CS 428 / 528

Language-Based Security

(Spring 2024)

Zhong Shao
Yale University

http://flint.cs.yale.edu/cs428

Problem and Approach

How to build truly “secure” software?

Conventional security:

- software is black box
- Encryption, firewalls, system calls/privileged mode

Language-based security:

- must reason about software (need formal methods)
- Information-flow control + dealing w. zero-day vulnerabilities
- How to verify a small amount of software to get the security guarantee for an entire system.

Course Overview

- Read papers, write reviews, discuss ideas in class, and work on a course project
  - **Tuesday classes**: discuss papers we read
  - **Thursday classes**: learn Coq, CertiKOS, DeepSEA, and CompCert and prepare for the final course projects
- A reading list will be made available soon
- Grading:
  - Class participation/discussion (10%)
  - Paper reviews and/or problem sets (25%)
  - Class presentation (15%)
  - Final course project (40%)

Course Objectives

- Learn *cutting-edge research* & *fundamental principles* for building secure and reliable system software

- Learn state-of-the-art tools for writing certified code
  - The Coq proof assistant
  - Certified C language & compiler (Clight & CompCert)
  - Certified OS kernels (CertiKOS and seL4)
  - DeepSEA and CCAL

- Study various language-based security technologies
  - Abstraction layers and formal specification & verification
  - OS kernel and hypervisor and secure enclave design
  - Capabilities & access control & information flow control
  - Reasoning about IPC, interrupts, atomicity, and transactions

Certified Heterogeneous Systems

- How to build efficient, scalable, and trustworthy heterogeneous systems?

  Need a high-level architectural design + stepwise refinement

- Correct-by-Construction or Secure-by-Construction
  - HW/SW Implementation → Deep/Fully-Abstract Functional Spec

    (VeriLog, C, Asm) (written in some formal logic)

    (semantics for these languages) (need formal proof assistant)
  - Mechanized proofs for the above “implements” relation

- Need a theory of component composition
  - What is a component? (HW vs. SW ones)
  - What is a “certified” component?
  - What are different ways of connecting/composing these components?

Sample Research Themes

- Shared-memory concurrency & concurrent objects
- Virtual memory management & spatial isolation
- File and storage systems and device drivers
- OS kernel and hypervisor for heterogeneous architecture
- Secure enclaves
- Web server
- Blockchains and smart contracts
- Consensus-based distributed systems
- Efficient proof-certificate checking

CS428/528 Summary

You will spend most of your time doing the following:

•Read papers and discuss with fellow 428/528 students

- learn *cutting-edge research* & *fundamental principles* on building secure and reliable system software

•Learn to write formal specs & proofs in Coq

- write certified C code inside a proof assistant & compile it using a certified C compiler
- work on an open-ended project

Warning: this is more of a “research-seminar” course; we need your active participation

First Two Weeks

- Jan 16 (Tuesday): Read the paper on “Hints on Programming Language Design” by Hoare.
- Jan 18 (Thursday): Coq Tutorial (Software Foundations)
- Jan 23 (Tuesday): Read the paper on “Hints and Principles for Computer System Design” by Lampson.
- Jan 25 (Thursday): Coq Tutorial (Software Foundations)

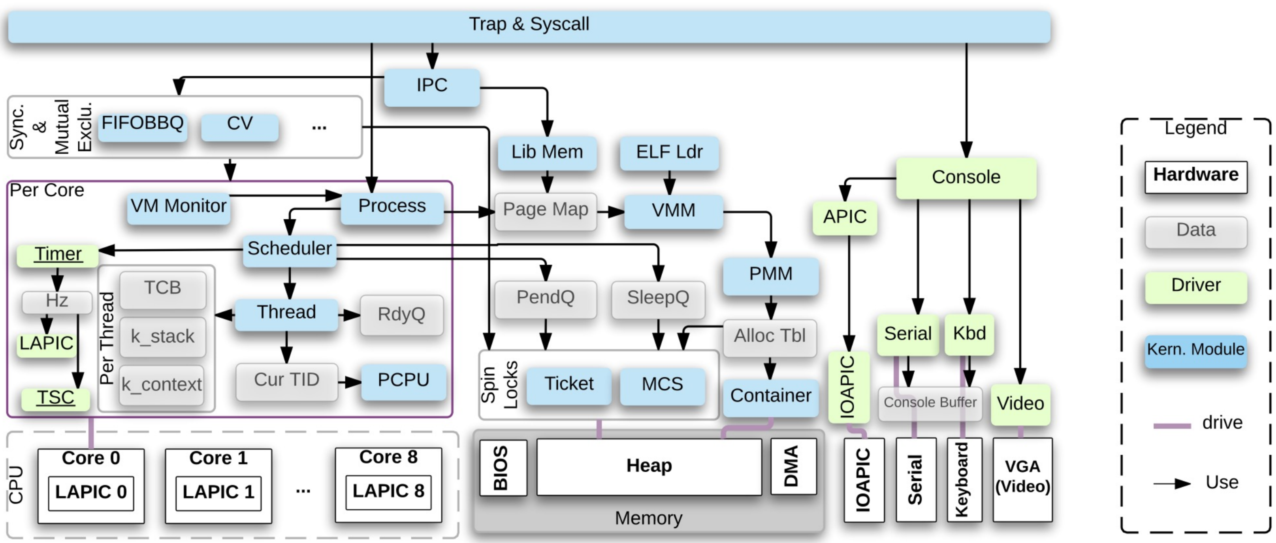
Problem Definition

- What is a certified OS kernel / hypervisor / security monitor?
  - a system binary ***implements*** its specification running over a HW machine model (w. devices & interrupts)?
  - what should the specification & the machine model be like?

- What properties do we want to prove?
  - safety & partial correctness properties
  - total functional correctness
  - security properties (isolation, confidentiality, integrity, availability)
  - resource usage properties (stack overflow, real time properties)
  - race-freedom, atomicity, and linearizability
  - liveness properties (deadlock-freedom, starvation freedom)

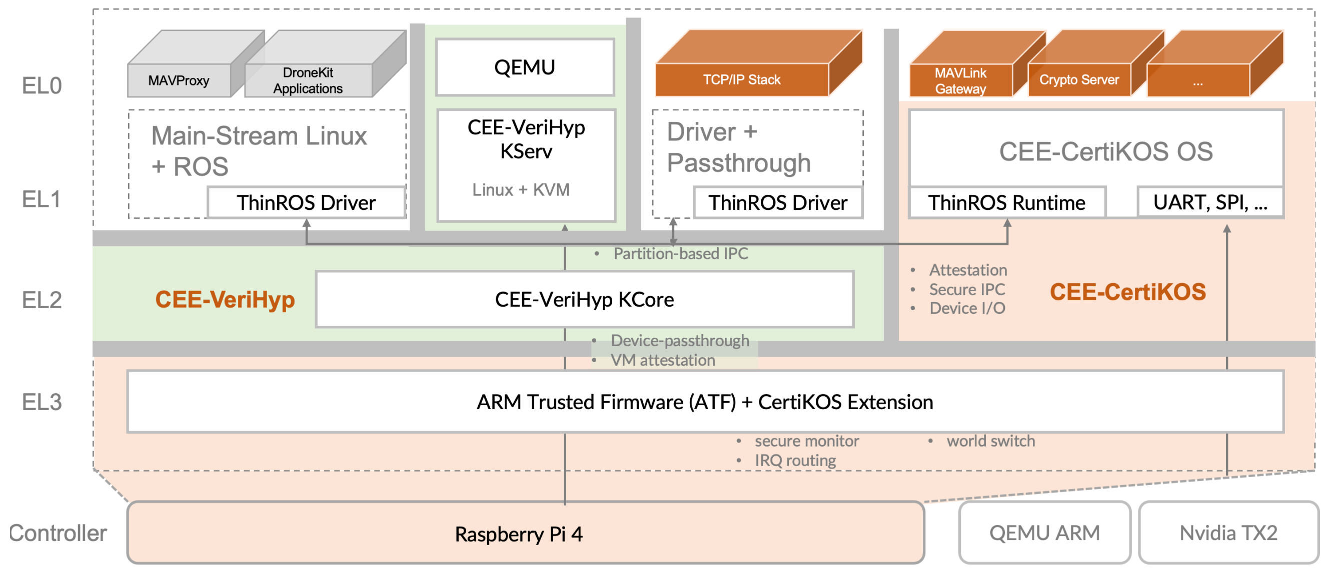
- How to cut down the cost of verification?

Problem Definition: Example OS Kernel



Formally Verified Concurrent CertiKOS (mC2) [OSDI 2016]

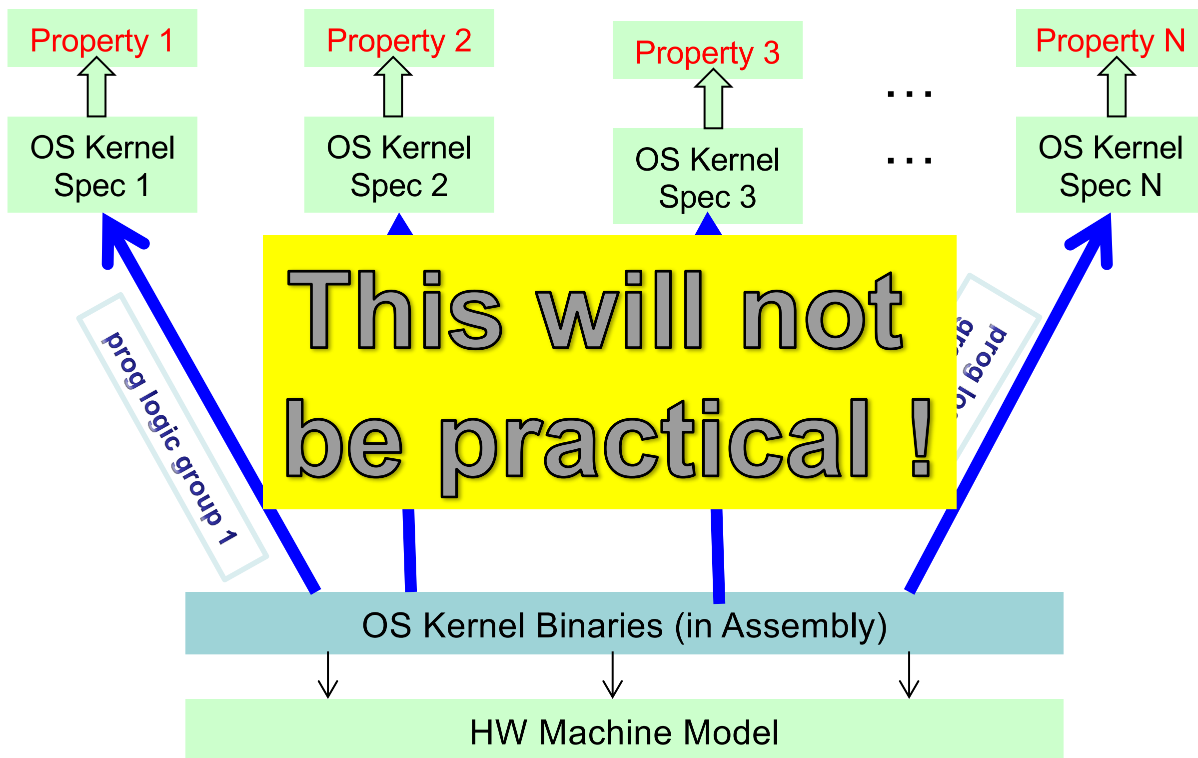
Problem Definition: Example Deployment



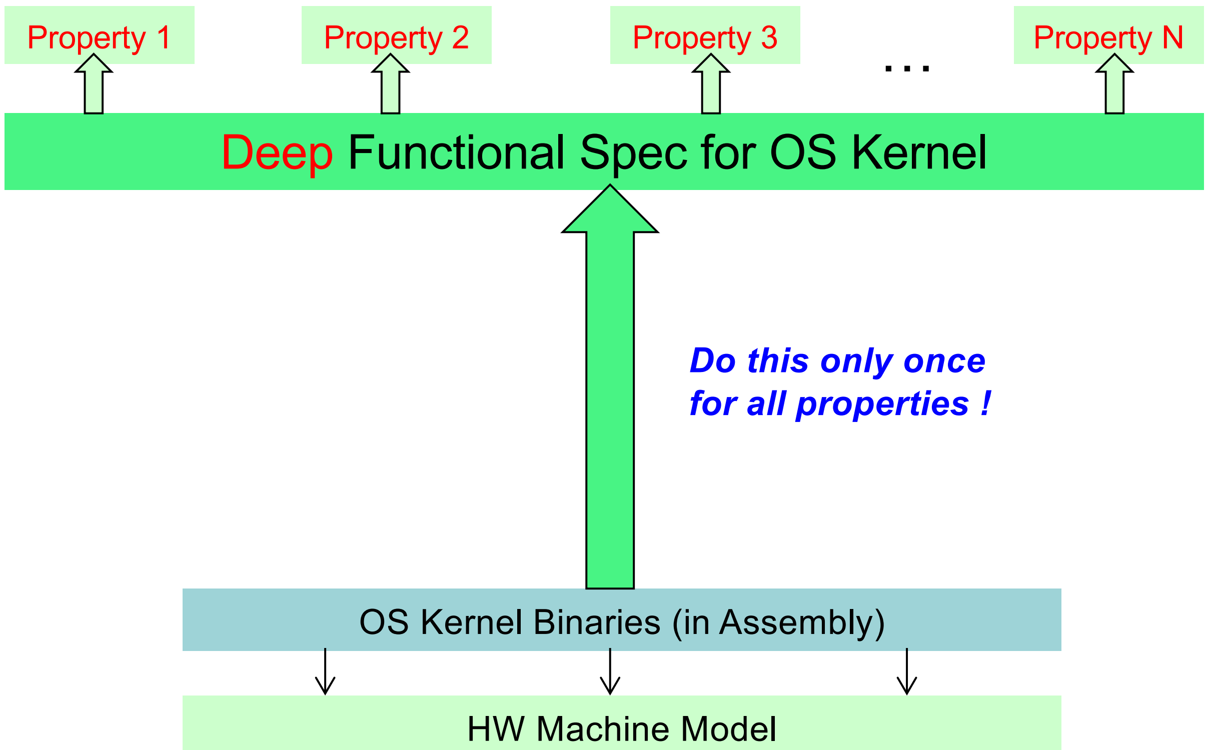
REFUEL: Formally Verified Composition of Secure Enclaves

[Joint w. Columbia U., DARPA V-SPELLS 2021-2025]

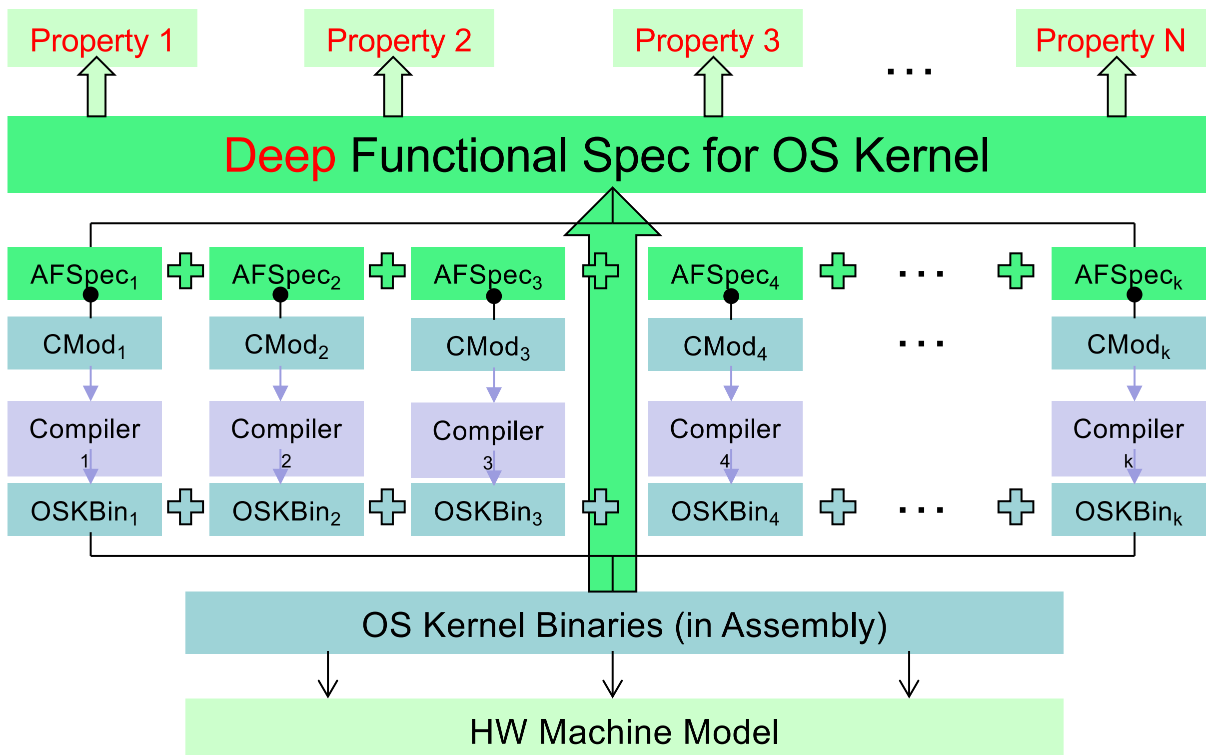
OS Verification: The Conventional Approach



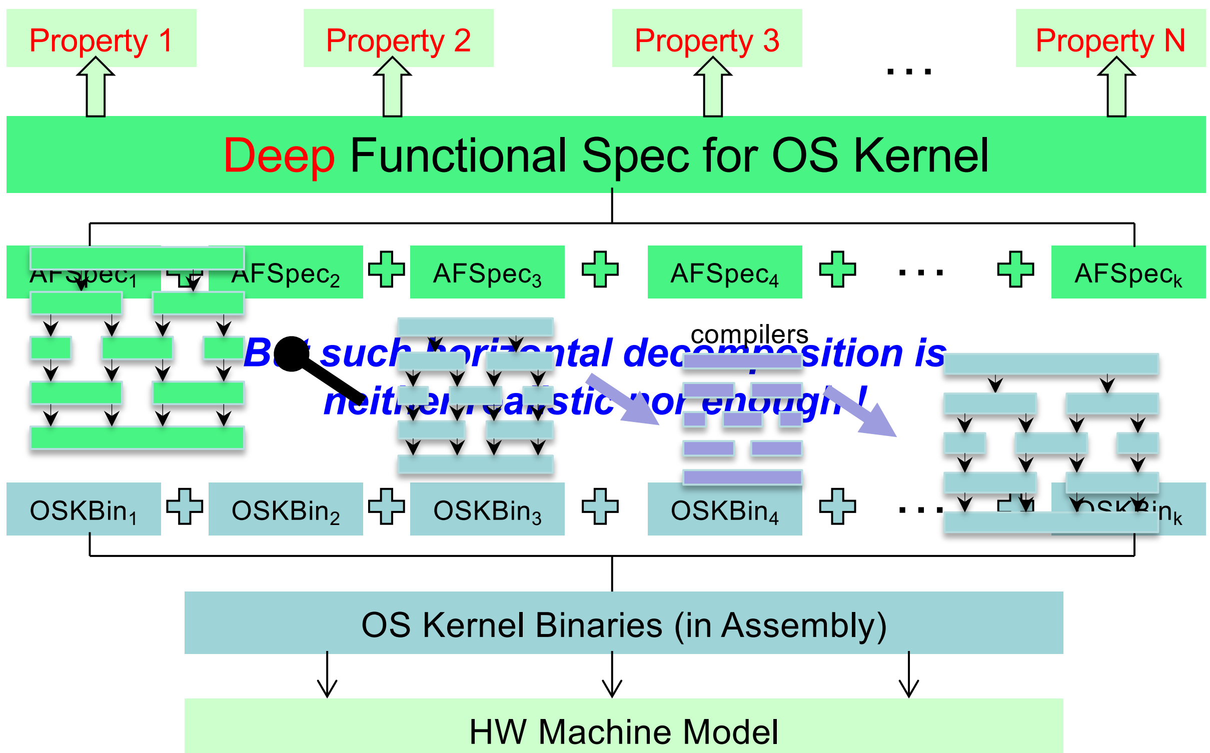
The CertiKOS Approach



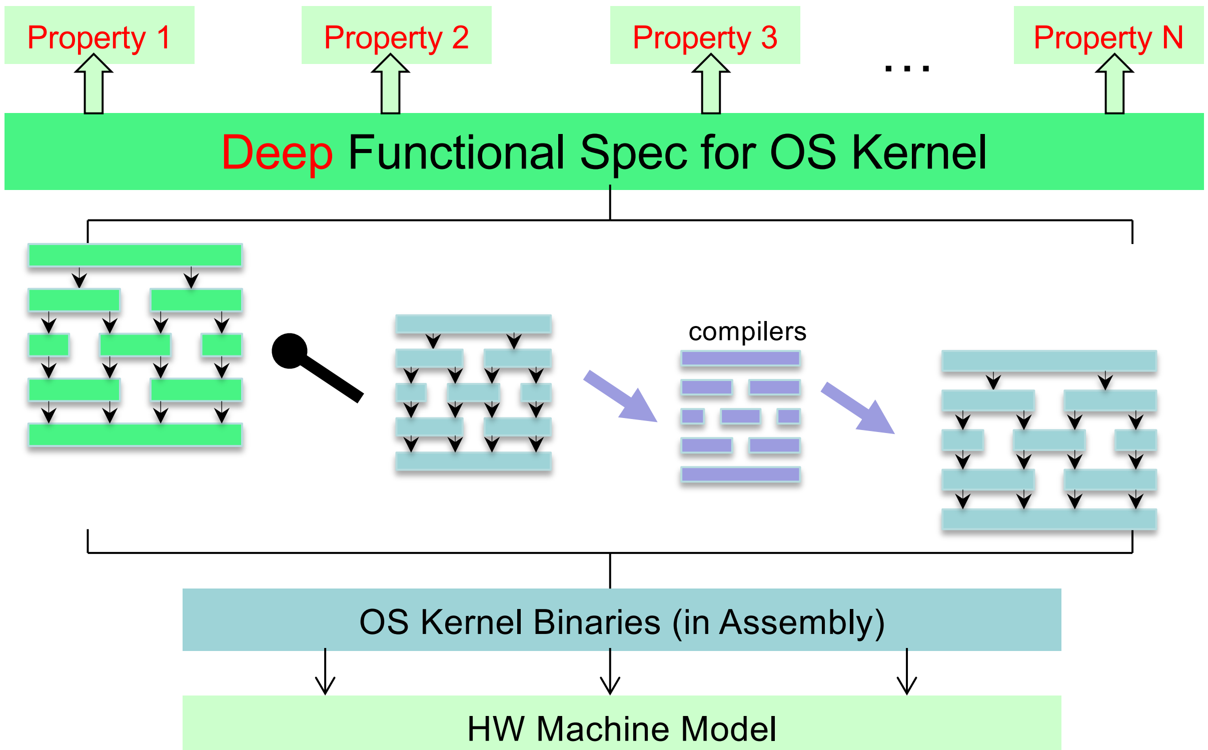
The CertiKOS Approach



The CertiKOS Approach



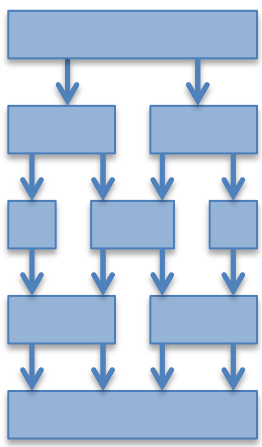
The CertiKOS Approach



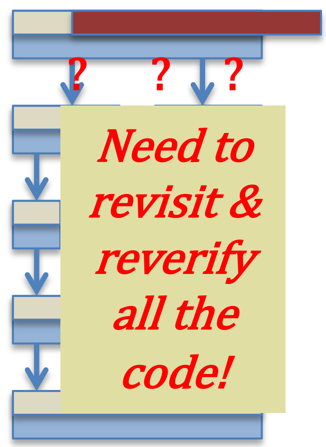
What is a Deep Spec?

C or Asm module | rich spec A | rich spec B

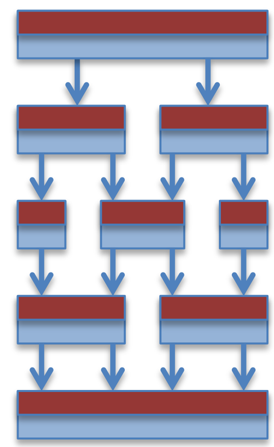
C & Asm Module Implementation



C & Asm Modules w. rich spec A



Want to prove another spec B ?



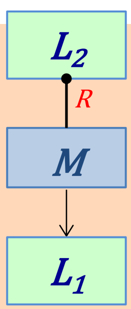
What is a Deep Spec?

$[\![M]\!]\, L_1 \sim_R L_2$

$[\![M]\!]\,(L_1)$ and L_2 simulates each other!

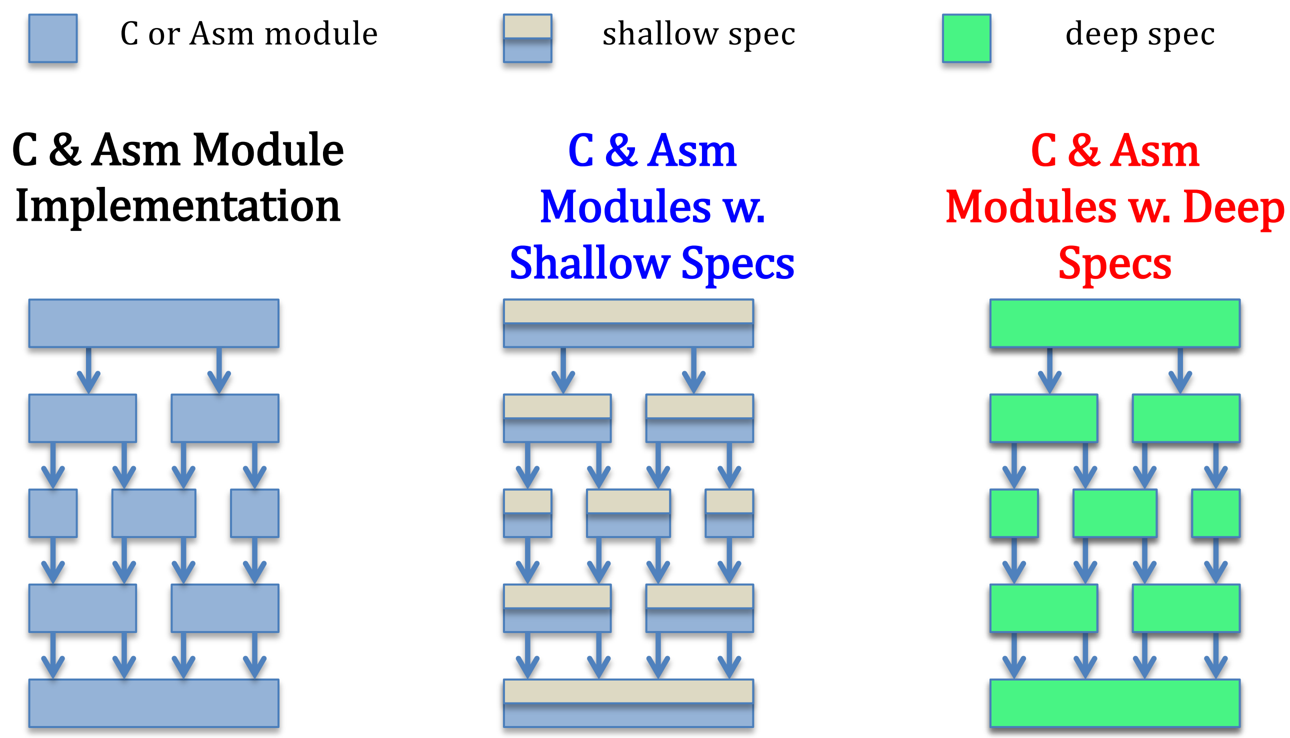
L_2 captures everything about running M over L_1

Making it "contextual" using the whole-program semantics 【•】



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

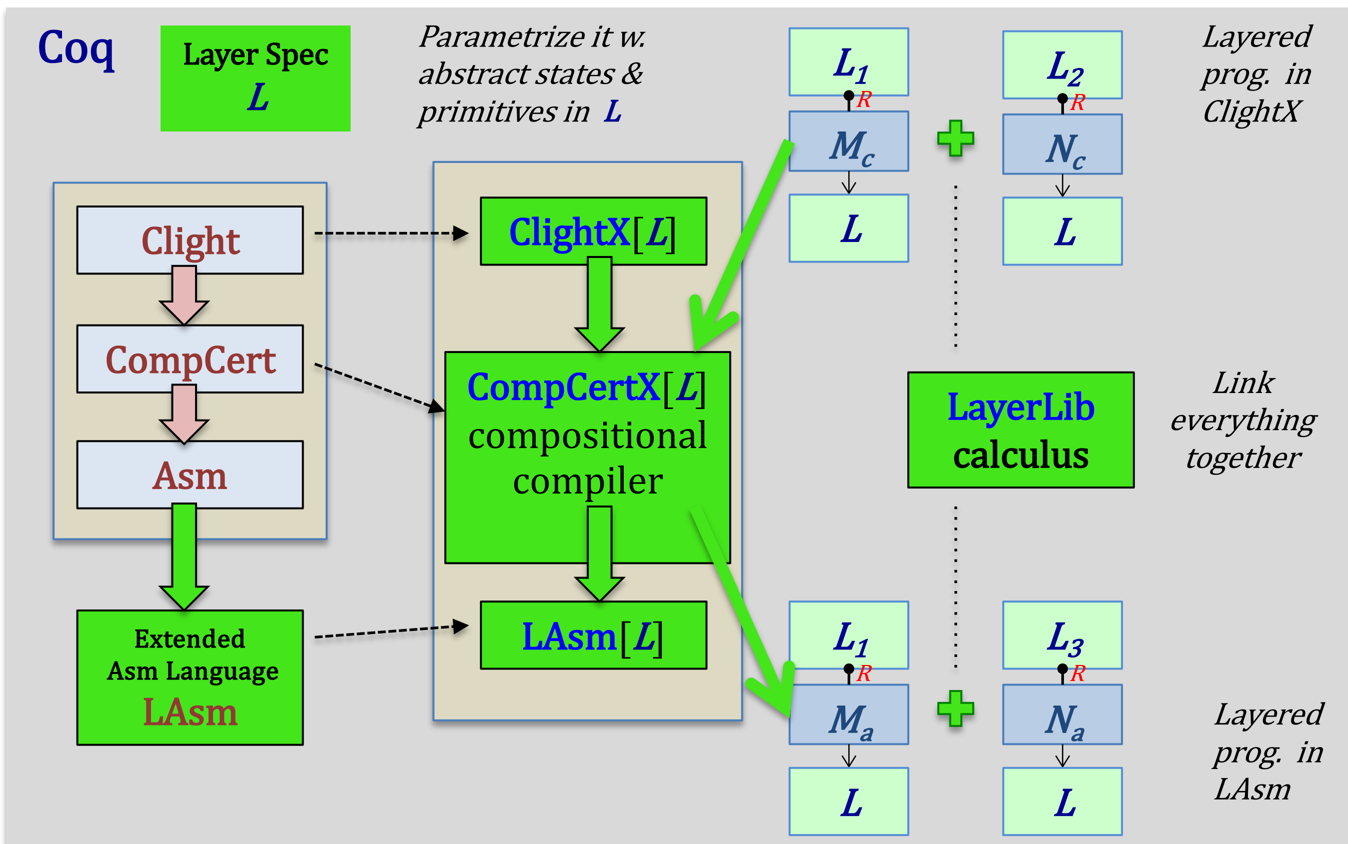
Shallow vs. Deep Specifications



The CertiKOS Approach

- We developed a language-based formalization of certified abstraction layers with deep specifications
- 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
  - The initial version has 37 layers and can boot Linux as a guest
  - The later versions support interrupts & multicore concurrency & security (spatial & temporal isolation w. real-time guarantee)

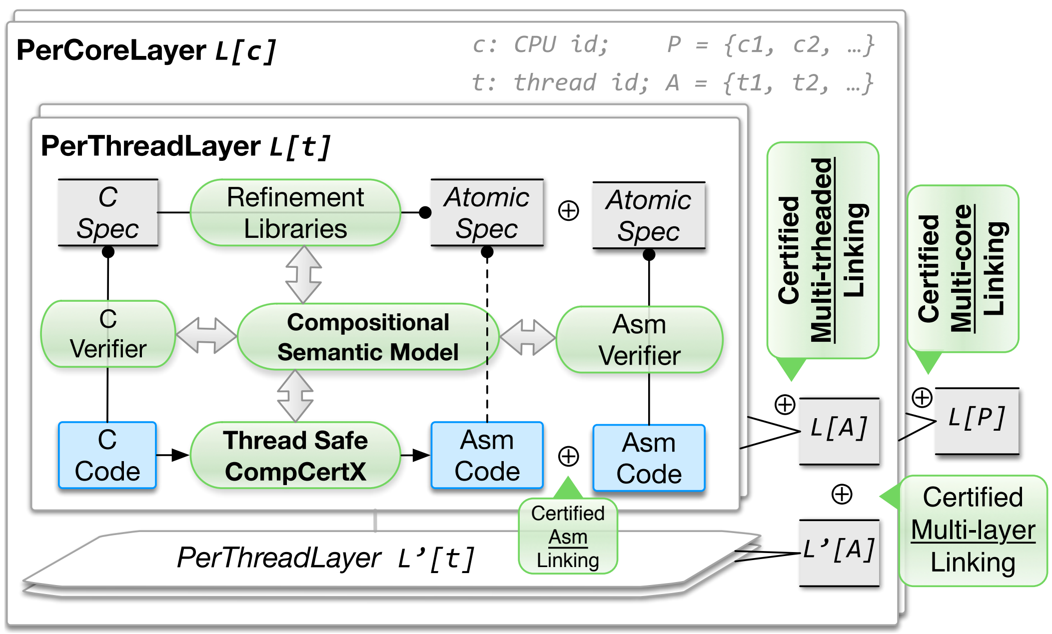
The CertiKOS Toolchain (CAL) [POPL'15]



The CertiKOS Toolchain (CCAL) [PLDI'18]

New programming toolkit w. certified multicore & multithreaded linking:

Composition = parallel composition + hiding (abstraction)



Other CCAL Use Cases

Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware

FUNCTIONAL REPRODUCED

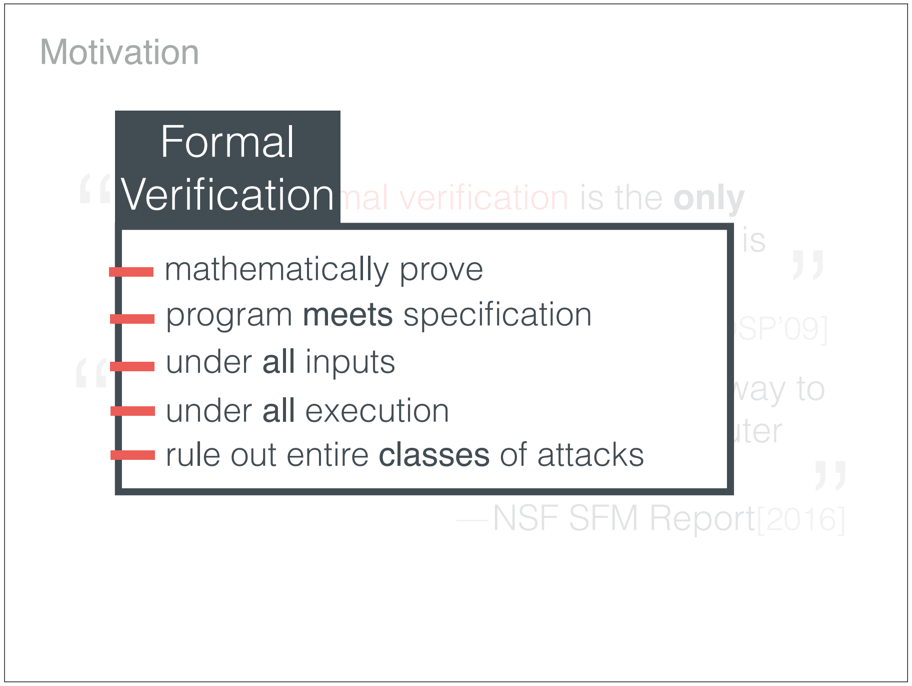
Design and Verification of the Arm Confidential Compute Architecture

Xupeng Li, *Columbia University*
Xuheng Li, *Columbia University*
Christoffer Dall, *Arm Ltd*
Ronghui Gu, *Columbia University*
Jason Nieh, *Columbia University*
Yousuf Sait, *Arm Ltd*
Gareth Stockwell, *Arm Ltd*

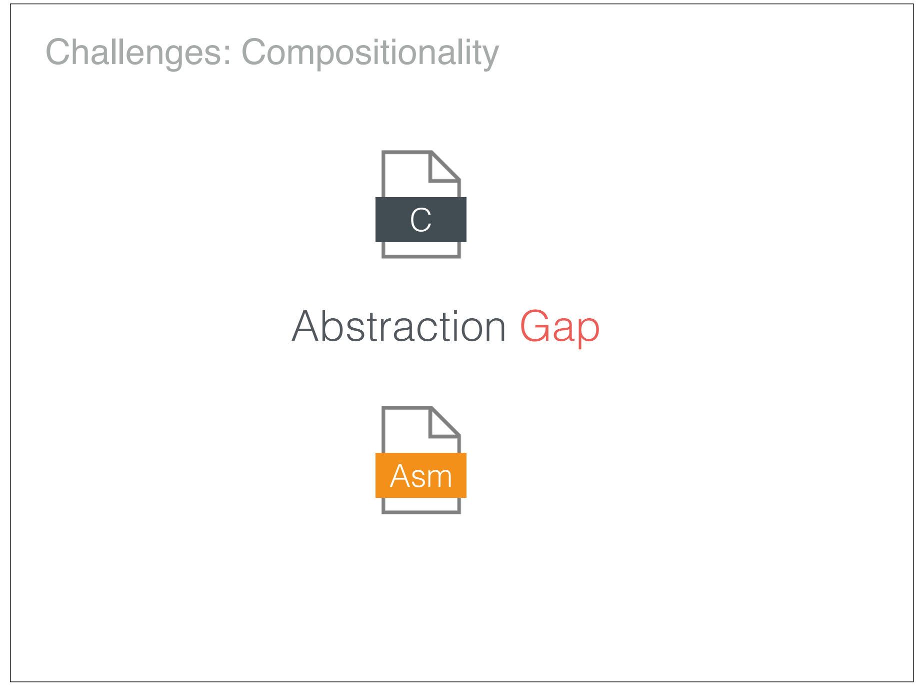
Abstract

The increasing use of sensitive private data in computing is matched by a growing concern regarding data privacy. System software such as hypervisors and operating systems are supposed to protect and isolate applications and their private data, but their large codebases contain many vulnerabilities that can risk data confidentiality and integrity. We introduce Realms, a new abstraction for confidential computing to protect the data confidentiality and integrity of virtual machines. Hardware creates and enforces Realm world, a new physical address space for Realms. Firmware controls the hardware to secure

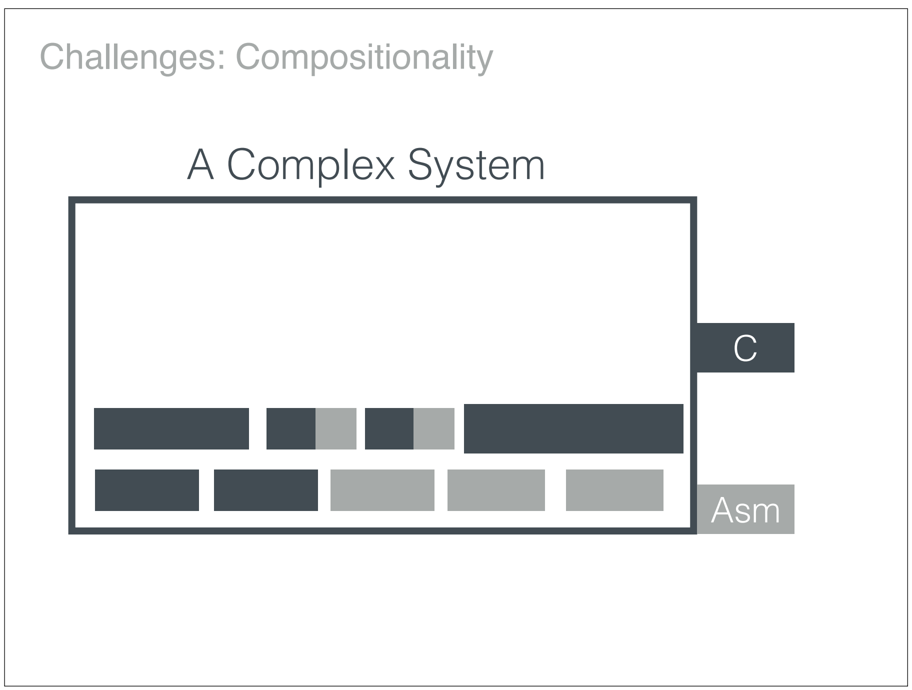
To address this problem, we introduce the *Arm Confidential Compute Architecture (Arm CCA)*. CCA provides *Realms*, secure execution environments that are completely opaque to privileged, untrusted system software such as OSes and hypervisors. CCA retains the ability of existing system software to manage hardware resources for Realms while preventing it from violating Realm confidentiality and integrity. For example, a hypervisor should retain its ability to dynamically allocate memory to or free memory from a Realm VM, but must never be allowed to access the protected memory contents of a Realm VM. CCA guarantees the confidentiality and integrity of Realm code and data in use, that is data in CPU



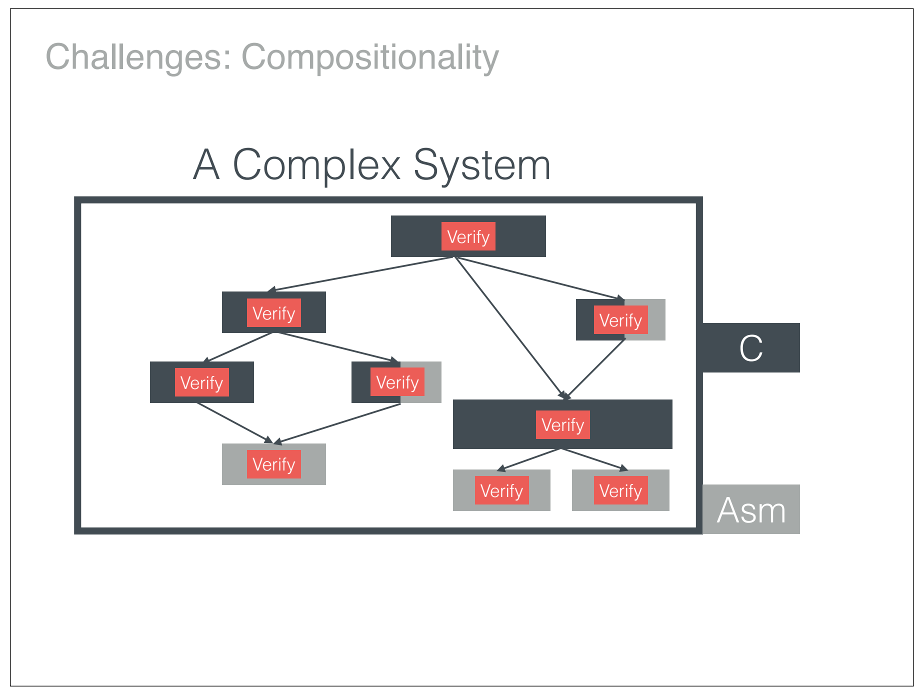
Motivation
Formal Verification
is the only
mathematically prove
program meets specification
under all inputs
under all execution
rule out entire classes of attacks
—NSF SFM Report[2016]



Challenges: Compositionality
C
Abstraction Gap
Asm

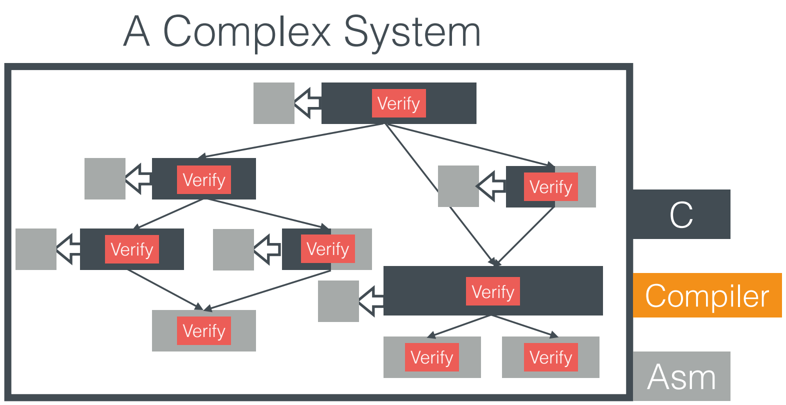


Challenges: Compositionality
A Complex System
C
Asm

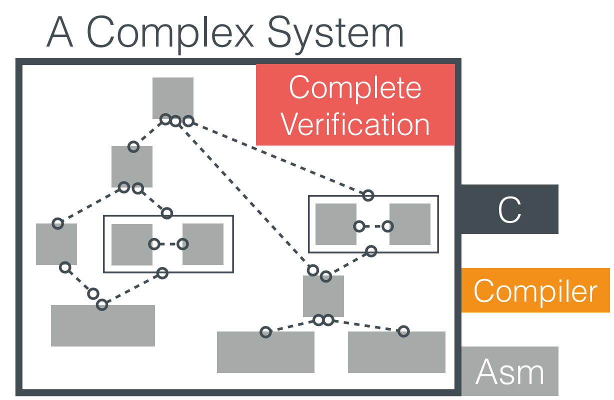


Challenges: Compositionality
A Complex System
Verify
Verify
Verify
Verify
Verify
Verify
Verify
Verify
Verify
Verify
C
Asm

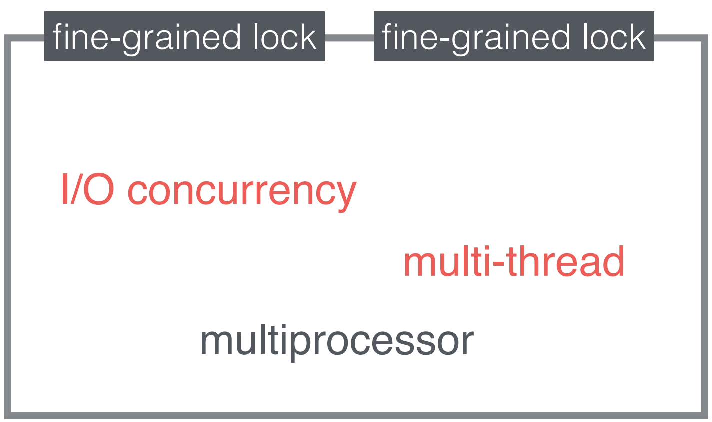
Challenges: Compositionality



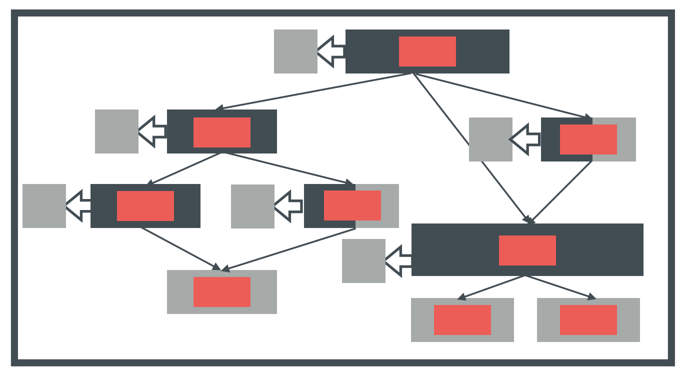
Challenges: Compositionality



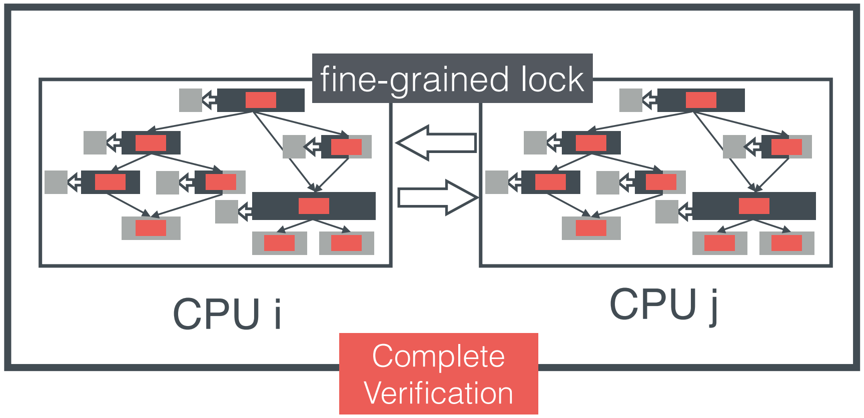
Challenges: Concurrency



Challenges: Concurrency

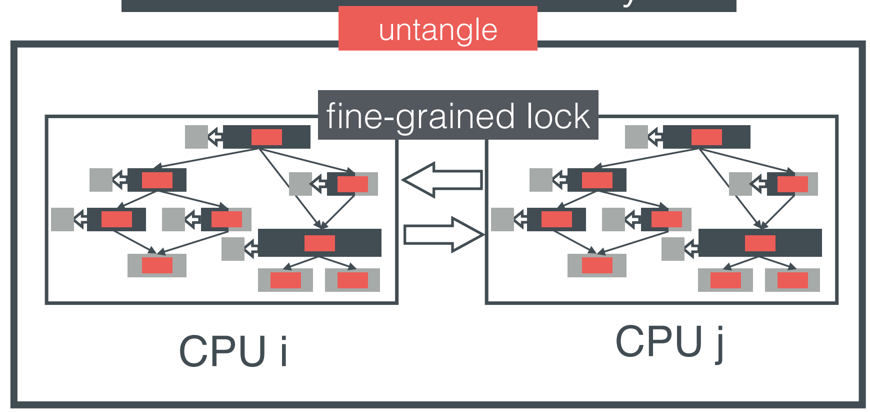


Challenges: Concurrency



Contribution

Certified Abstraction Layers



Contribution

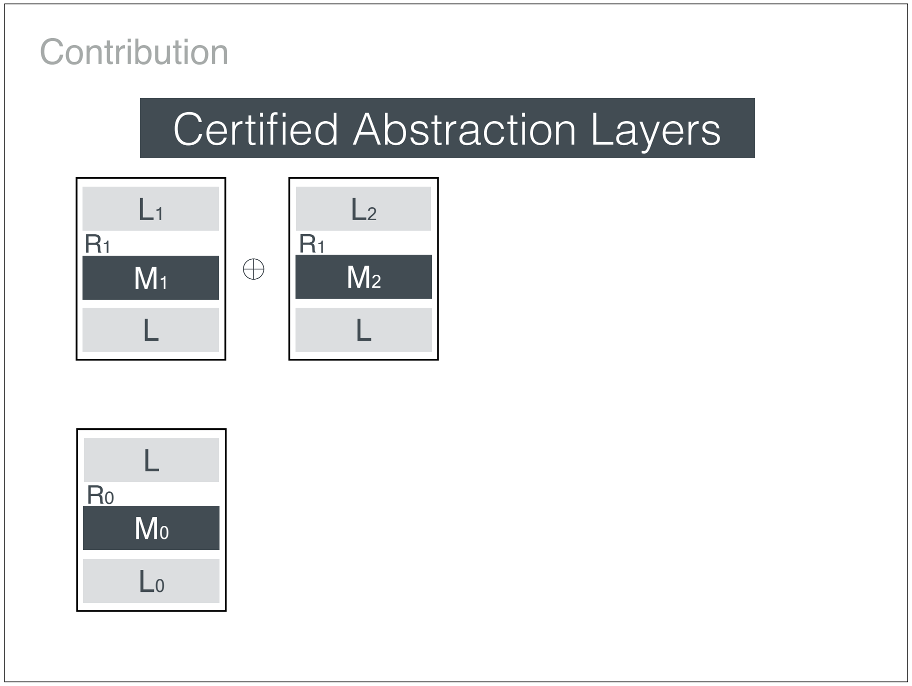
Certified Abstraction Layers

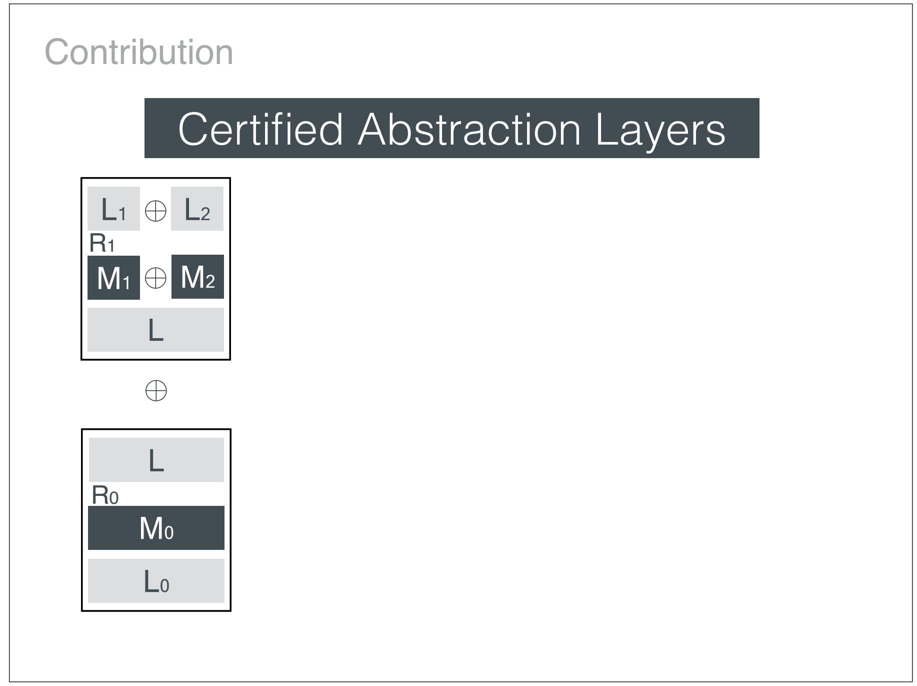
- verify existing systems
- build the next generation heterogeneous systems **designed** to be reliable and secure

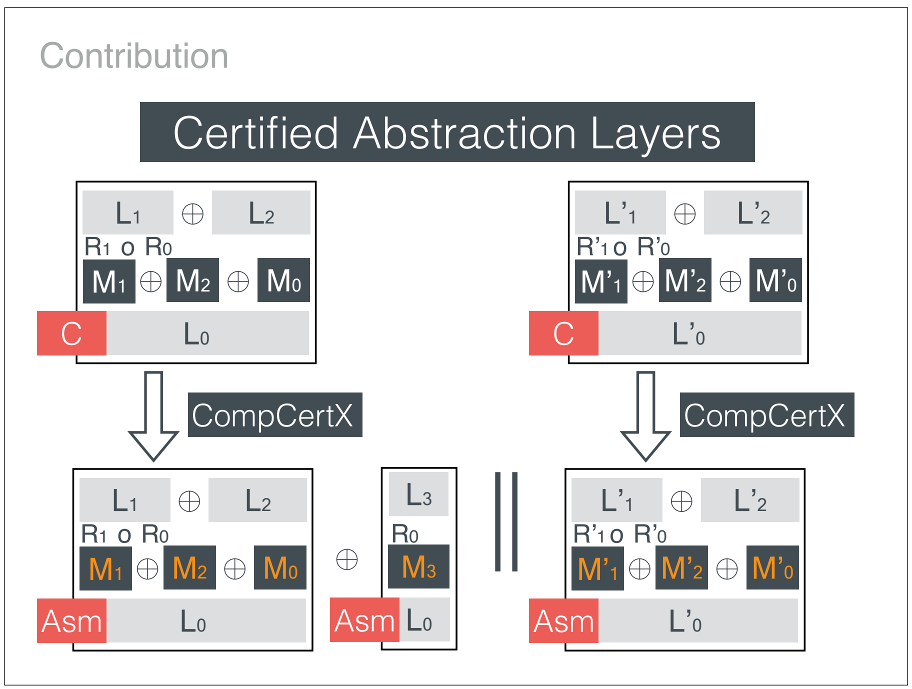
Contribution

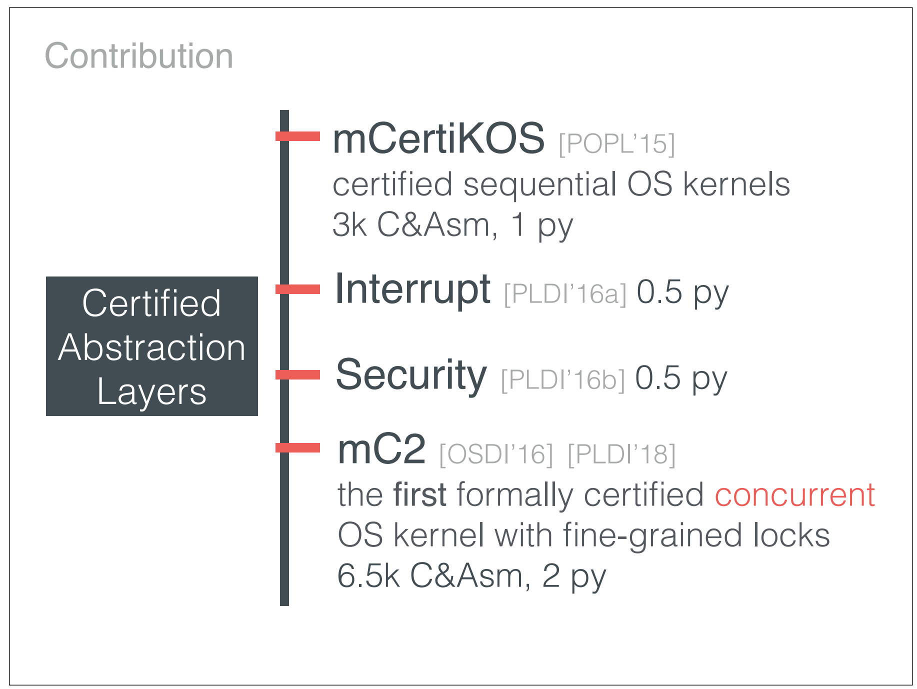
Certified Abstraction Layers

- verify existing systems
- build certified heterogeneous systems


Contribution
Certified Abstraction Layers
L1
R1
M1
L
⊕
L2
R1
M2
L
L
R0
M0
L0


Contribution
Certified Abstraction Layers
L1 ⊕ L2
R1
M1 ⊕ M2
L
⊕
L
R0
M0
L0


Contribution
Certified Abstraction Layers
L1 ⊕ L2
R1 o R0
M1 ⊕ M2 ⊕ M0
C
L0
CompCertX
L1 ⊕ L2
R1 o R0
M1 ⊕ M2 ⊕ M0
Asm
L0
⊕
L3
R0
M3
Asm
L0
L'1 ⊕ L'2
R'1 o R'0
M'1 ⊕ M'2 ⊕ M'0
C
L'0
CompCertX
L'1 ⊕ L'2
R'1 o R'0
M'1 ⊕ M'2 ⊕ M'0
Asm
L'0

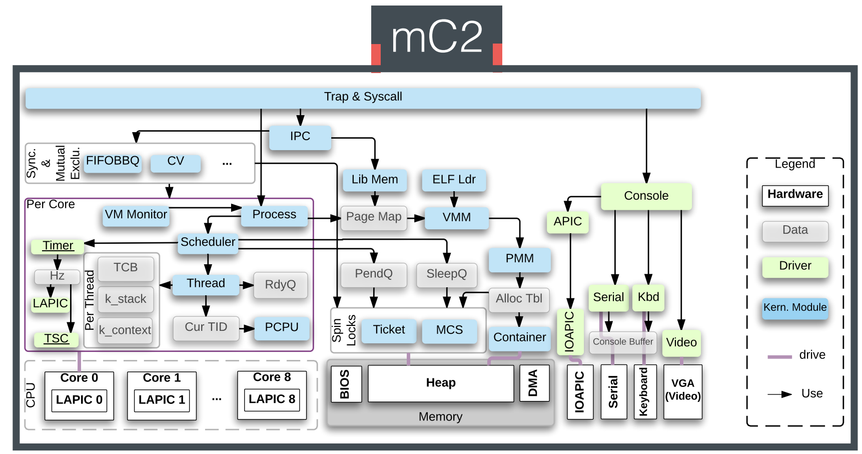

Contribution
Certified Abstraction Layers
mCertiKOS [POPL'15]
certified sequential OS kernels
3k C&Asm, 1 py
Interrupt [PLDI'16a] 0.5 py
Security [PLDI'16b] 0.5 py
mC2 [OSDI'16] [PLDI'18]
the first formally certified concurrent
OS kernel with fine-grained locks
6.5k C&Asm, 2 py

Contribution

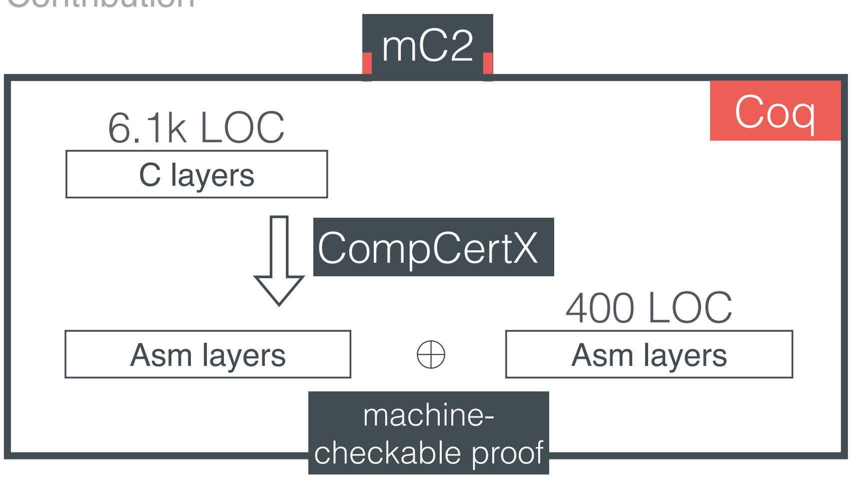
Certified System Software

- functional **correctness**
- **liveness**
- **no** stack/integer/buffer overflow
- **no** race condition

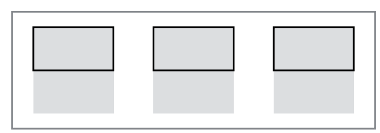
Contribution



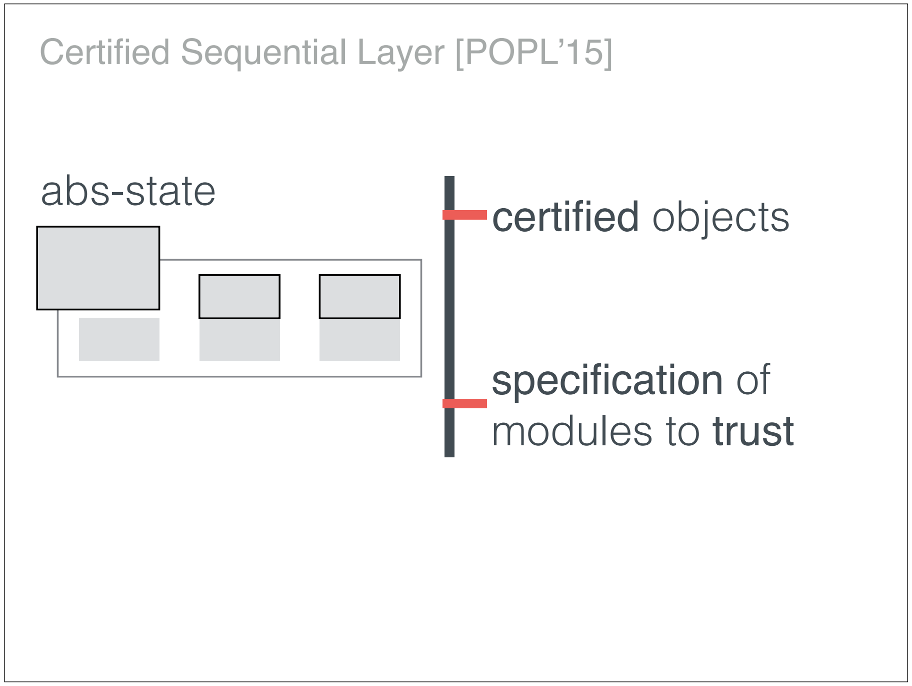
Contribution

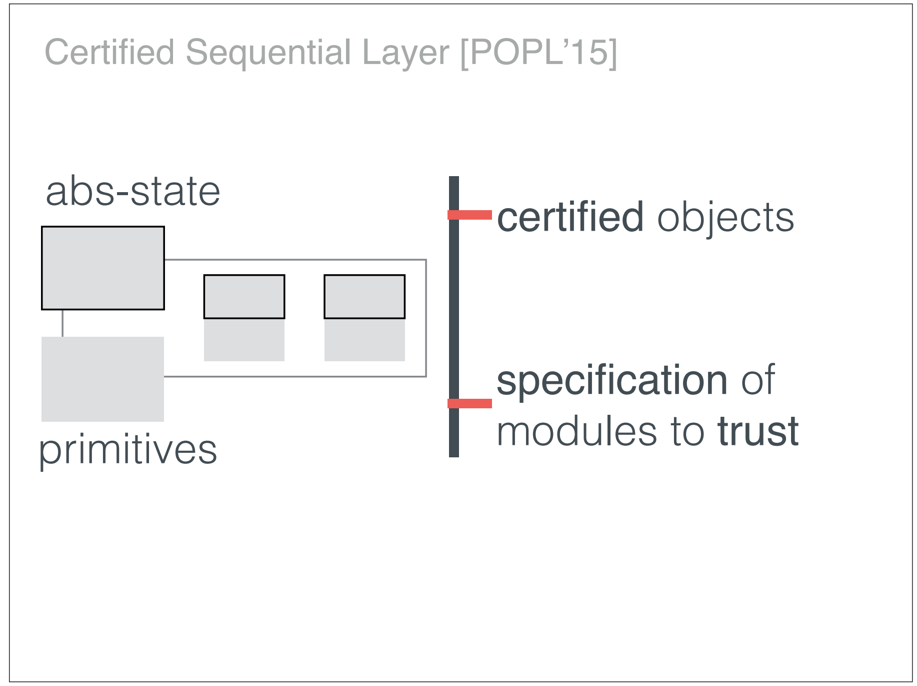


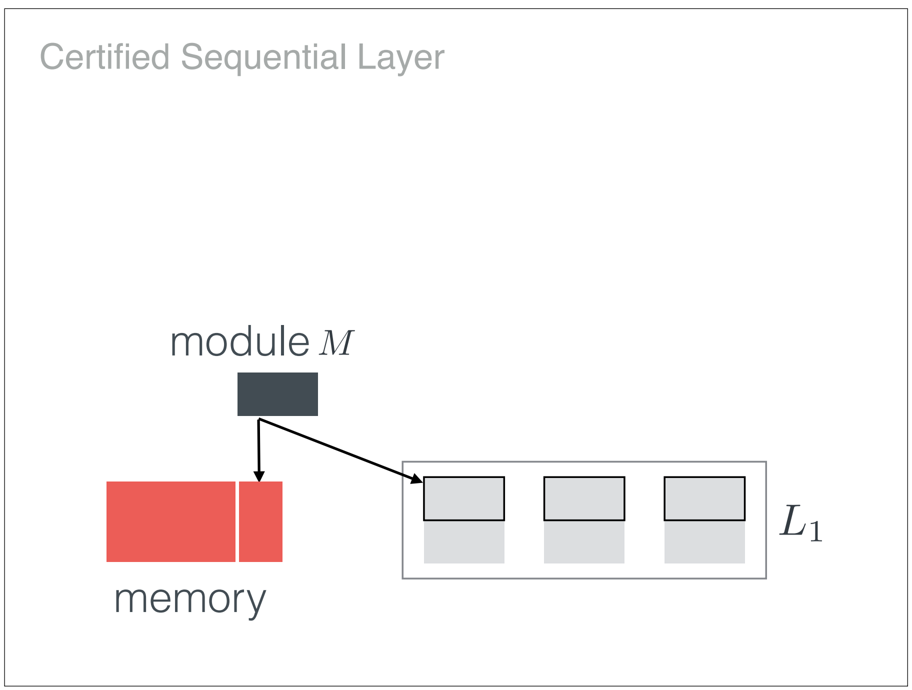
Certified Sequential Layer [POPL'15]

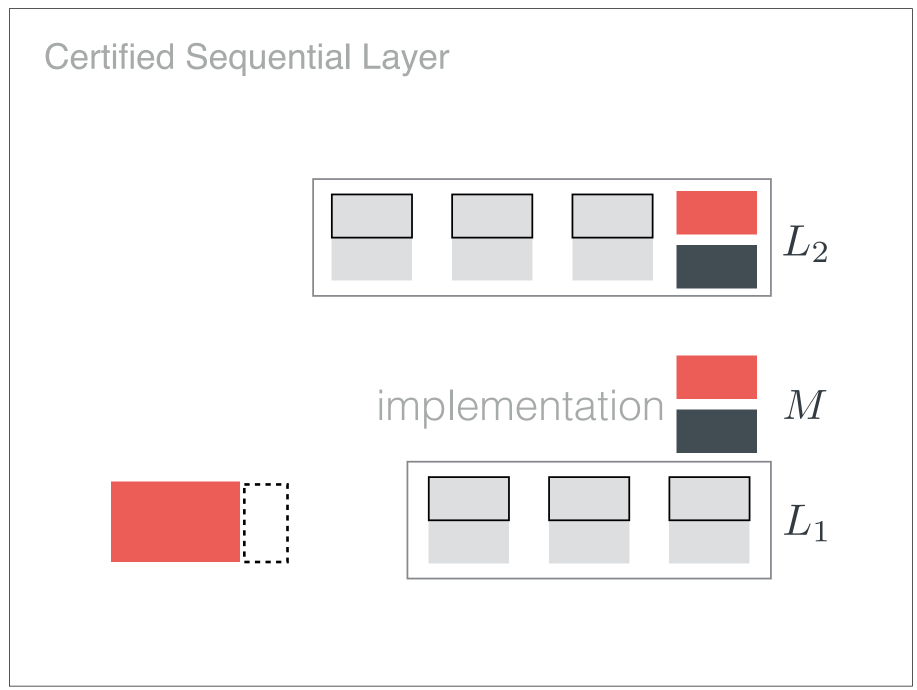


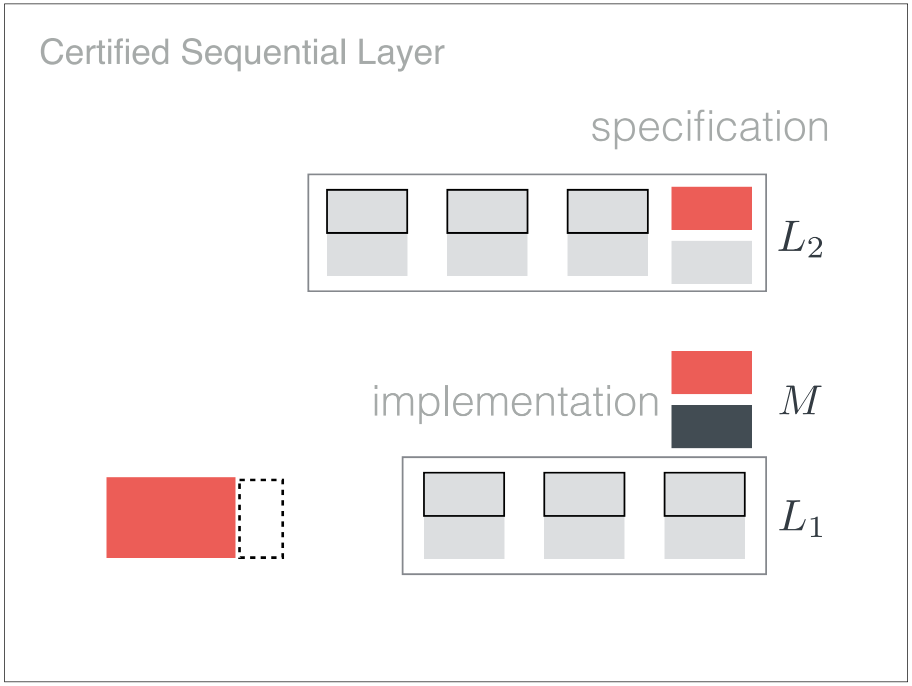
- **certified** objects
- **specification** of modules to **trust**

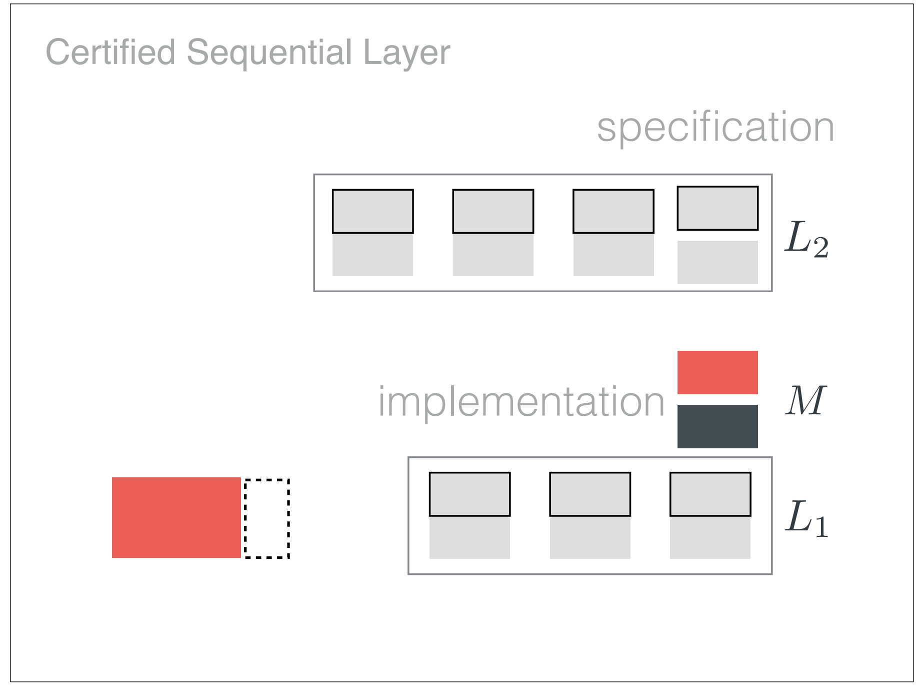

Certified Sequential Layer [POPL'15]
abs-state
certified objects
specification of modules to trust

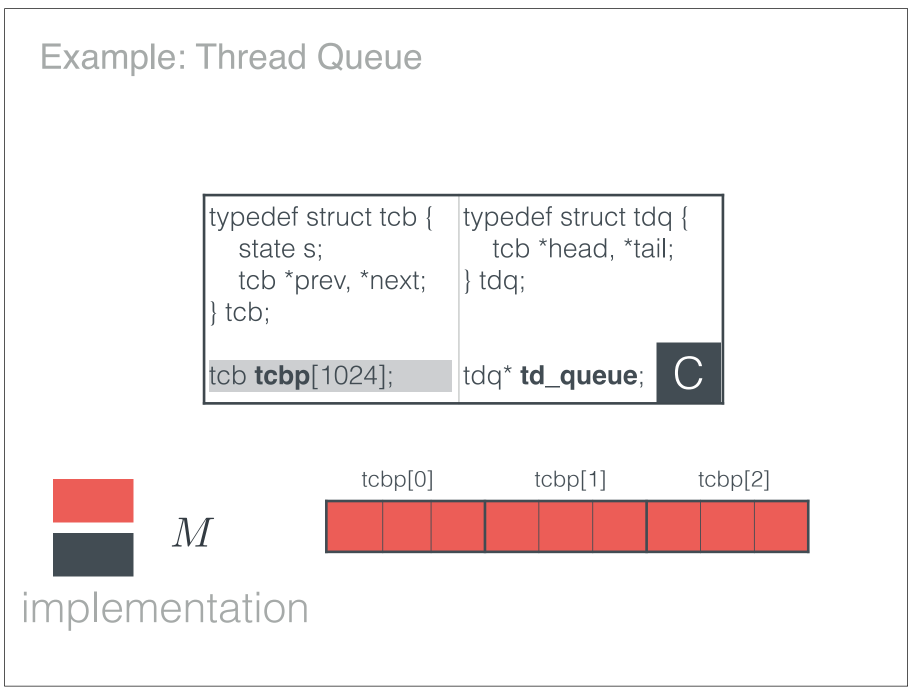

Certified Sequential Layer [POPL'15]
abs-state
primitives
certified objects
specification of modules to trust

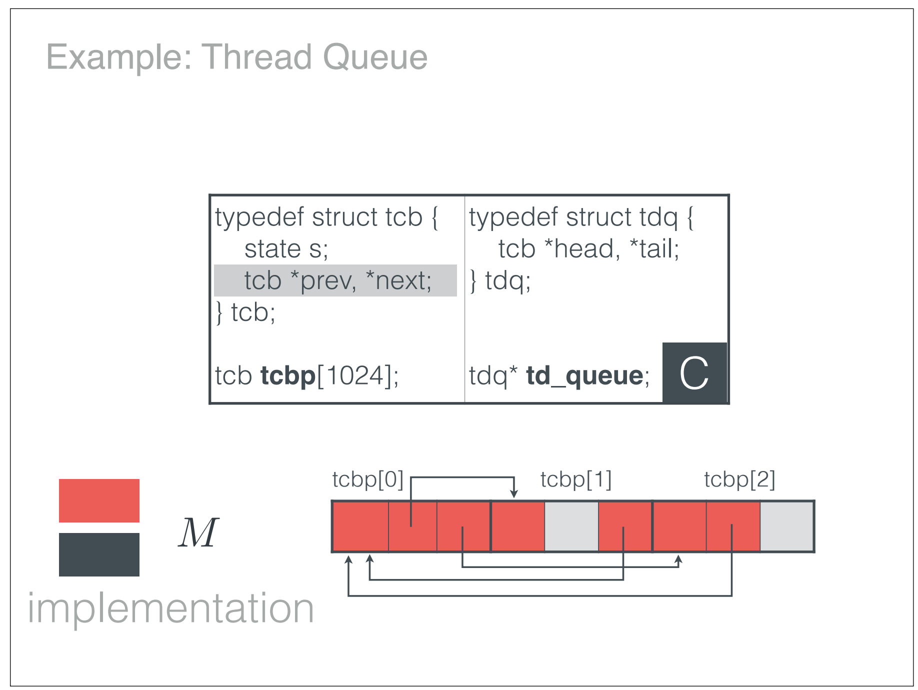

Certified Sequential Layer
module M
memory
L_1


Certified Sequential Layer
L_2
implementation
M
L_1


Certified Sequential Layer
specification
L_2
implementation
M
L_1


Certified Sequential Layer
specification
L_2
implementation
M
L_1


Example: Thread Queue
typedef struct tcb {
state s;
tcb *prev, *next;
} tcb;
tcb tcbp[1024];
typedef struct tdq {
tcb *head, *tail;
} tdq;
tdq* td_queue;
C
M
implementation
tcbp[0]
tcbp[1]
tcbp[2]


Example: Thread Queue
typedef struct tcb {
state s;
tcb *prev, *next;
} tcb;
tcb tcbp[1024];
typedef struct tdq {
tcb *head, *tail;
} tdq;
tdq* td_queue;
C
M
implementation
tcbp[0]
tcbp[1]
tcbp[2]

Example: Thread Queue

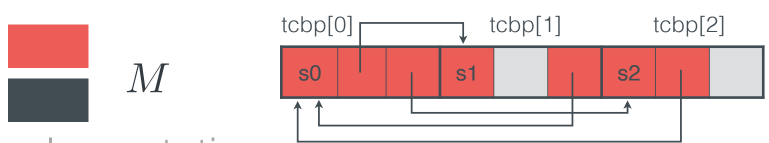
```
typedef struct tcb {
    state s;
    tcb *prev, *next;
} tcb;

tcb tcbp[1024];
```

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

tdq* td_queue;
```

C



implementation

Example: Thread Queue

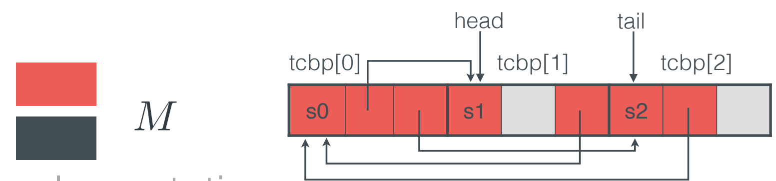
```
typedef struct tcb {
    state s;
    tcb *prev, *next;
} tcb;

tcb tcbp[1024];
```

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

tdq* td_queue;
```

C

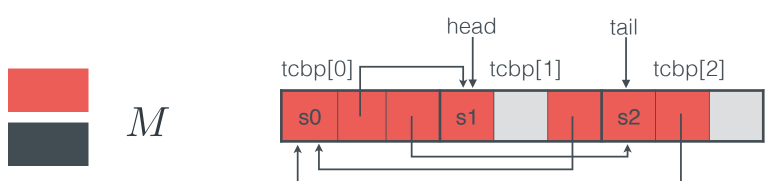


implementation

Example: Thread Queue

```
tcb* dequeue(tdq* q) {
    tcb *head, *next;
    tcb *i = null;
    if (!q) return i;
    head = q -> head;
    if (!head) return i;
    i = head;
    next = i -> next;
    if (!next) {
        q -> head = null;
        q -> tail = null;
    } else {
        next -> prev = null;
        q -> head = next;
    }
    return i;
}
```

C

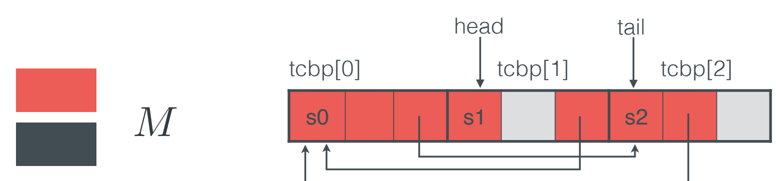


implementation

Example: Thread Queue

```
tcb* dequeue(tdq* q) {
    tcb *head, *next;
    tcb *i = null;
    if (!q) return i;
    head = q -> head;
    if (!head) return i;
    i = head;
    next = i -> next;
    if (!next) {
        q -> head = null;
        q -> tail = null;
    } else {
        next -> prev = null;
        q -> head = next;
    }
    return i;
}
```

C



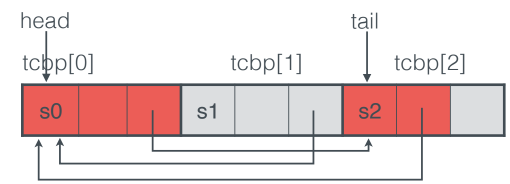
implementation

Example: Thread Queue

```
tcb* dequeue(tdq* q) {
    tcb *head, *next;
    tcb *i = null;
    if (!q) return i;
    head = q -> head;
    if (!head) return i;
    i = head;
    next = i -> next;
    if (!next) {
        q -> head = null;
        q -> tail = null;
    } else {
        next -> prev = null;
        q -> head = next;
    }
    return i;
}
```

C

M



implementation

Example: Thread Queue

specification

L_2

Definition **tcbp** := ZMap.t state.
Definition **td_queue** := List Z.

Coq

Example: Thread Queue

specification

L_2

tcbp(0) tcbp(1) tcbp(2)

s0 s1 s2

Definition **tcbp** := ZMap.t state.
Definition **td_queue** := List Z.

Coq

Example: Thread Queue

specification

L_2

tcbp(0) tcbp(1) tcbp(2)

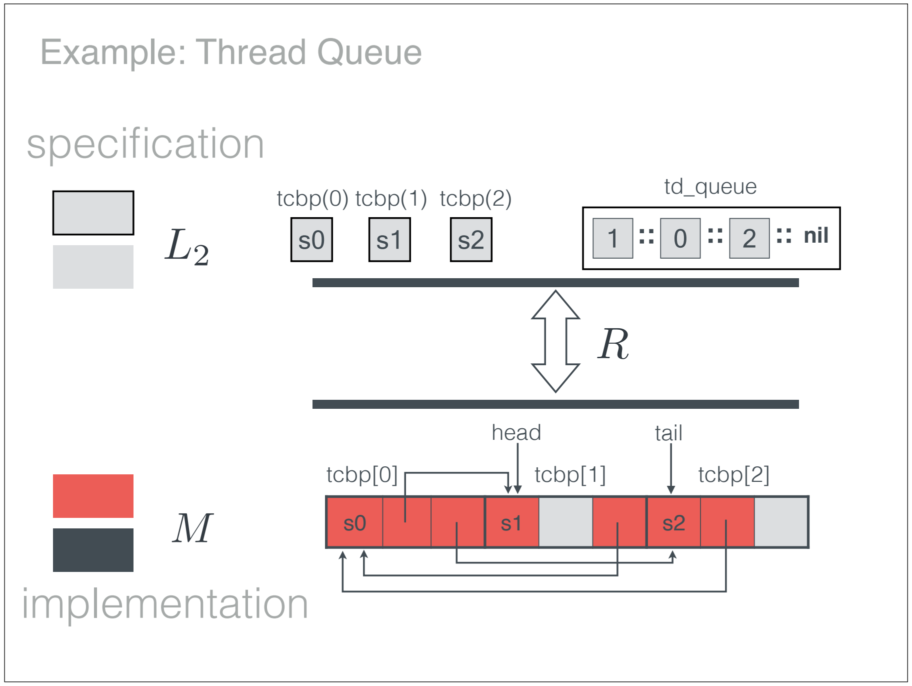
s0 s1 s2

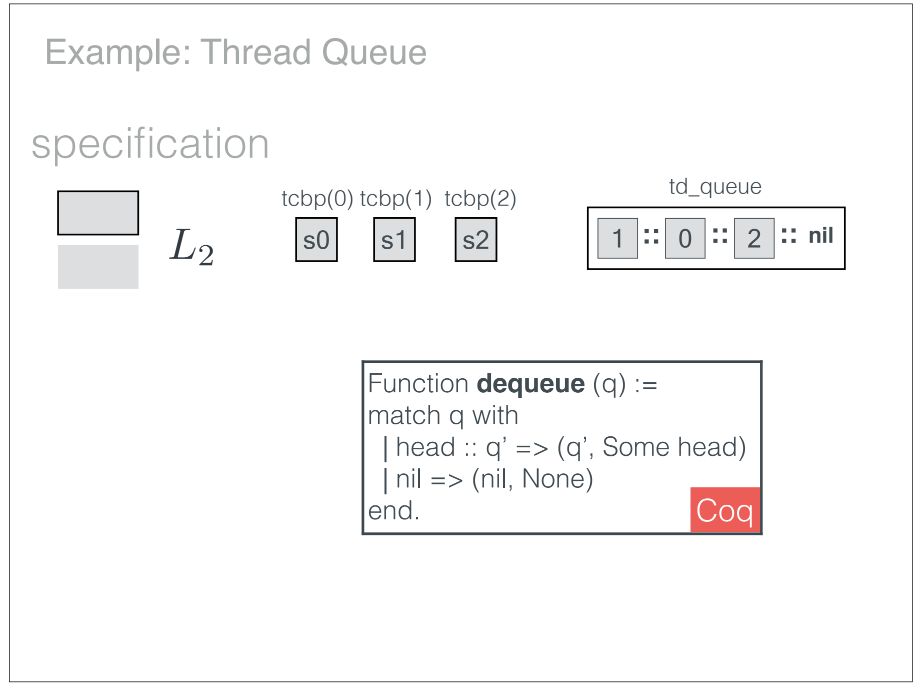
td_queue

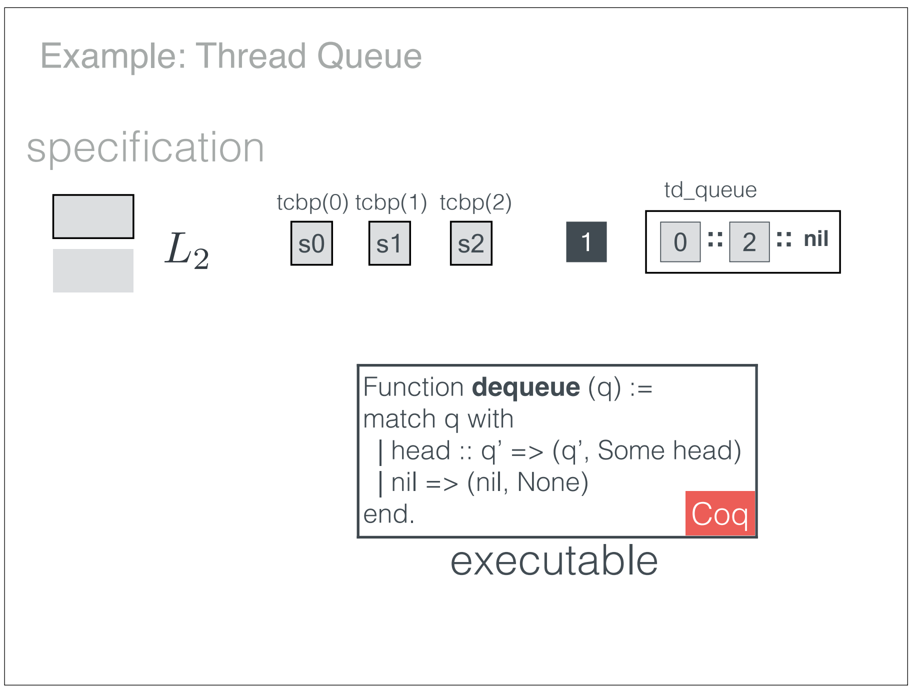
1 :: 0 :: 2 :: **nil**

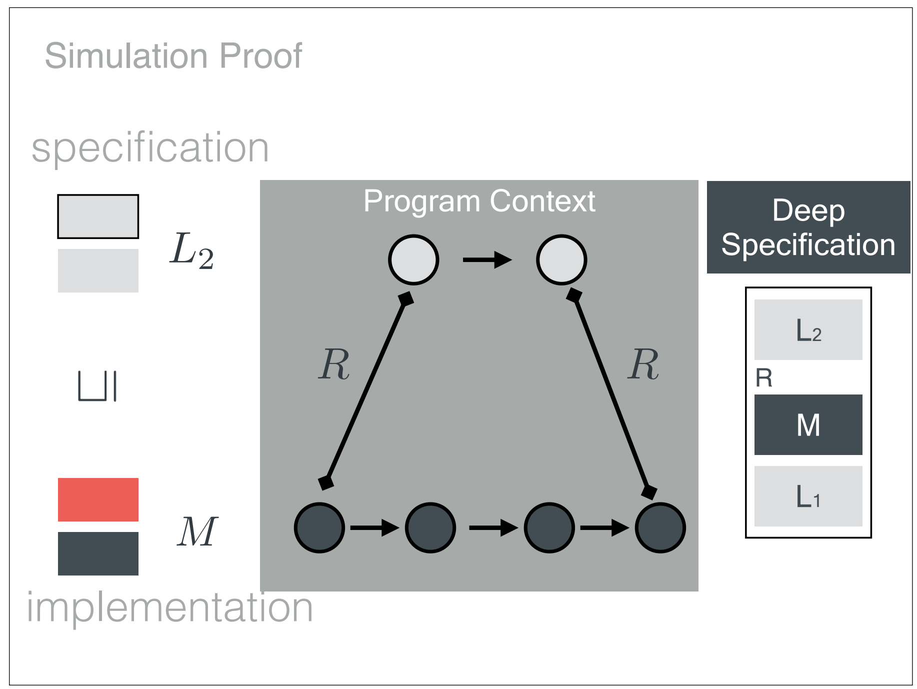
Definition **tcbp** := ZMap.t state.
Definition **td_queue** := List Z.

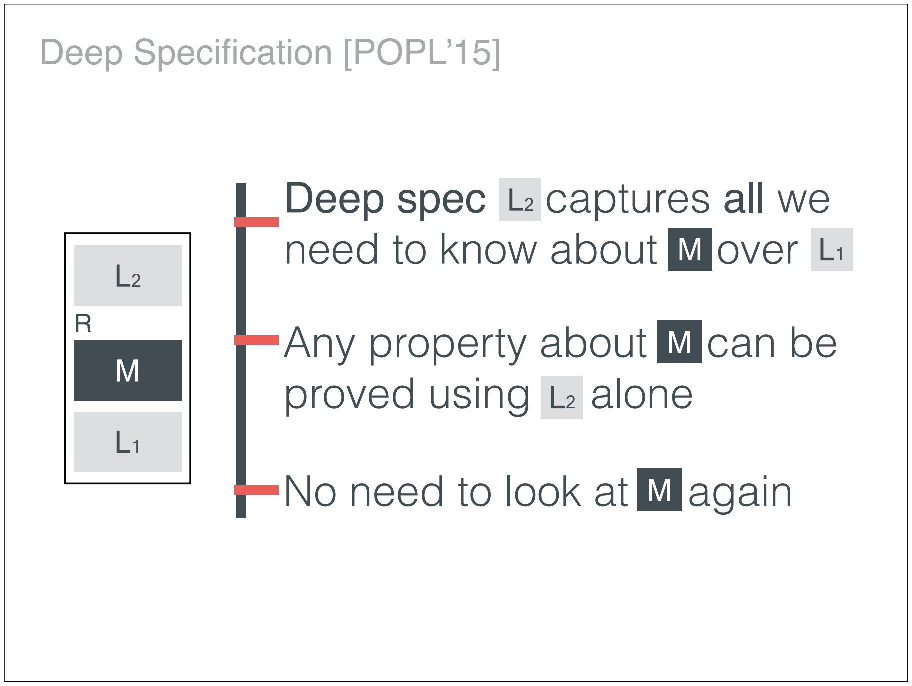
Coq

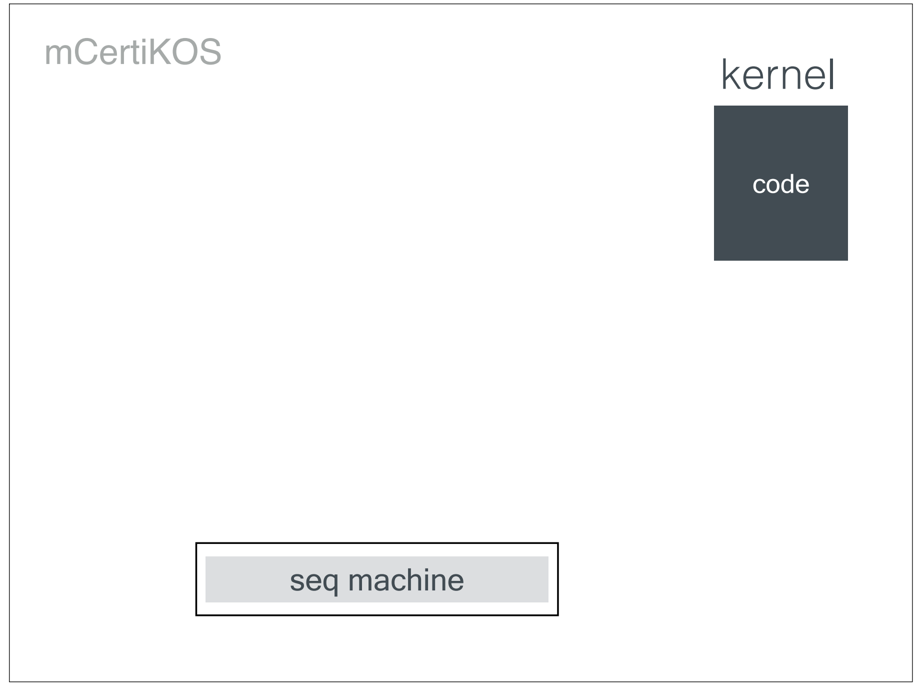

Example: Thread Queue
specification
L2
tcbp(0) tcbp(1) tcbp(2)
s0 s1 s2
td_queue
1 :: 0 :: 2 :: nil
R
head
tail
tcbp[0]
tcbp[1]
tcbp[2]
s0 s1 s2
M
implementation

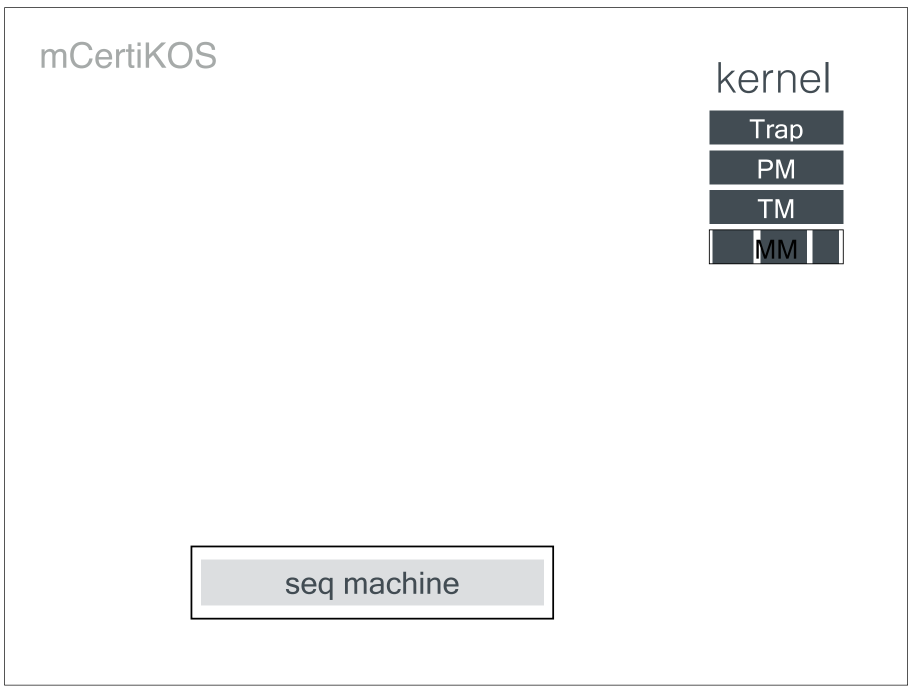

Example: Thread Queue
specification
L2
tcbp(0) tcbp(1) tcbp(2)
s0 s1 s2
td_queue
1 :: 0 :: 2 :: nil
Function dequeue (q) :=
match q with
| head :: q' => (q', Some head)
| nil => (nil, None)
end.
Coq

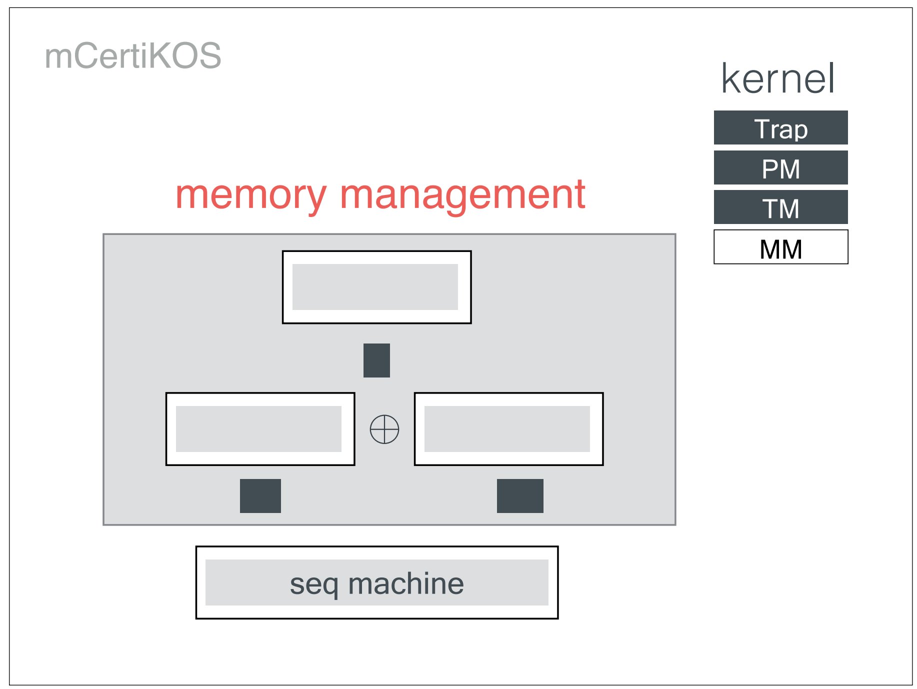

Example: Thread Queue
specification
L2
tcbp(0) tcbp(1) tcbp(2)
s0 s1 s2
1
td_queue
0 :: 2 :: nil
Function dequeue (q) :=
match q with
| head :: q' => (q', Some head)
| nil => (nil, None)
end.
Coq
executable

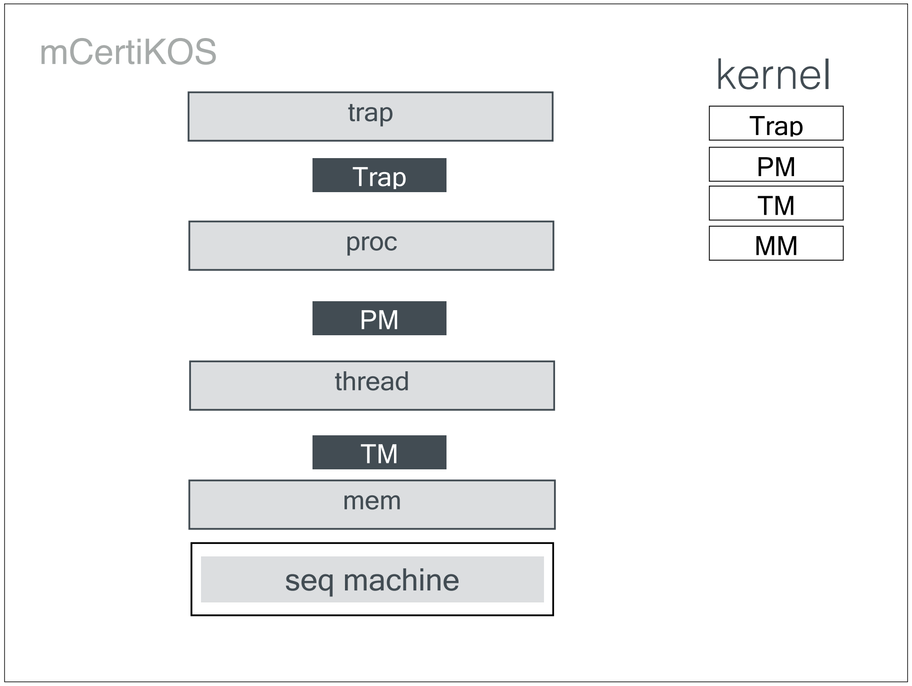

Simulation Proof
specification
L2
⊑
M
implementation
Program Context
R
R
Deep Specification
L2
R
M
L1

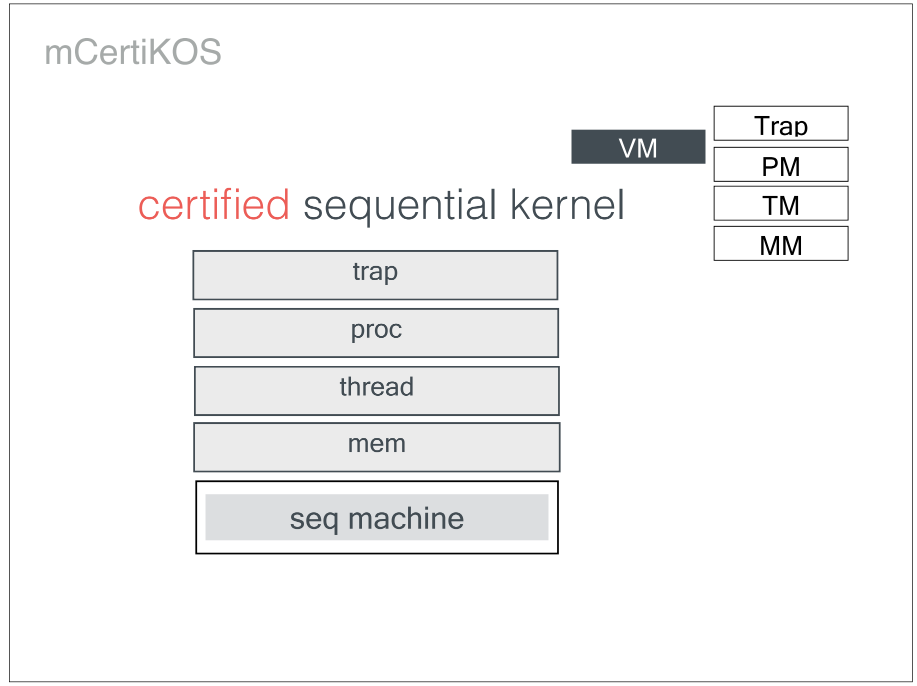

Deep Specification [POPL'15]
L2
R
M
L1
Deep spec L2 captures all we need to know about M over L1
Any property about M can be proved using L2 alone
No need to look at M again

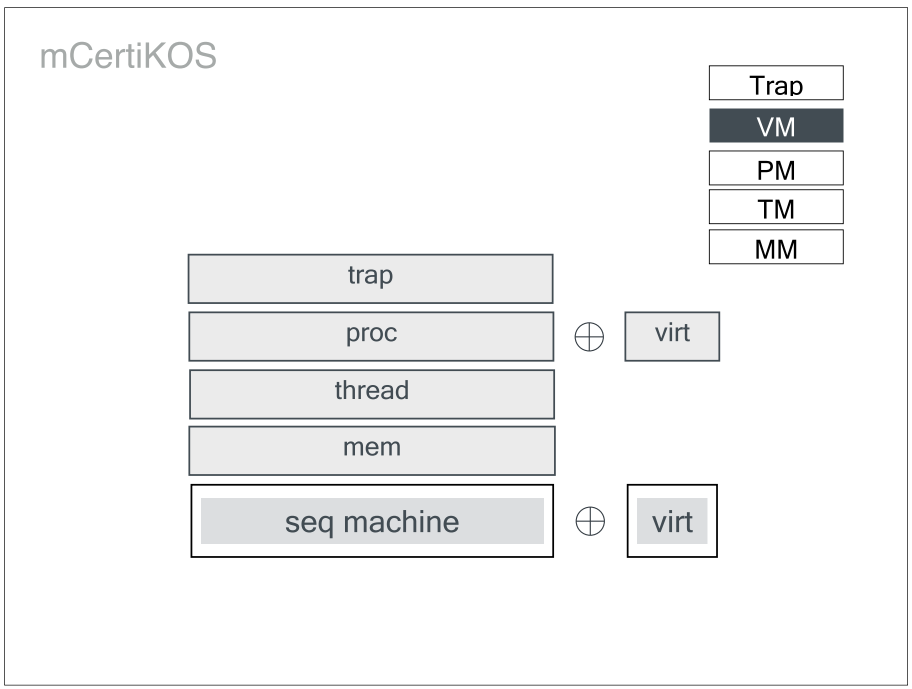

mCertiKOS
kernel
code
seq machine

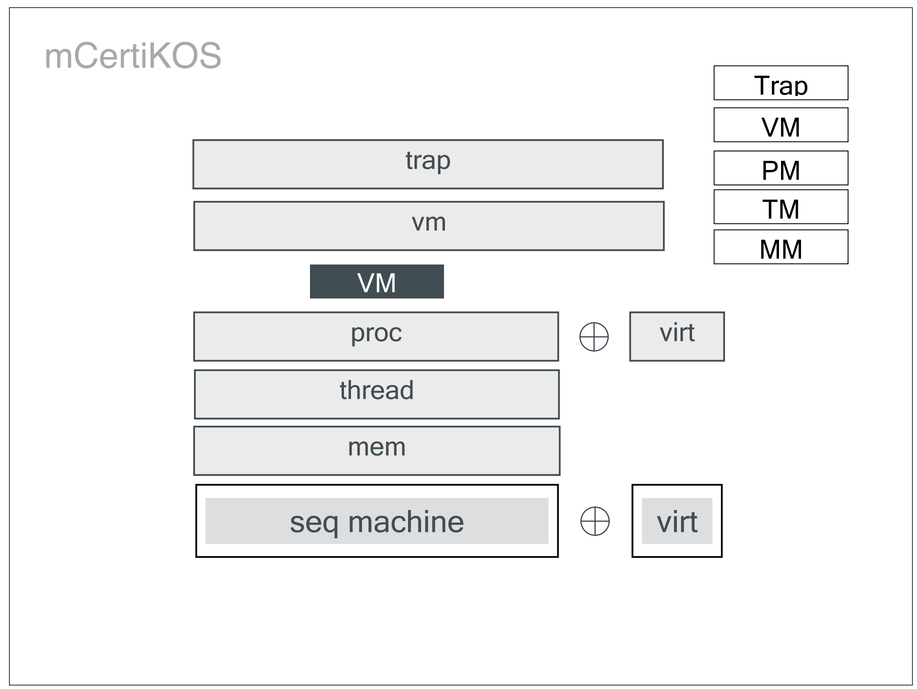

mCertiKOS
kernel
Trap
PM
TM
MM
seq machine

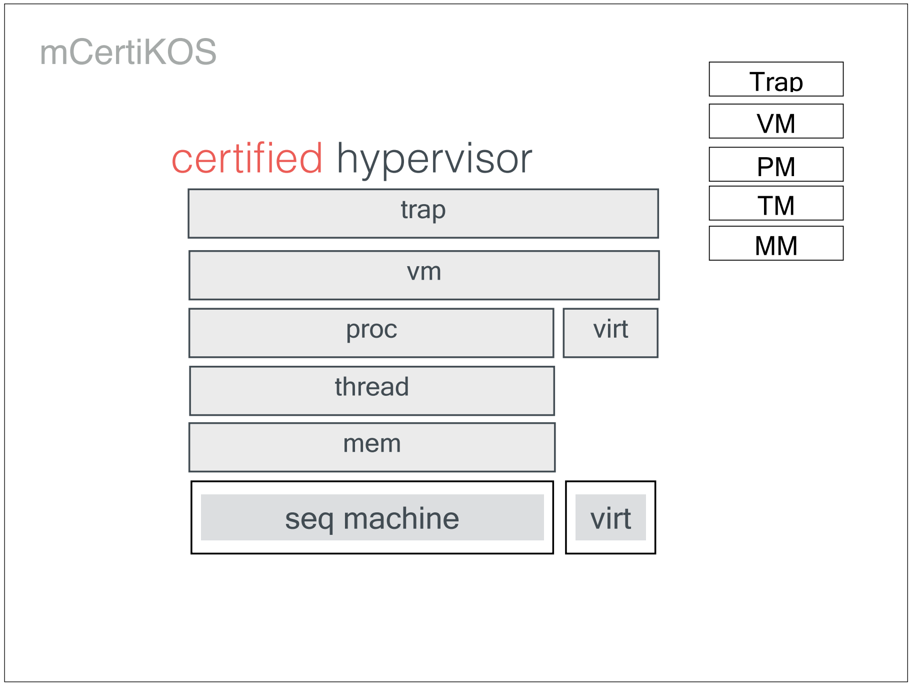

mCertiKOS
kernel
Trap
PM
TM
MM
memory management
seq machine

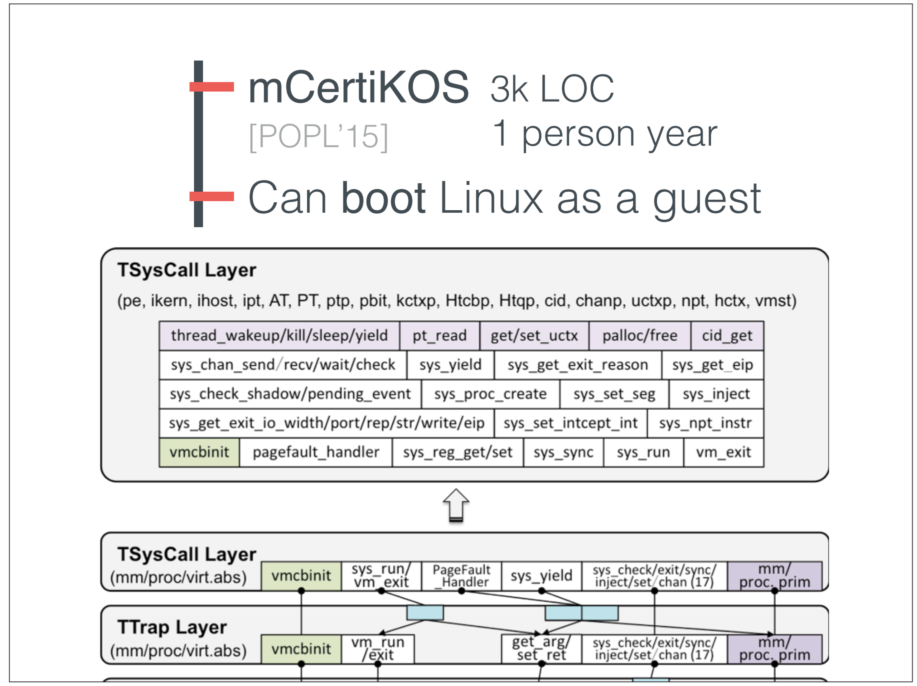

mCertiKOS
kernel
Trap
PM
TM
MM
trap
Trap
proc
PM
thread
TM
mem
seq machine

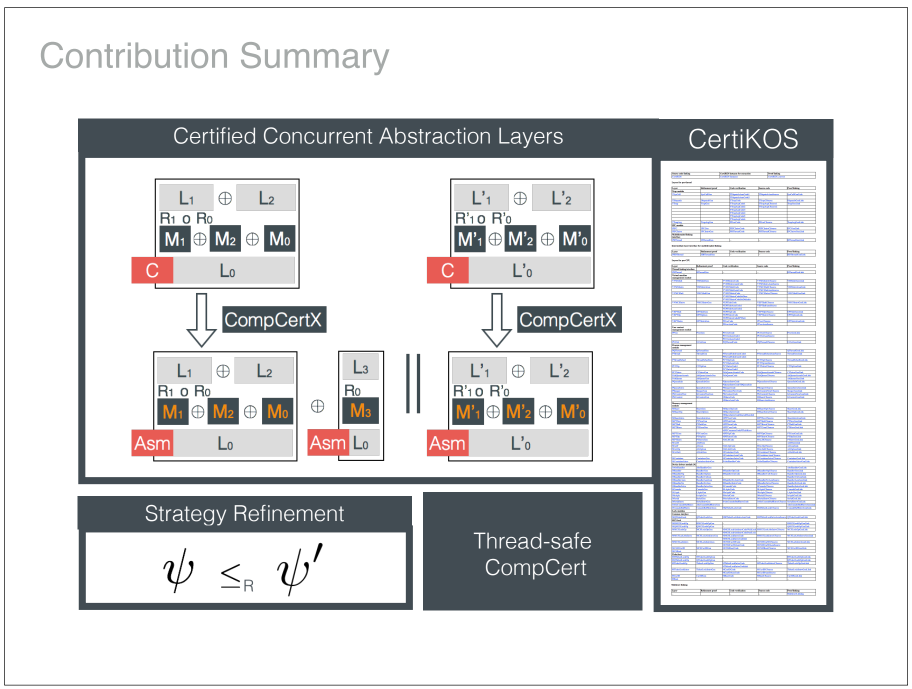

mCertiKOS
VM
Trap
PM
TM
MM
certified sequential kernel
trap
proc
thread
mem
seq machine

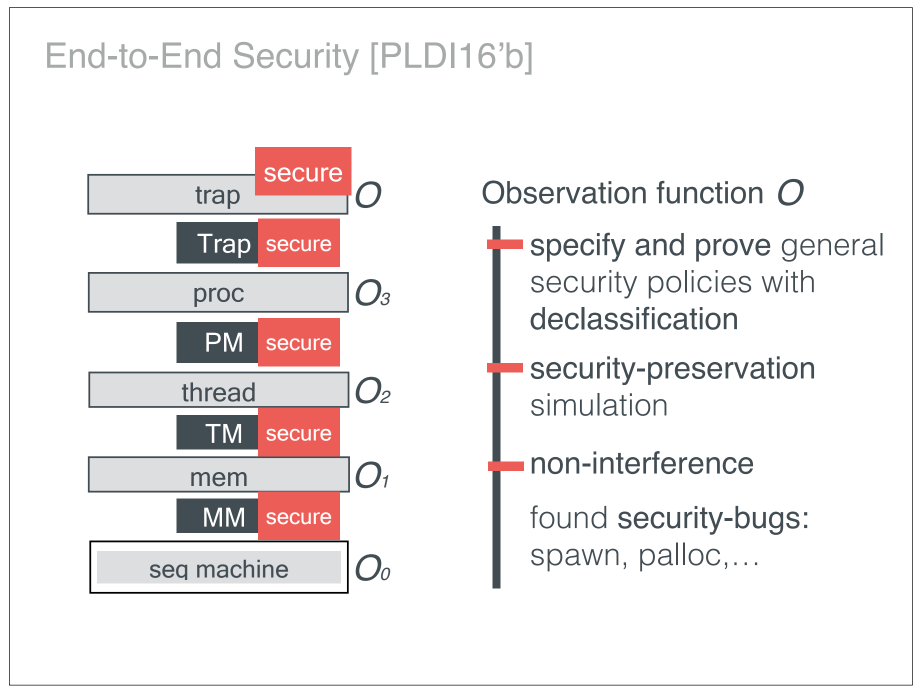

mCertiKOS
Trap
VM
PM
TM
MM
trap
proc
virt
thread
mem
seq machine
virt


mCertiKOS
Trap
VM
PM
TM
MM
trap
vm
VM
proc
virt
thread
mem
seq machine
virt


mCertiKOS
certified hypervisor
trap
vm
proc
virt
thread
mem
seq machine
virt
Trap
VM
PM
TM
MM

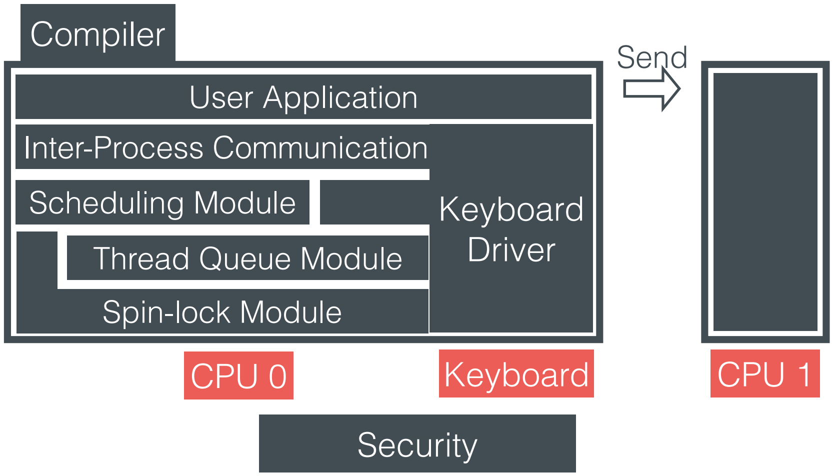

mCertiKOS 3k LOC
[POPL'15] 1 person year
Can boot Linux as a guest
TSysCall Layer
(pe, ikern, ihost, ipt, AT, PT, ptp, pbit, kctxp, Htcbp, Htqp, cid, chanp, uctxp, npt, hctx, vmst)
thread_wakeup/kill/sleep/yield
pt_read
get/set_uctx
palloc/free
cid_get
sys_chan_send/recv/wait/check
sys_yield
sys_get_exit_reason
sys_get_eip
sys_check_shadow/pending_event
sys_proc_create
sys_set_seg
sys_inject
sys_get_exit_io_width/port/rep/str/write/eip
sys_set_intcept_int
sys_npt_instr
vmcbinit
pagefault_handler
sys_reg_get/set
sys_sync
sys_run
vm_exit
TSysCall Layer
(mm/proc/virt.abs)
vmcbinit
sys_run/ vm_exit
PageFault Handler
sys_yield
sys_check/exit/sync/ inject/set_chan (17)
mm/ proc_prim
TTrap Layer
(mm/proc/virt.abs)
vmcbinit
vm_run /exit
get_arg/ set_ret
sys_check/exit/sync/ inject/set_chan (17)
mm/ proc_prim


Contribution Summary
Certified Concurrent Abstraction Layers
CertiKOS
L1 ⊕ L2
R1 o R0
M1 ⊕ M2 ⊕ M0
C
L0
CompCertX
L1 ⊕ L2
R1 o R0
M1 ⊕ M2 ⊕ M0
Asm
L0
⊕
L3
R0
M3
Asm
L0
||
L'1 ⊕ L'2
R'1 o R'0
M'1 ⊕ M'2 ⊕ M'0
C
L'0
CompCertX
L'1 ⊕ L'2
R'1 o R'0
M'1 ⊕ M'2 ⊕ M'0
Asm
L'0
Strategy Refinement
ψ ≤R ψ'
Thread-safe CompCert


End-to-End Security [PLDI16'b]
trap
secure
O
Trap
secure
proc
O3
PM
secure
thread
O2
TM
secure
mem
O1
MM
secure
seq machine
O0
Observation function O
specify and prove general security policies with declassification
security-preservation simulation
non-interference
found security-bugs: spawn, palloc,…

Case Study

Build a Certified System



Summary: The CertiKOS / DeepSpec Project

Killer-app: high-assurance “heterogeneous” systems of systems!

Conjecture: today’s PLs fail because they ignored OS, and today’s OSes fail because they get little help from PLs

New Insights:

- deepspec & certified abstraction layers;
- a unifying framework for composing heterogeneous components (via game semantics + linear logic connectives)

Opportunities:

- New certified system software stacks (CertiKOS ++)
- New certifying programming languages (DeepSEA vs. C & Asm)
- New certified programming tools
- New certified modeling & arch. description lang. (DeepSEA)
- We verify all interesting properties (correctness, safety, security, availability, …)