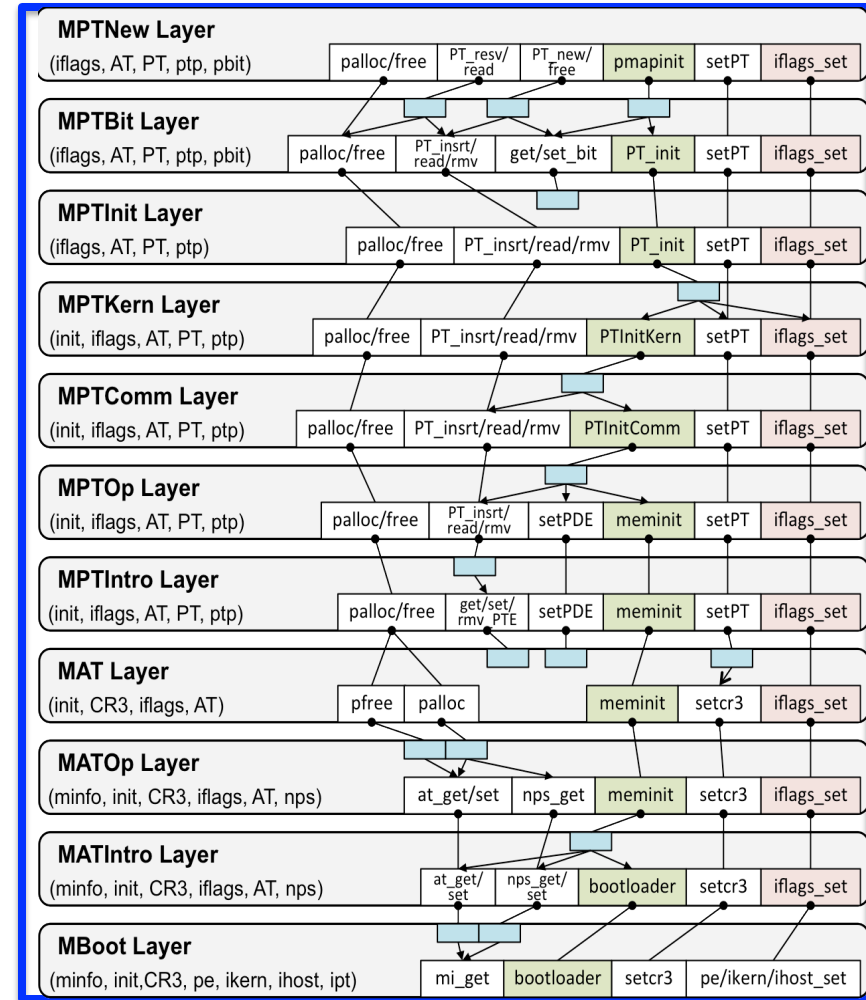


Decomposing mCertiKOS

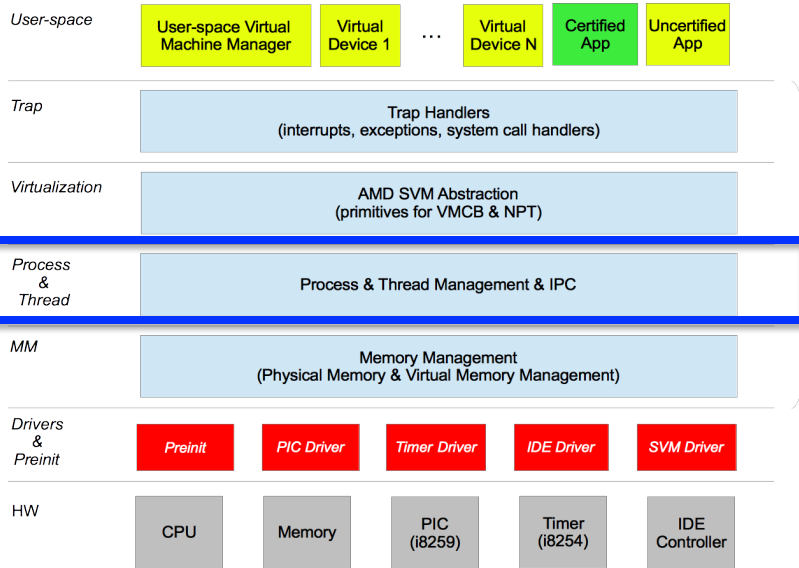


Physical Memory and Virtual Memory Management (11 Layers)

Based on the abstract machine provided by boot loader

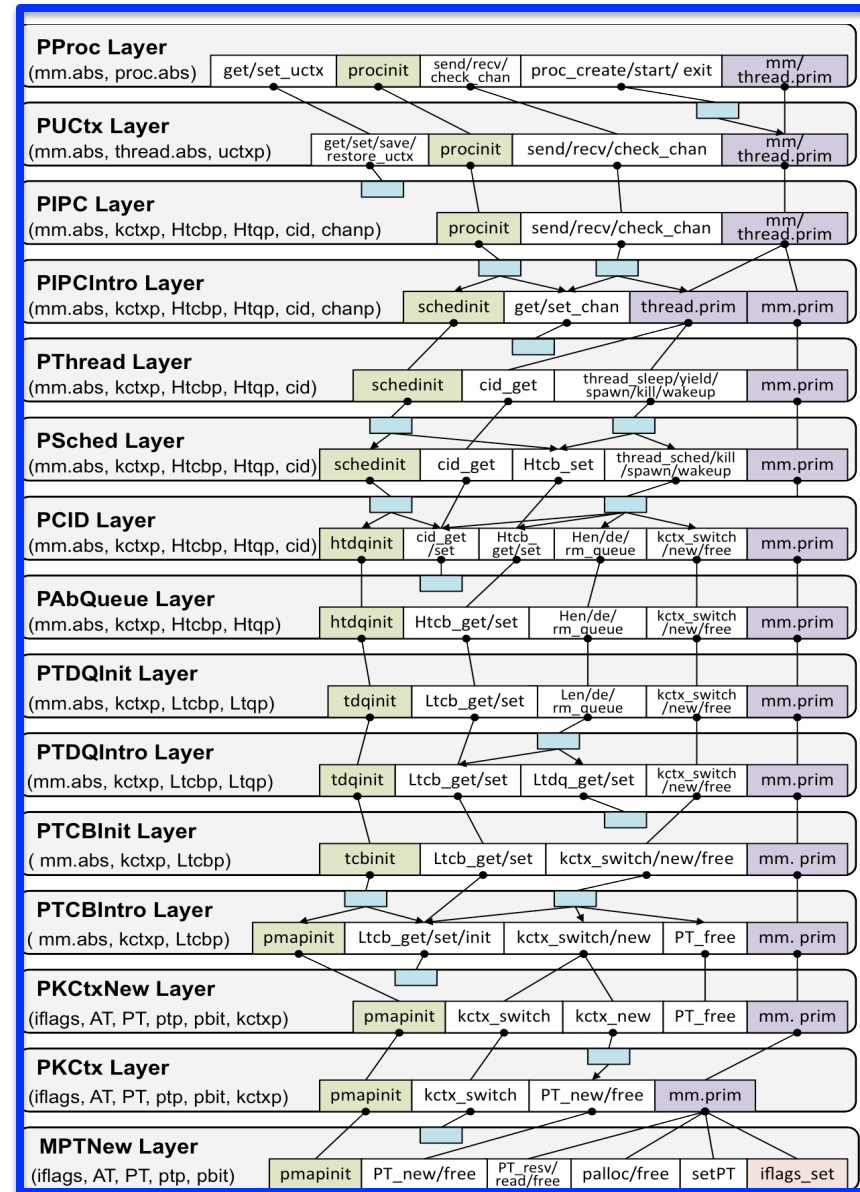


Decomposing mCertiKOS (cont'd)

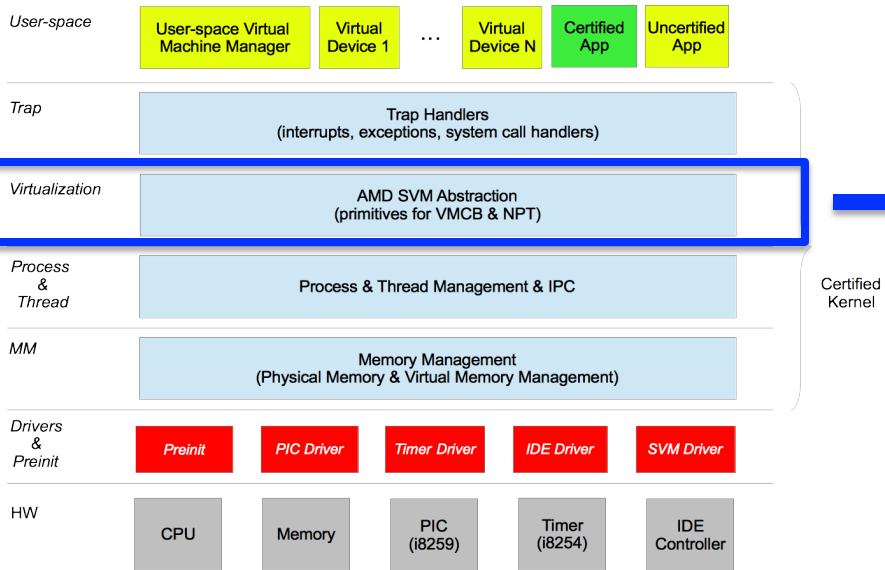


Kernel

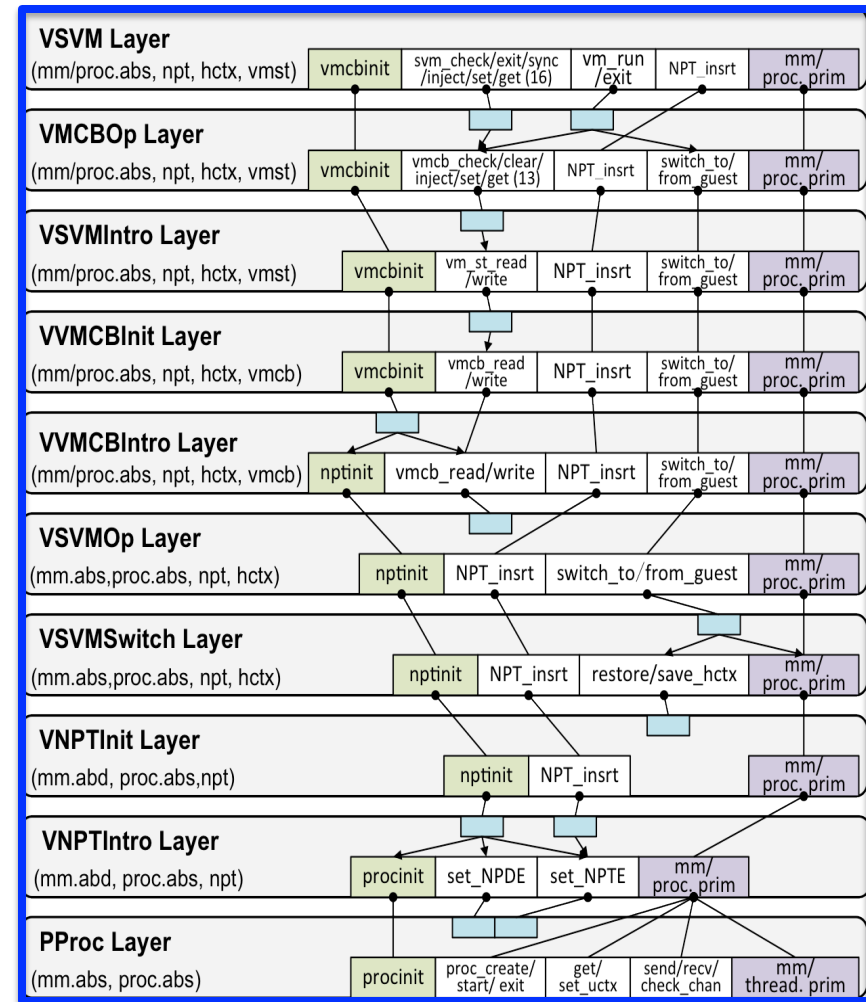
Thread and Process Management (14 Layers)



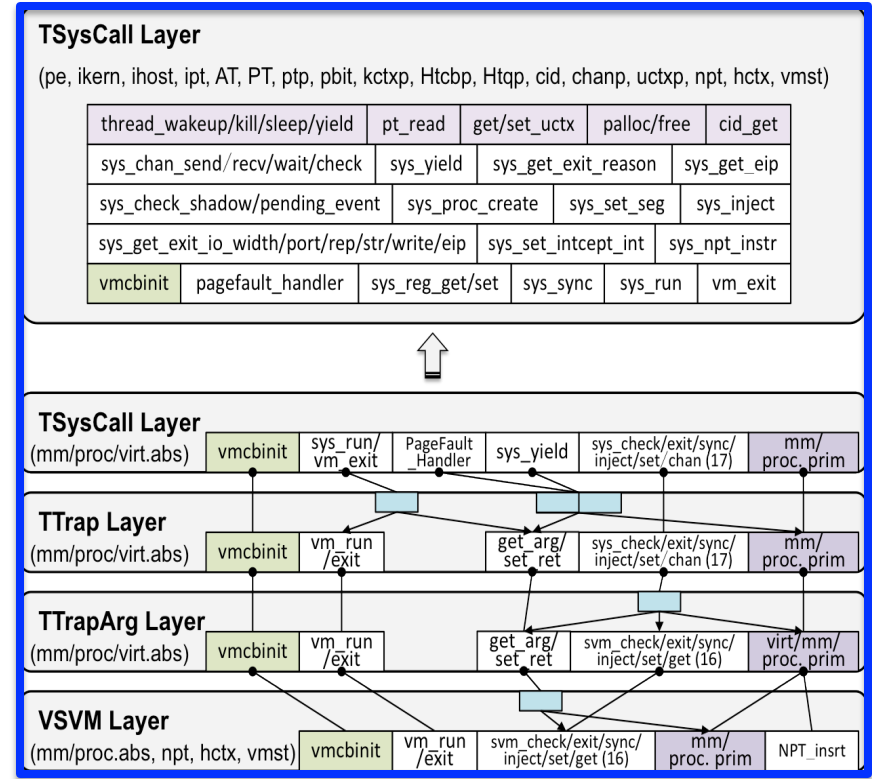
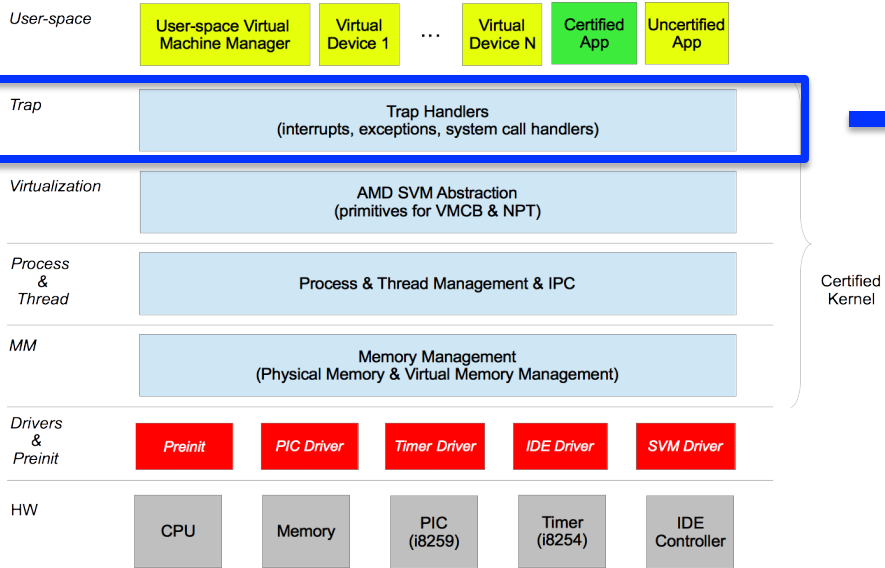
Decomposing mCertiKOS (cont'd)



Virtualization Support (9 Layers)

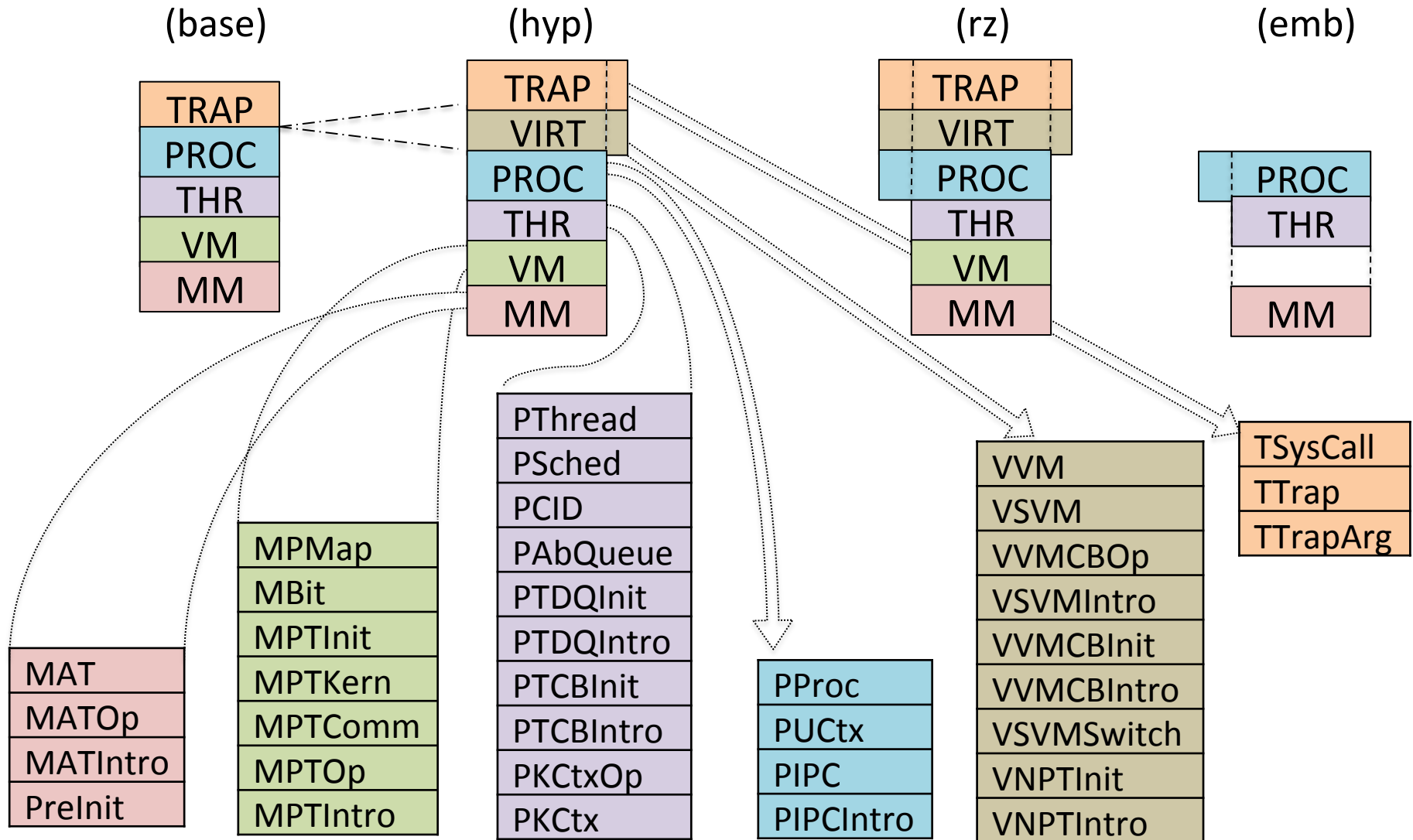


Decomposing mCertiKOS (cont'd)

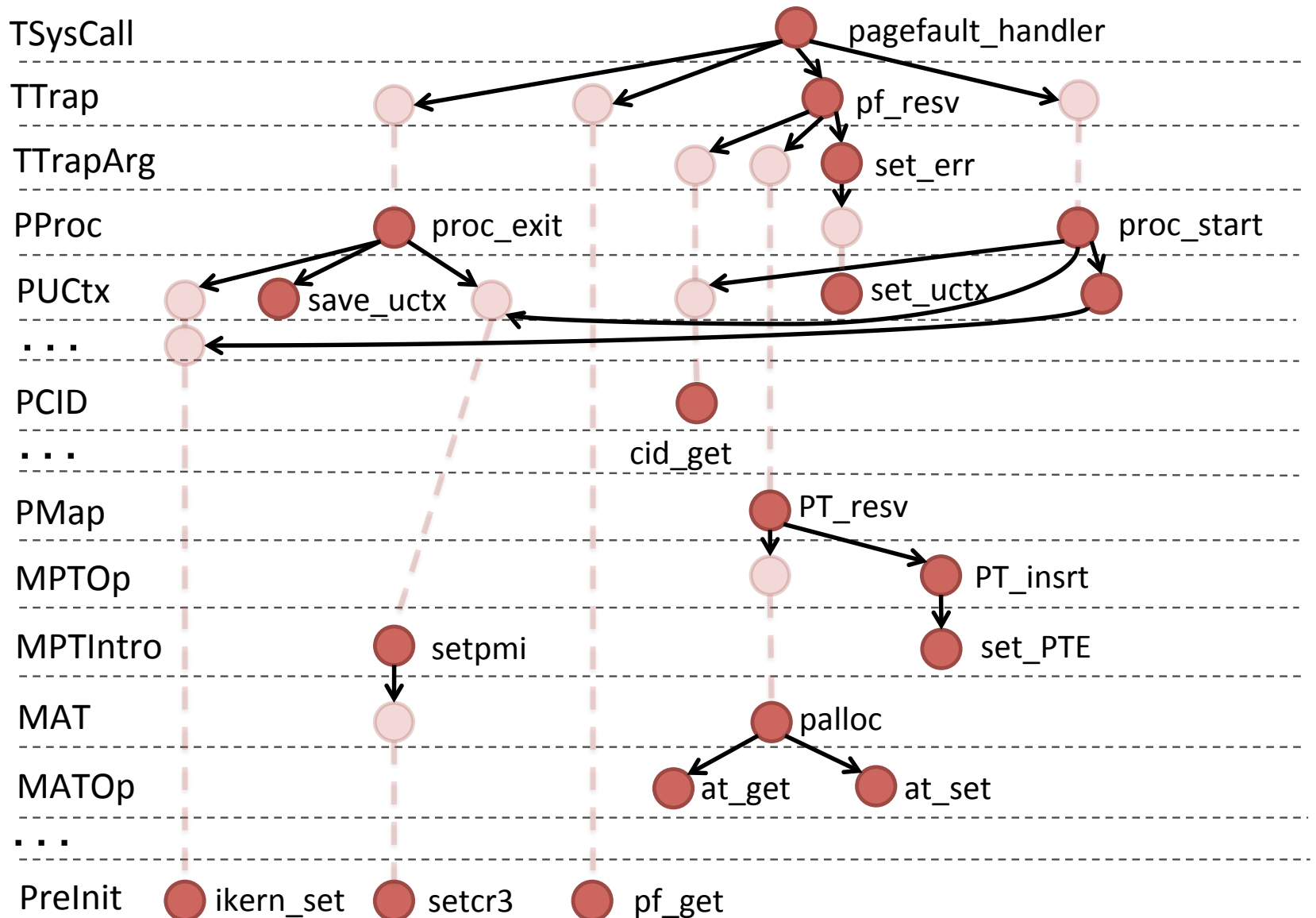


Syscall and Trap Handlers (3 Layers)

Variants of mCertikOS Kernels



Example: Page Fault Handler



Conclusions

- Great success w. today's **system software** ... but why?
- We identify, sharpen, & **formalize** two possible ingredients
 - abstraction over **deep specs**
 - a **compositional layered** methodology
- We build new lang. & tools to make **layered programming** *rigorous & certified* --- this leads to **huge benefits**:
 - simplified design & spec; reduced proof effort; better extensibility
- They also help *verification in the small*
 - hiding implementation details as soon as possible
- Still need better PL and tool support (Coq / ClightX / LAsm)

Thank You!

*Interested in working on the CertiKOS project?
we are hiring & recruiting at all levels:*

postdocs,

research scientists,

PhD students, and visitors

A Subtlety for LAsm

Some functions (e.g., [kernel context switch](#)) do not follow the C calling convention and must be programmed in $L\text{Asm}[L]$.

$$L \vdash_R M_a : L_2 \quad \longrightarrow \quad L_2 \leq_R \llbracket M_a \rrbracket_{L\text{Asm}}(L)$$

Problem: per-module semantics $\llbracket M_a \rrbracket_{L\text{Asm}}(L)$ is *NOT deterministic* relative to external events

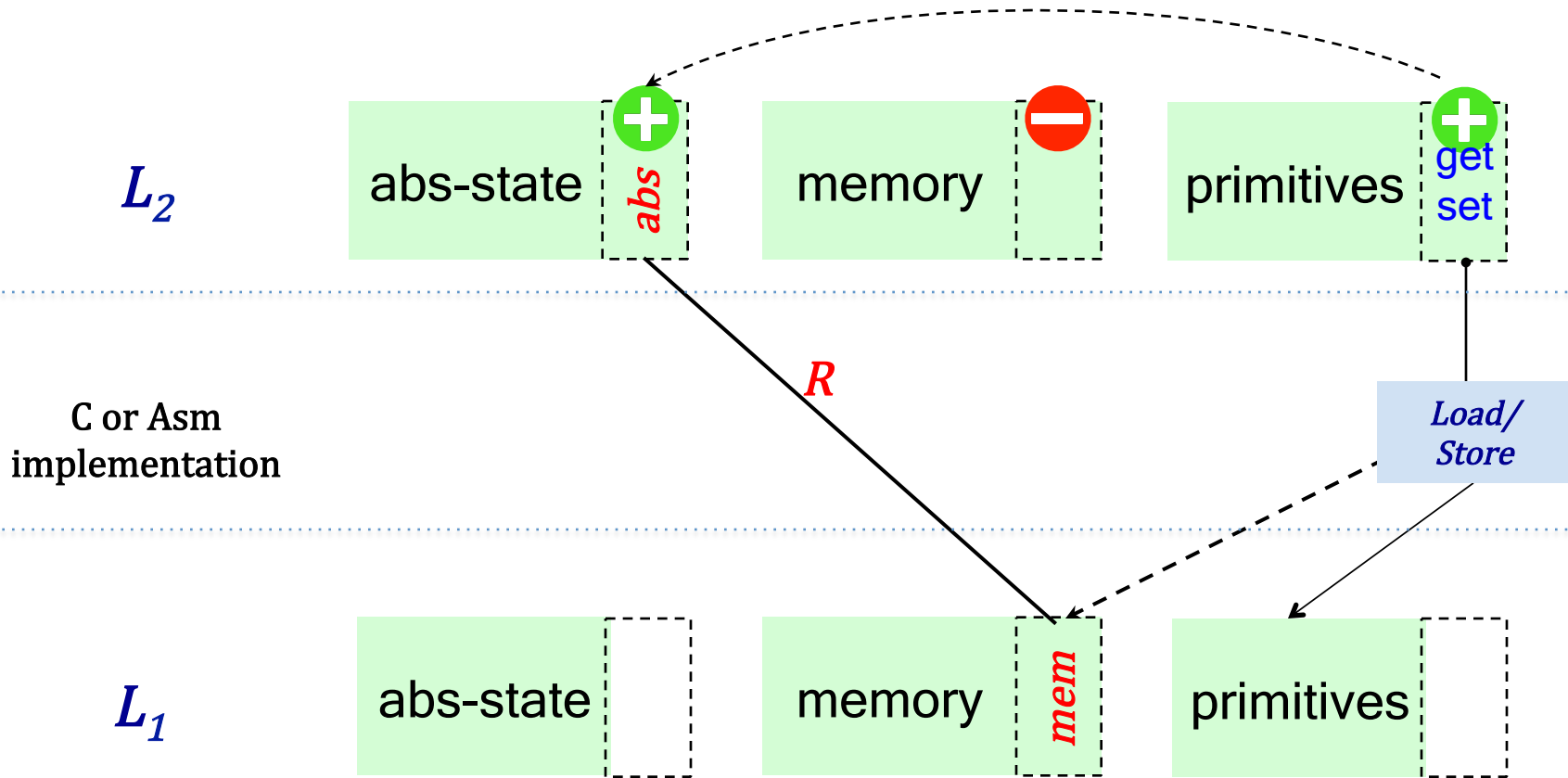


$$\llbracket M_a \rrbracket_{L\text{Asm}}(L) \leq_R L_2$$

Fortunately, whole-machine semantics $\llbracket \bullet \rrbracket_{L\text{Asm}}(L)$ is deterministic relative to external events, so it can still be reversed:

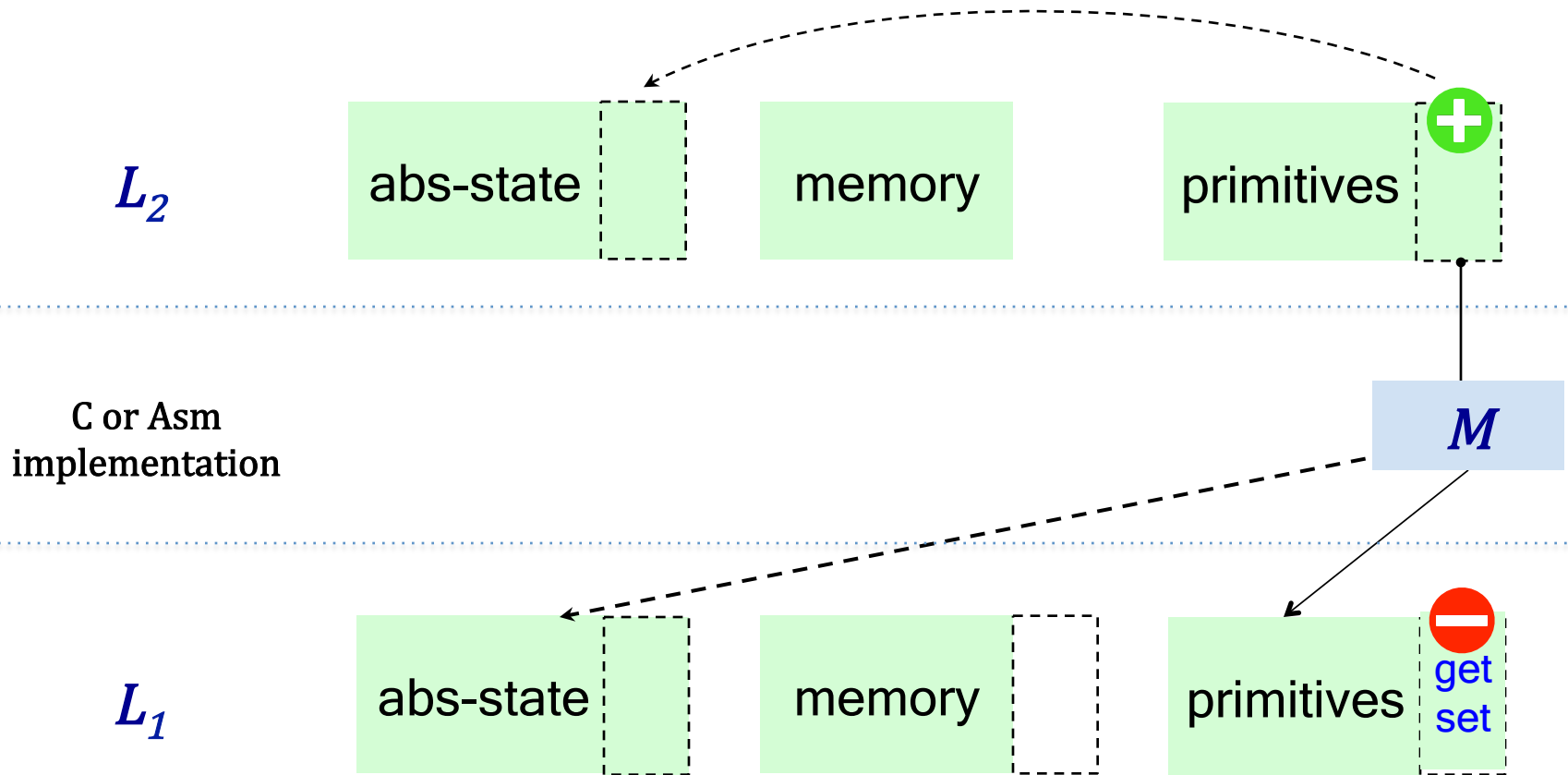
$$\forall P. \llbracket P \oplus M_a \rrbracket_{L\text{Asm}}(L) \sim_R \llbracket P \rrbracket_{L\text{Asm}}(L_2)$$

Layer Pattern 1: Getter/Setter



Hide concrete memory; replace it with Abstract State
Only the **getter** and **setter** primitives can access memory

Layer Pattern 2: AbsFun



Memory does not change

New implementation code does not access memory directly!