NICTA

L4.verified

Gerwin Klein

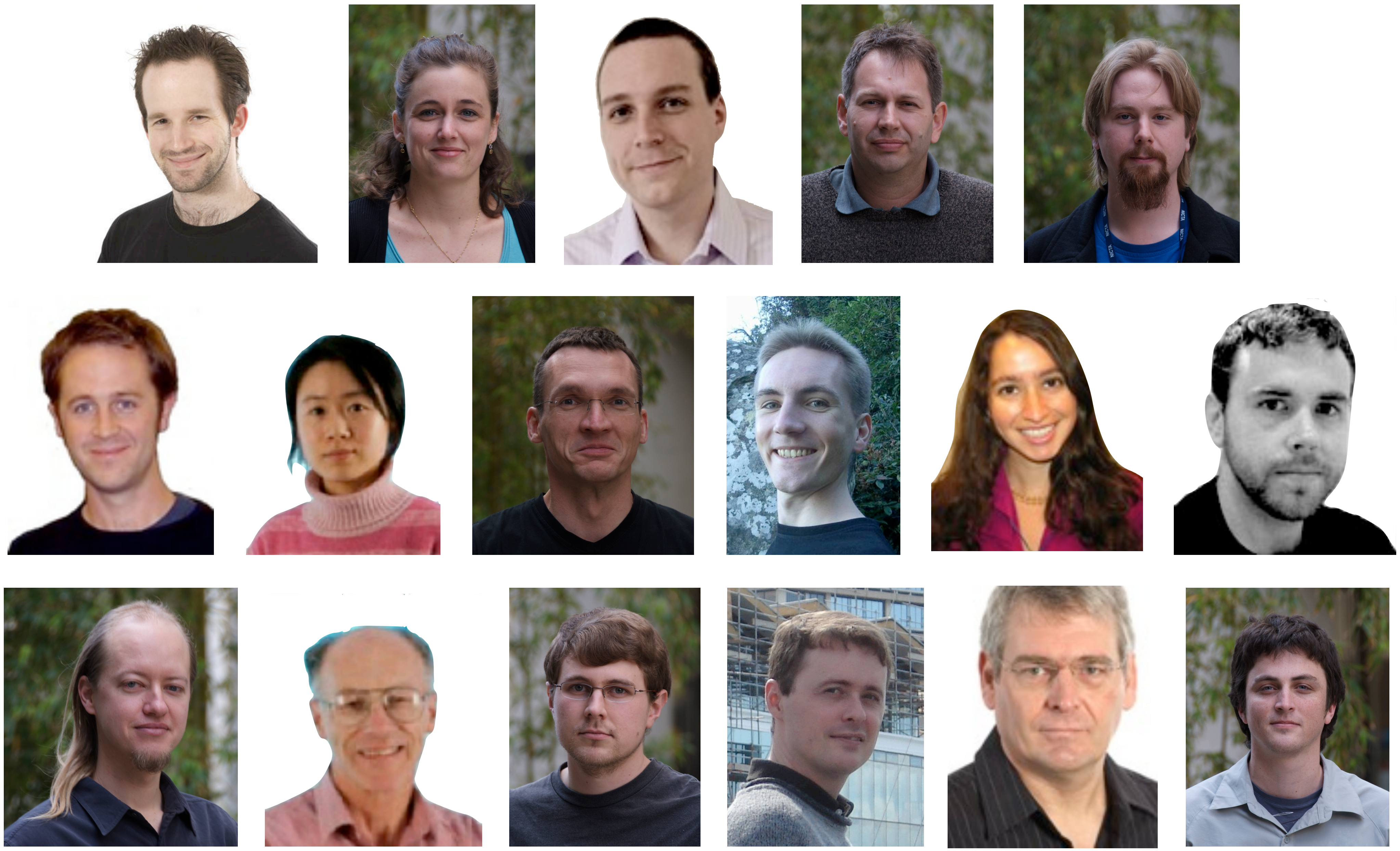
Australian Government
Department of Communications, Information Technology and the Arts
Australian Research Council

NICTA Members
ANU THE AUSTRALIAN NATIONAL UNIVERSITY
UNSW THE UNIVERSITY OF NEW SOUTH WALES
NSW Department of State and Regional Development

Victoria The Place To Be
THE UNIVERSITY OF MELBOURNE
The University of Sydney
Queensland Government
Griffith UNIVERSITY
QUT Queensland University of Technology
THE UNIVERSITY OF QUEENSLAND AUSTRALIA
NICTA Partners

The Team

The Team



L4.Verified

L4.Verified

1 microkernel

8,700 lines of C

0 bugs*

qed

*conditions apply

Windows

An exception 06 has occured at 0028:C11B3ADC in VxD DiskTSD(03) + 00001660. This was called from 0028:C11B40C8 in VxD voltrack(04) + 00000000. It may be possible to continue normally.

* Press any key to attempt to continue.
* Press CTRL+ALT+RESET to restart your computer. You will lose any unsaved information in all applications.

Press any key to continue


CLEAR


Windows Vista
Stunning. Breakthrough. Entertaining.
HP TouchSmart PC and Microsoft Windows Vista deliver you a PC experience designed to fit wherever life happens.
Innovative solutions brought to you by:
Microsoft
hp
BEST BUY


the Bay
the Bay

The Problem

Motivation

Annoying Problem

Real Problem

Small Kernels

Small trustworthy foundation

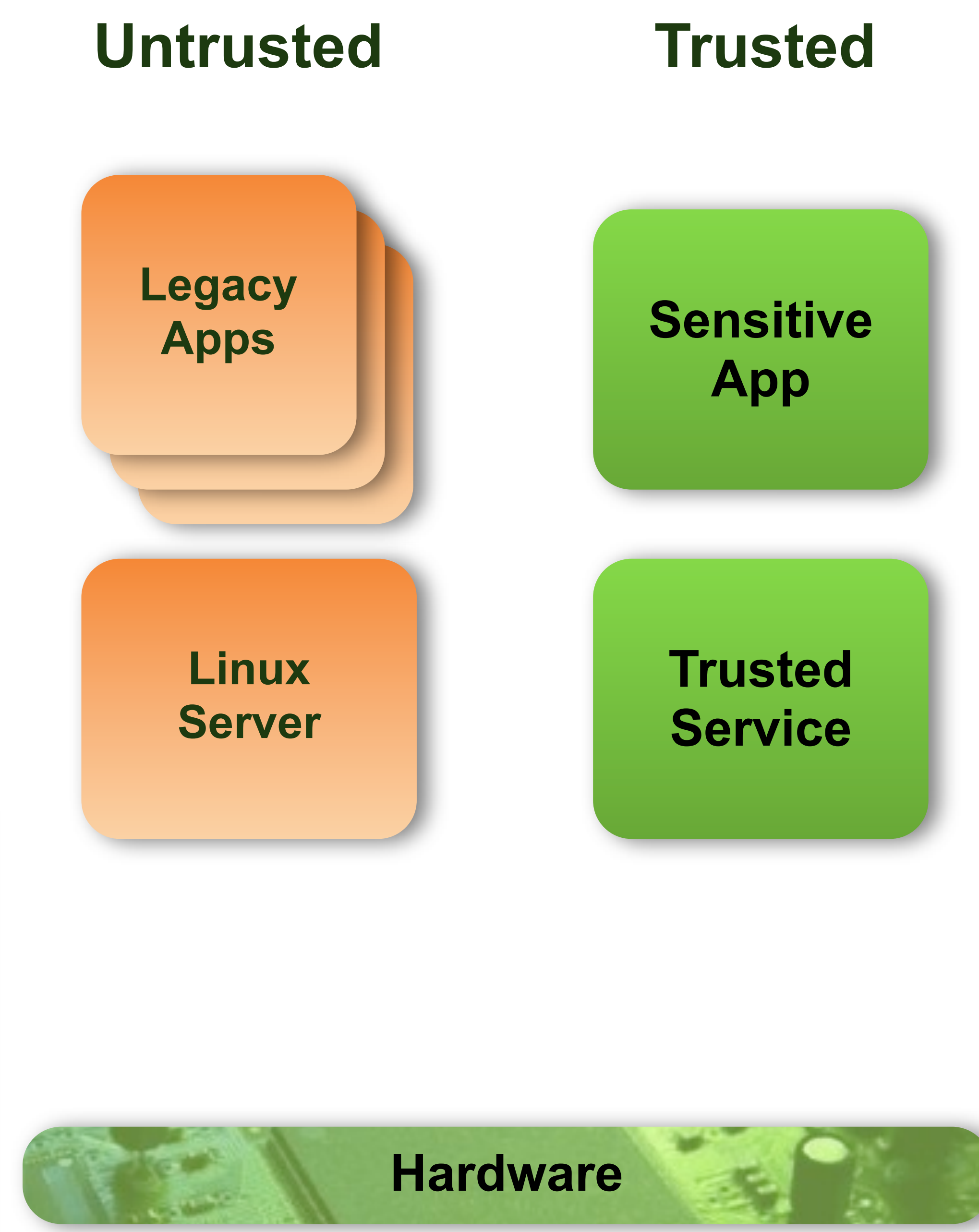
- hypervisor, microkernel, nano-kernel, virtual machine, separation kernel, exokernel ...
- High assurance components in presence of other components

seL4 API:

- IPC
- Threads
- VM
- IRQ
- Capabilities

Platforms:

- **ARMv6 (verified)**
- x86
- x86/IOMMU
- x86/SMP



Small Kernels

Small trustworthy foundation

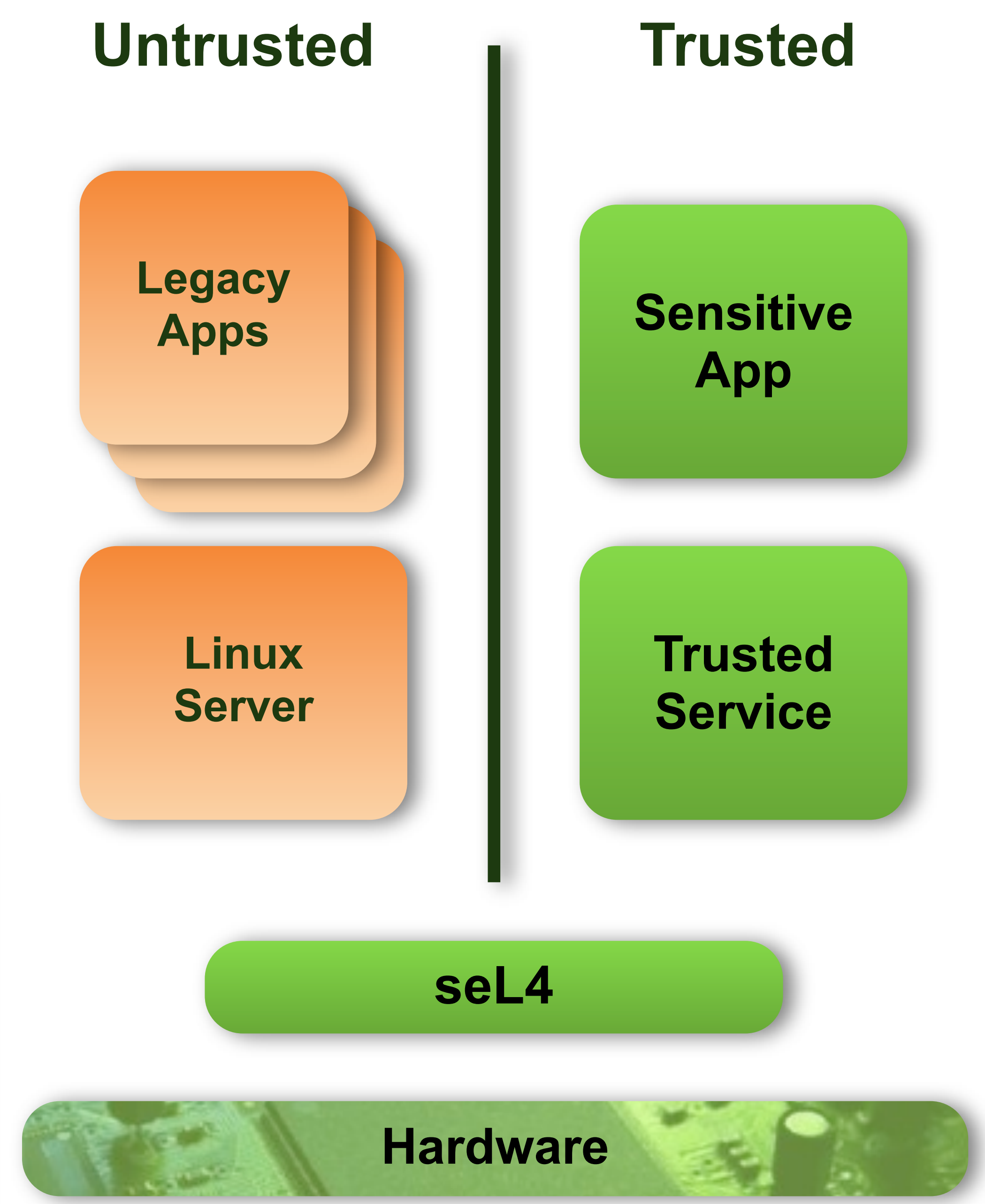
- hypervisor, microkernel, nano-kernel, virtual machine, separation kernel, exokernel ...
- High assurance components in presence of other components

seL4 API:

- IPC
- Threads
- VM
- IRQ
- Capabilities

Platforms:

- **ARMv6 (verified)**
- x86
- x86/IOMMU
- x86/SMP

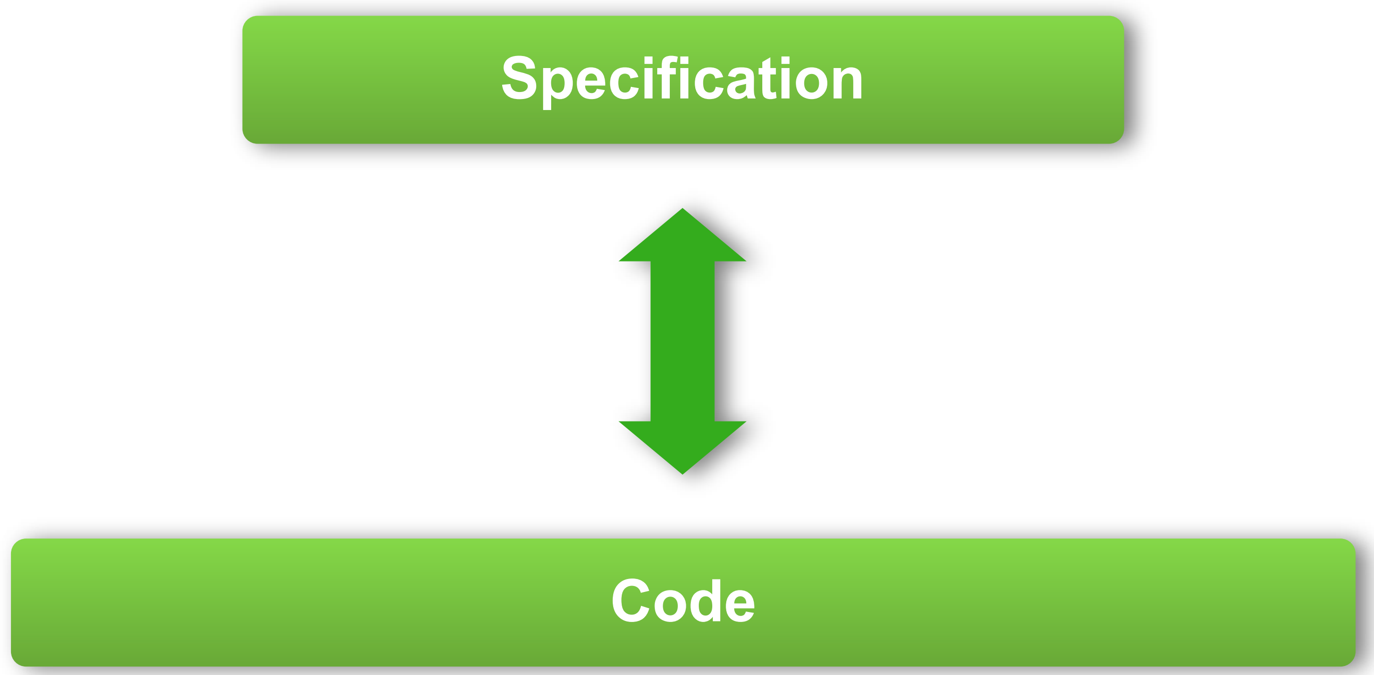


The Proof

The Proof

Functional Correctness

Proof

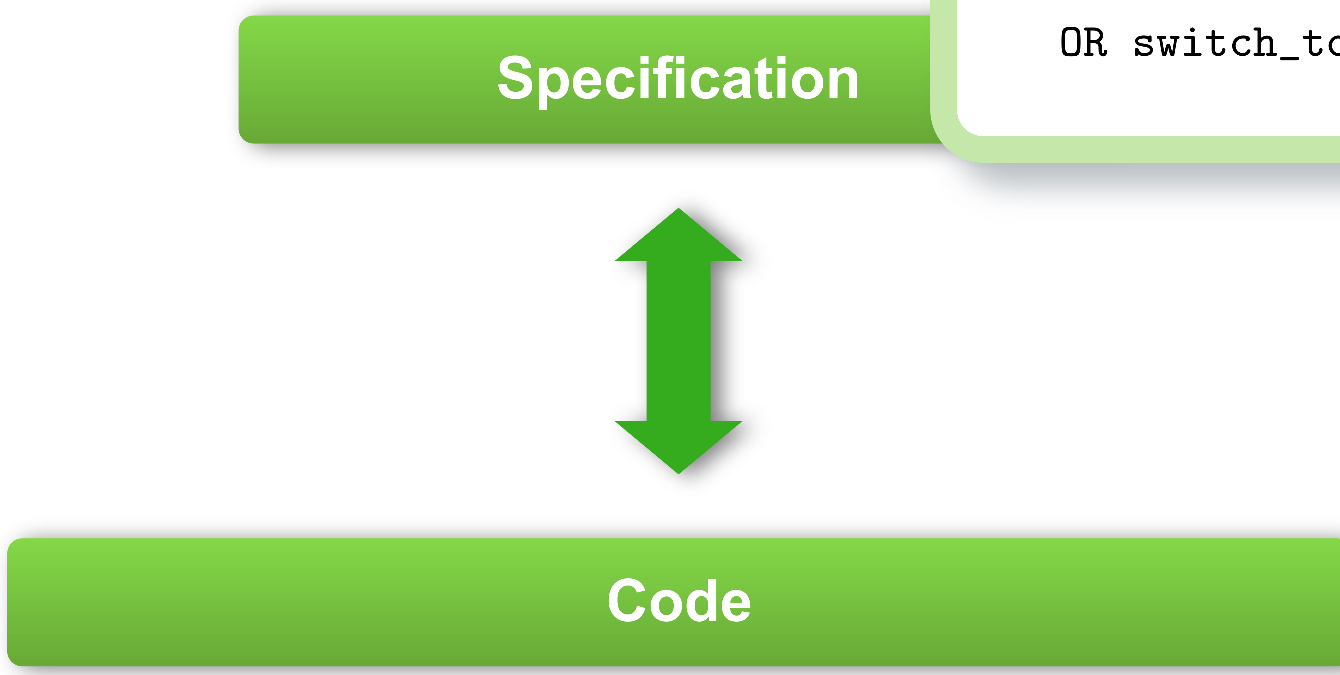


Functional Correctness

What

```
definition
  schedule :: unit s_monad where
  schedule ≡ do
    threads ← allActiveTCBs;
    thread ← select threads;
    switch_to_thread thread
  od
  OR switch_to_idle_thread
```

Proof



Functional Correctness

What

Specification

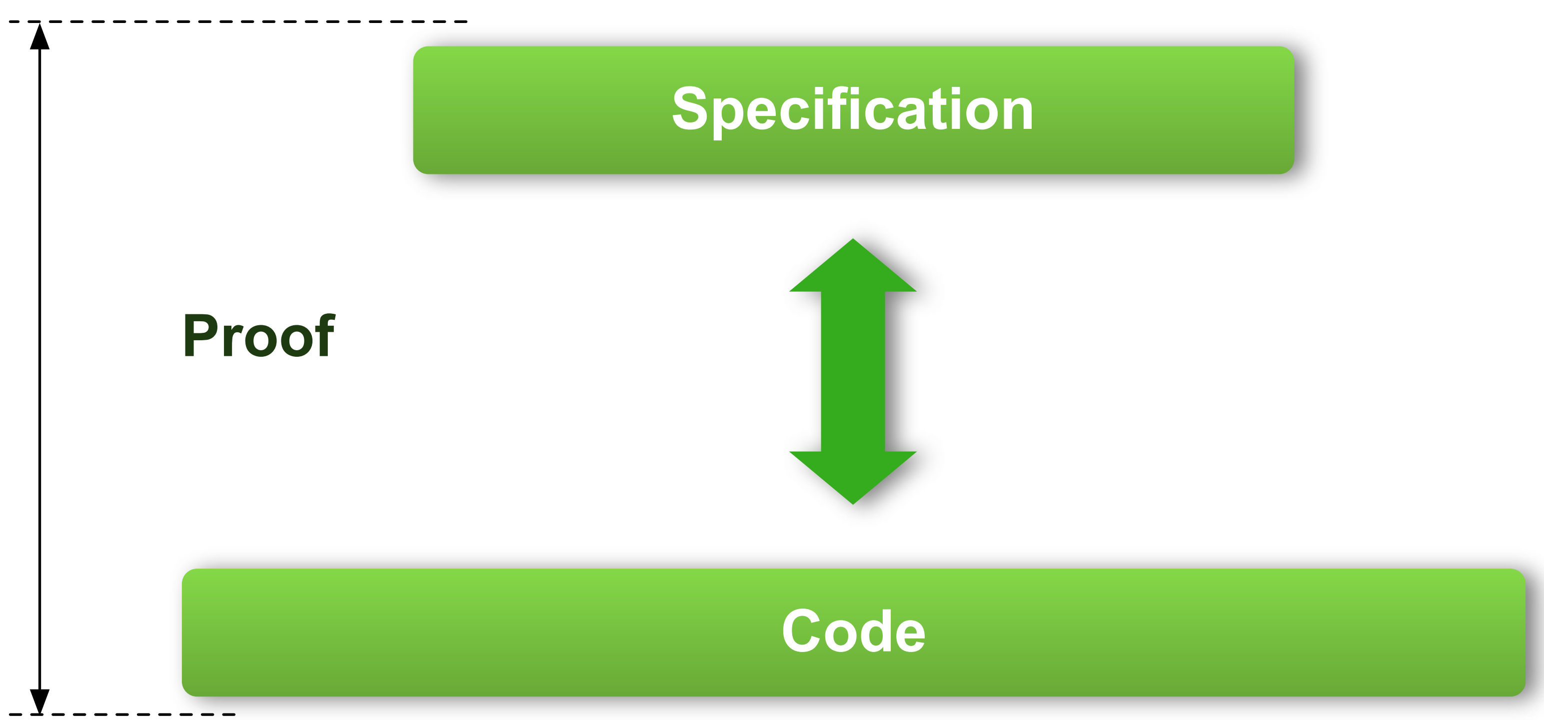
```
definition
  schedule :: unit s_monad where
  schedule ≡ do
    threads ← allActiveTCBs;
    thread ← select threads;
    switch_to_thread thread
  od
  OR switch_to_idle_thread
```

Proof

How

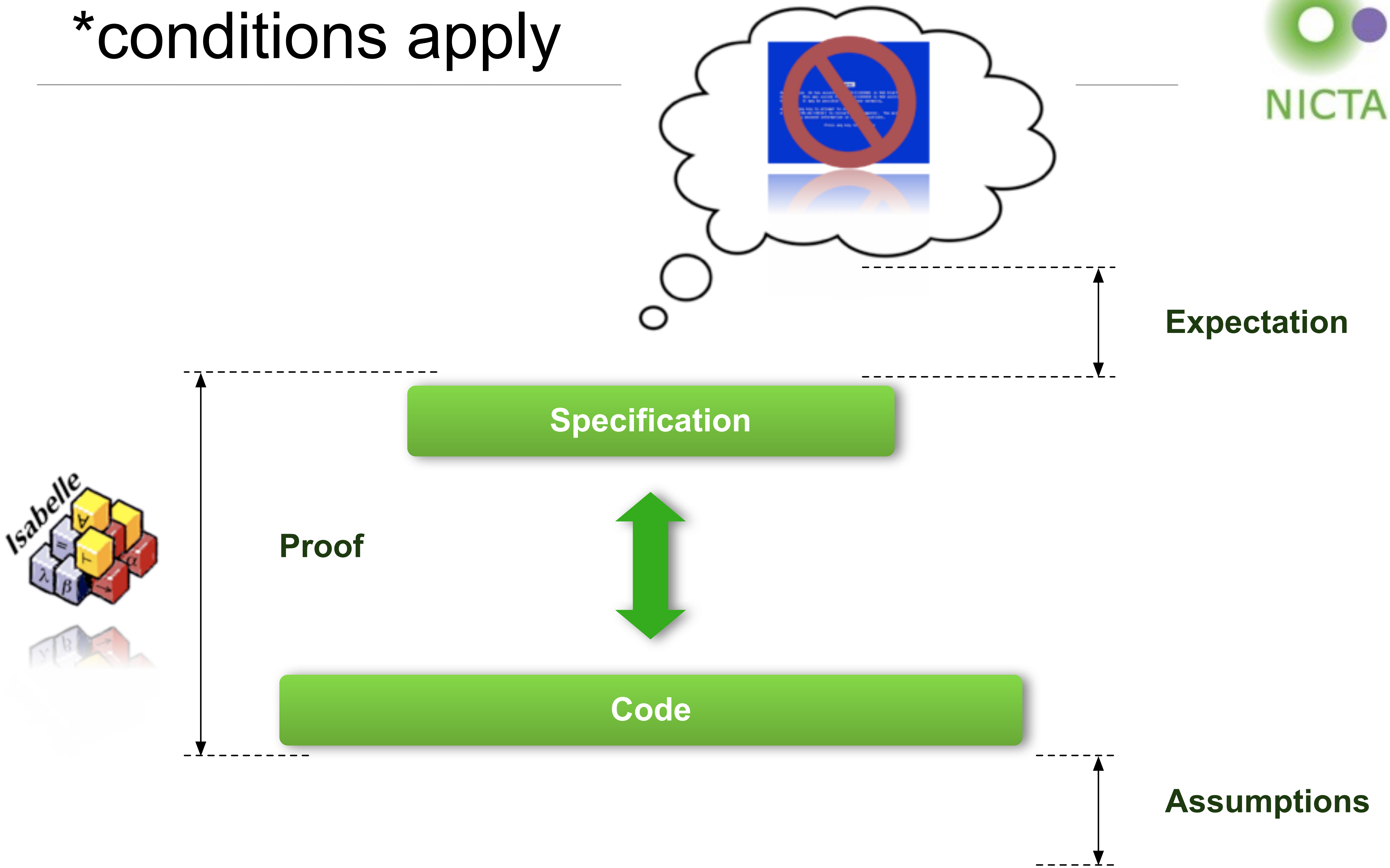
```
void
schedule(void) {
    switch ((word_t)ksSchedulerAction) {
        case (word_t)SchedulerAction_ResumeCurrentThread:
            break;

        case (word_t)SchedulerAction_ChooseNewThread:
            chooseThread();
            ksSchedulerAction = SchedulerAction_ResumeCurrentThread;
            break;

        default: /* SwitchToThread */
            switchToThread(ksSchedulerAction);
            ksSchedulerAction = SchedulerAction_ResumeCurrentThread;
            break;
    }
}

void
chooseThread(void) {
    prio_t prio;
    tcb_t *thread, *next;
```

*conditions apply



*conditions apply

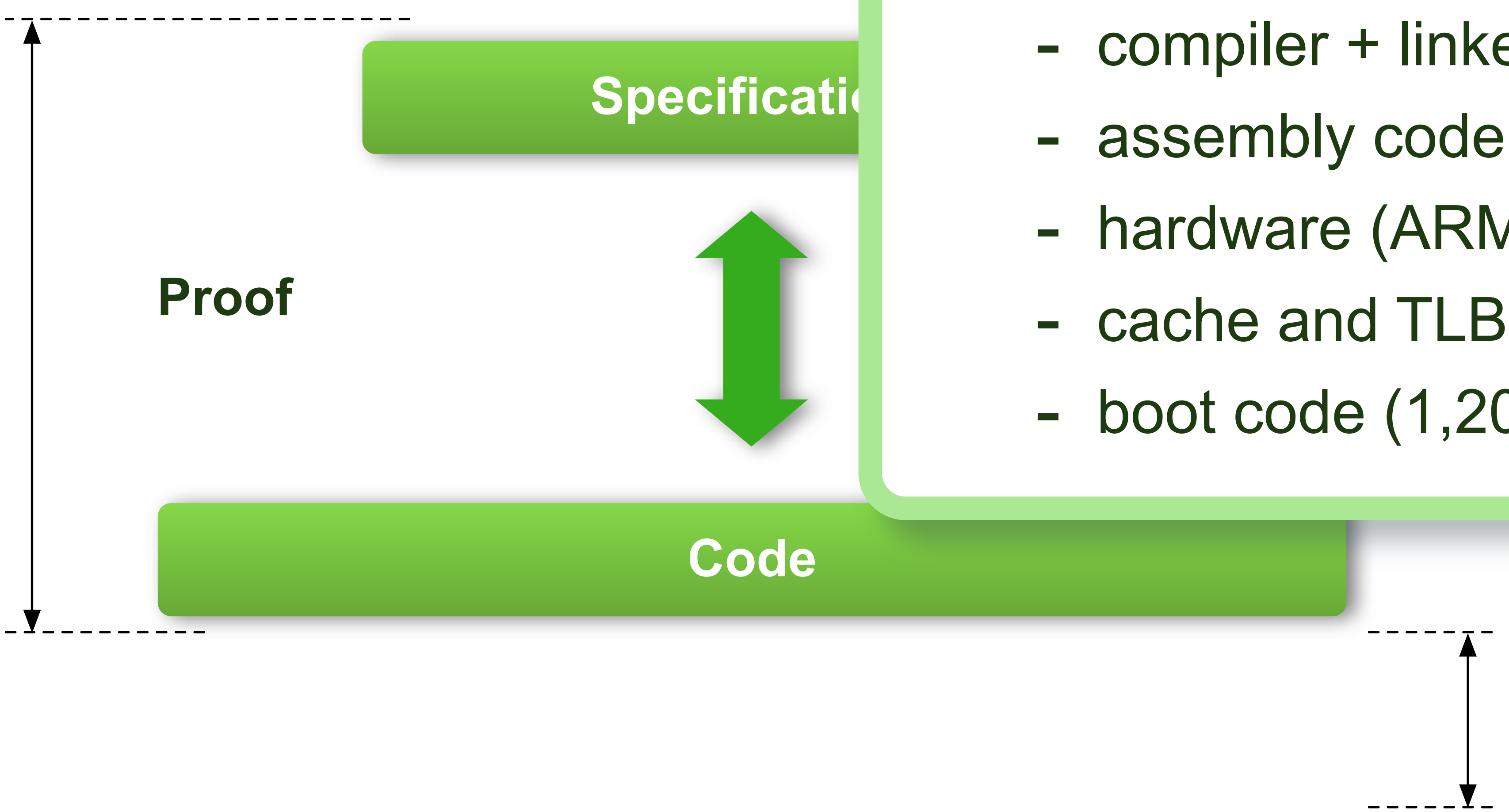
NICTA



*conditions apply

Assume correct:

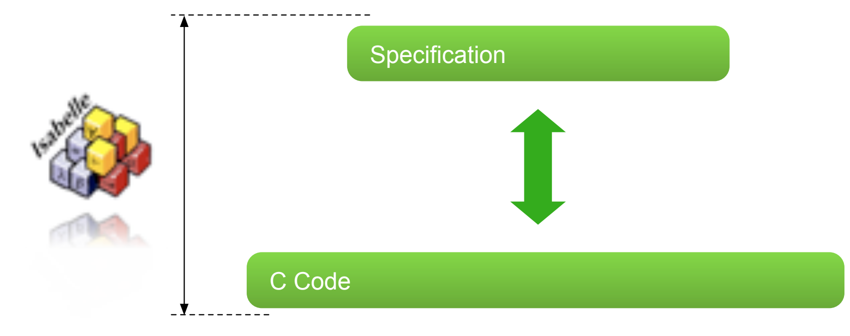
- compiler + linker (wrt. C op-sem)
- assembly code (600 loc)
- hardware (ARMv6)
- cache and TLB management
- boot code (1,200 loc)



Implications

Execution always defined:

- no null pointer de-reference
- no buffer overflows
- no code injection
- no memory leaks/out of kernel memory
- no div by zero, no undefined shift
- no undefined execution
- no infinite loops/recursion



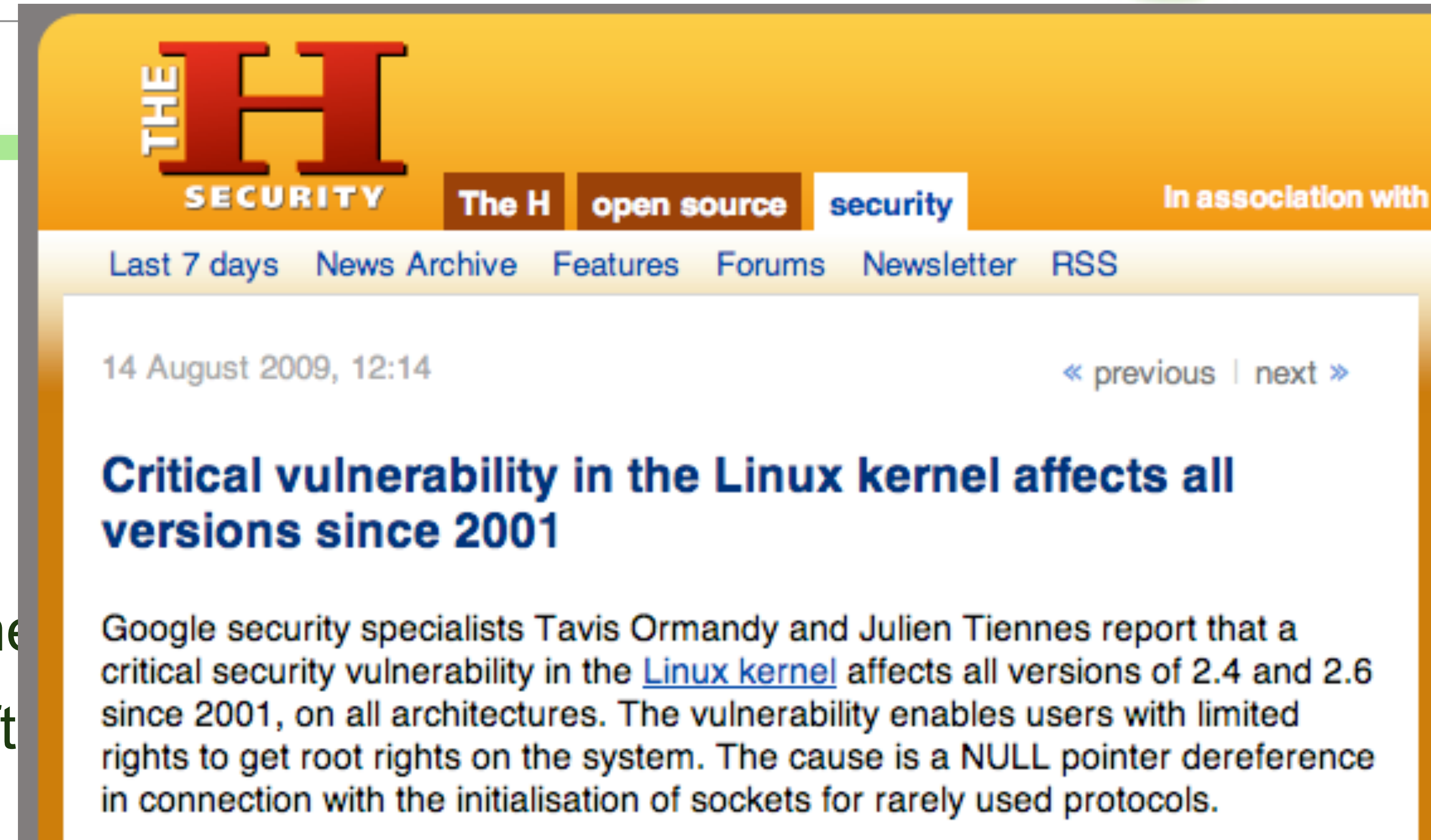
Not implied:

- "secure" (define secure)
- zero bugs from expectation to physical world
- covert channel analysis

Implications

Execution always defined:

- no null pointer de-reference
- no buffer overflows
- no code injection
- no memory leaks/out of kernel me
- no div by zero, no undefined shift
- no undefined execution
- no infinite loops/recursion


THE H SECURITY

The H | open source | security

In association with

Last 7 days | News Archive | Features | Forums | Newsletter | RSS

14 August 2009, 12:14

« previous | next »

Critical vulnerability in the Linux kernel affects all versions since 2001

Google security specialists Tavis Ormandy and Julien Tiennes report that a critical security vulnerability in the Linux kernel affects all versions of 2.4 and 2.6 since 2001, on all architectures. The vulnerability enables users with limited rights to get root rights on the system. The cause is a NULL pointer dereference in connection with the initialisation of sockets for rarely used protocols.

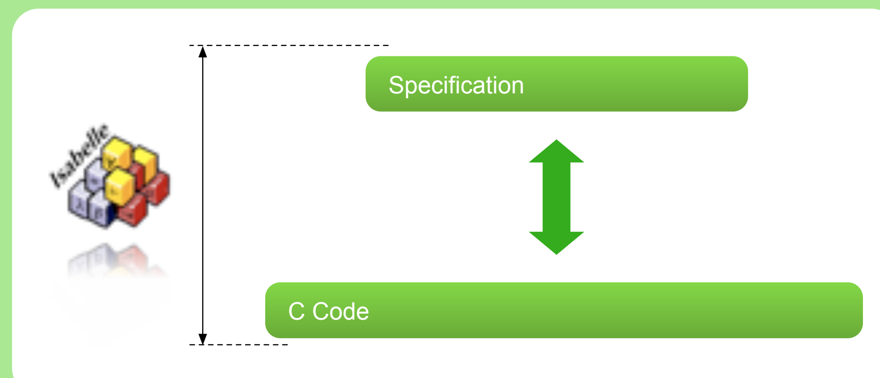
Not implied:

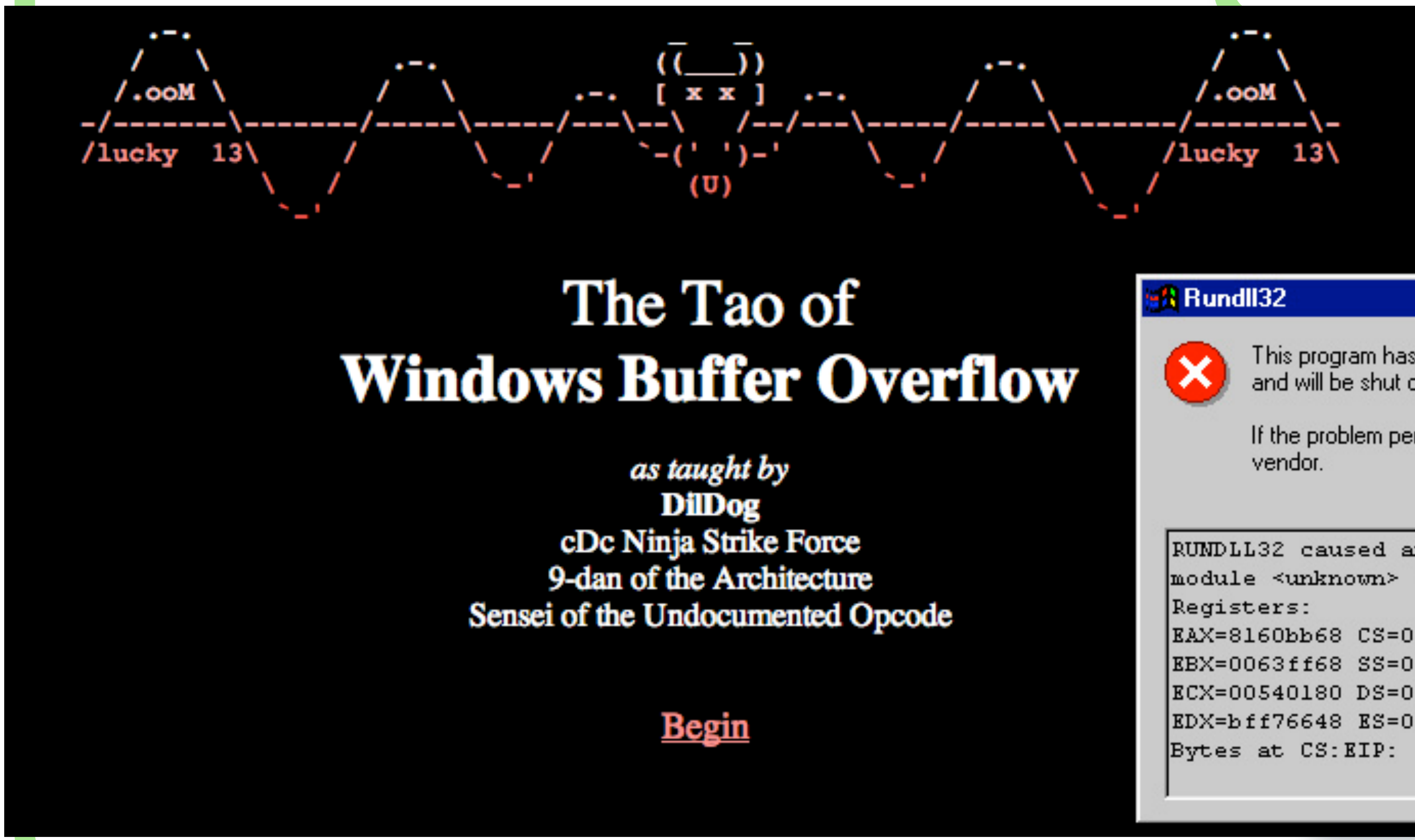
- “secure” (define secure)
- zero bugs from expectation to physical world
- covert channel analysis

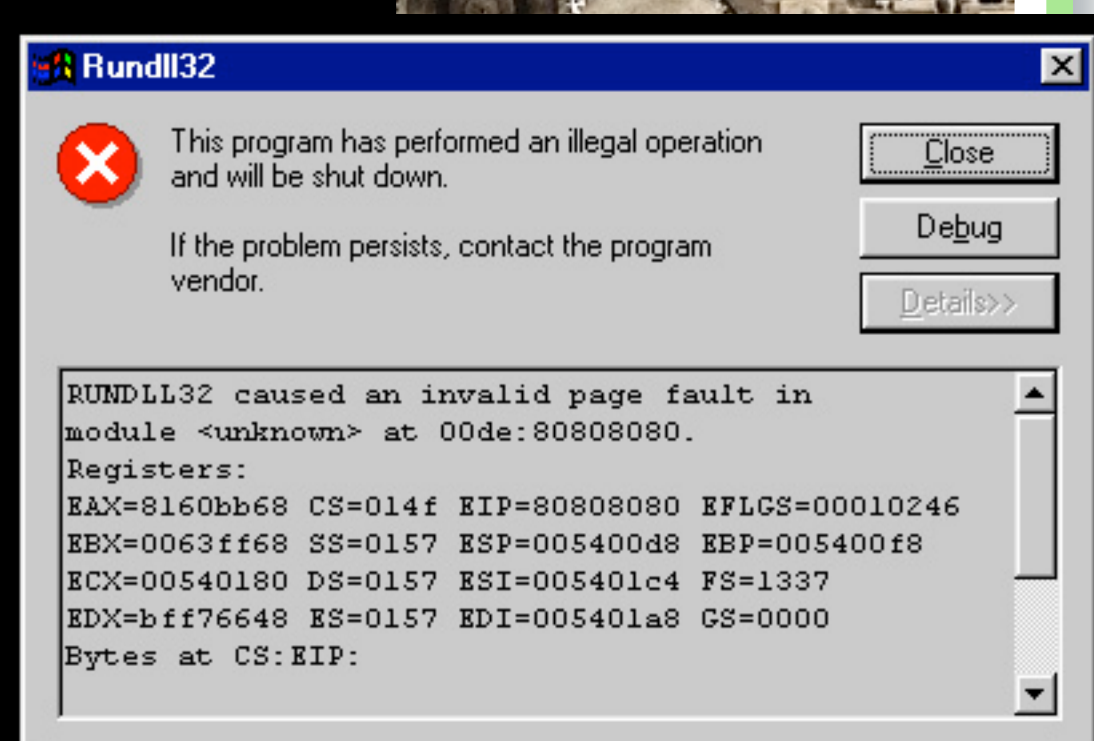
Implications

Execution always defined:

- no null pointer de-reference
- no buffer overflows



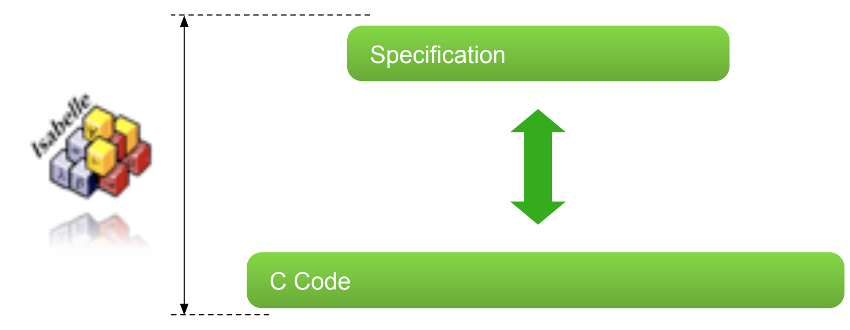




Implications

Execution always defined:

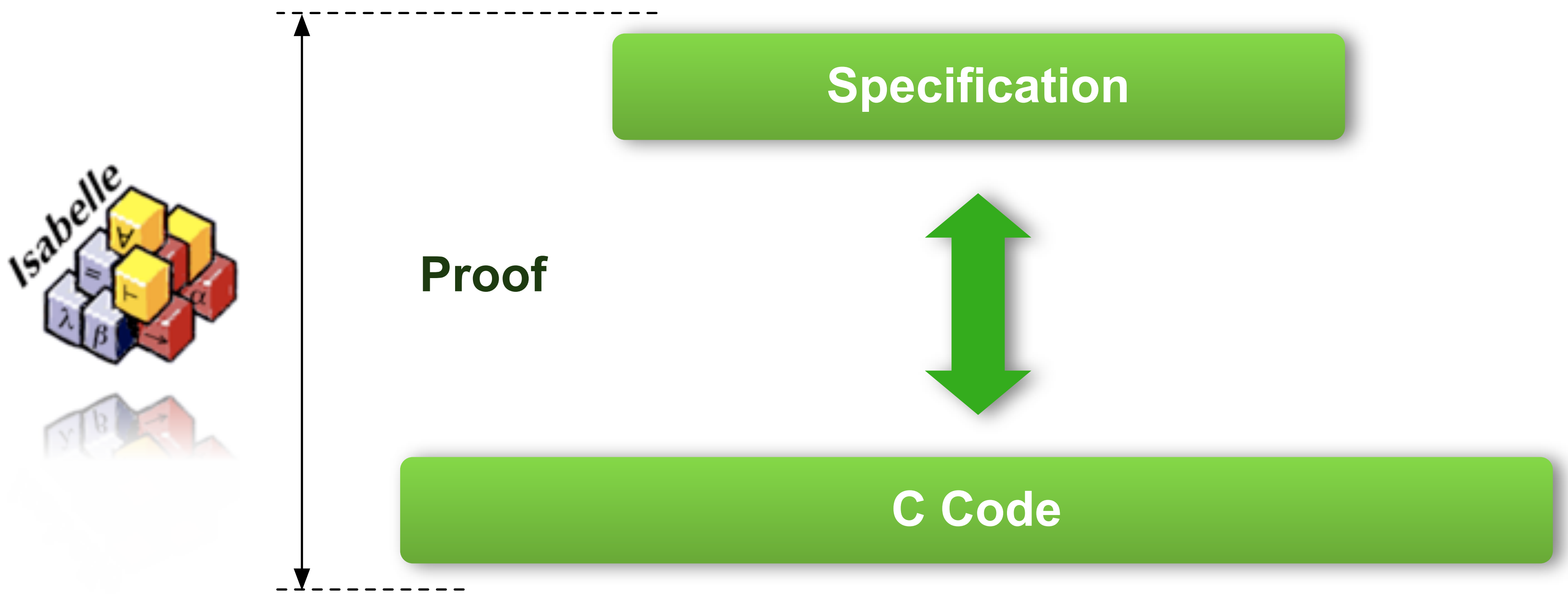
- no null pointer de-reference
- no buffer overflows
- no code injection
- no memory leaks/out of kernel memory
- no div by zero, no undefined shift
- no undefined execution
- no infinite loops/recursion



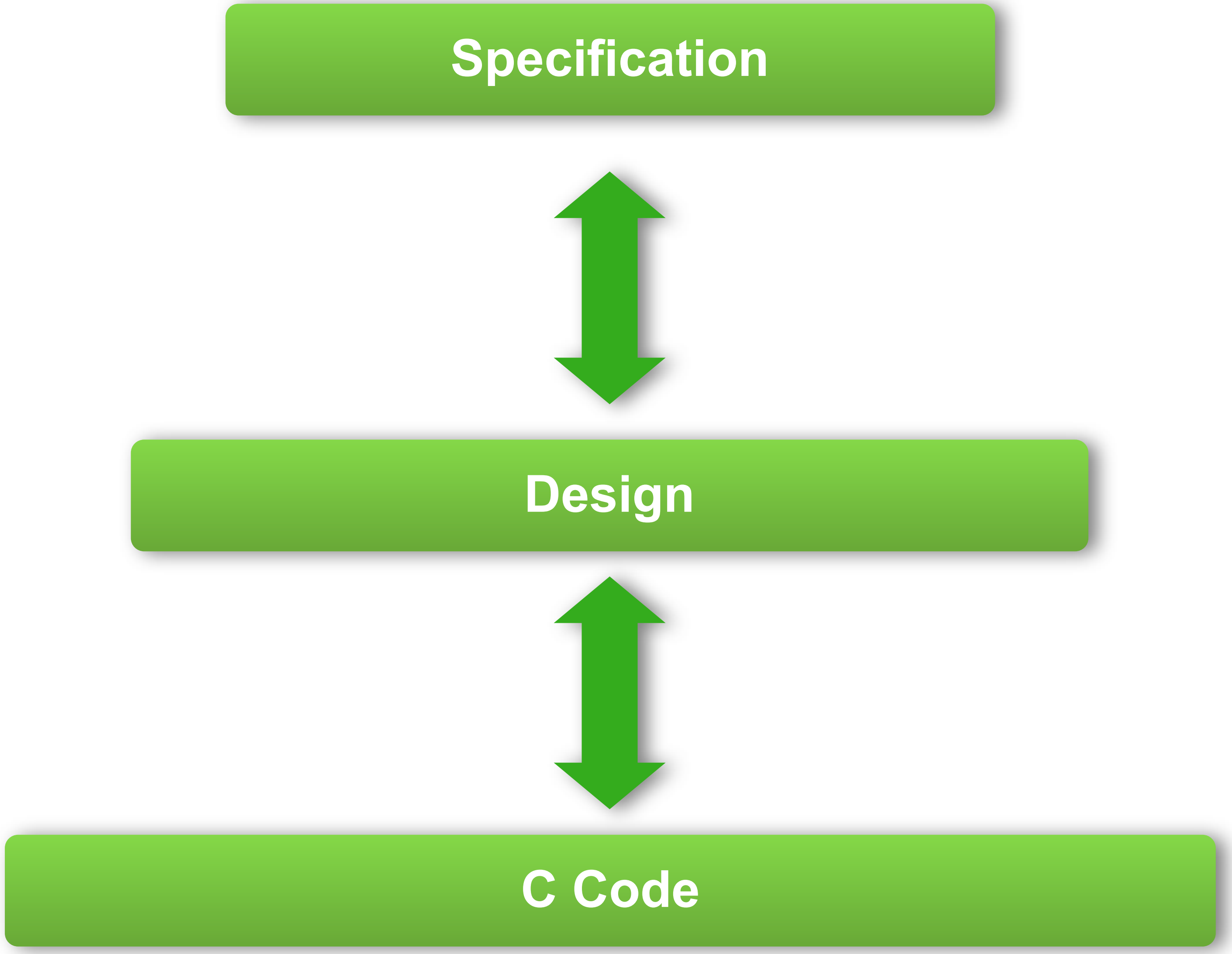
Not implied:

- “secure” (define secure)
- zero bugs from expectation to physical world
- covert channel analysis

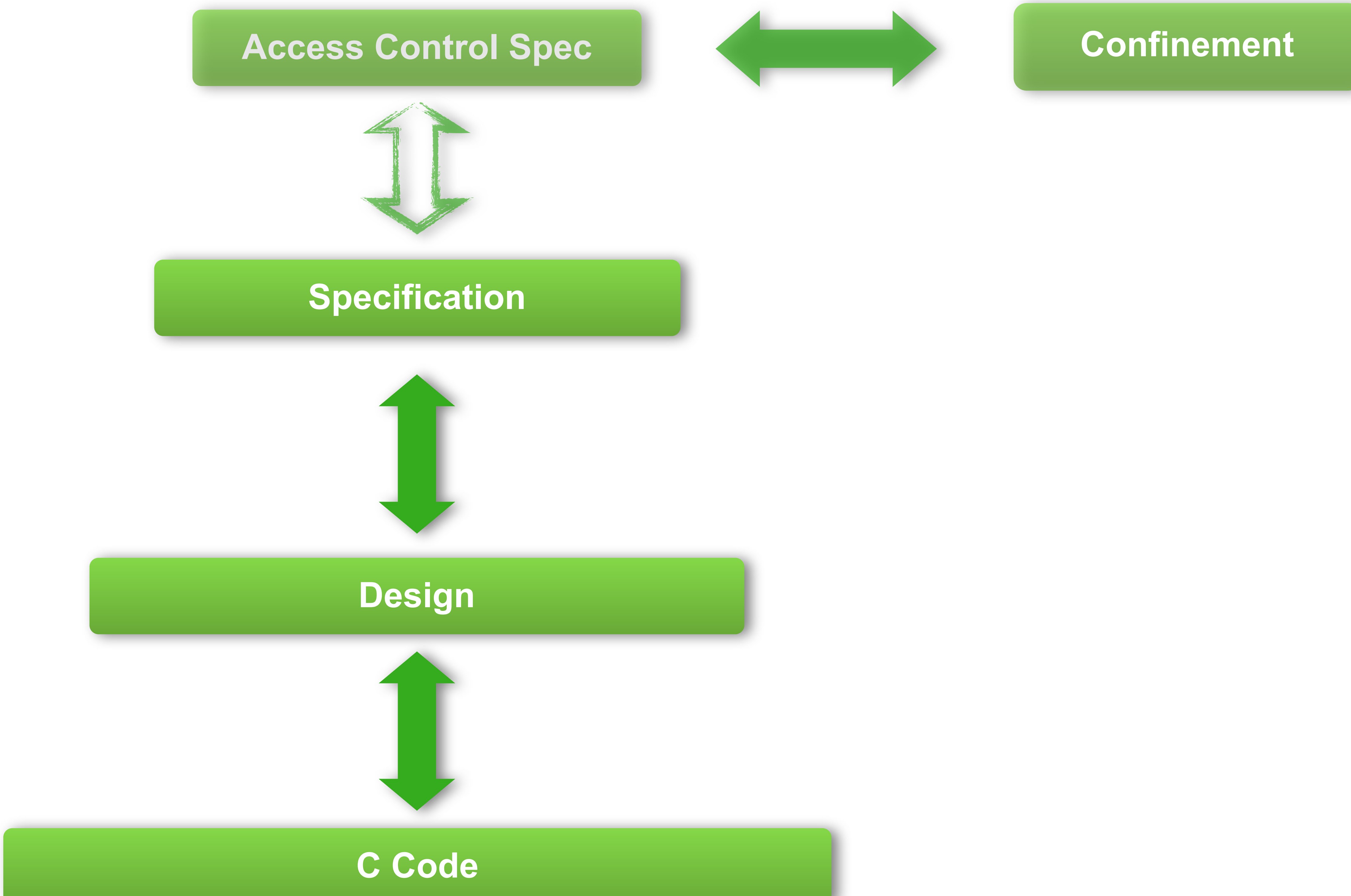
Proof Architecture



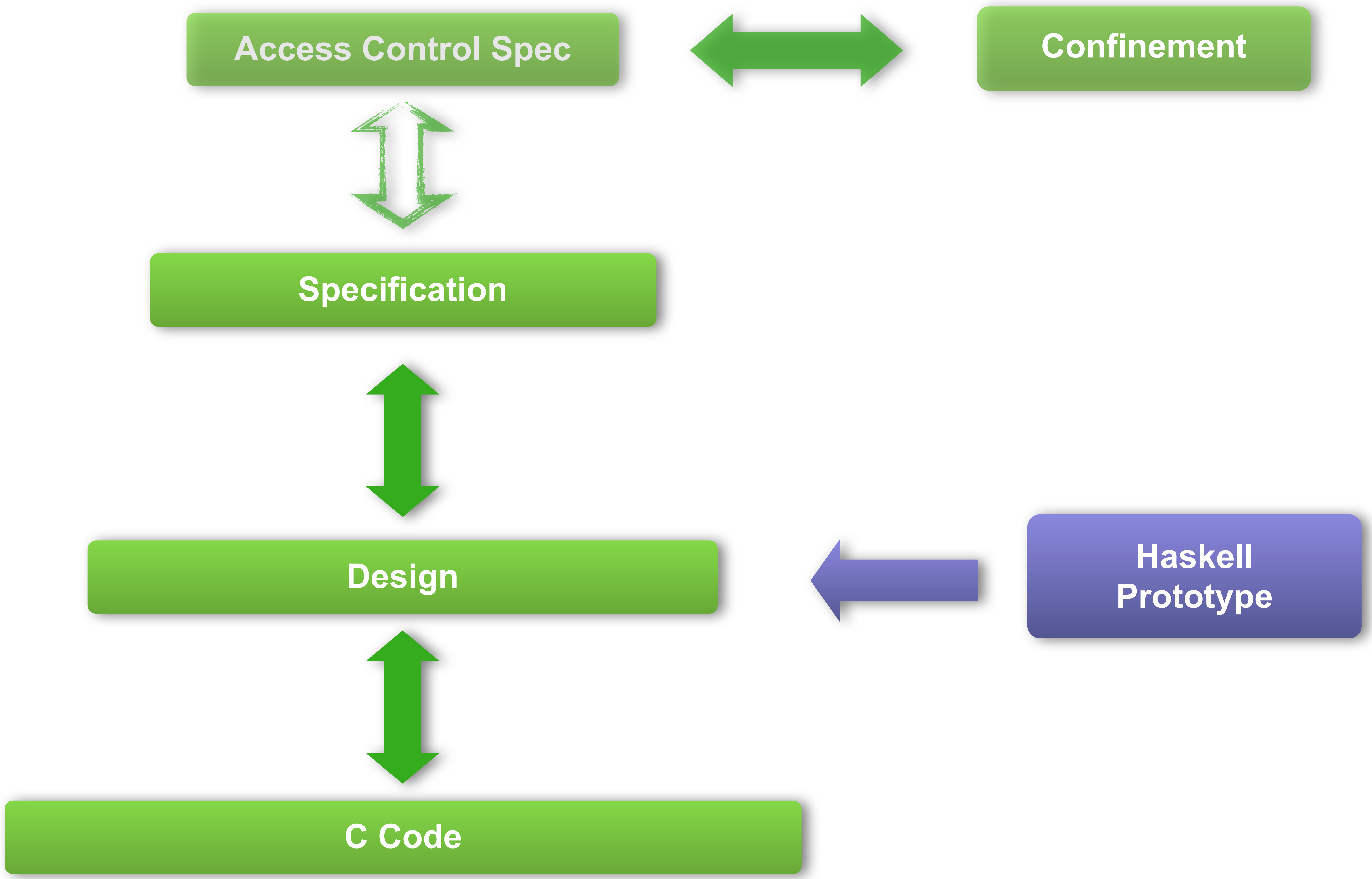
Proof Architecture



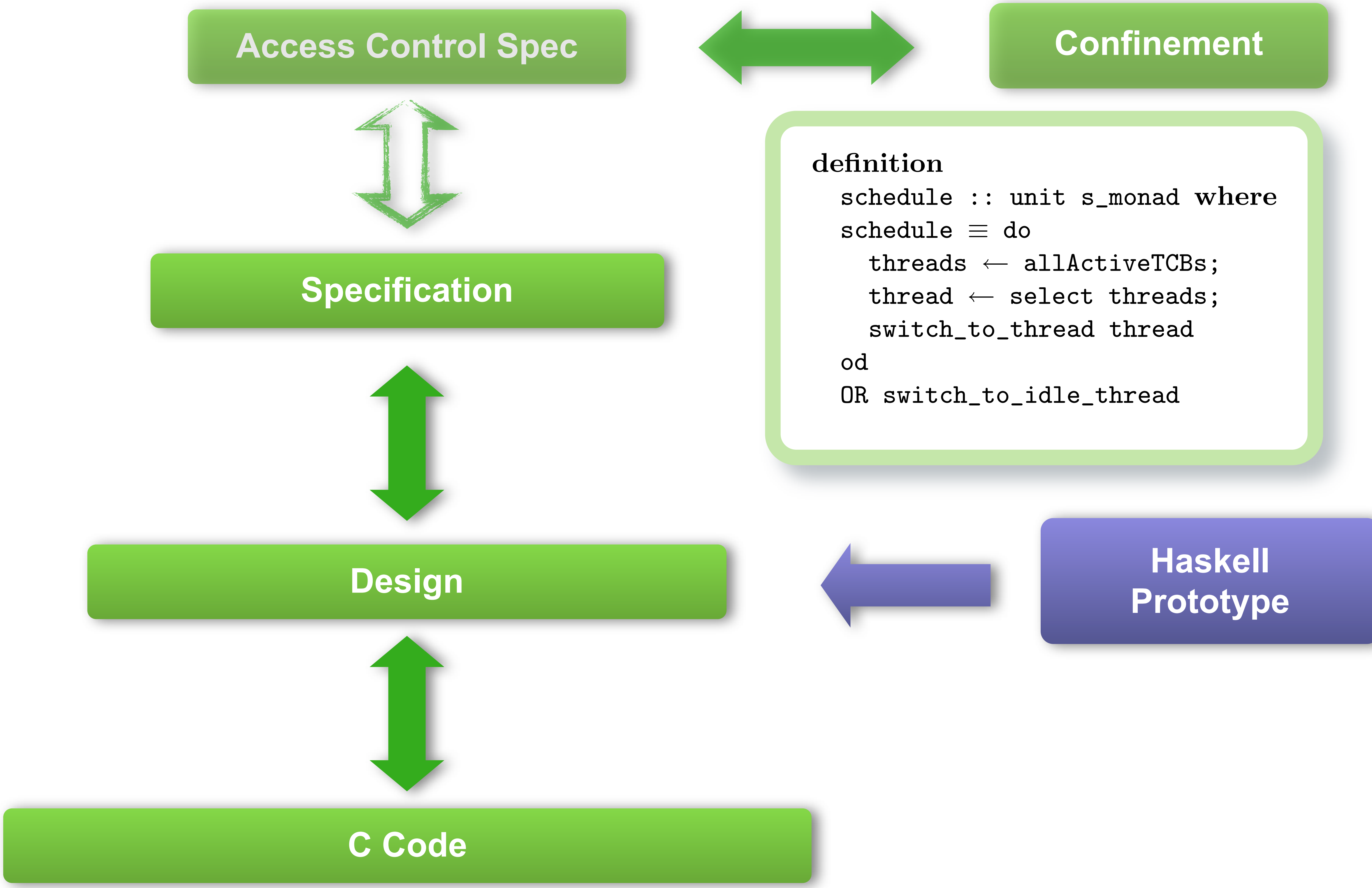
Proof Architecture



Proof Architecture

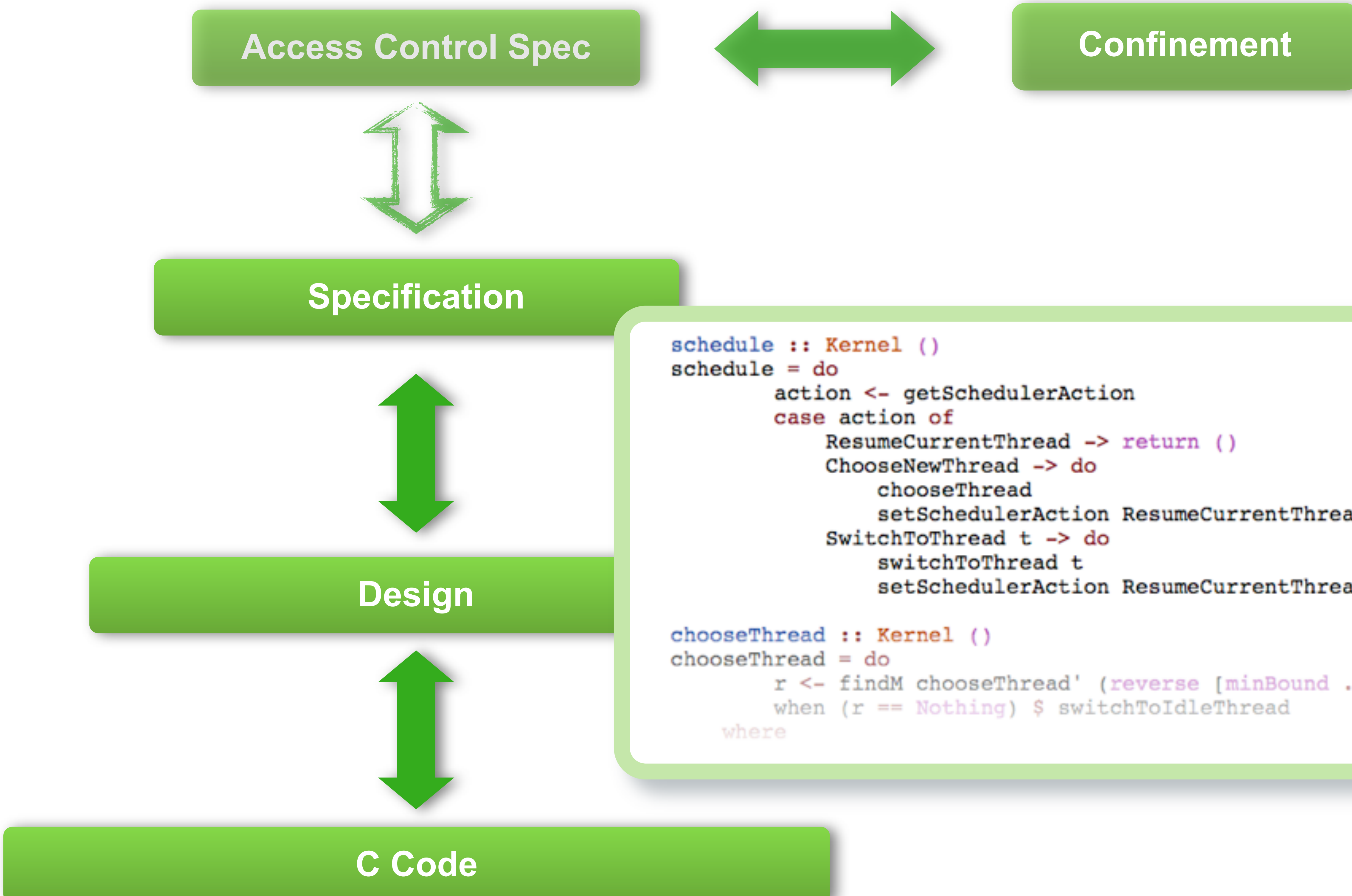


Proof Architecture



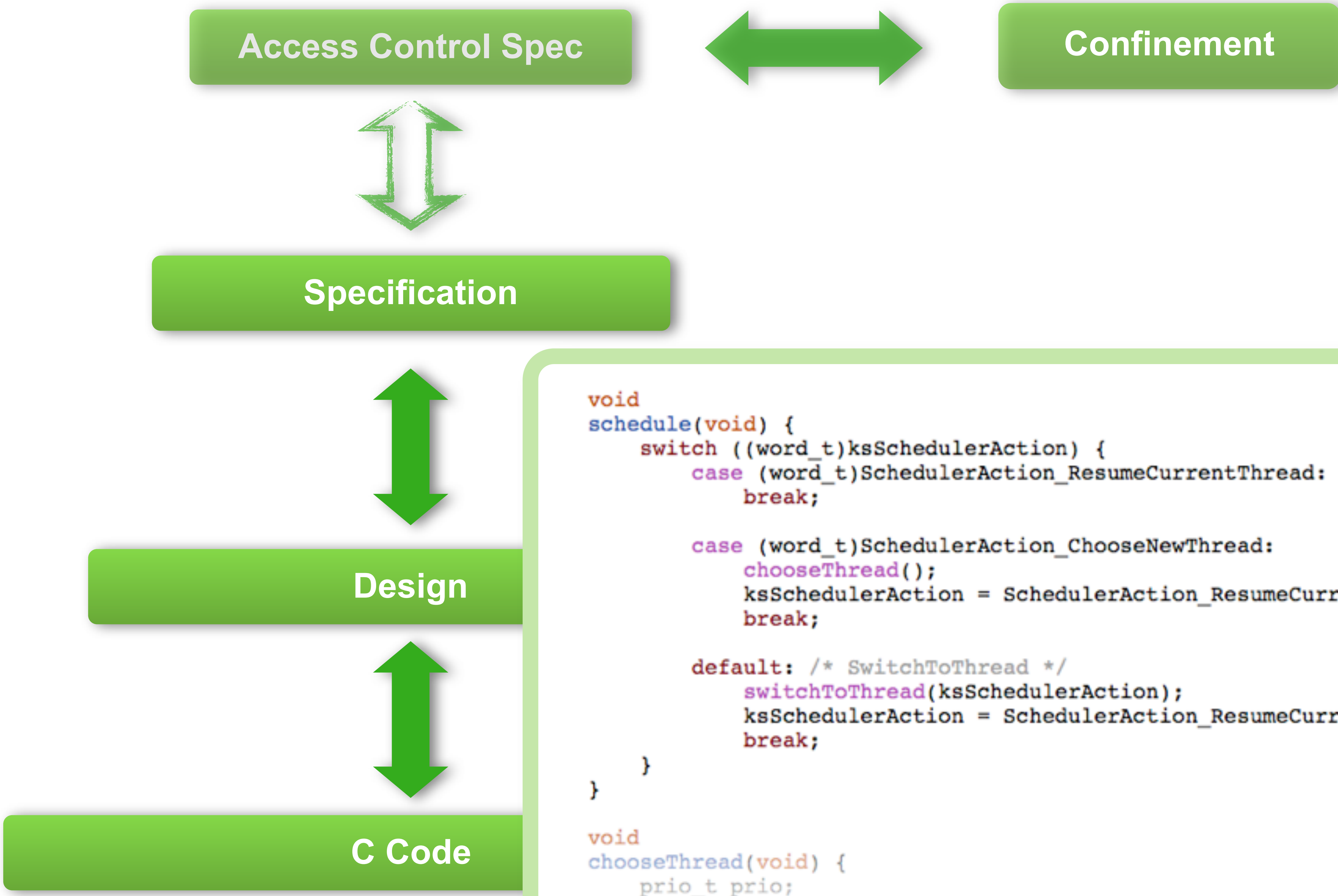

```
definition
  schedule :: unit s_monad where
  schedule ≡ do
    threads ← allActiveTCBs;
    thread ← select threads;
    switch_to_thread thread
  od
  OR switch_to_idle_thread
```

Proof Architecture




```
schedule :: Kernel ()
schedule = do
        action <- getSchedulerAction
        case action of
            ResumeCurrentThread -> return ()
            ChooseNewThread -> do
                chooseThread
                setSchedulerAction ResumeCurrentThread
            SwitchToThread t -> do
                switchToThread t
                setSchedulerAction ResumeCurrentThread

chooseThread :: Kernel ()
chooseThread = do
        r <- findM chooseThread' (reverse [minBound .. maxBound])
        when (r == Nothing) $ switchToIdleThread
    where
```

Proof Architecture




```
void
schedule(void) {
    switch ((word_t)ksSchedulerAction) {
        case (word_t)SchedulerAction_ResumeCurrentThread:
            break;

        case (word_t)SchedulerAction_ChooseNewThread:
            chooseThread();
            ksSchedulerAction = SchedulerAction_ResumeCurrentThread;
            break;

        default: /* SwitchToThread */
            switchToThread(ksSchedulerAction);
            ksSchedulerAction = SchedulerAction_ResumeCurrentThread;
            break;
    }
}

void
chooseThread(void) {
    prio_t prio;
    tcb_t *thread, *next;
```

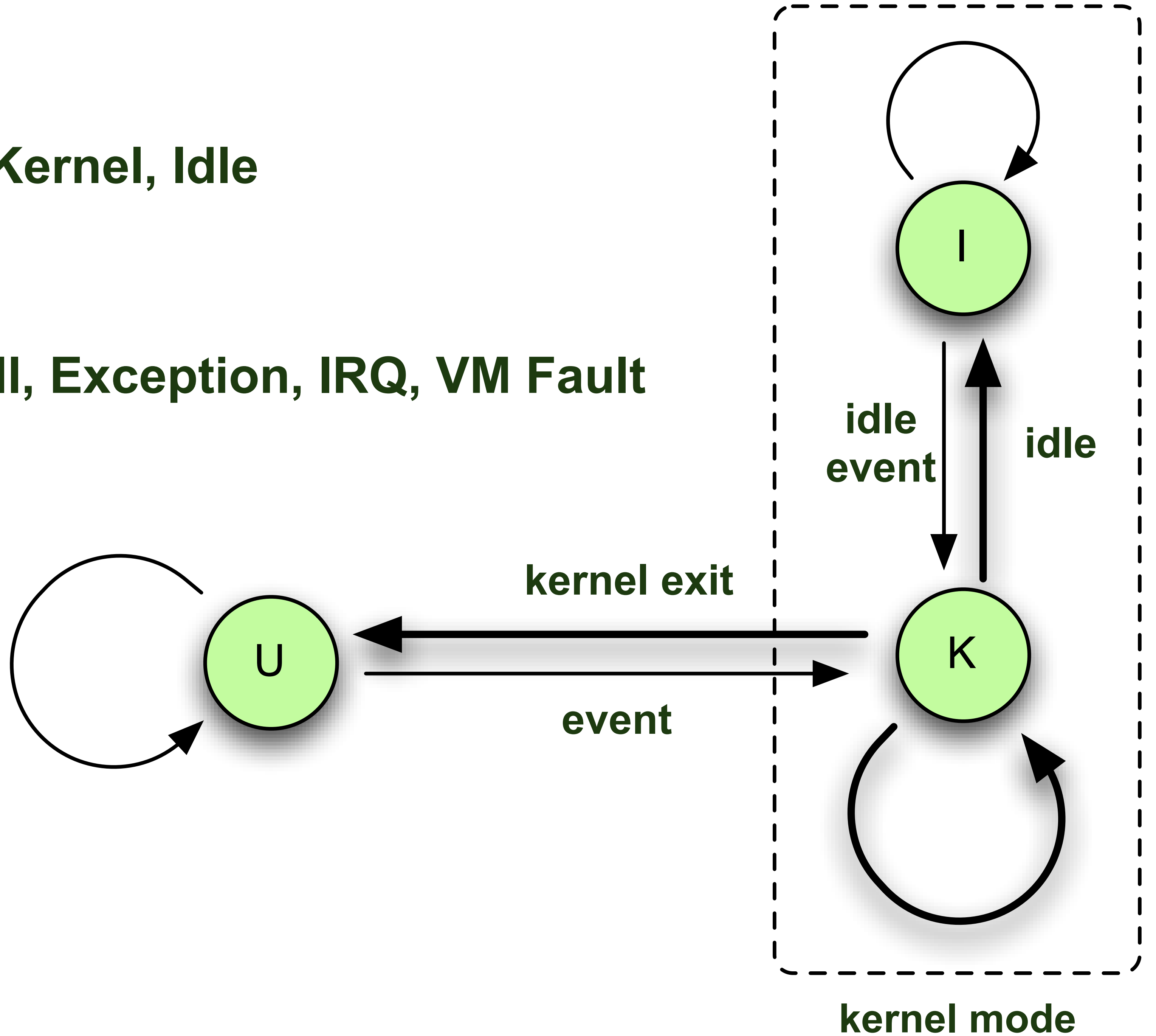
System Model

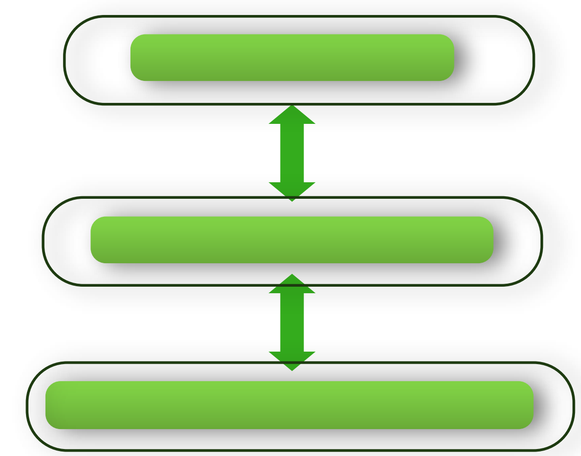
States:

User, Kernel, Idle

Events:

Syscall, Exception, IRQ, VM Fault





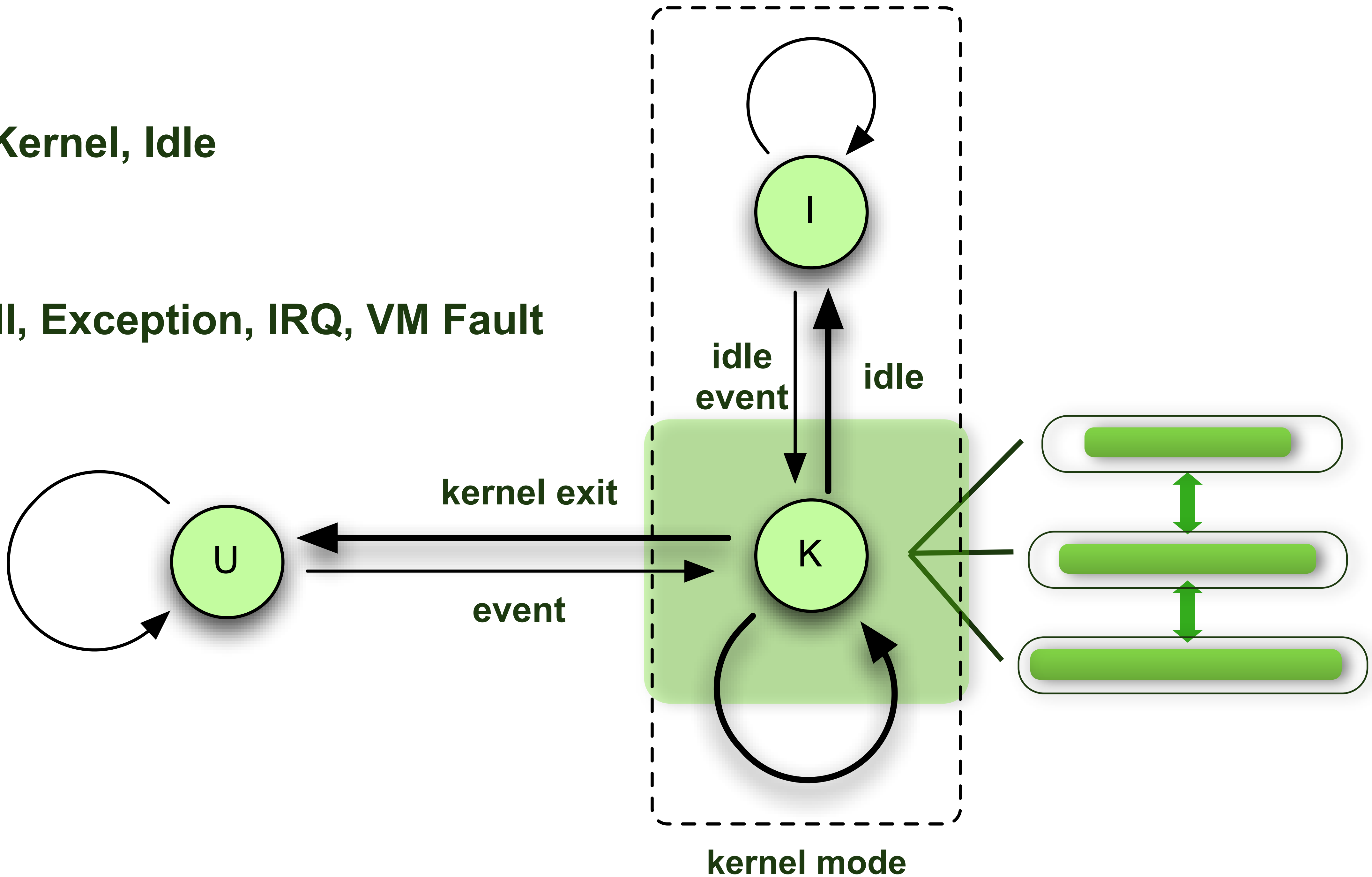
System Model

States:

User, Kernel, Idle

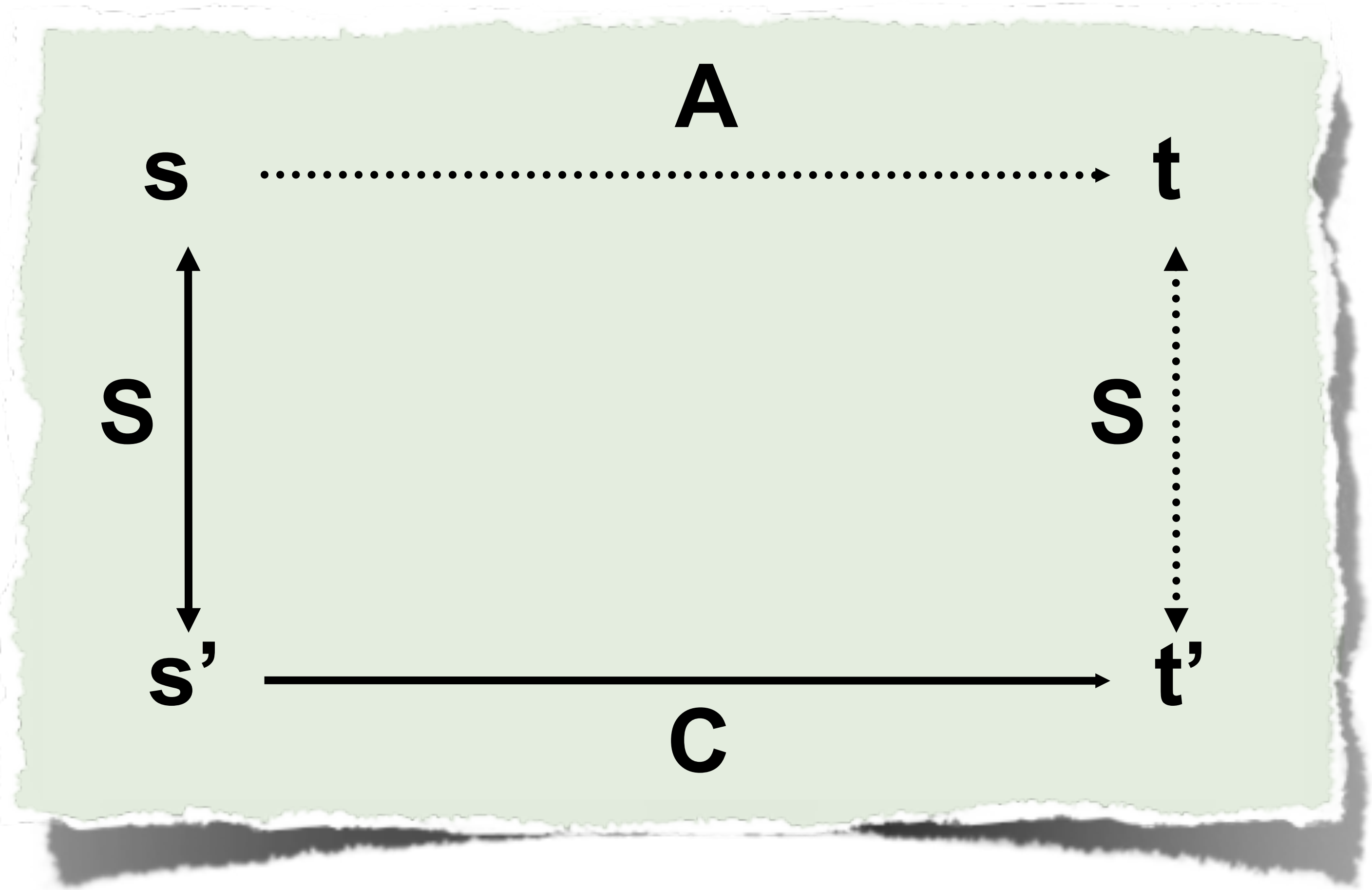
Events:

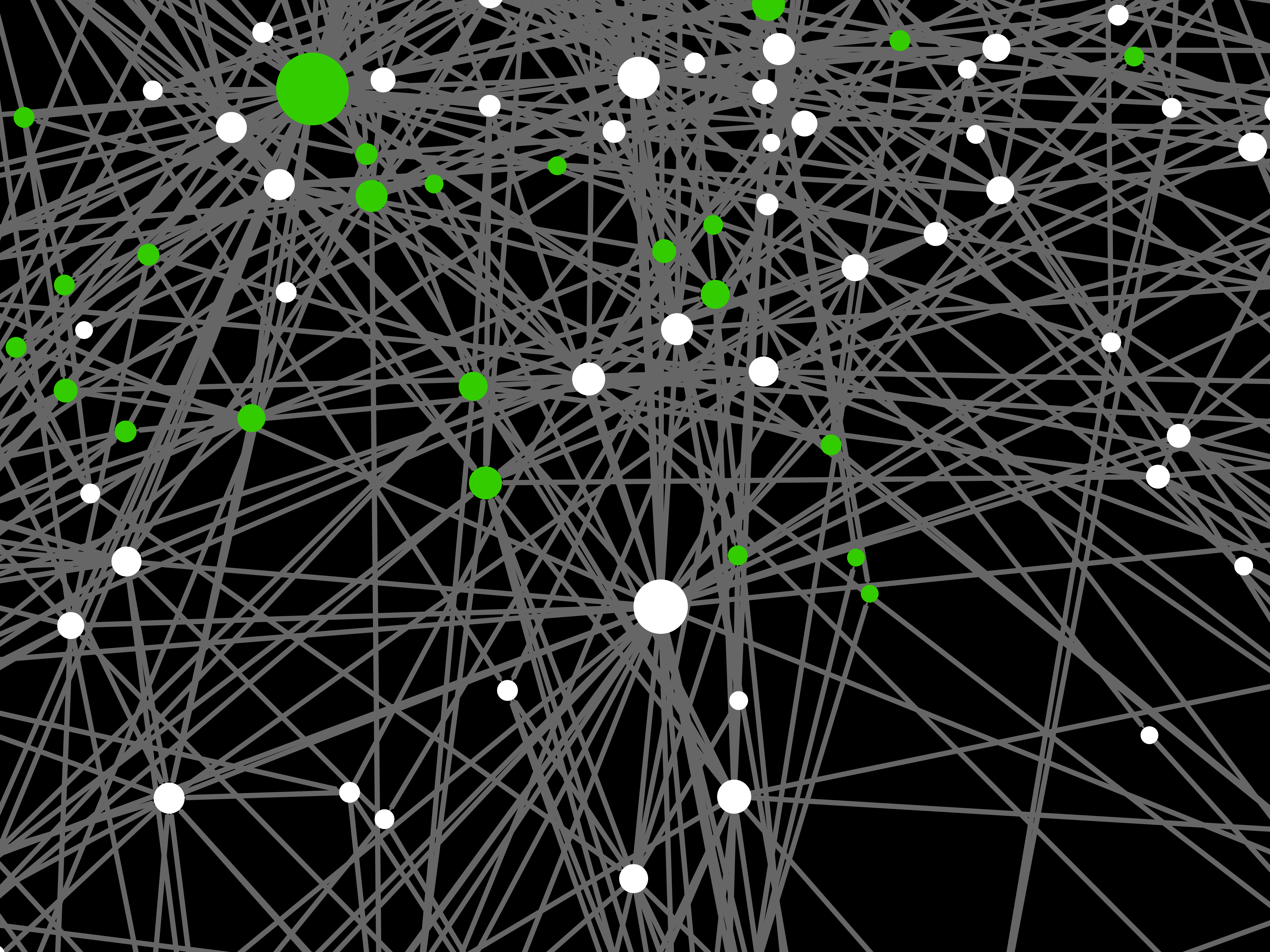
Syscall, Exception, IRQ, VM Fault



Old Story: Refinement

- C refines A if all behaviours of C are contained in A
- Sufficient: forward simulation

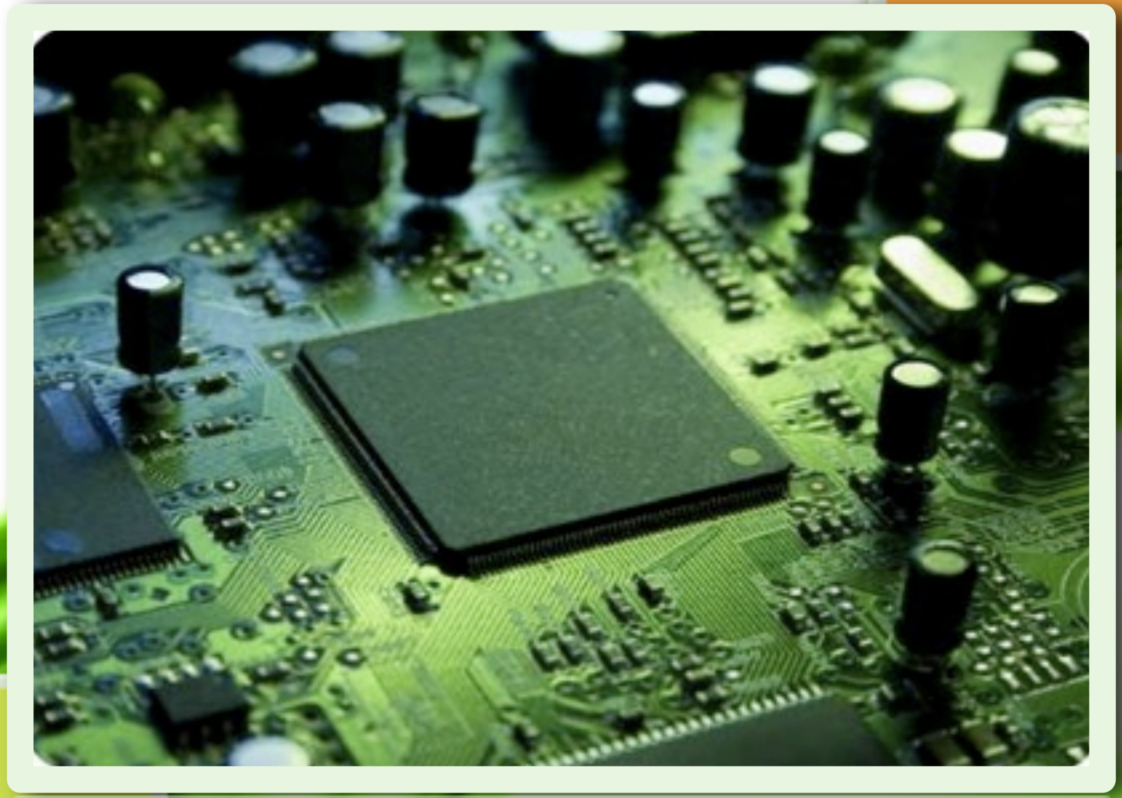




seL4

Designing and Formalising a Microkernel

Designing and Formalising a Microkernel



Two Teams

Formal Methods Practitioners

Kernel Developers

Two Teams

Formal Methods Practitioners

Kernel Developers

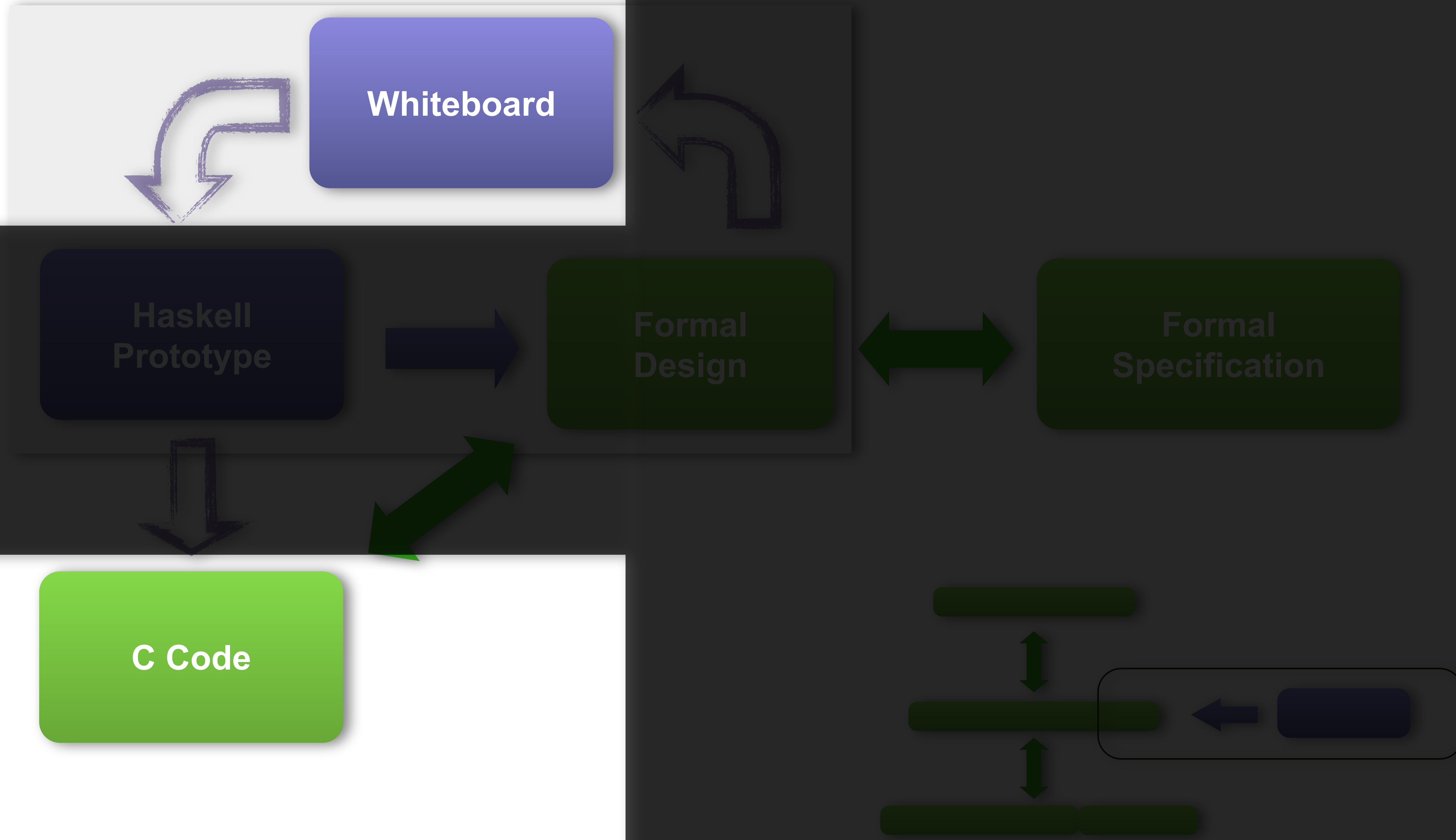
The Power of Abstraction

(Liskov 09)

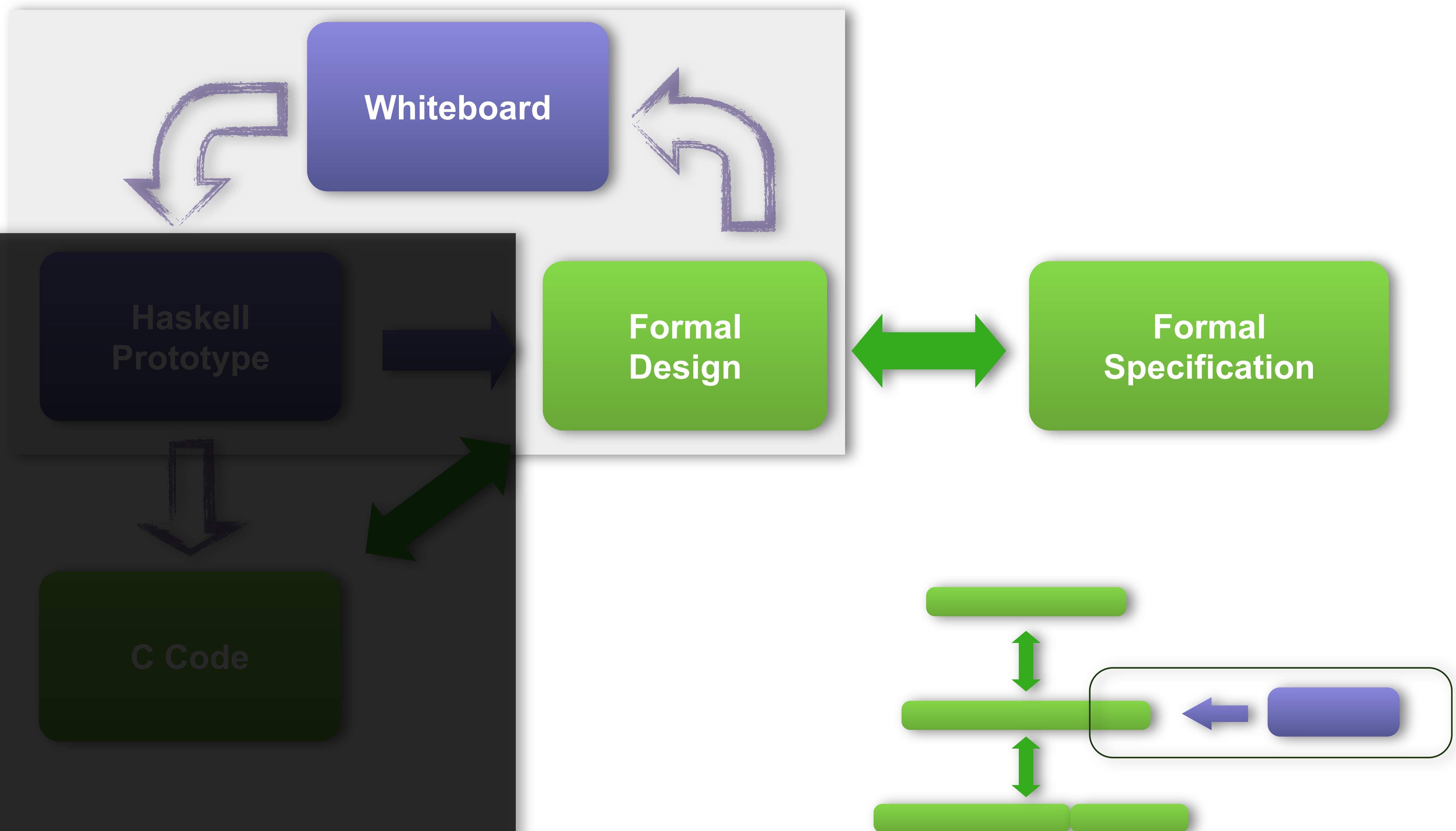
Exterminate All OS Abstractions!

(Engler 95)

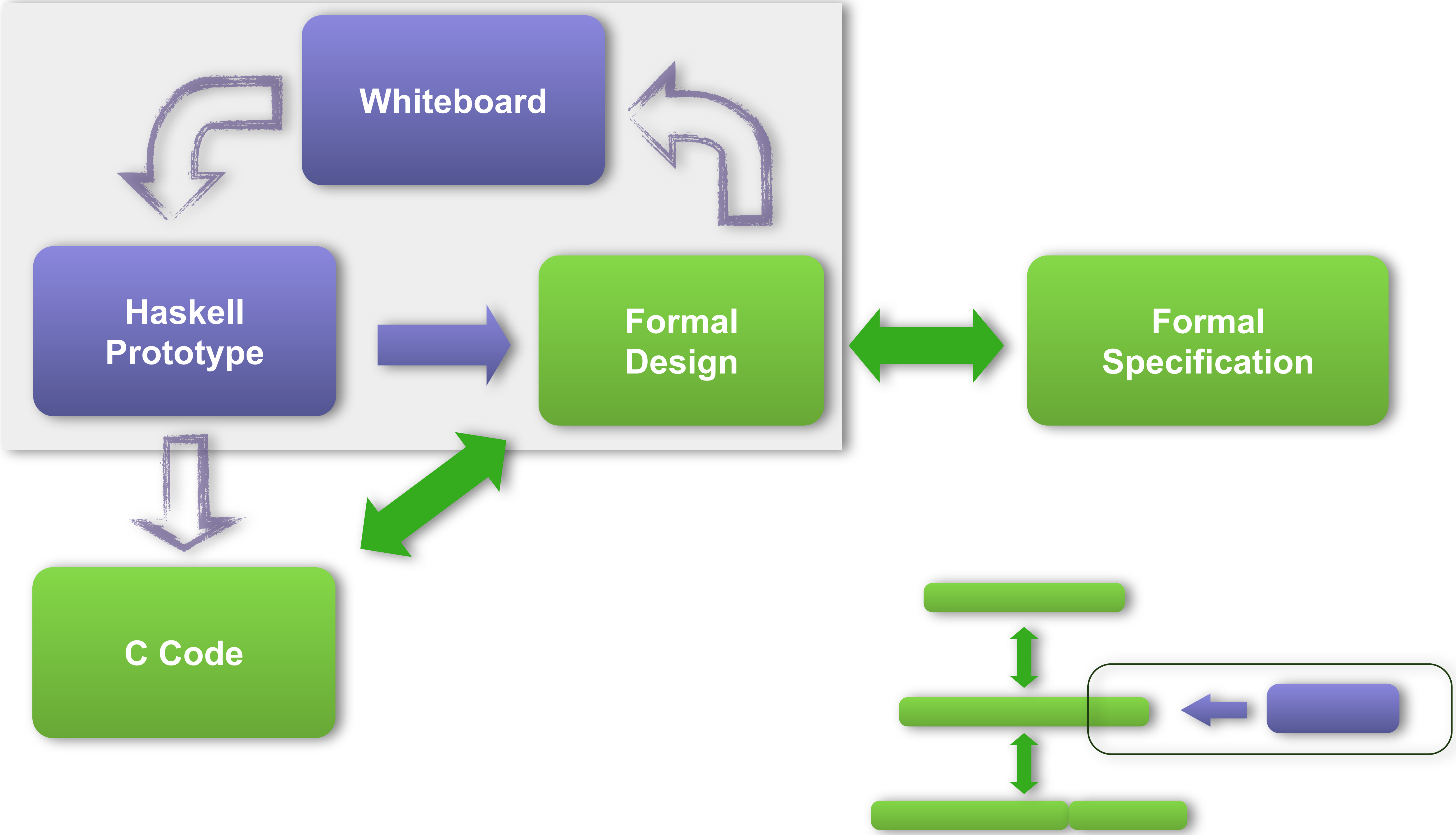
Iterative Design and Formalisation



Iterative Design and Formalisation



Iterative Design and Formalisation



Design for Verification

Reducing Complexity

Hardware

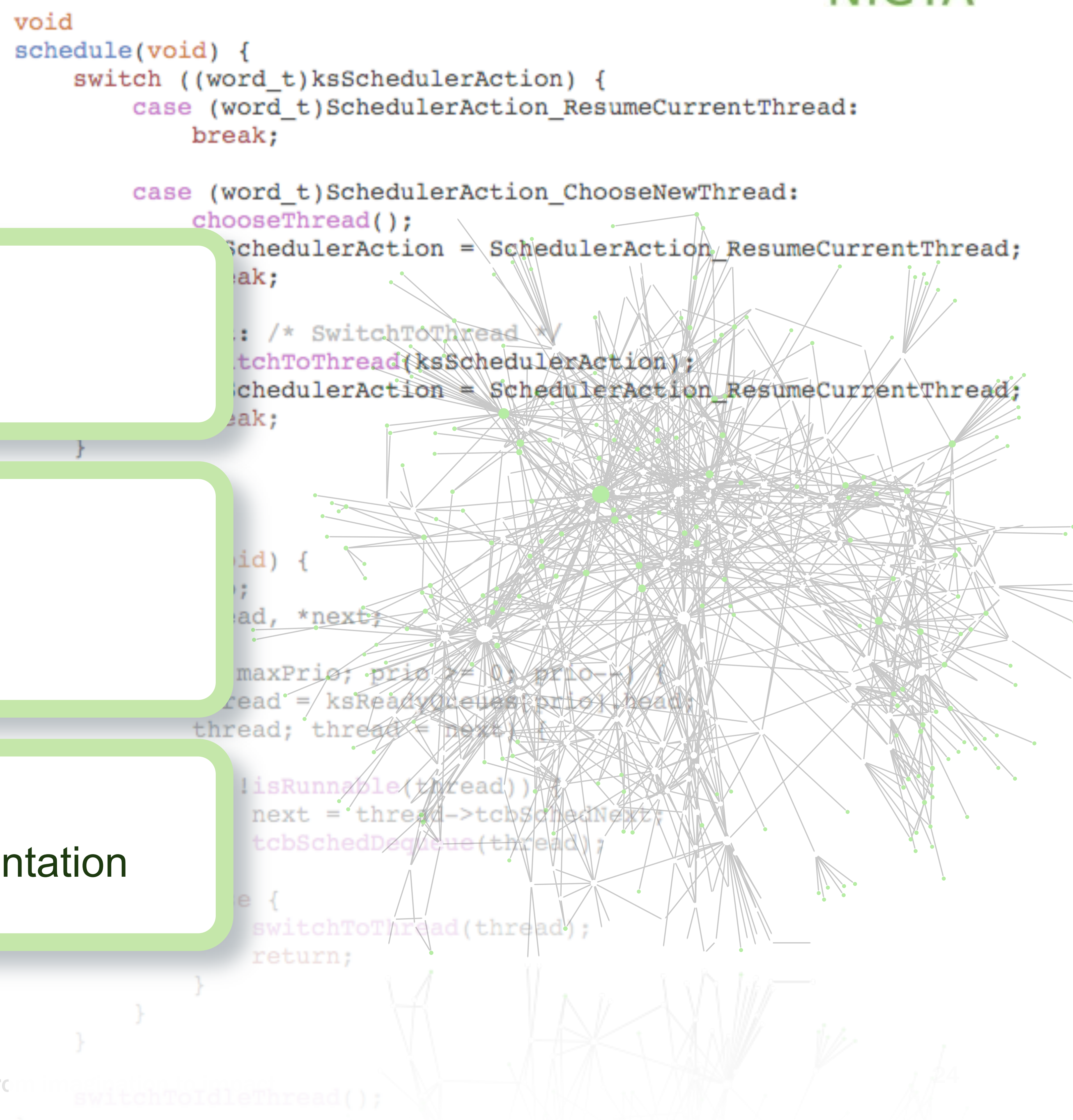
- drivers outside kernel

Concurrency

- event based kernel
- limit preemption

Code

- derive from functional representation



C subset

```
void
schedule(void) {
    switch ((word_t)ksSchedulerAction) {
        case (word_t)SchedulerAction_ResumeCurrentThread:
            break;

        case (word_t)SchedulerAction_ChooseNewThread:
            chooseThread();
```

Everything from C standard

- **including**:
  - pointers, casts, pointer arithmetic
  - data types
  - structs, padding
  - pointers into structs
  - precise finite integer arithmetic

- **minus:**
  - goto, switch fall-through
  - reference to local variable
  - side-effects in expressions
  - function pointers (restricted)
  - unions

- **plus** compiler assumptions on:
  - data layout, encoding, endianess

```
thread; thread = next) {

f(!isRunnable(thread)) {
    next = thread->tcbSchedNext;
    tcbSchedDequeue(thread);

lse {
    switchToThread(thread);
    return;
}
        }
    }
    switchToIdleThread();
}
```

Common Criteria

EAL	Requirem.	Funct Spec	TDS	Implem.
EAL1		Informal		
EAL2		Informal	Informal	
EAL3		Informal	Informal	
EAL4		Informal	Informal	Informal
EAL5		Semiformal	Semiformal	Informal
EAL6	Formal	Semiformal	Semiformal	Informal
EAL7	Formal	Formal	Formal	Informal
L4.verified	Formal	Formal	Formal	Formal

Did you find any Bugs?

Bugs found

during testing: 16

during verification:

- in C: 160
- in design: ~150
- in spec: ~150

460 bugs

Effort

Haskell design	2	py
First C impl.	2	weeks
Debugging/Testing	2	months
Kernel verification	12	py
Formal frameworks	10	py
Total	25	py

```
void
schedule(void) {
    switch ((word_t)ksSchedulerAction) {
        case (word_t)SchedulerAction_ResumeCurrentThread:
                                                          read;
                                                          read;
    }
}

void
chooseTh
    prio
    tcb_t *thread, *next;

    for(prio = maxPrio; prio >= 0; prio--) {
        for(thread = ksReadyQueues[prio].head;
            thread; thread = next) {

            if(!isRunnable(thread)) {
                next = thread->tcbSchedNext;
                tcbSchedDequeue(thread);
            }
            else {
                switchToThread(thread);
                return;
            }
        }
    }
    switchToIdleThread();
}
```

Did you find any Bugs?

Bugs found

```
void
schedule(void) {
    switch ((word_t)ksSchedulerAction) {
        case (word_t)SchedulerAction_ResumeCurrentThread:
```

during testing: 16

during verification:

- in C: 160
- in design: ~150
- in spec: ~150

460 bugs

Effort

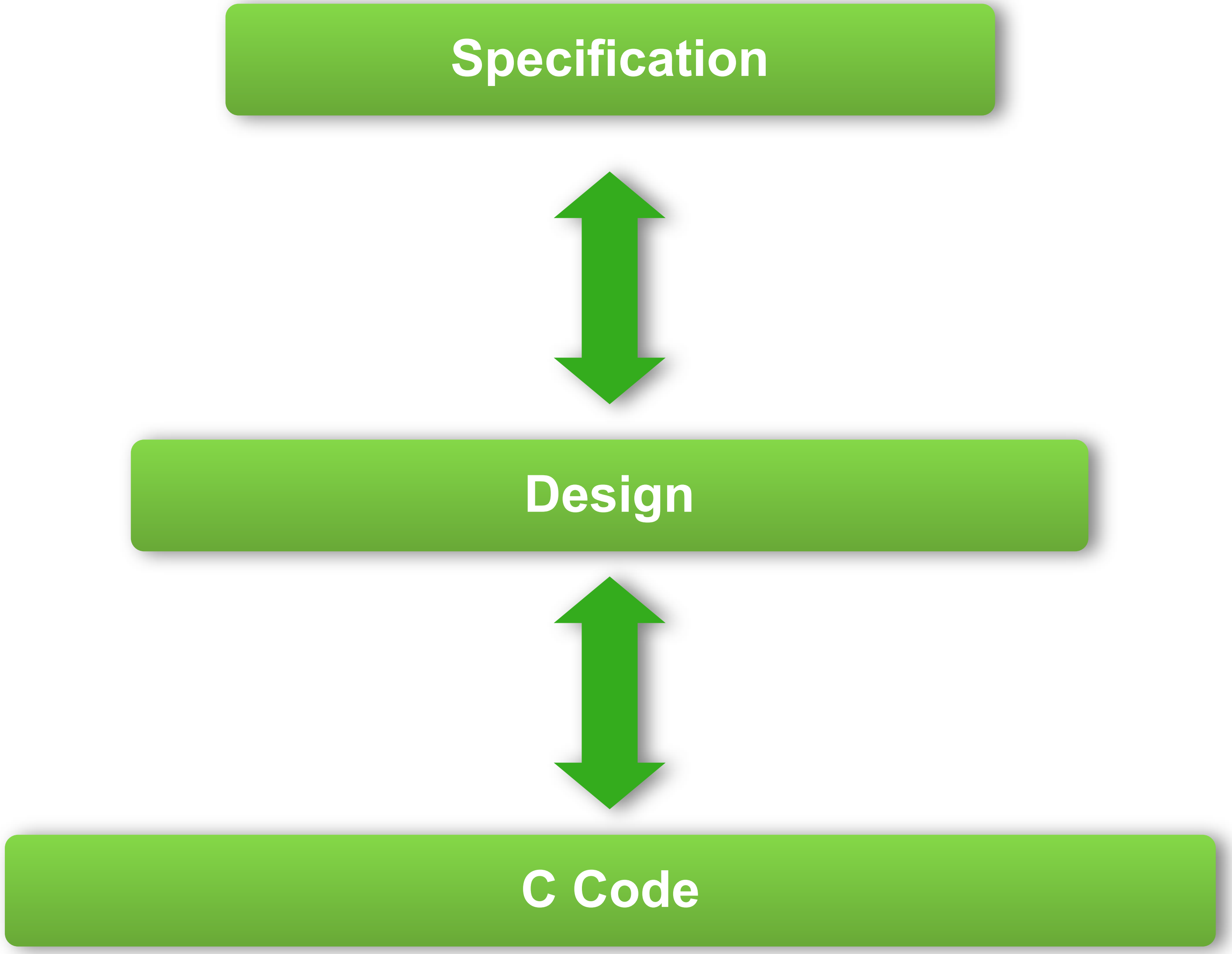
Haskell design	2	py
First C impl.	2	weeks
Debugging/Testing	2	months
Kernel verification	12	py
Formal frameworks	10	py
Total	25	py

```
void
chooseTh
    prio_
    tcb_t *thread, *next;

    for(prio = maxPrio; prio >= 0; prio--) {
        for(thread = ksReadyQueues[prio].head;
            thread; thread = next) {

            if(!isRunnable(thread)) {
                next = thread->tcbSchedNext;
                tcbSchedDequeue(thread);
            }
            else {
                switchToThread(thread);
                return;
            }
        }
    }
```

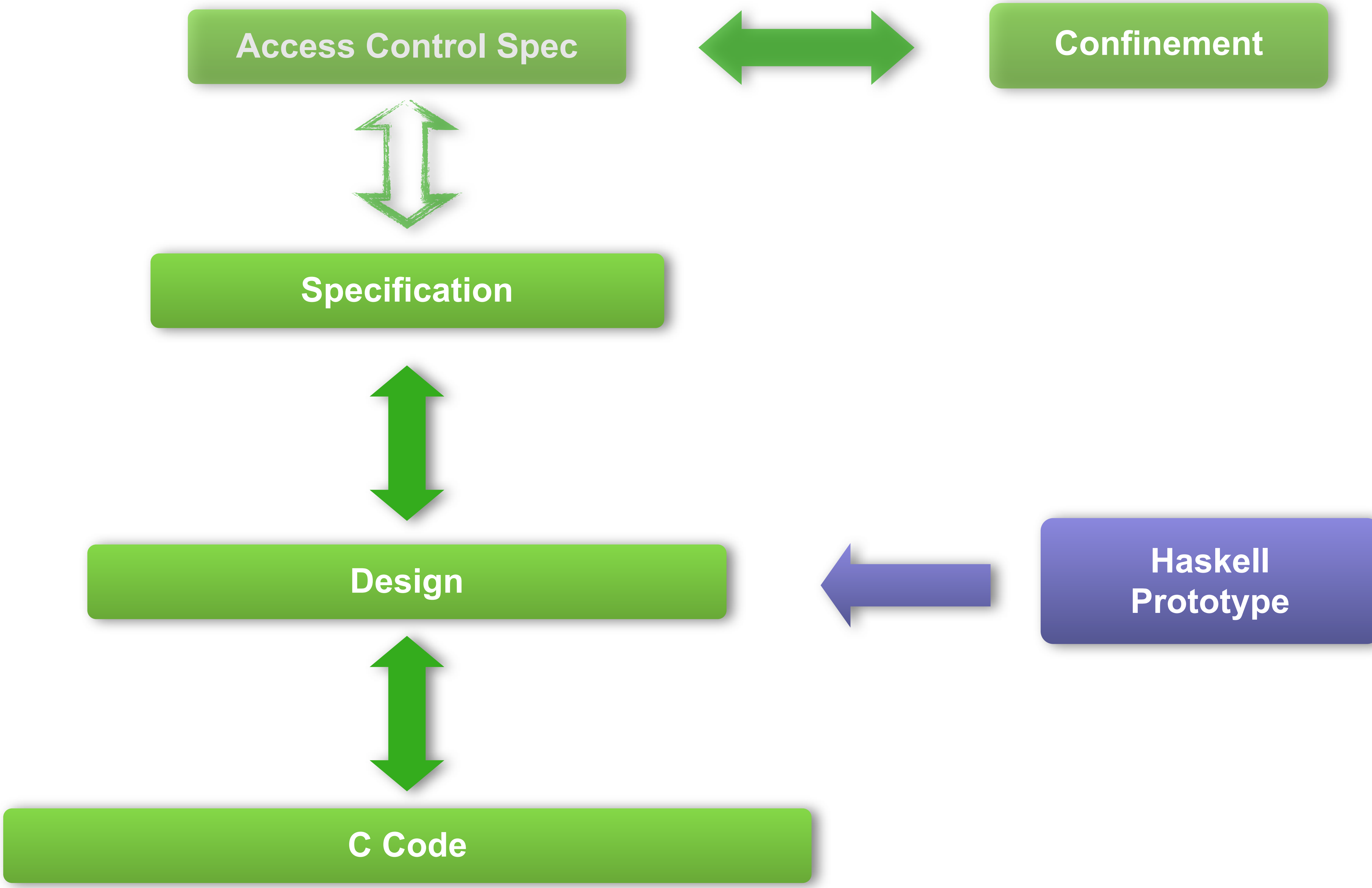
Access Control

Access Control

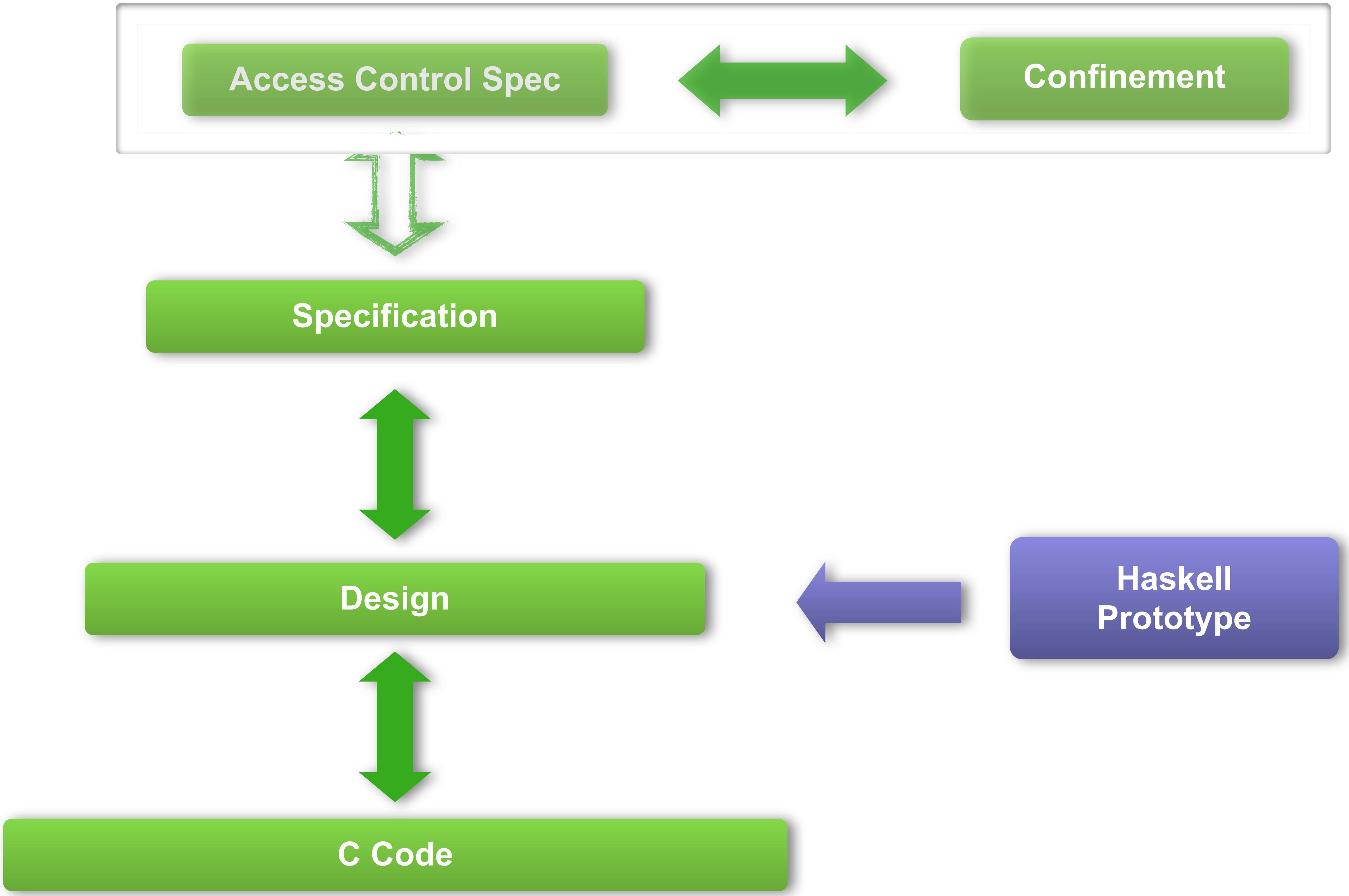
Proof Architecture



Proof Architecture



Proof Architecture



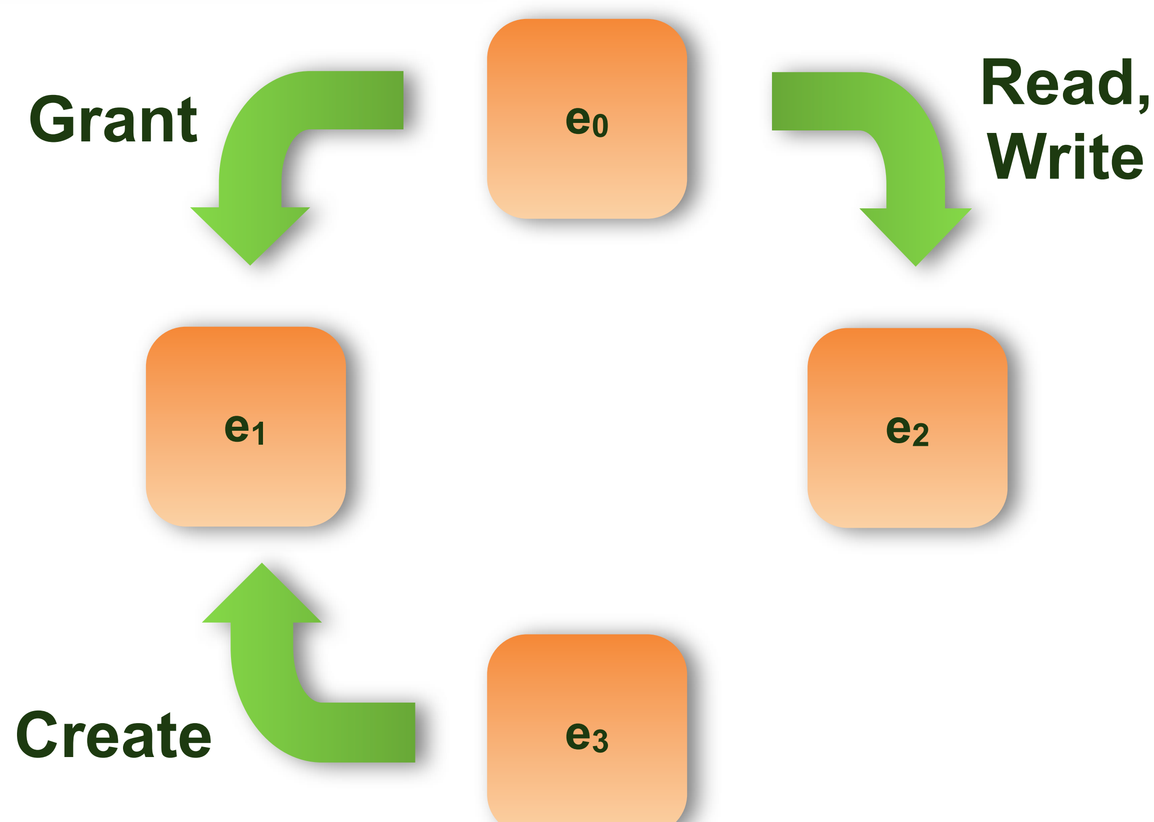
Take-Grant model

NICTA

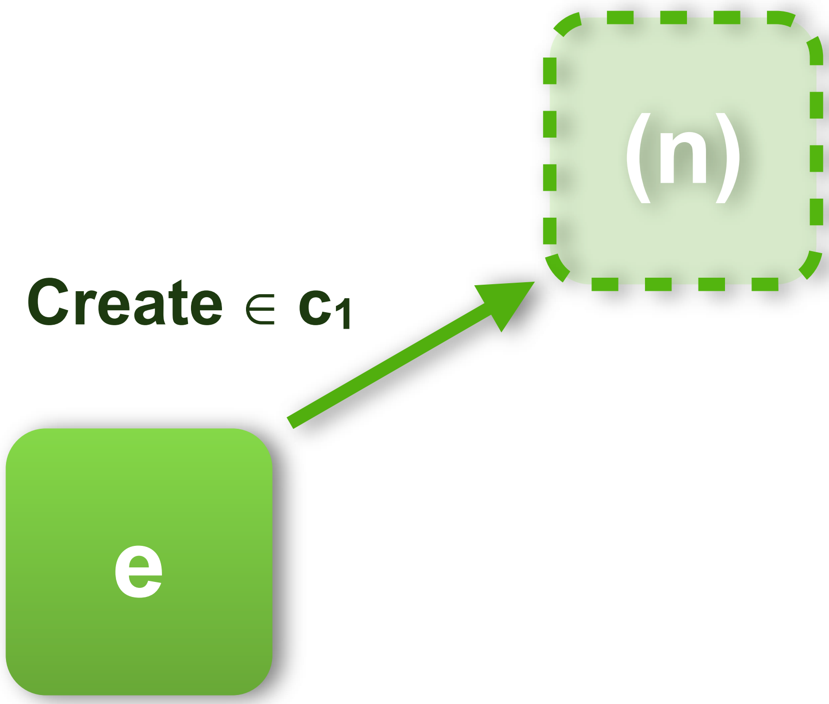
Lipton and Snyder:

- entities represented as nodes of a graph
- capabilities represented as edges of a graph
- rights are contained in capabilities

The Rights: **Read**
Write
Take
Grant
Create

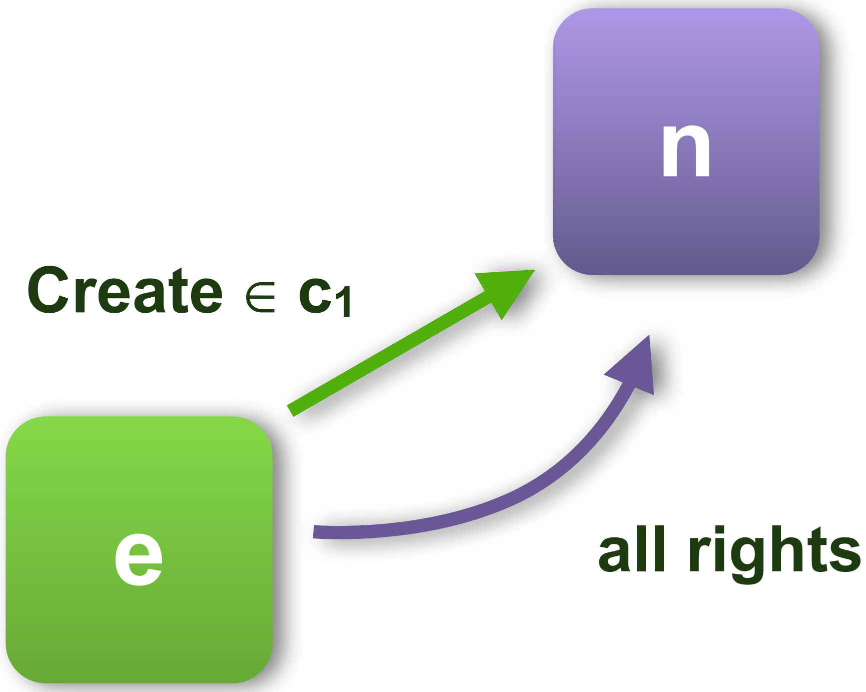


Operations - Create



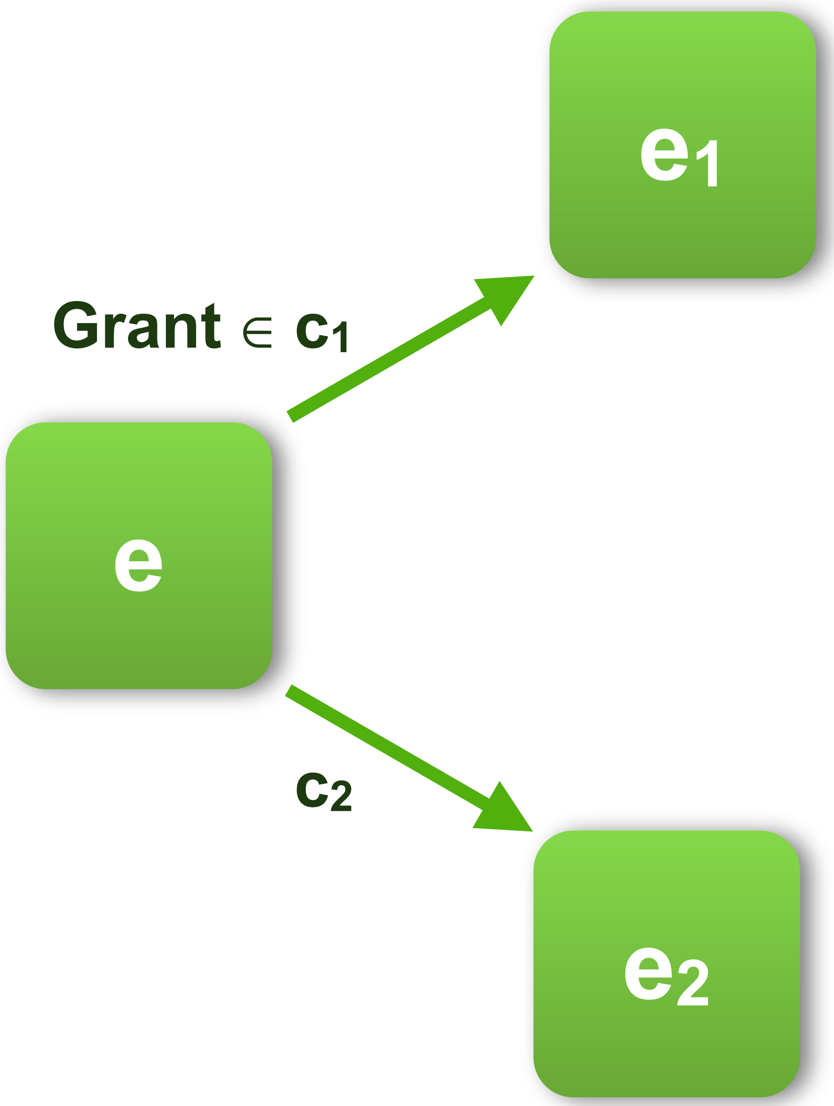
Create new entity

Operations - Create



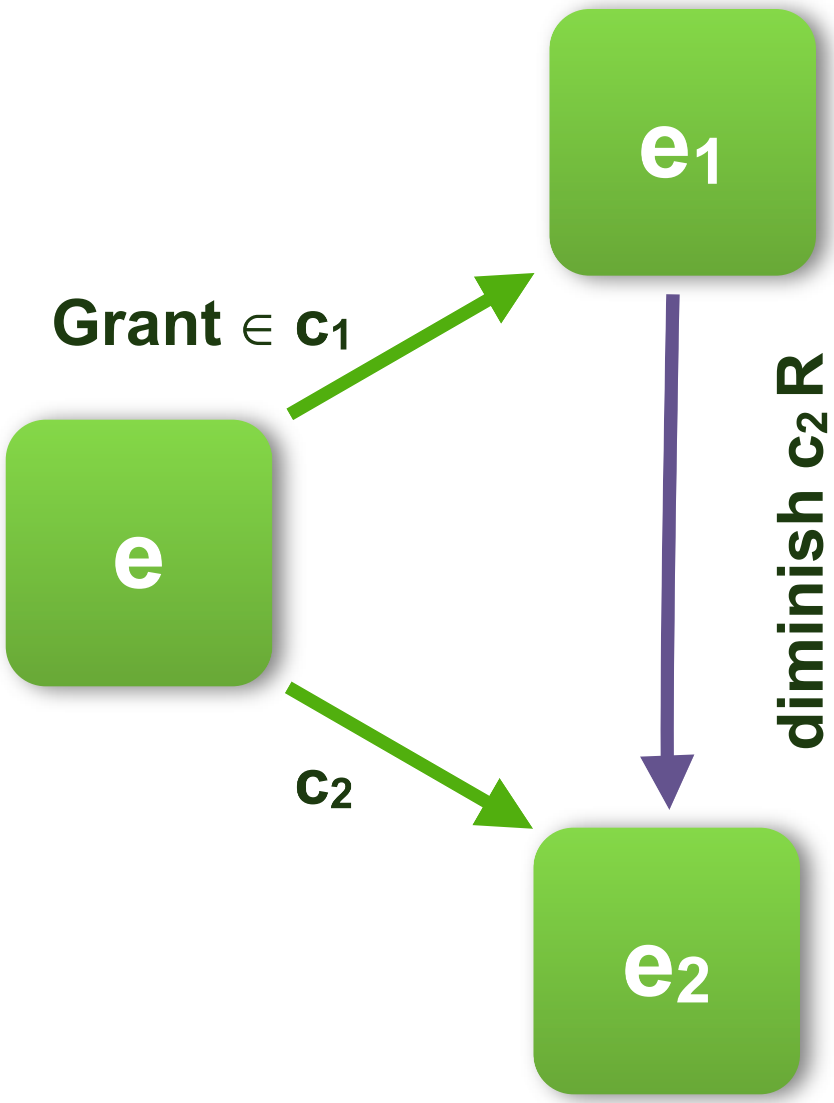
Create new entity

Operations - Grant



Grant c_2 to e_1
with mask R

Operations - Grant



Grant c_2 to e_1
with mask R

Operations - Remove/Delete

Remove capability c_2

Delete entity e_2

Operations - Remove/Delete

Remove capability c_2

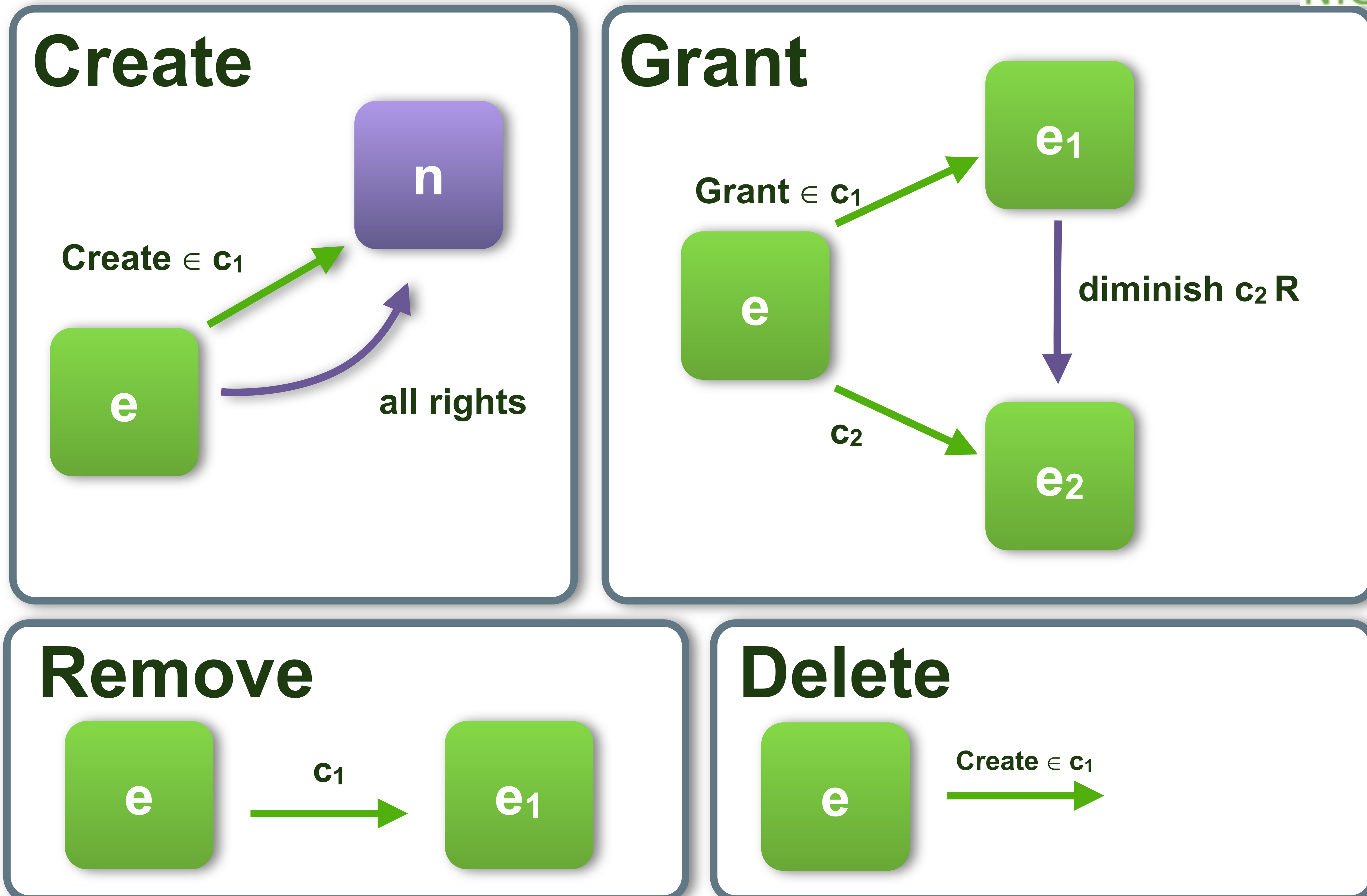
Delete entity e_2

Operations - Remove/Delete

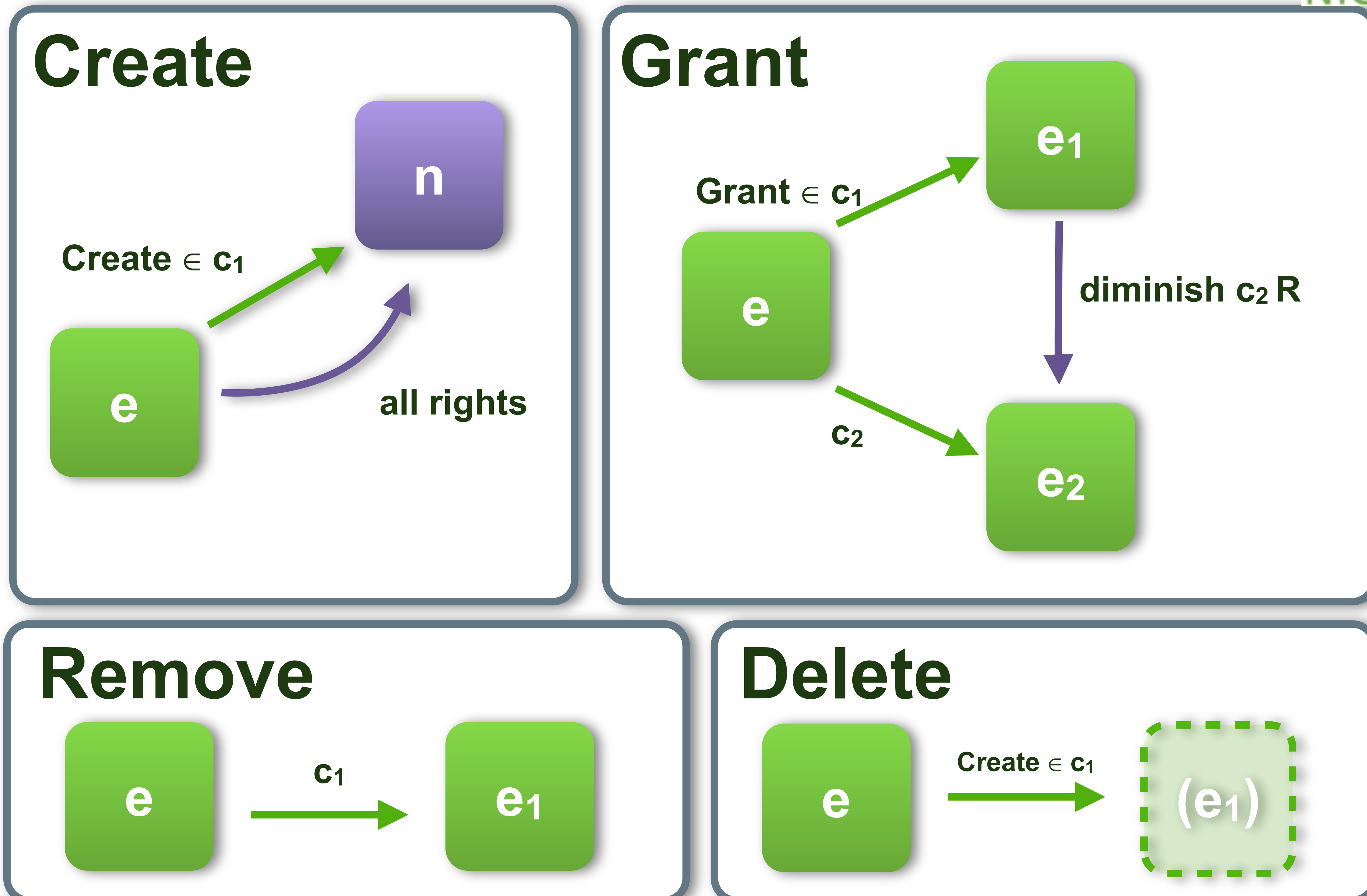
Remove capability c_2

Delete entity e_2

Operations Summary



Operations Summary



Questions

For any state in the future:

- Can entity **E** do **X?**
- Can **E** gain authority to do **X**?
- Can **E** gain more **authority** than it has?
- **How** much more?
- Can **information** flow from A to B?

Leaking authority

x can ***leak*** to y

Leaking authority

x can ***leak*** to y

Leak: $s \vdash x \rightarrow y \equiv \text{grant-cap } y :< \text{caps-of } s\ x$

Leaking authority

x can ***leak*** to y

Leak: $s \vdash x \rightarrow y \equiv$ grant-cap y :< caps-of s x

Connected: $s \vdash x \leftrightarrow y \equiv s \vdash x \rightarrow y \vee s \vdash y \rightarrow x$

Leaking authority

x can ***leak*** to y

Leak: $s \vdash x \rightarrow y \equiv \text{grant-cap } y :< \text{caps-of } s\ x$

Connected: $s \vdash x \leftrightarrow y \equiv s \vdash x \rightarrow y \lor s \vdash y \rightarrow x$

Subsystems: $\text{subsys } s\ x = \{e.\ s \vdash e \leftrightarrow^* x\}$

Leaking authority

x can ***leak*** to y

Leak: $s \vdash x \rightarrow y \equiv$ grant-cap y :< caps-of s x

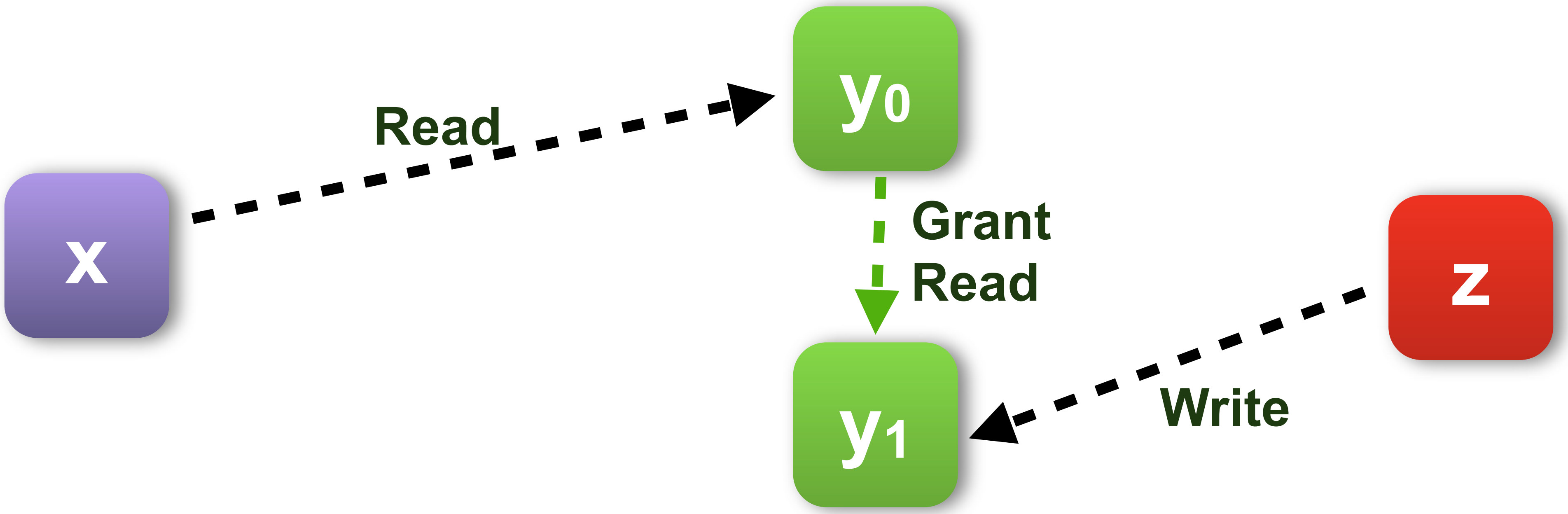
Connected: $s \vdash x \leftrightarrow y \equiv s \vdash x \rightarrow y \vee s \vdash y \rightarrow x$

Subsystems: subsys s x = $\{e.\ s \vdash e \leftrightarrow^* x\}$

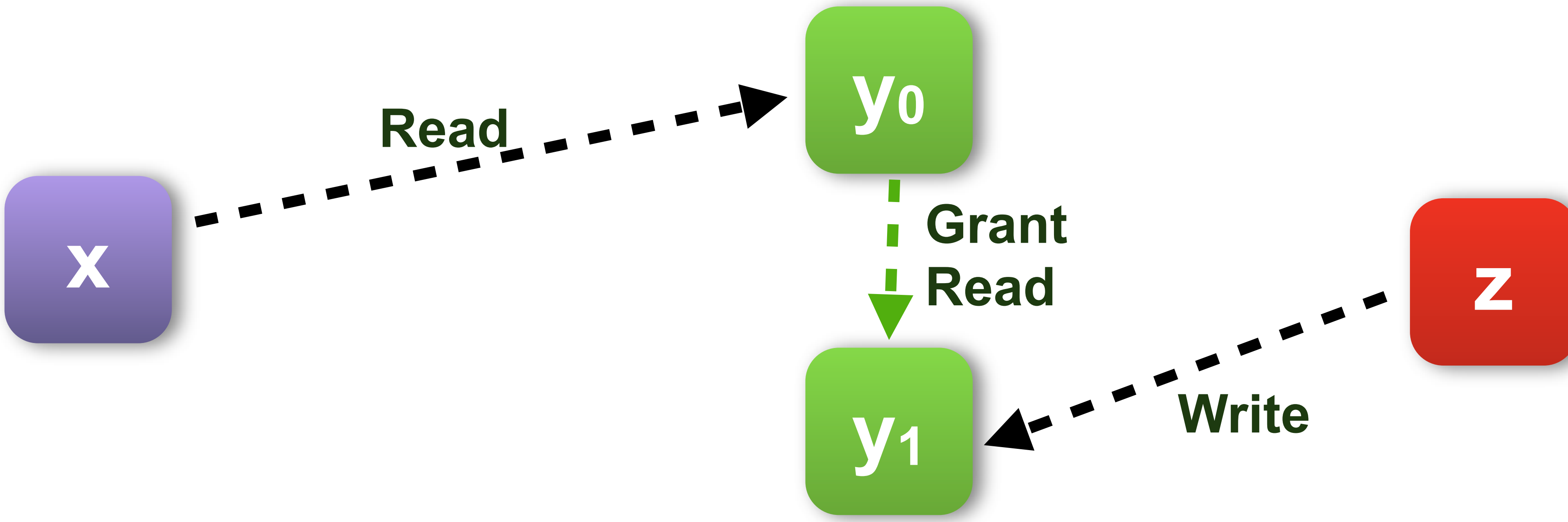
Theorems: $s' \in$ execute cmds s $\wedge\ s' \vdash x \leftrightarrow^* y \Rightarrow s \vdash x \leftrightarrow^* y$

$s' \in$ execute cmds s $\wedge$ c :> subsys-caps s x $\Rightarrow$ c :> subsys-caps s' x

Explicit information flow

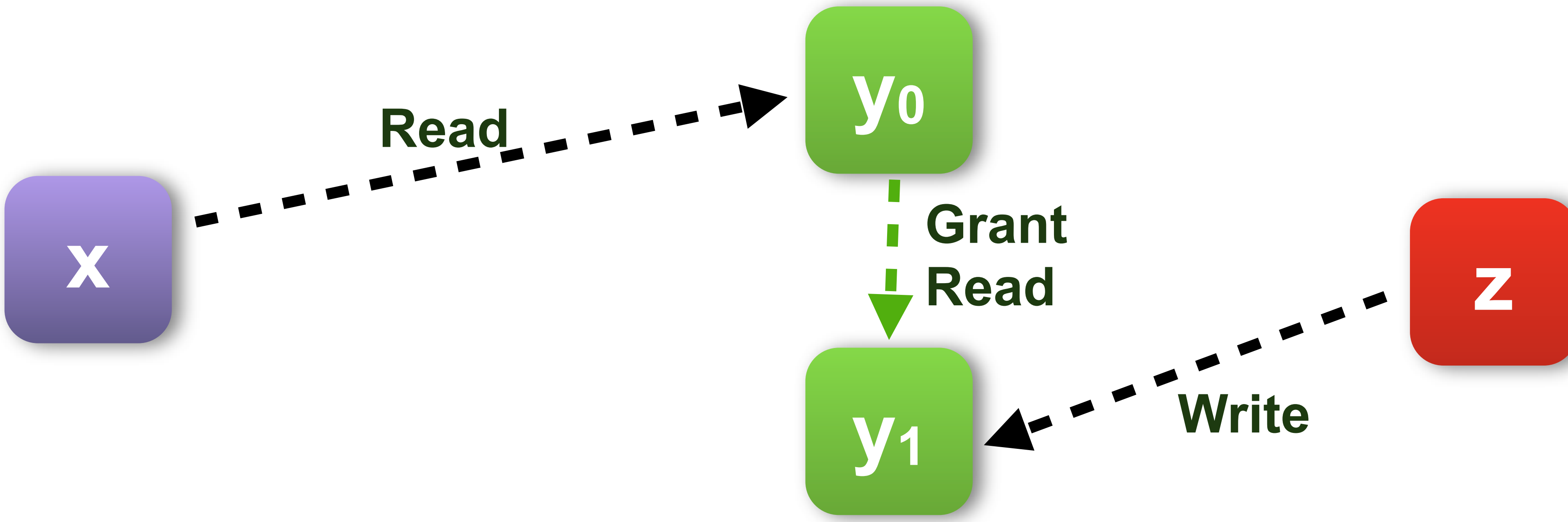


Explicit information flow



Like Bishop's analysis of *islands*, examine information flow between *subsystems*

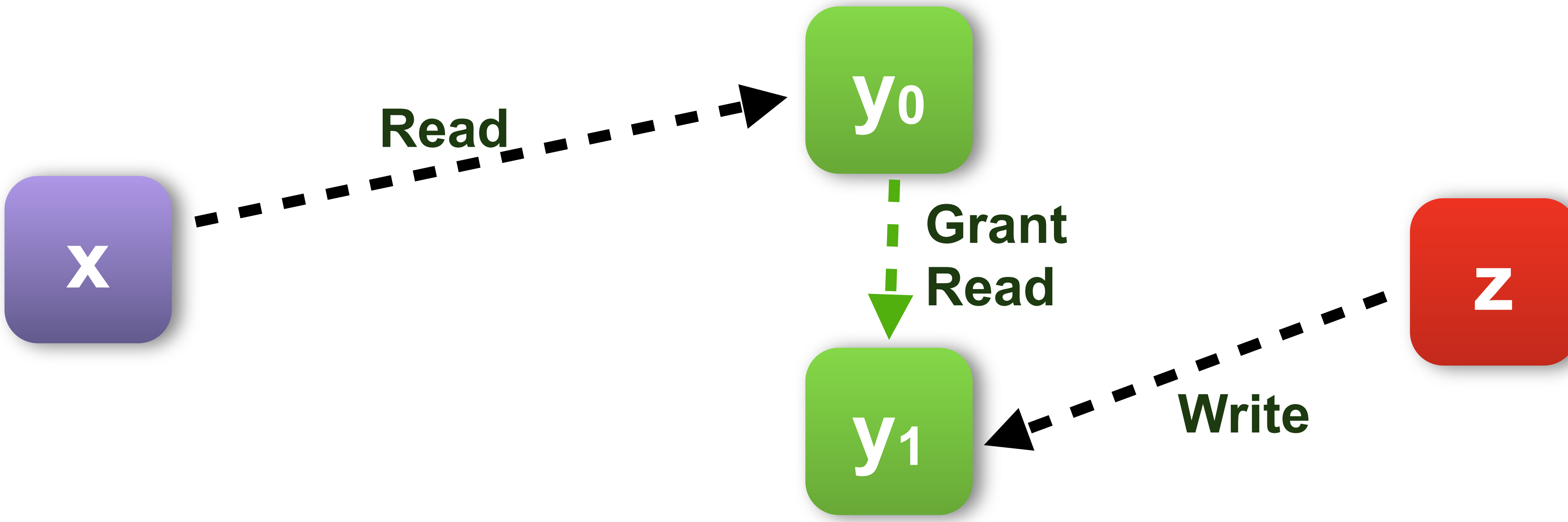
Explicit information flow



Like Bishop's analysis of *islands*, examine information flow between *subsystems*

Flow: $s \vdash x \rightarrow y = \exists x' \in \text{subsys } s\ x.\ \exists y' \in \text{subsys } s\ y.$
$\text{read-cap } x' :< \text{caps-of } s\ y' \vee \text{write-cap } y' :< \text{caps-of } s\ x'$

Explicit information flow

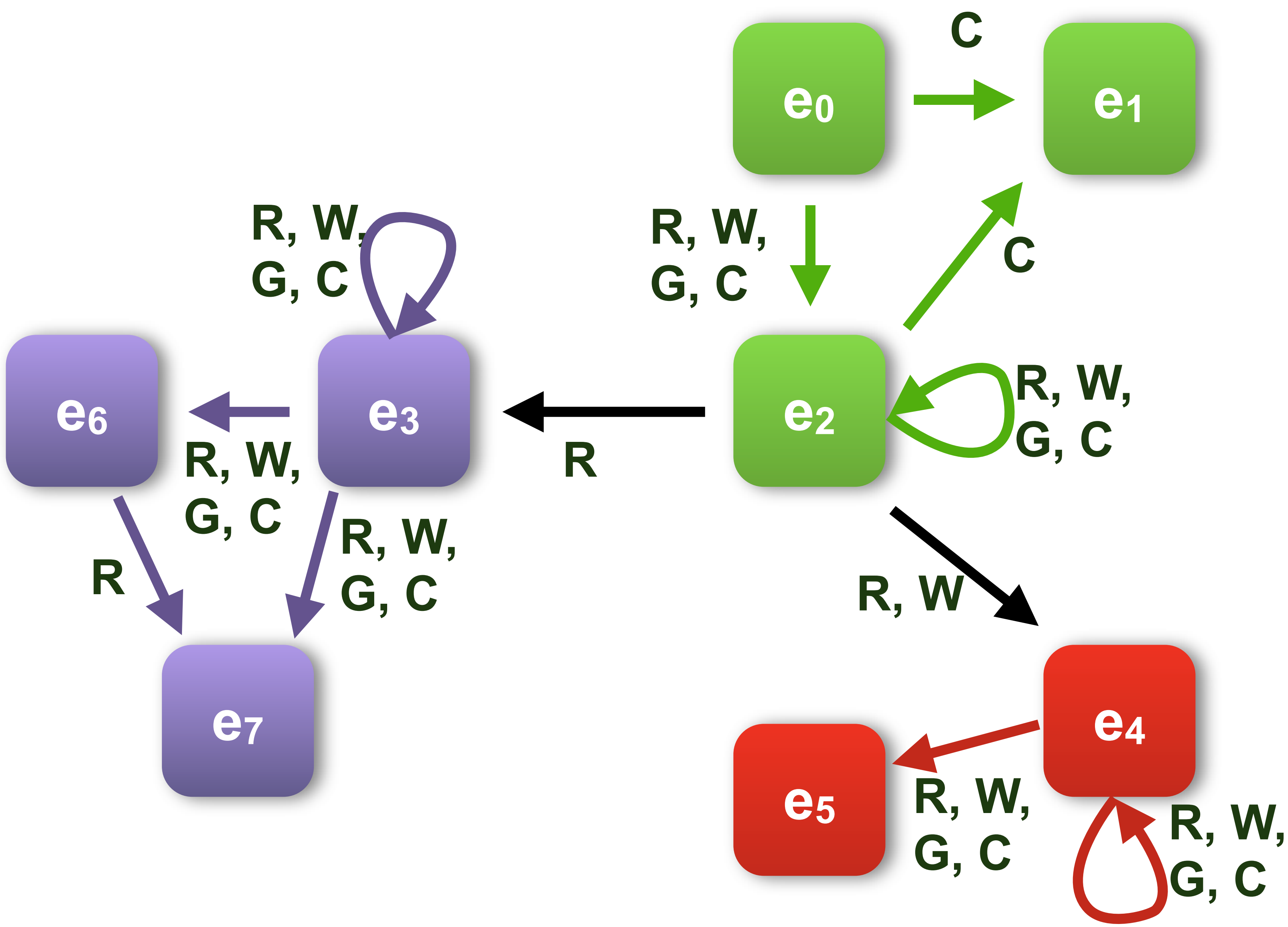


Like Bishop's analysis of *islands*, examine information flow between *subsystems*

Flow: $s \vdash x \rightsquigarrow y = \exists x' \in \text{subsys } s\ x.\ \exists y' \in \text{subsys } s\ y.$
$\text{read-cap } x' :< \text{caps-of } s\ y' \lor \text{write-cap } y' :< \text{caps-of } s\ x'$

Theorems: $s' \in \text{execute cmds } s \land \neg\, s \vdash x \rightsquigarrow^* y \Rightarrow \neg\, s' \vdash x \rightsquigarrow^* y$

Example

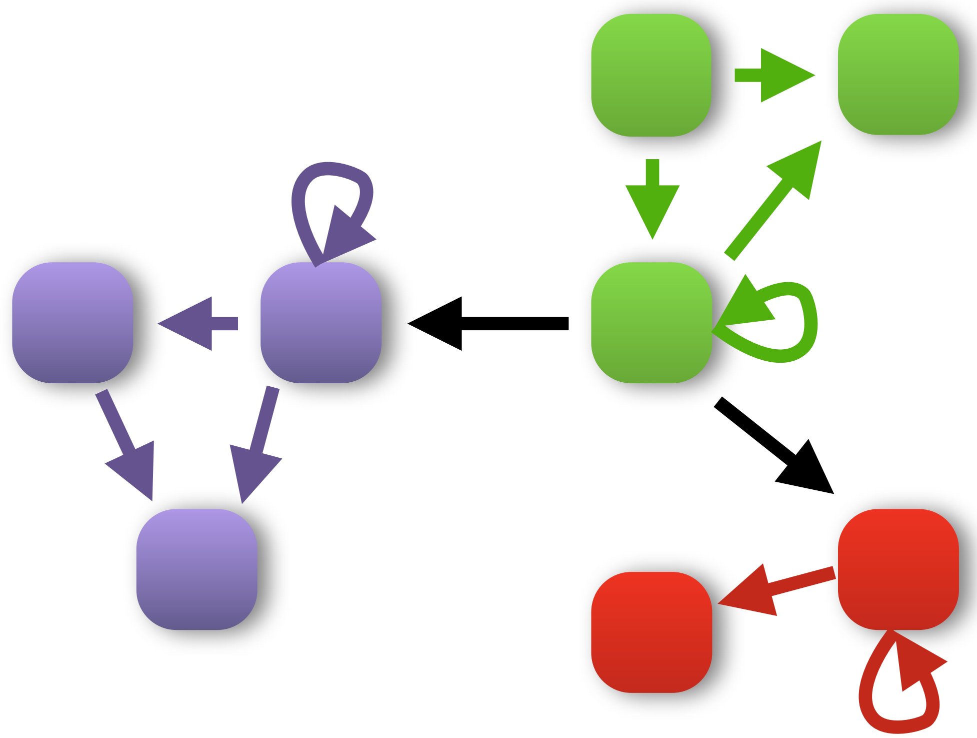

e0
C
e1
R, W,
G, C
C
R, W,
G, C
e6
e3
R
e2
R, W,
G, C
R, W,
G, C
R, W,
G, C
R
R, W
e7
e4
e5
R, W,
G, C
R, W,
G, C

Take-Grant Summary

- Simple capability model

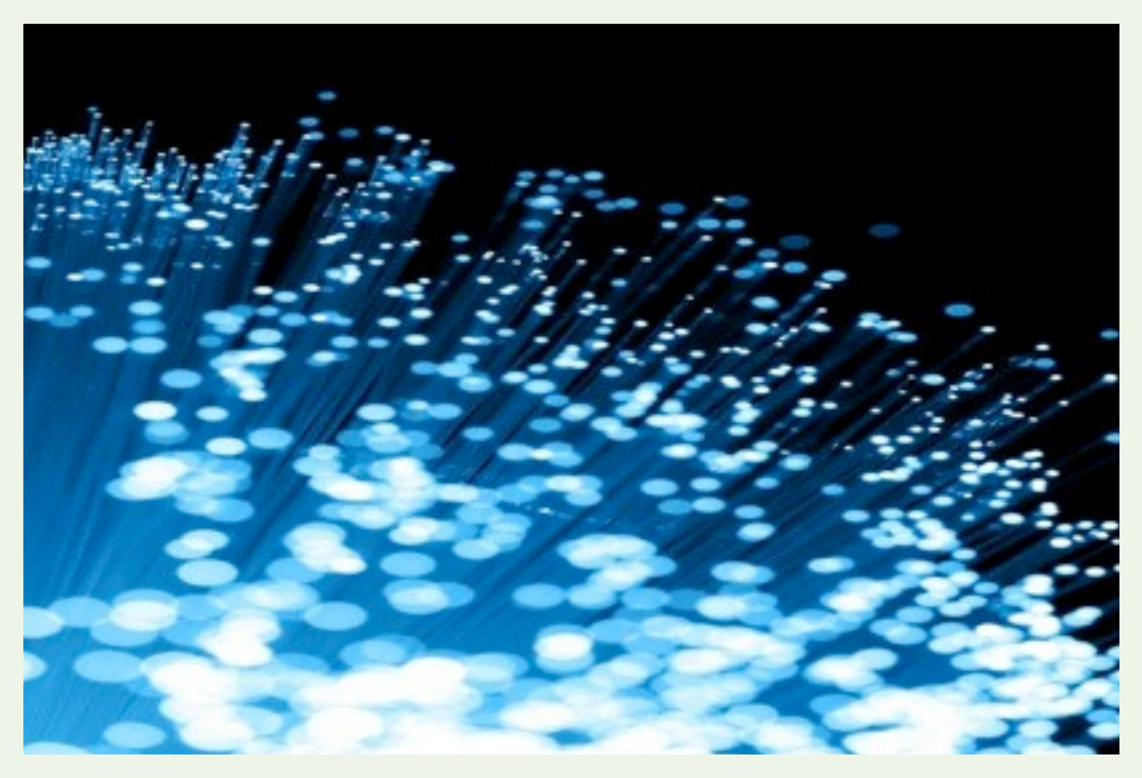
- Decidable access control
  - Basic information flow model
  - Isolated subsystems

- Proof in progress:
  - seL4 implements this model



What's next?

What's next?



Trustworthy Embedded Systems

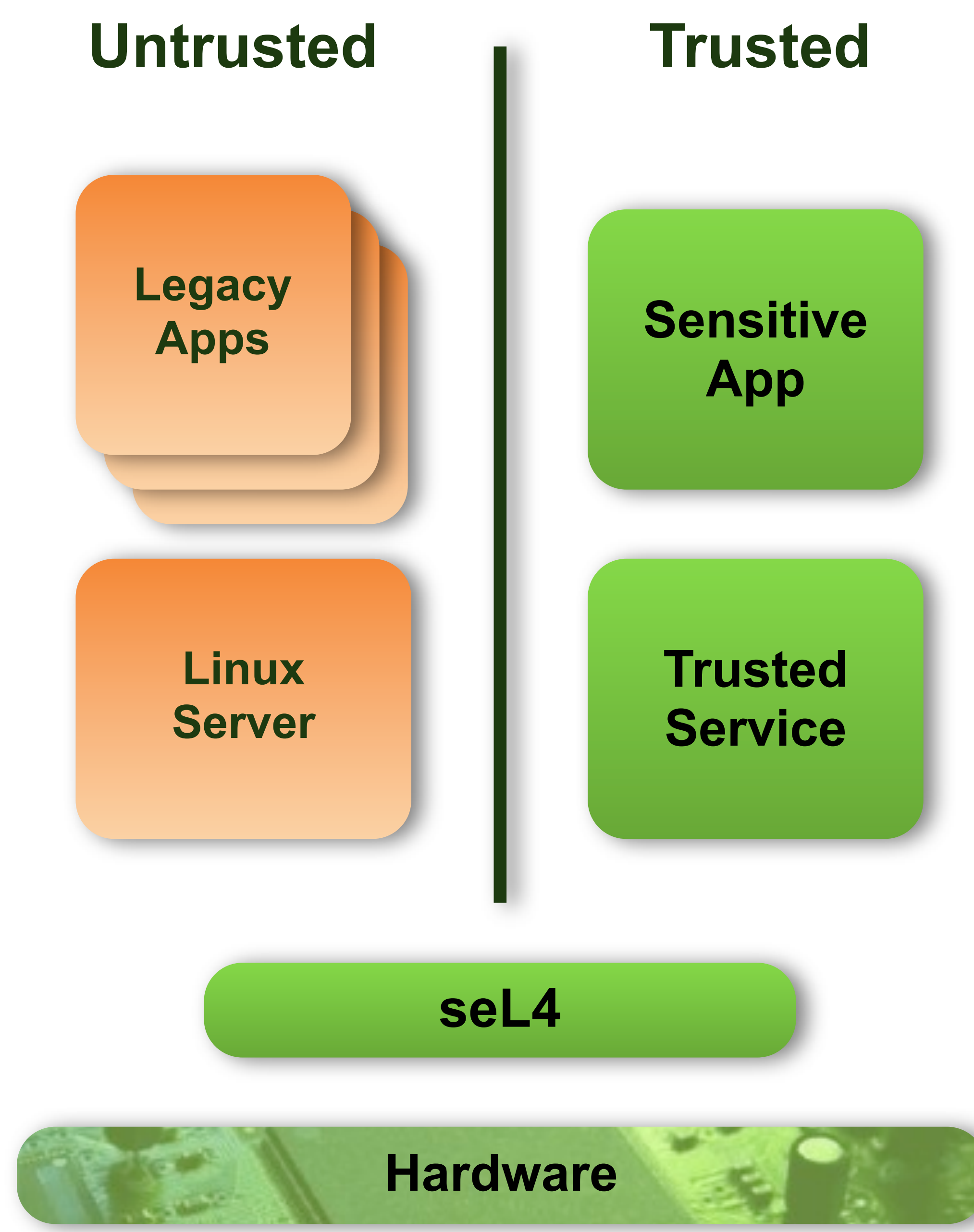
- **L4.verified:**
  functional correctness for
  10,000 loc

- **Next step:**
  formal guarantees for
  > 1,000,000 loc

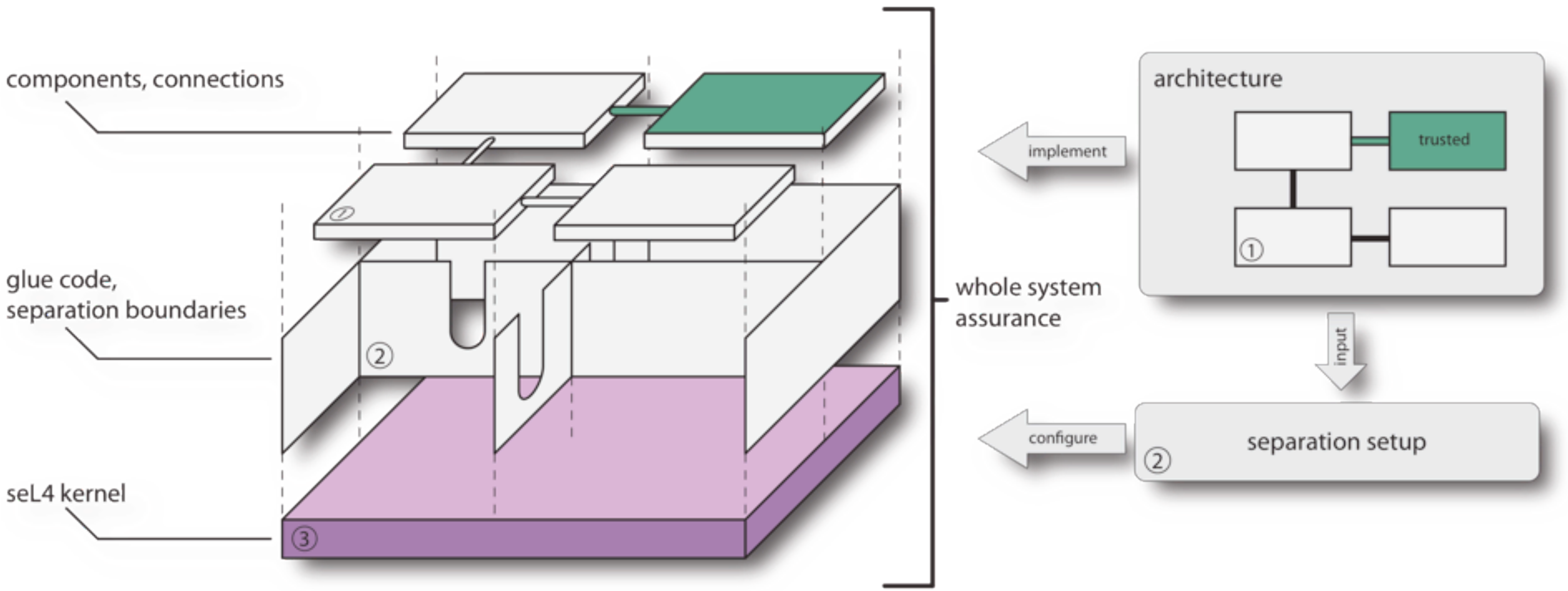
How?

Exploit:

- seL4 isolation
- verified properties
- MILS architectures



Global picture



- **Build system with minimal TCB**
- **Formalise and prove security properties about architecture**
- **Prove correctness of trusted components**
- **Prove correctness of setup**

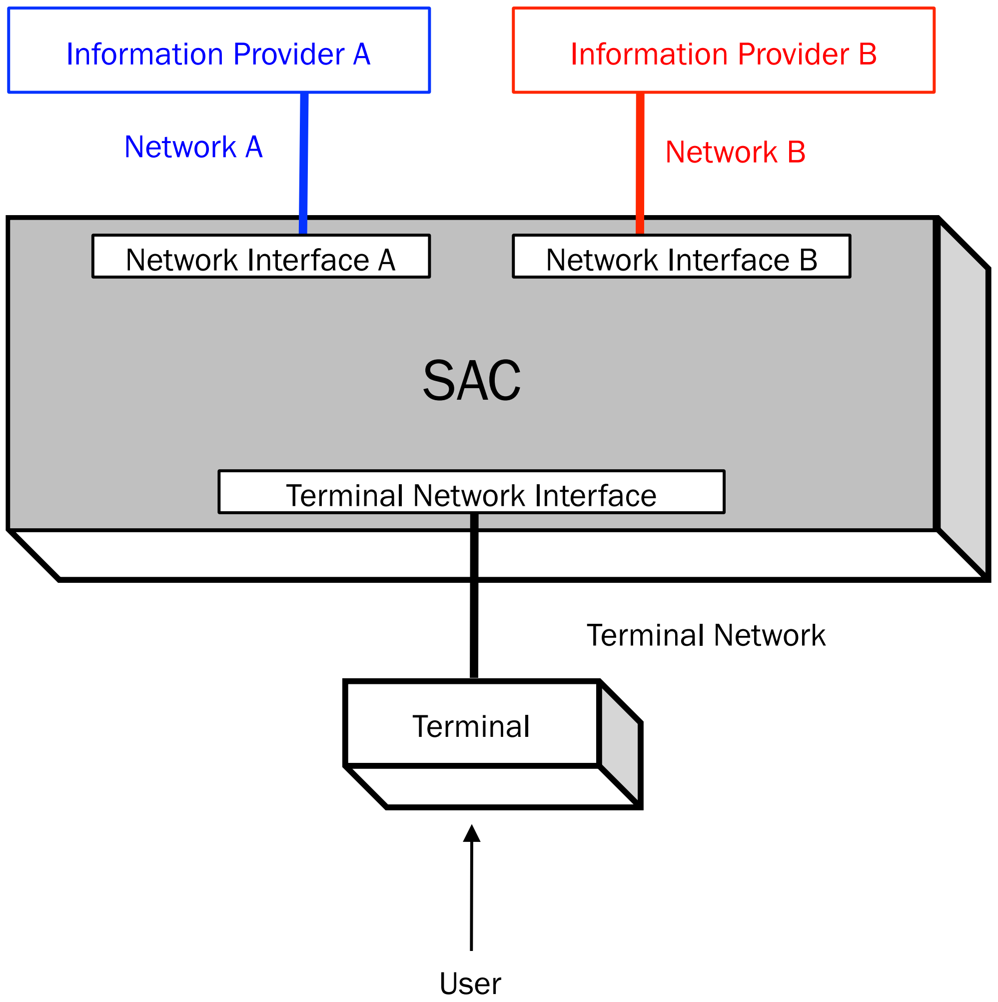
Example System

- Multilevel Secure Access Device

No information flow
between providers **A** and **B**

through SAC

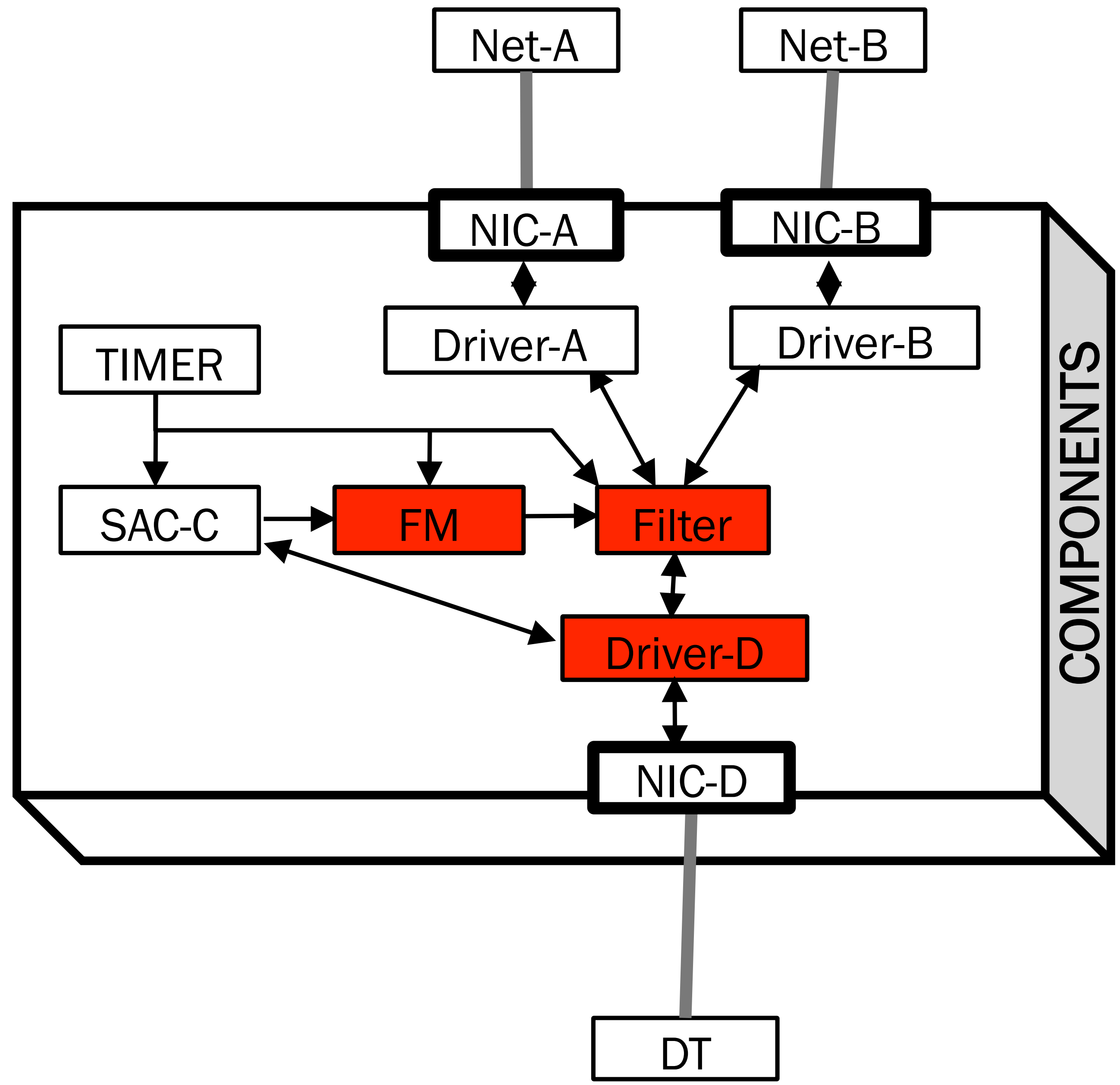
even if they collaborate



First Design

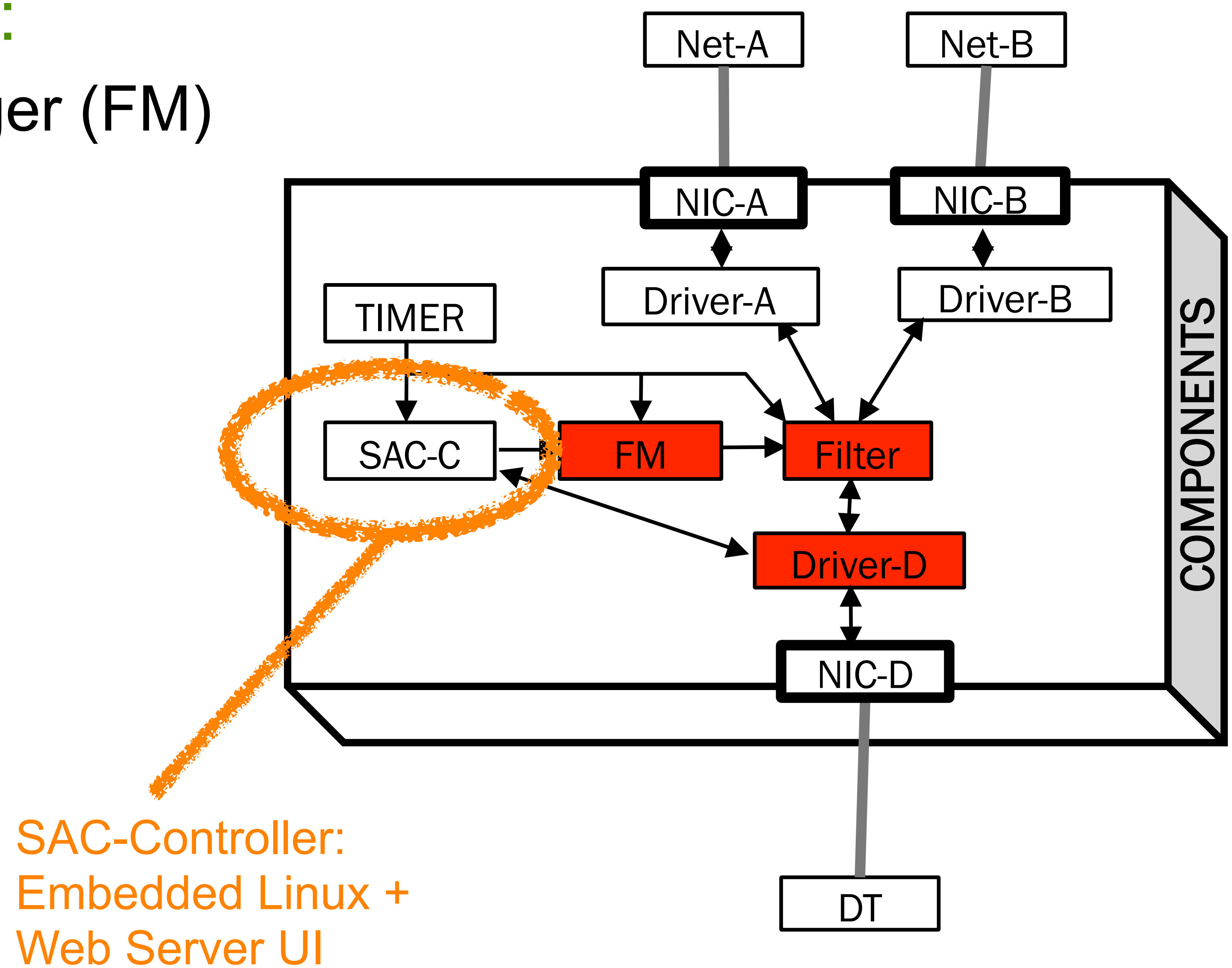
- Minimal TCB:
  - Filter Manager (FM)
  - Filter
  - Driver for D

- (and)
  - kernel
  - booter
  - hardware



First Design

- Minimal TCB:
  - Filter Manager (FM)
  - Filter
  - Driver for D
- (and)
  - kernel
  - booter
  - hardware



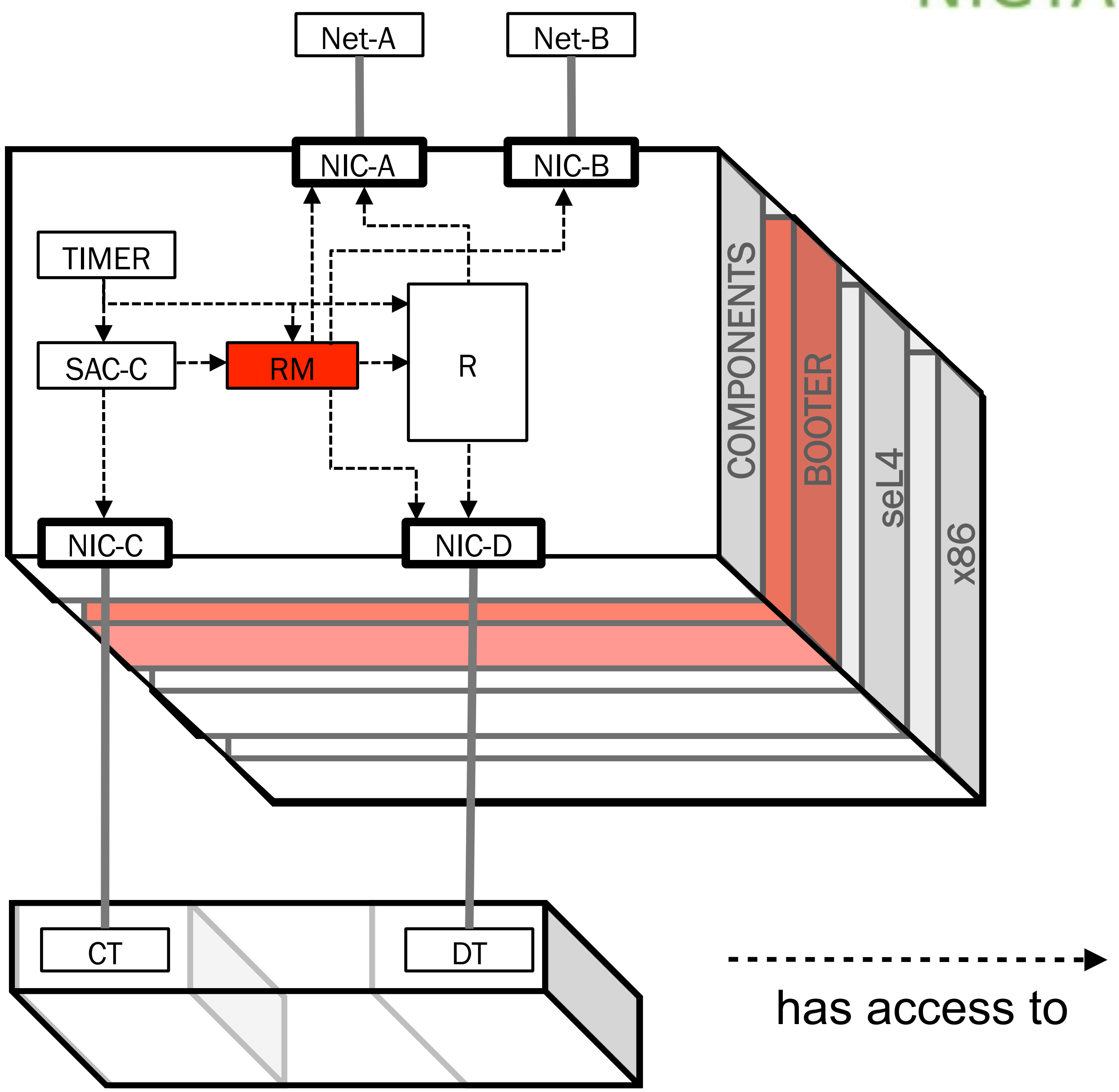
First Design

- **Minimal TCB?**
  - Filter Manager (FM)
  - Filter
  - Driver for D
- (and)
  - kernel
  - booter
  - hardware

We can do better!

- **Even smaller TCB**
  - Router Manager (RM)

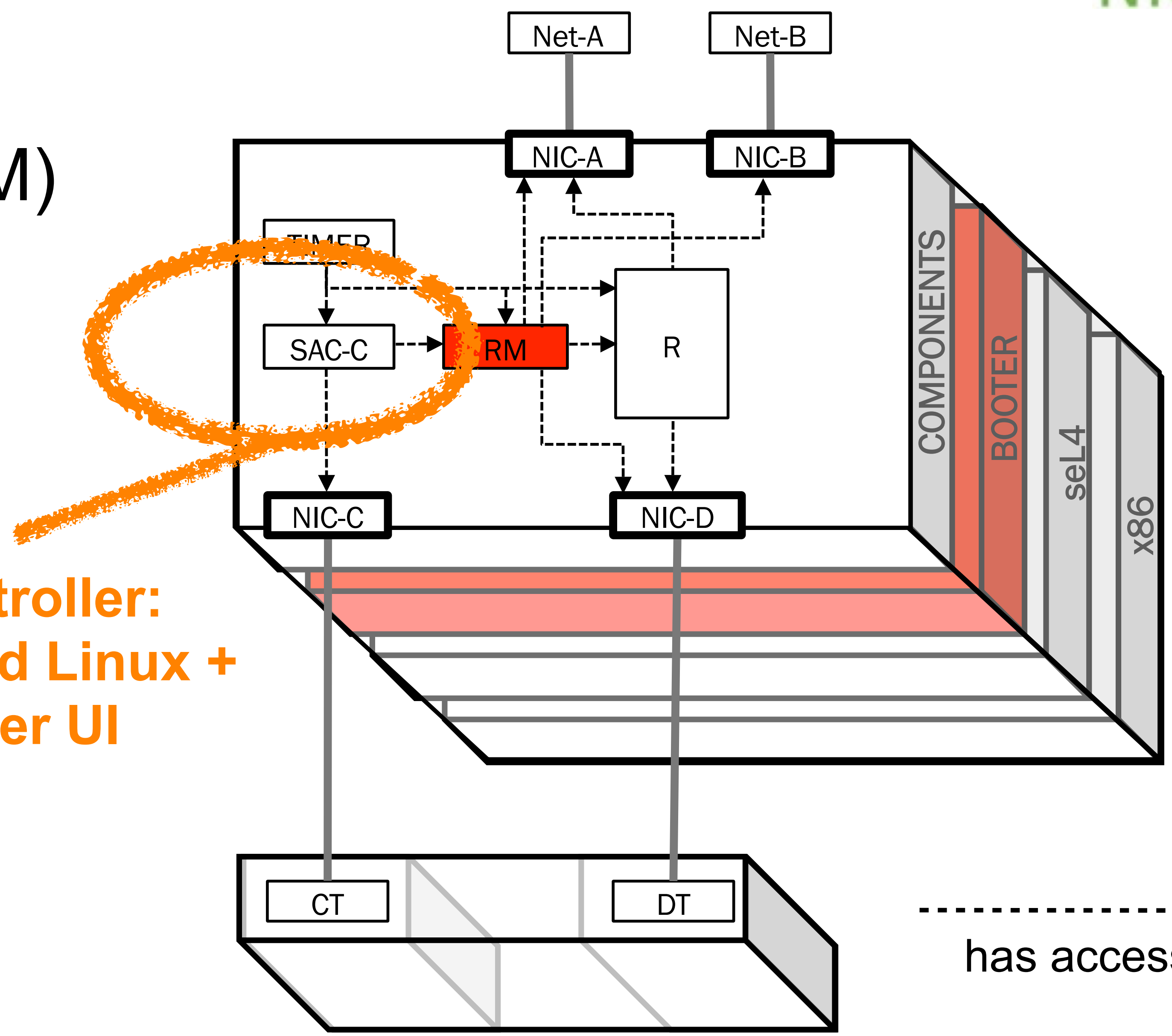
- (and)
  - kernel
  - booter
  - hardware



Net-A = Network A
Net-B = Network B
NIC-A = Network Card for Network A
NIC-B = Network Card for Network B
NIC-C = Control Network Card
NIC-D = Data Network Card
CT = Control Terminal
DT = Data Terminal
R= Router
RM = Router Manager
SAC-C = SAC Controller

We can do better!

- **Even smaller TCB**
  - Router Manager (RM)



- (and)
  - kernel
  - booter
  - hardware

SAC-Controller: Embedded Linux + Web Server UI

has access to

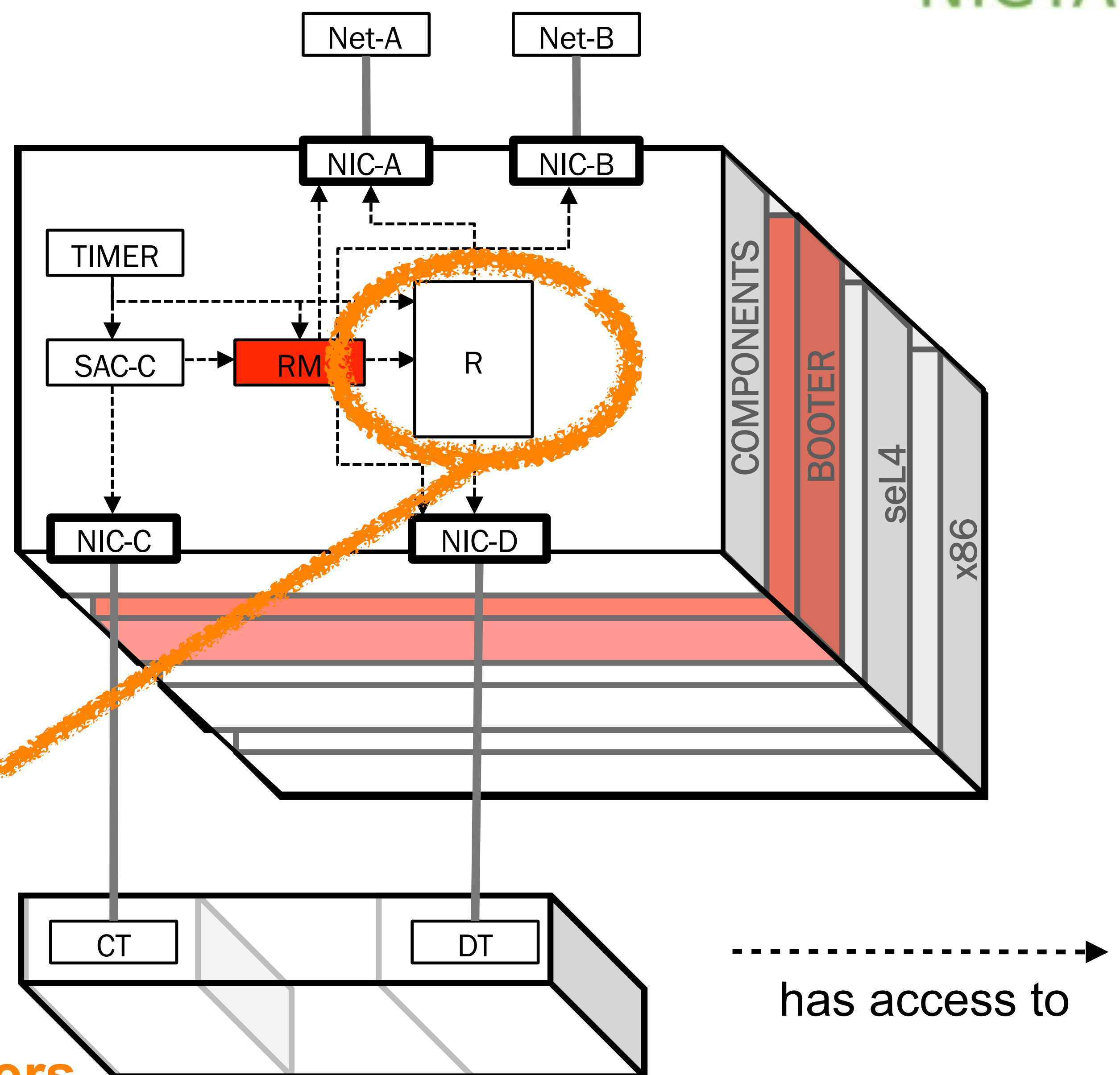
Net-A = Network A
Net-B = Network B
NIC-A = Network Card for Network A
NIC-B = Network Card for Network B
NIC-C = Control Network Card
NIC-D = Data Network Card
CT = Control Terminal
DT = Data Terminal
R= Router
RM = Router Manager
SAC-C = SAC Controller

We can do better!

- **Even smaller TCB**
  - Router Manager (RM)

- (and)
  - kernel
  - booter
  - hardware

Router:
Embedded Linux/
Network Routing + Drivers

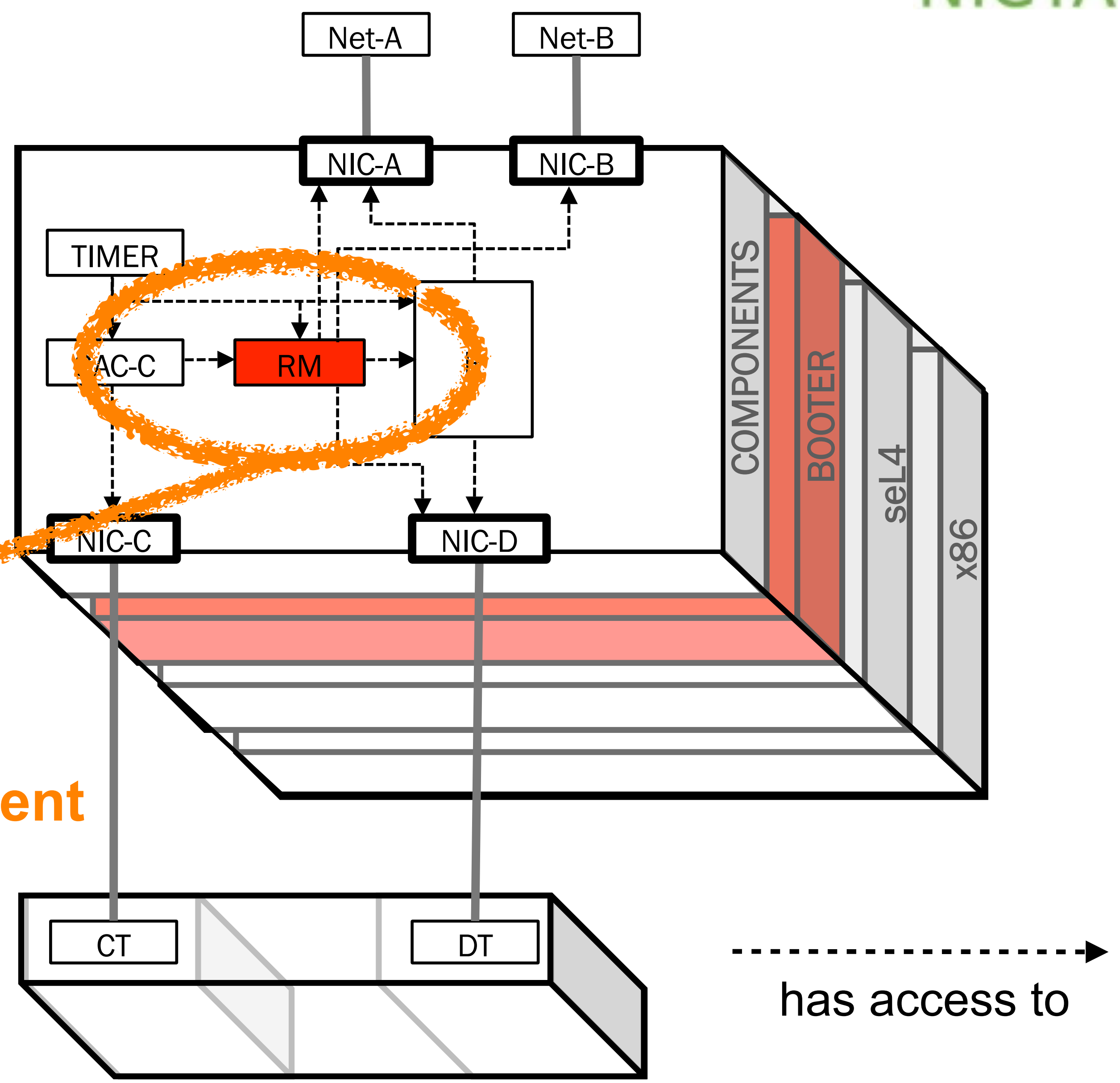


Net-A = Network A	NIC-C = Control Network Card	R= Router
Net-B = Network B	NIC-D = Data Network Card	RM = Router Manager
NIC-A = Network Card for Network A	CT = Control Terminal	SAC-C = SAC Controller
NIC-B = Network Card for Network B	DT = Data Terminal	

We can do better!

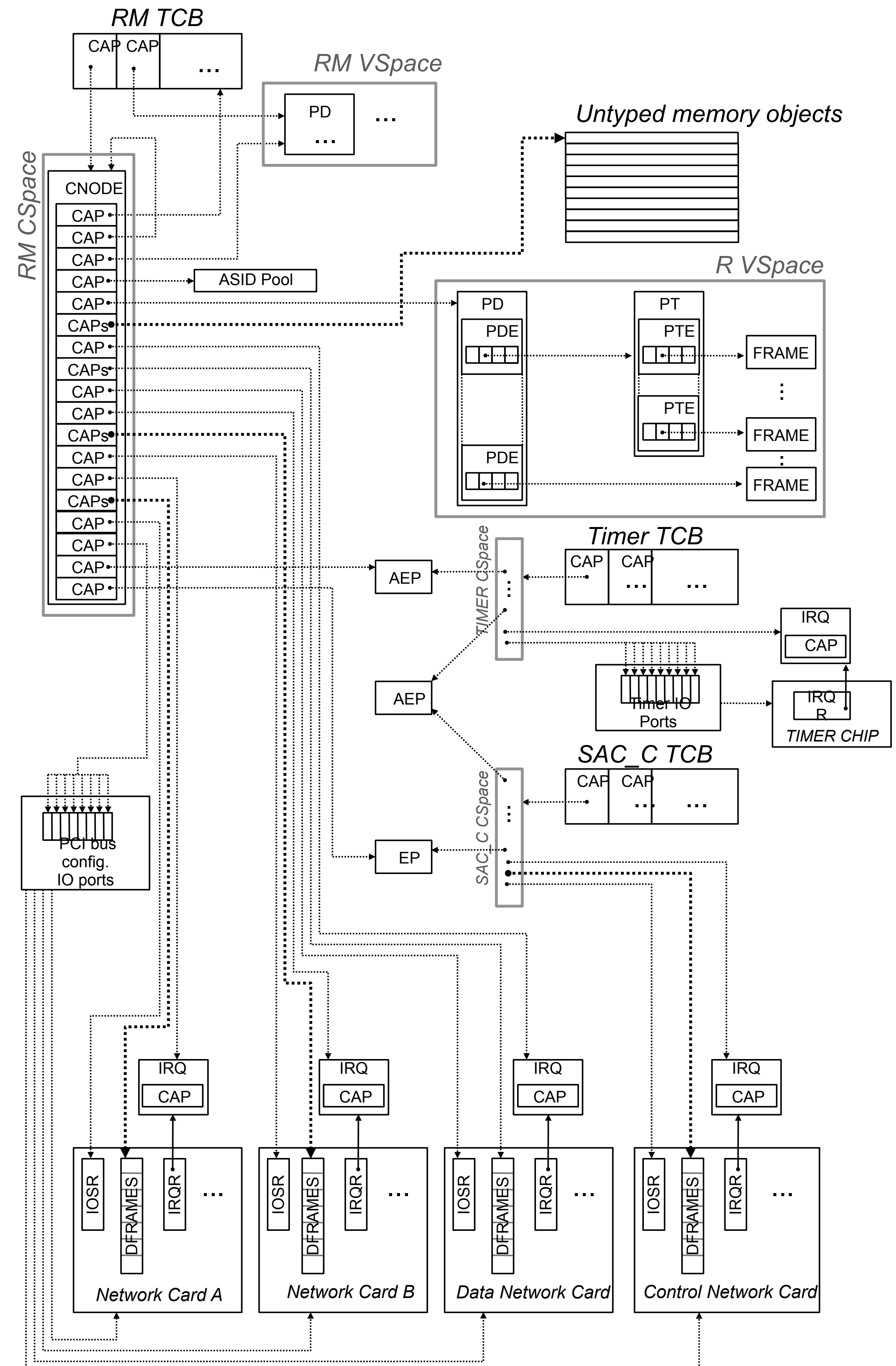
- **Even smaller TCB**
  – Router Manager (RM)
- (and)
  – kernel
  – booter
  – hardware

Router Manager:
< 2kloc
only trusted component



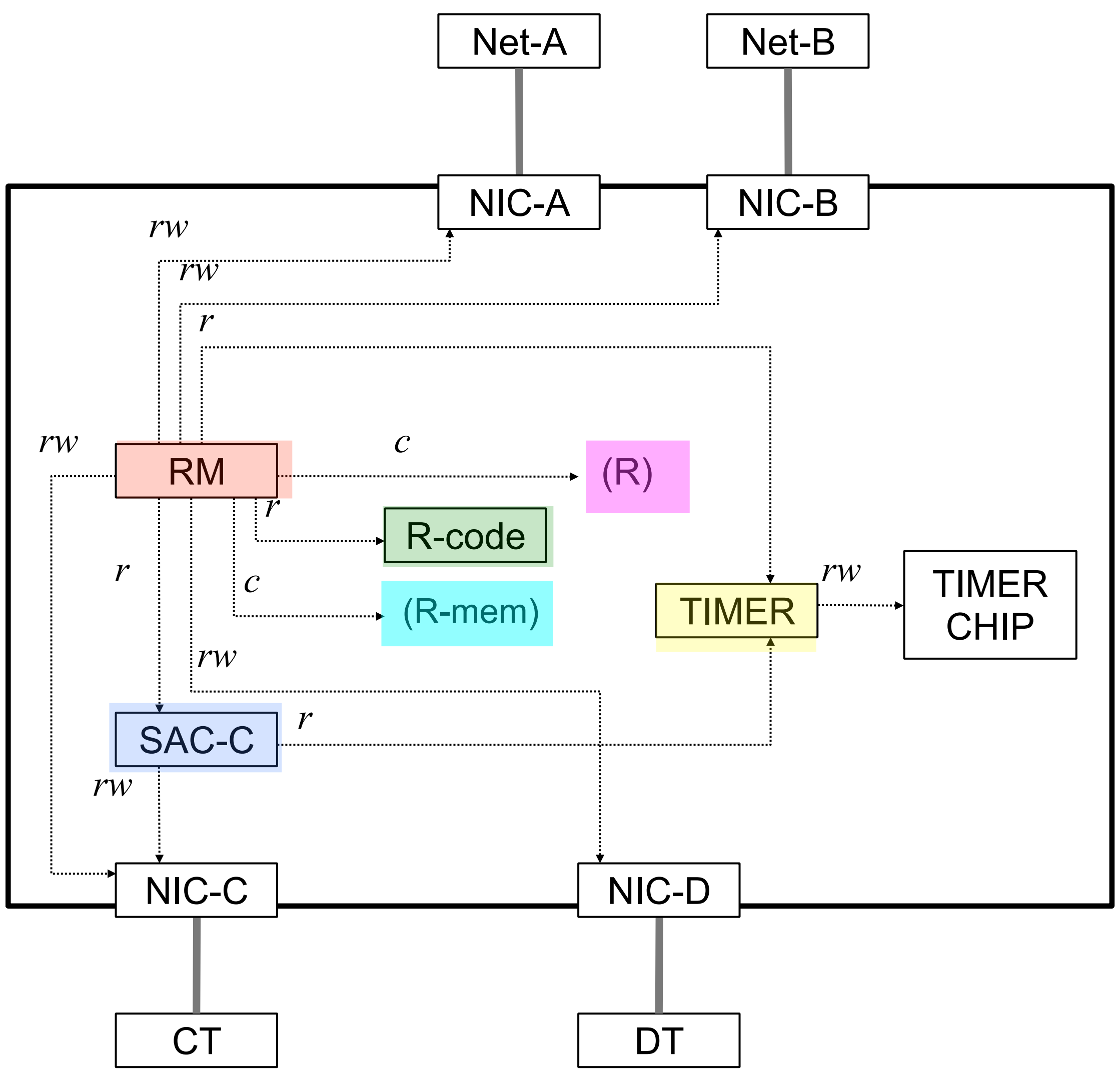
Net-A = Network A	NIC-C = Control Network Card	R= Router
Net-B = Network B	NIC-D = Data Network Card	RM = Router Manager
NIC-A = Network Card for Network A	CT = Control Terminal	SAC-C = SAC Controller
NIC-B = Network Card for Network B	DT = Data Terminal	

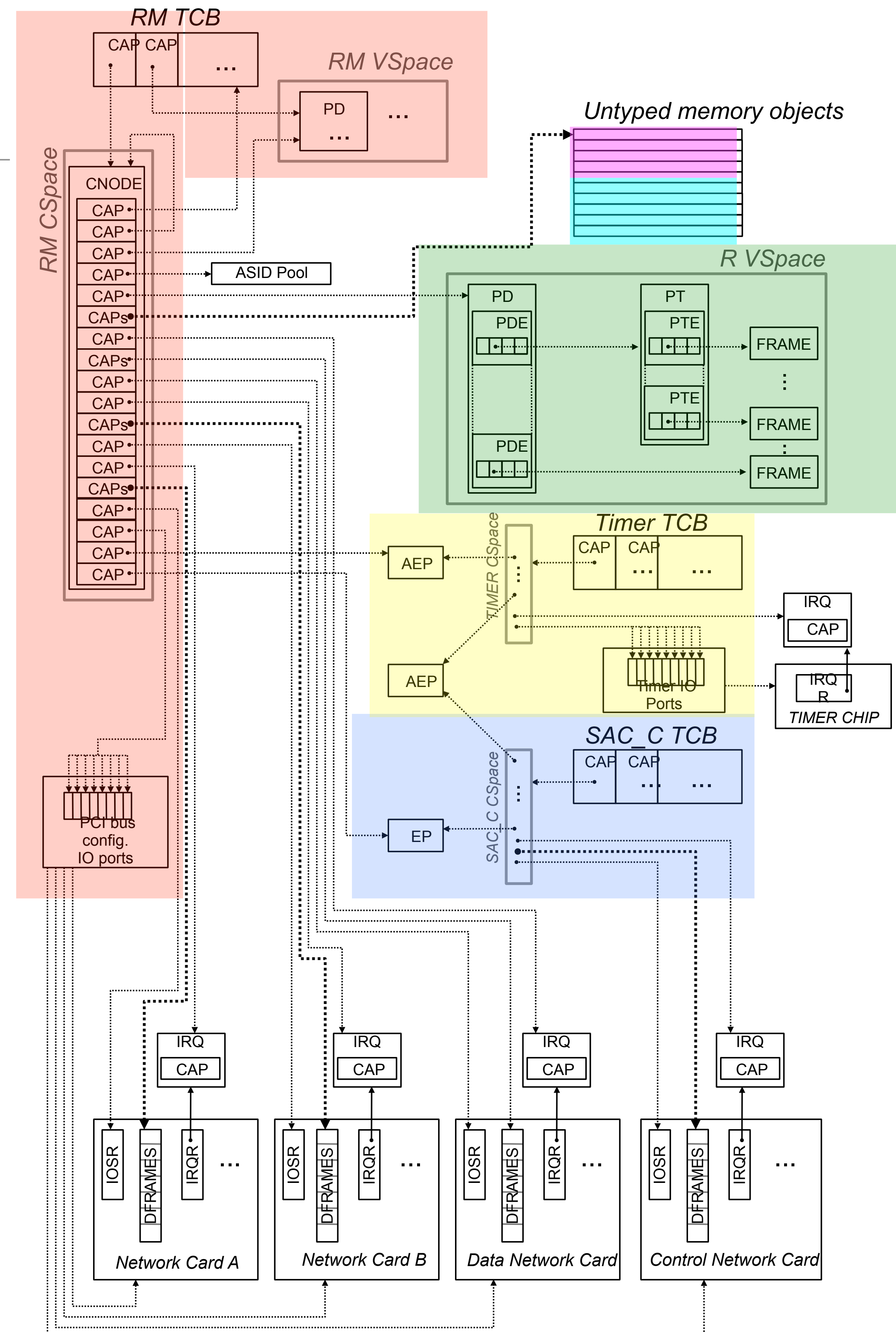
Low-Level Design



IOSR = IOSpace Root Pointer
IRQR = IRQ register reference
DFRAMES = Device Frames

Abstraction

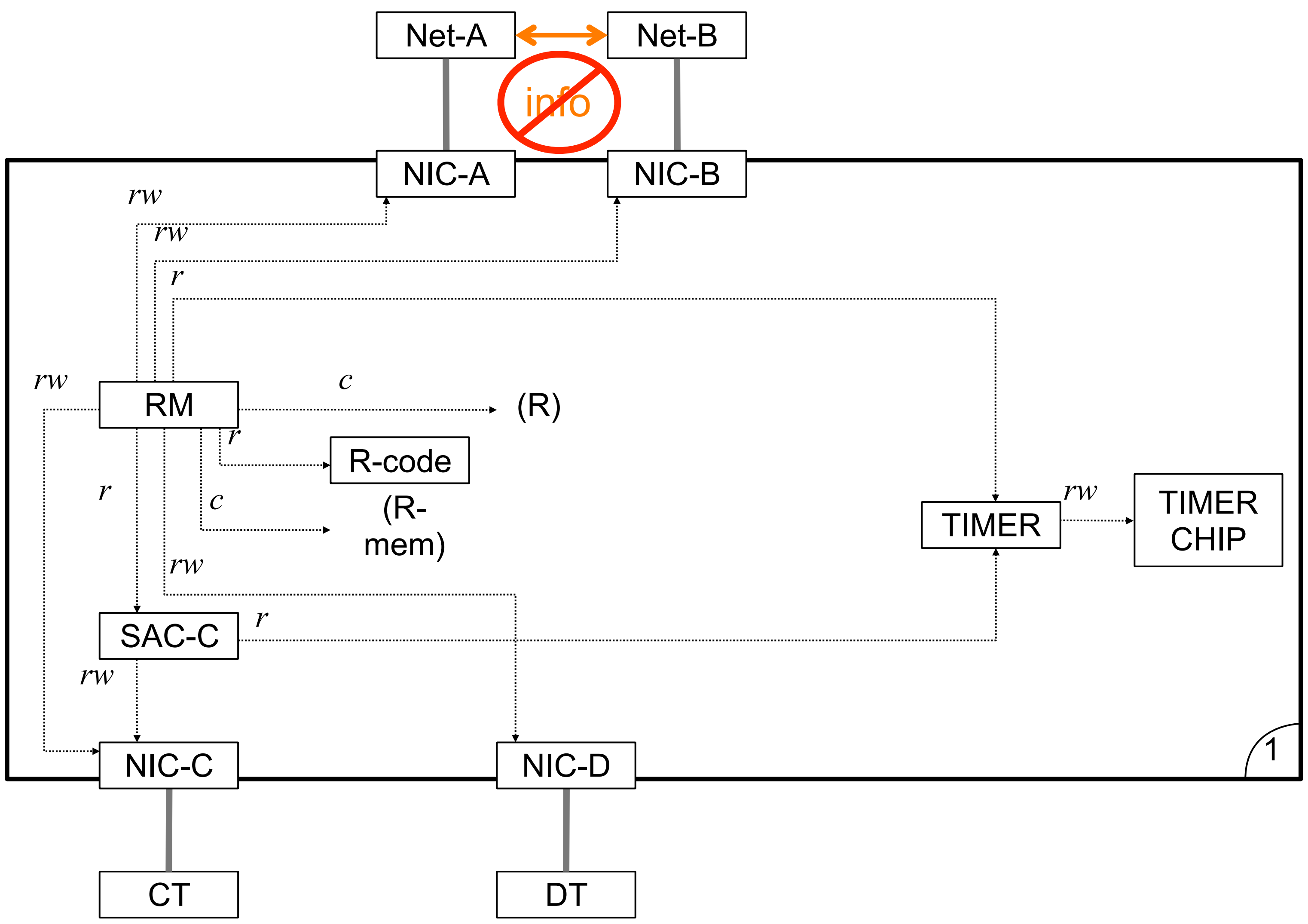




IOSR = IOSpace Root Pointer
IRQR = IRQ register reference
DFRAMES = Device Frames

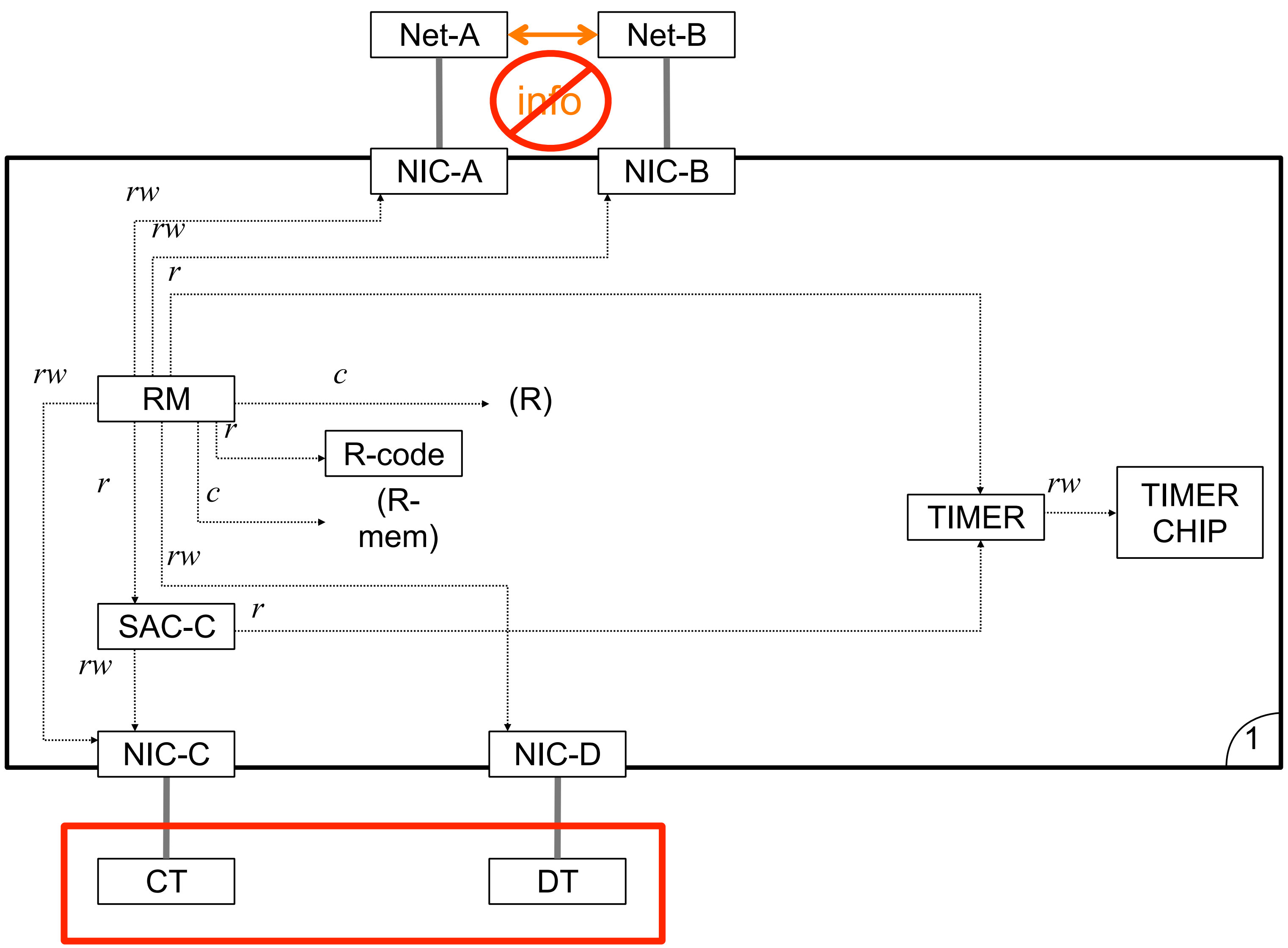
Security Goal

NICTA



Goal: No information flowing between providers A and B

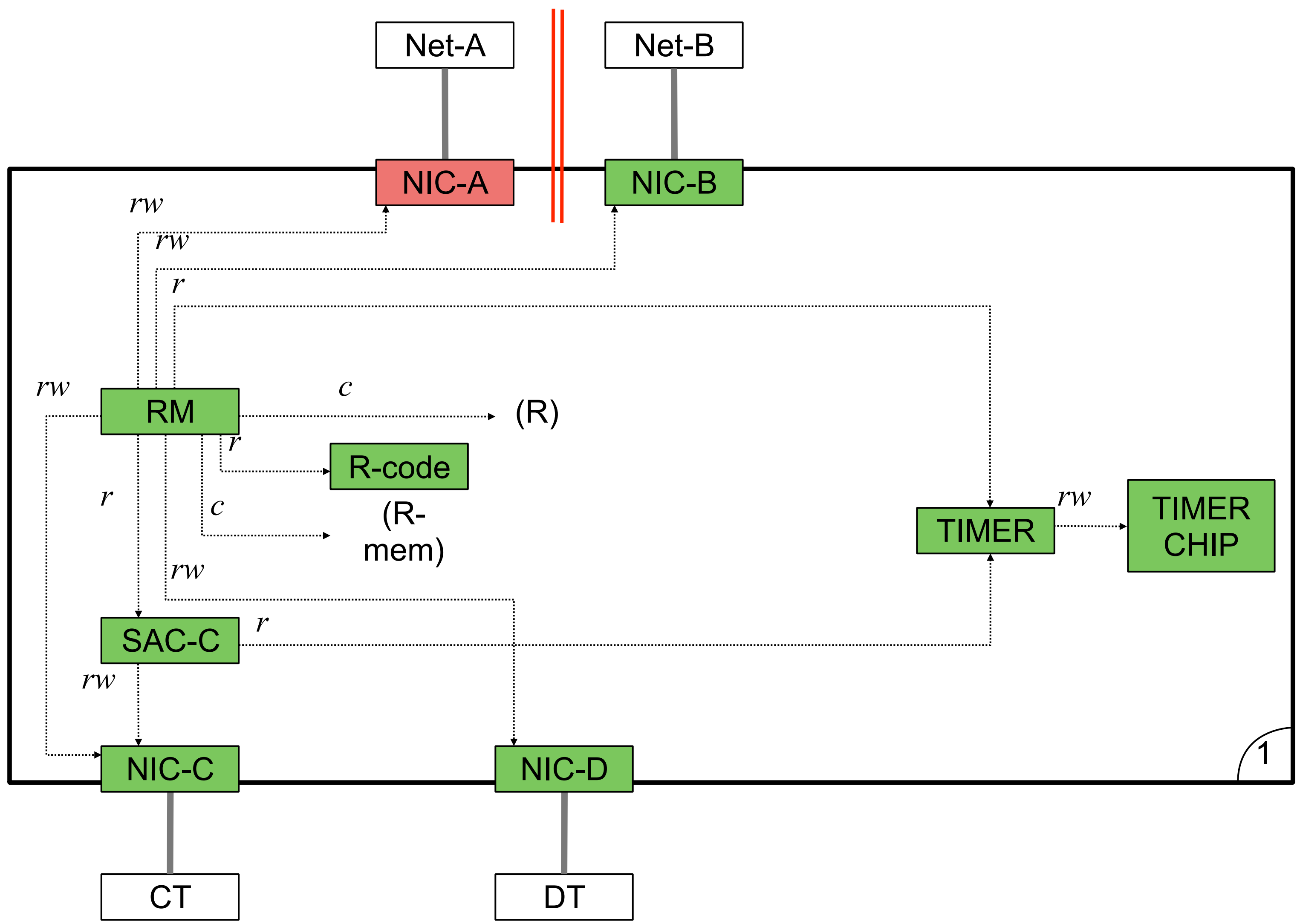
Security Goal



Goal: No information flowing between providers A and B

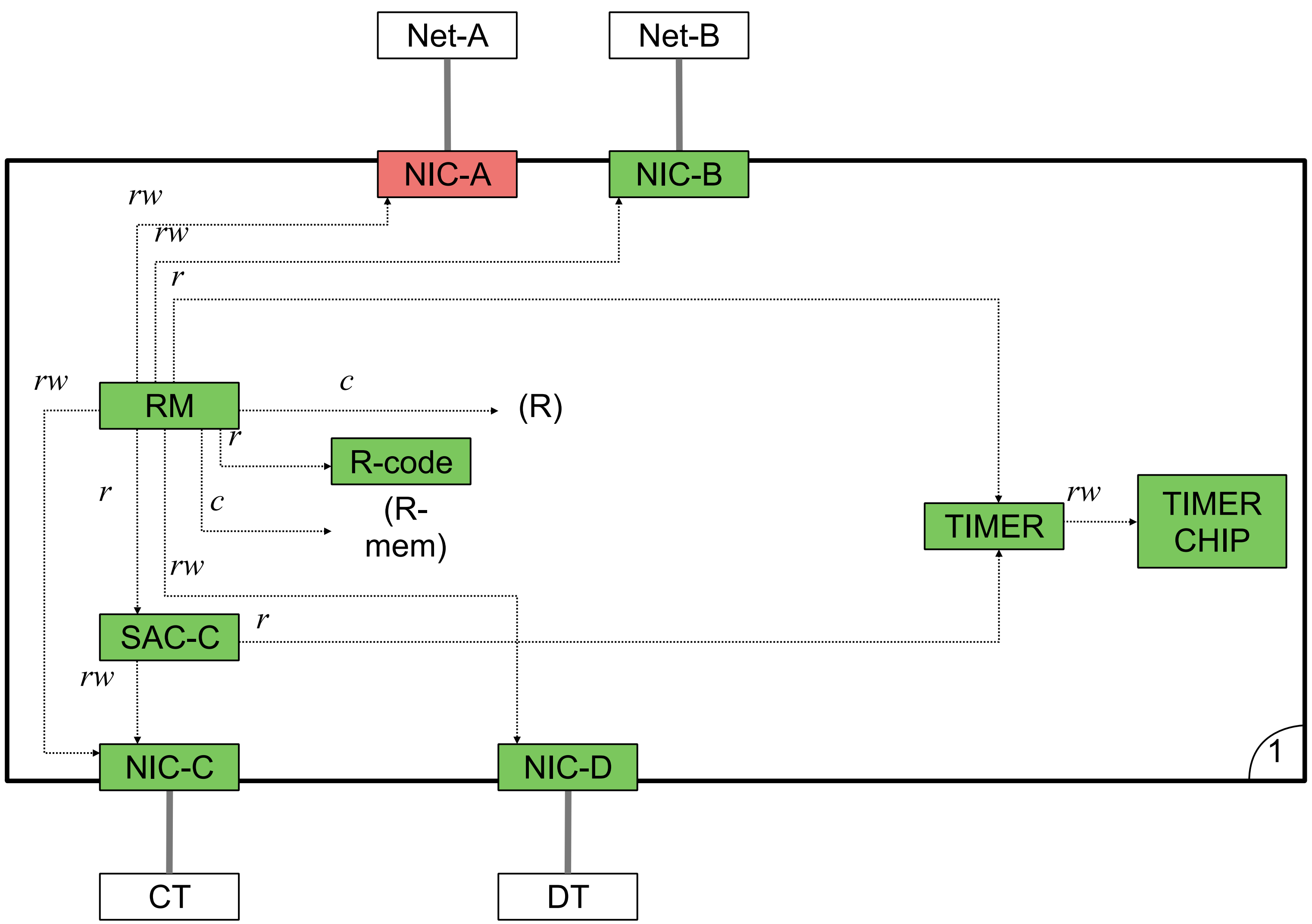
Assumption: Info flow through front-end terminal is trusted

Security Goal

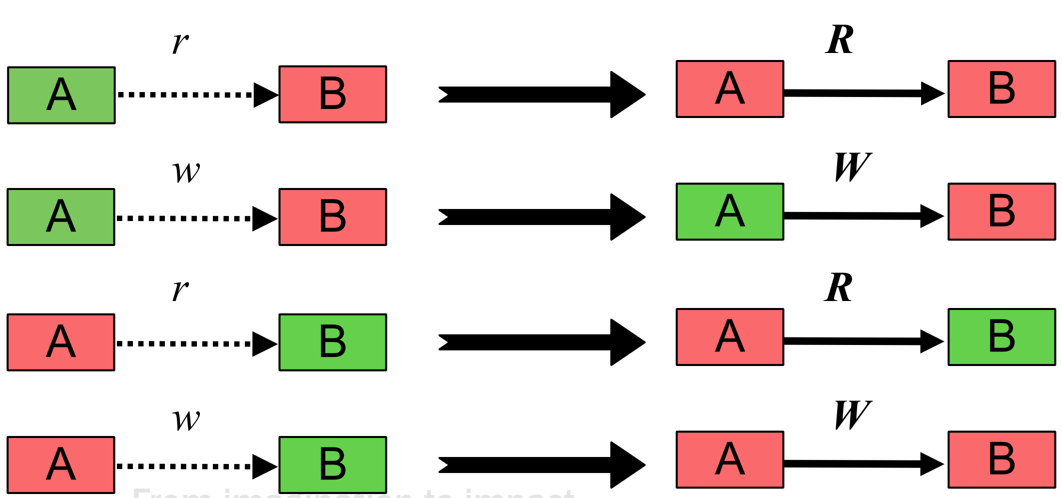


Approach:
- data from Net-A confidential; should not be read by Net-B
- label-based security:
  entities tagged 'contaminated' if may contain data from Net-A
  NIC-A always contaminated
- Goal: prove NIC-B never contaminated (always 'not contaminated')

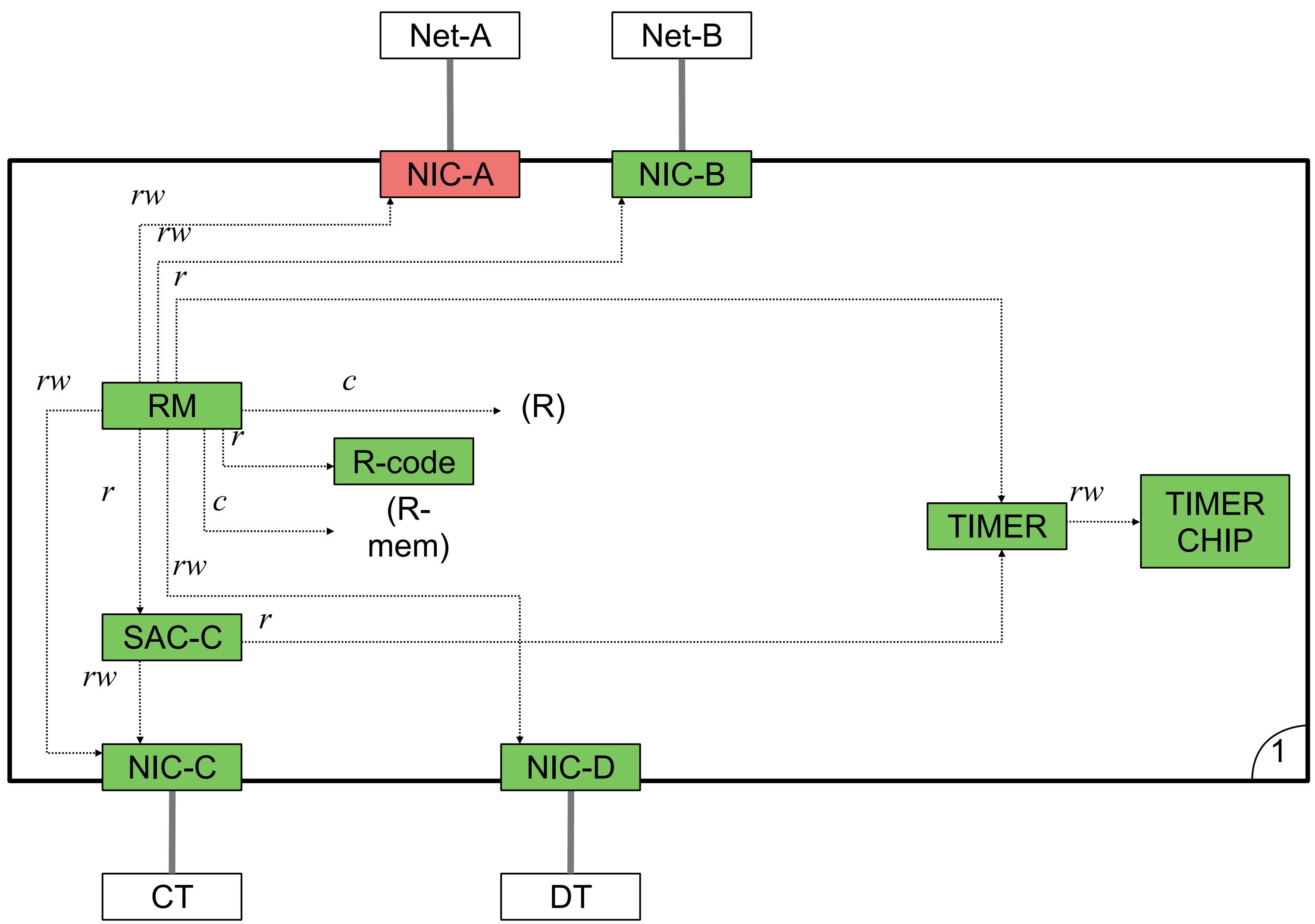
Security Analysis

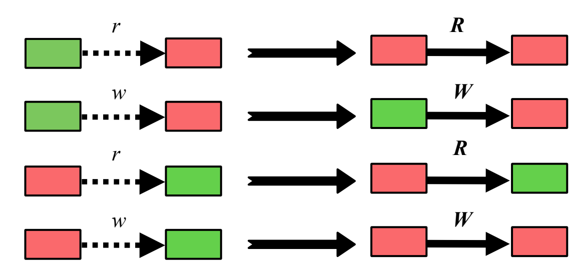


Rules:



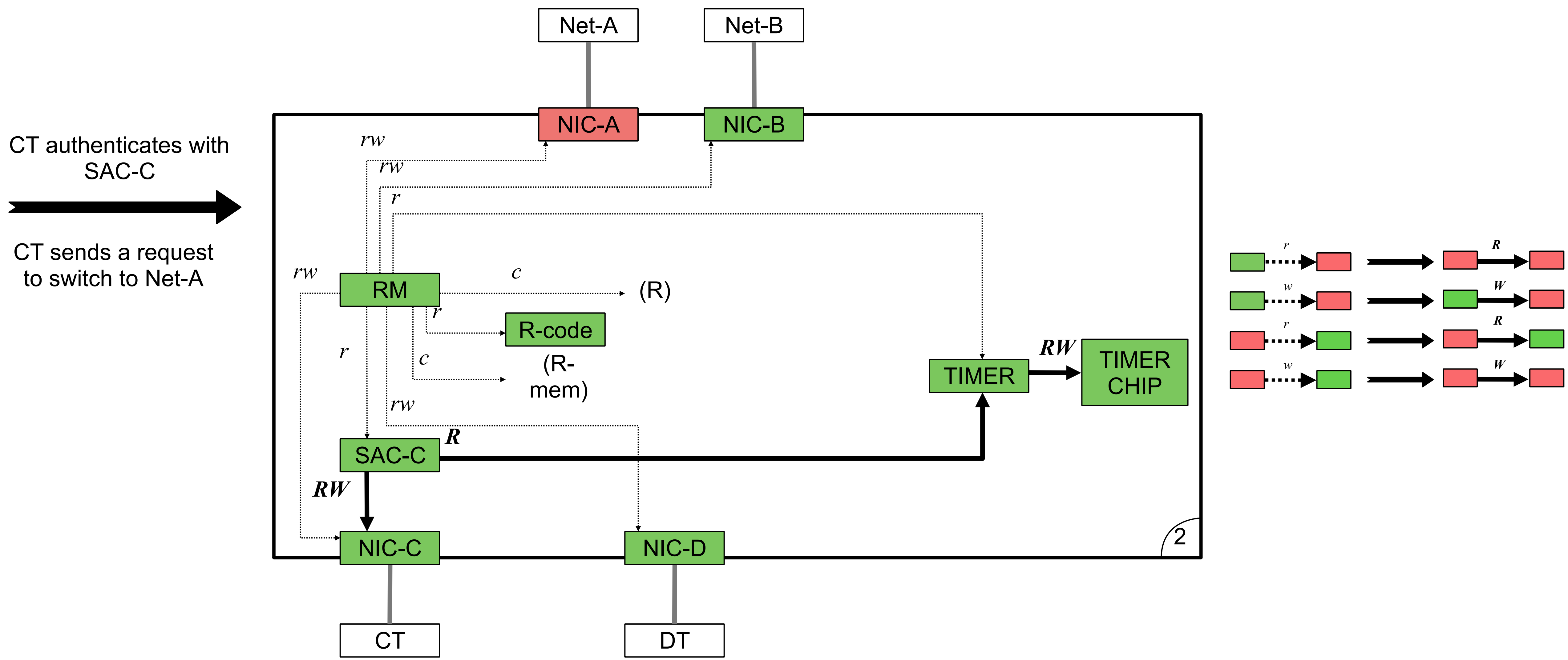
Life Cycle






```
RM_id         -> Some ( {rw_to_NIC_A, rw_to_NIC_B, …}, not_contaminated )
SAC_C_id      -> Some ( {rw_to_NIC_C, r_to_TIMER}, not_contaminated )
TIMER_id      -> Some ( {rw_to_TIMER_CHIP}, not_contaminated )
R_id          -> None
R_mem_id      -> None
NIC_A_id      -> Some ( {}, contaminated )
NIC_B_id      -> Some ( {}, not_contaminated )
NIC_C_id      -> Some ( {}, not_contaminated )
NIC_D_id      -> Some ( {}, not_contaminated )
R_code_id     -> Some ( {}, not_contaminated )
TIMER_CHIP_id -> Some ( {}, not_contaminated )
```

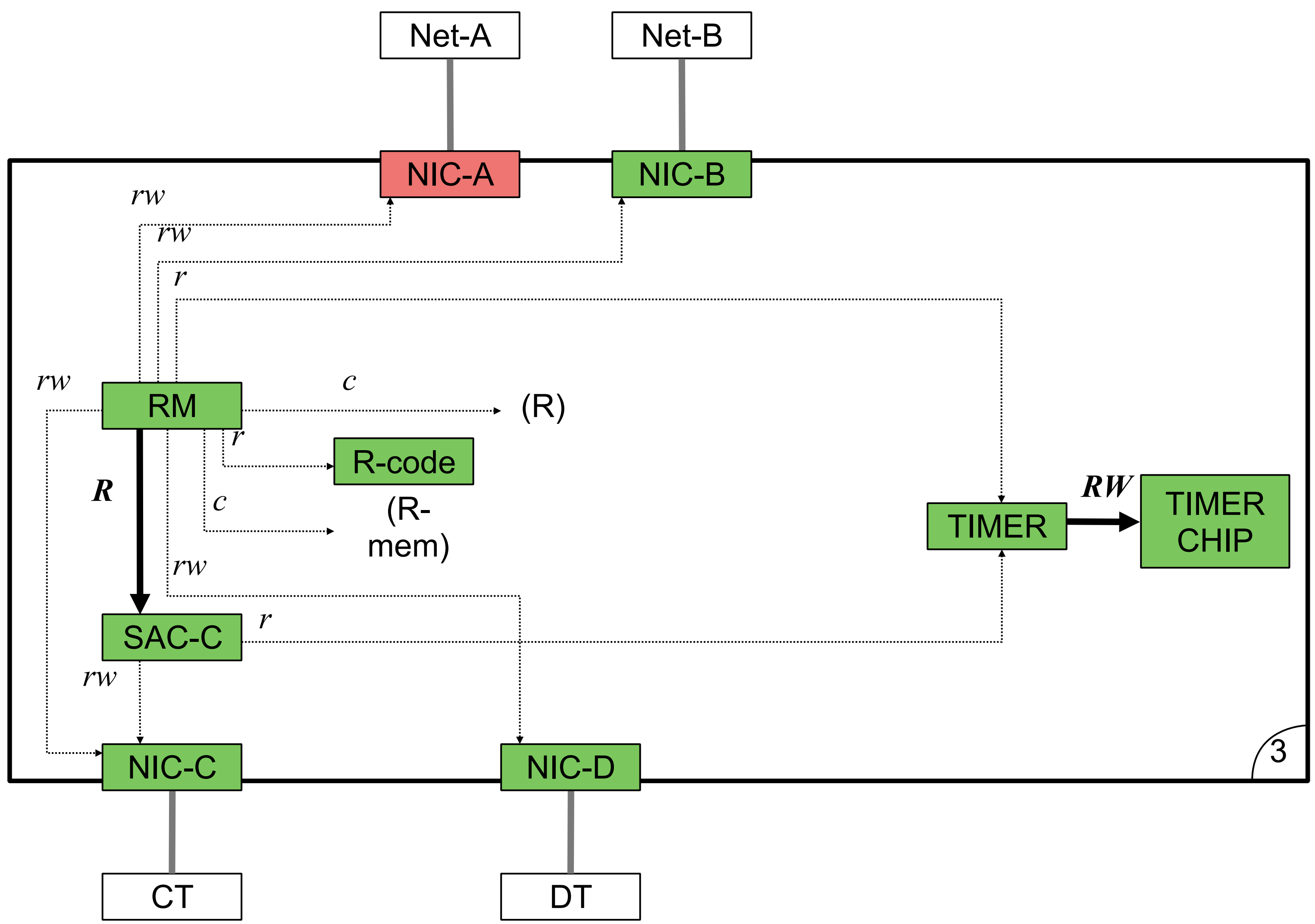
Life Cycle

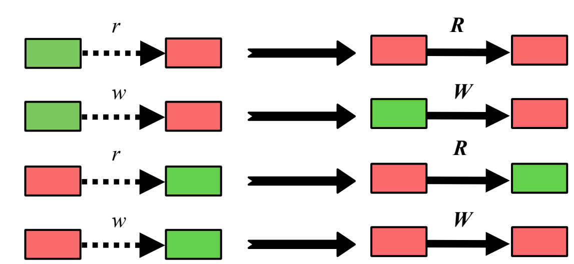



```
RM_id          -> Some ( {rw_to_NIC_A, rw_to_NIC_B, …}, not_contaminated )
SAC_C_id       -> Some ( {rw_to_NIC_C, r_to_TIMER}, not_contaminated )
TIMER_id       -> Some ( {rw_to_TIMER_CHIP}, not_contaminated )
R_id           -> None
R_mem_id       -> None
NIC_A_id       -> Some ( {}, contaminated )
NIC_B_id       -> Some ( {}, not_contaminated )
NIC_C_id       -> Some ( {}, not_contaminated )
NIC_D_id       -> Some ( {}, not_contaminated )
R_code_id      -> Some ( {}, not_contaminated )
TIMER_CHIP_id  -> Some ( {}, not_contaminated )
```

Life Cycle

RM receives request to switch to Net-A



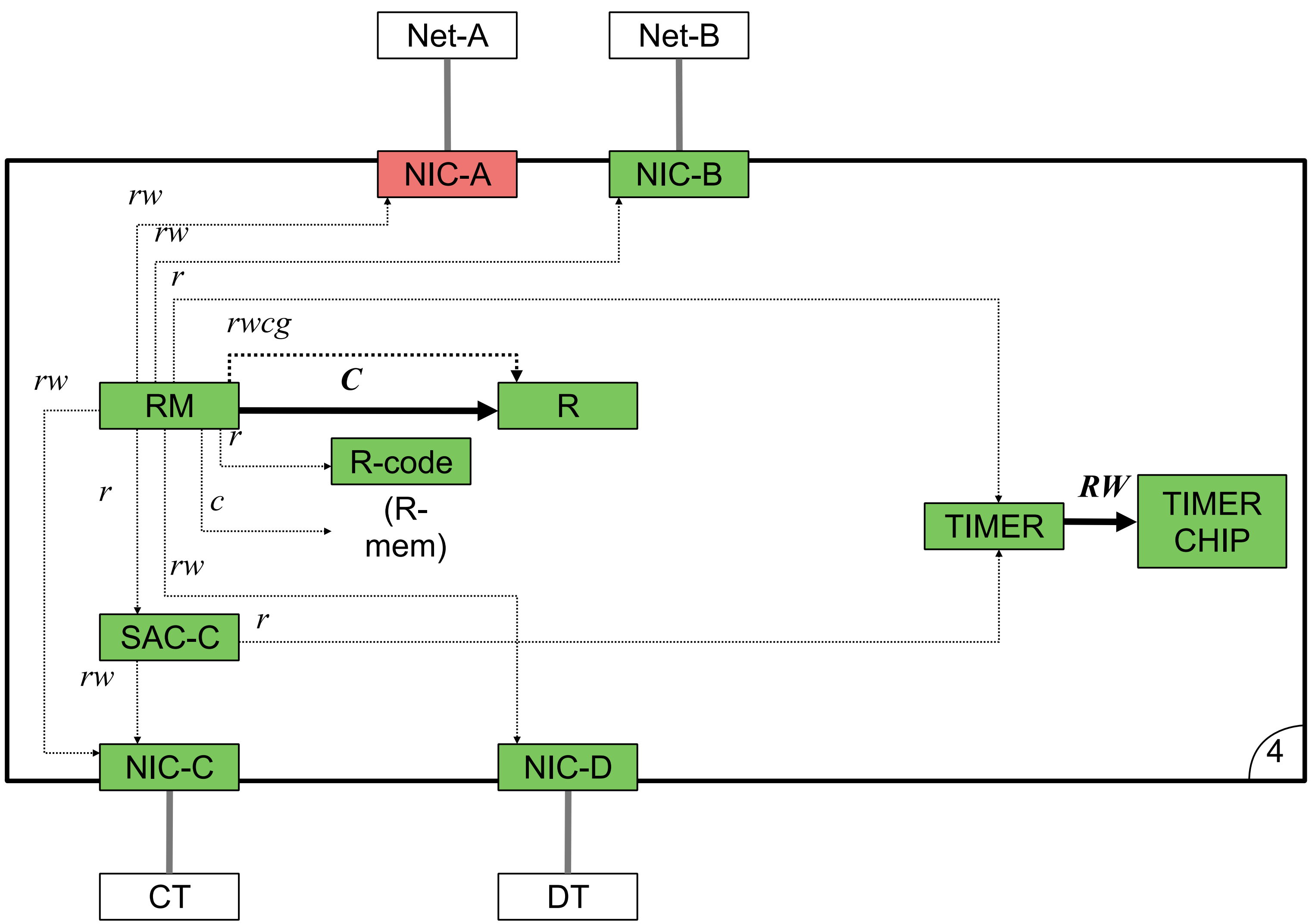


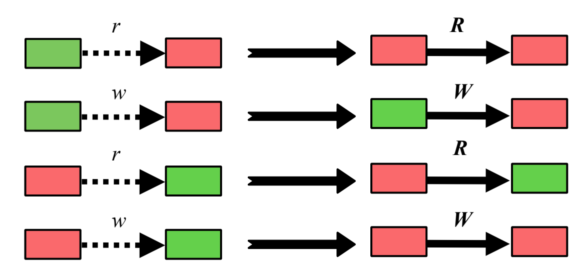

```
RM_id          -> Some ( {rw_to_NIC_A, rw_to_NIC_B, …}, not_contaminated )
SAC_C_id      -> Some ( {rw_to_NIC_C, r_to_TIMER}, not_contaminated )
TIMER_id       -> Some ( {rw_to_TIMER_CHIP}, not_contaminated )
R_id           -> None
R_mem_id       -> None
NIC_A_id      -> Some ( {}, contaminated )
NIC_B_id       -> Some ( {}, not_contaminated )
NIC_C_id       -> Some ( {}, not_contaminated )
NIC_D_id       -> Some ( {}, not_contaminated )
R_code_id     -> Some ( {}, not_contaminated )
TIMER_CHIP_id -> Some ( {}, not_contaminated )
```

Life Cycle

RM creates a new Router instance R

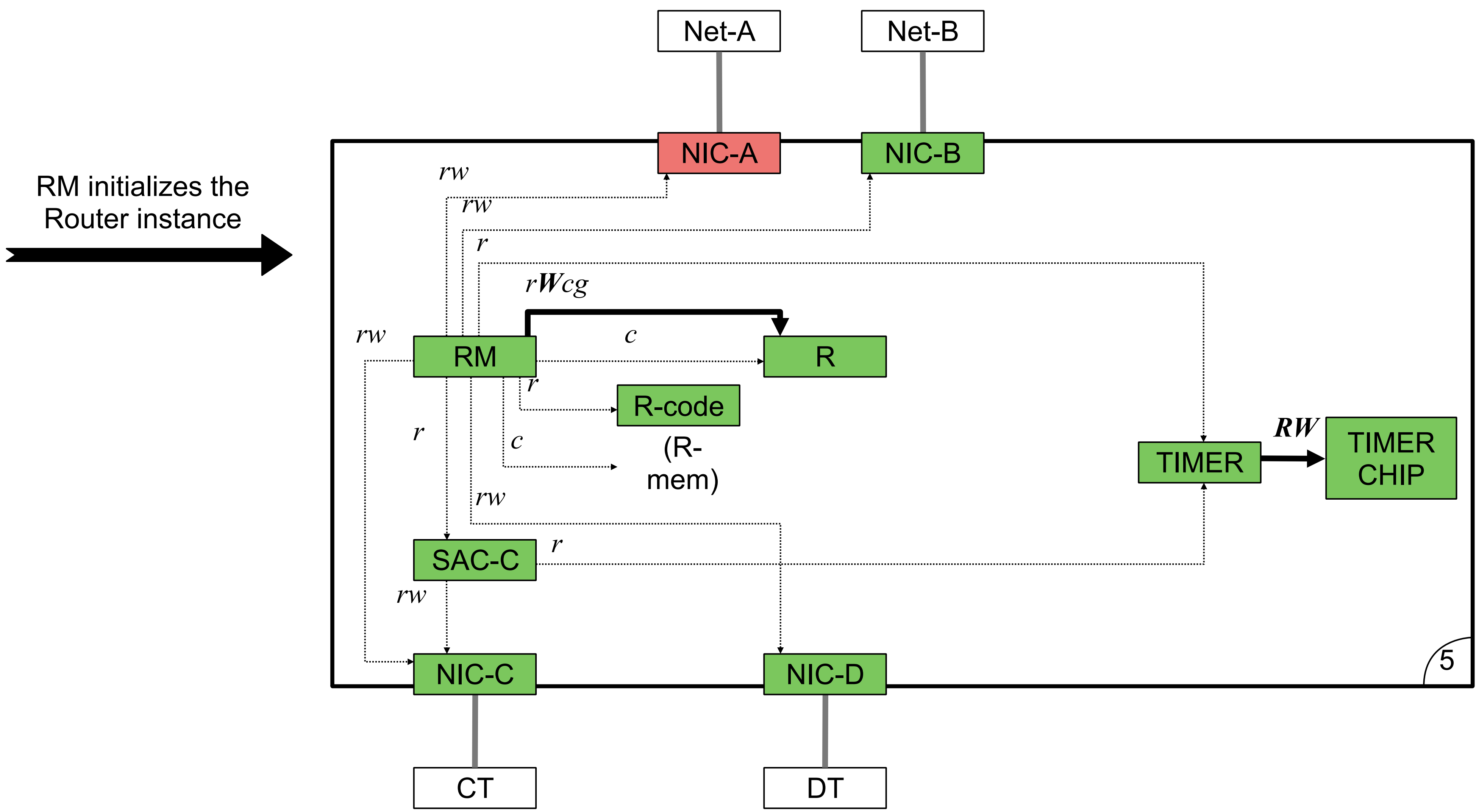
(and gets full rights to the newly created object)

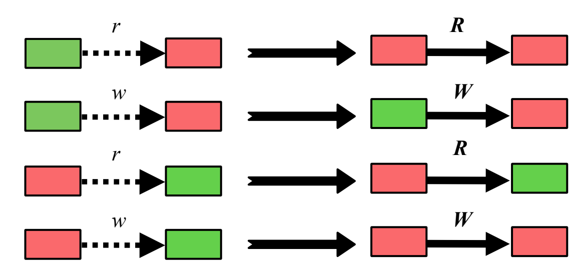





```
RM_id          -> Some ( {rw_to_NIC_A, rw_to_NIC_B, …, rwcg_to_R}, not_contaminated )
SAC_C_id       -> Some ( {rw_to_NIC_C, r_to_TIMER}, not_contaminated )
TIMER_id       -> Some ( {rw_to_TIMER_CHIP}, not_contaminated )
R_id           -> Some ( {}, not_contaminated  )
R_mem_id       -> None
NIC_A_id       -> Some ( {}, contaminated )
NIC_B_id       -> Some ( {}, not_contaminated )
NIC_C_id       -> Some ( {}, not_contaminated )
NIC_D_id       -> Some ( {}, not_contaminated )
R_code_id      -> Some ( {}, not_contaminated )
TIMER_CHIP_id  -> Some ( {}, not_contaminated )
```

Life Cycle



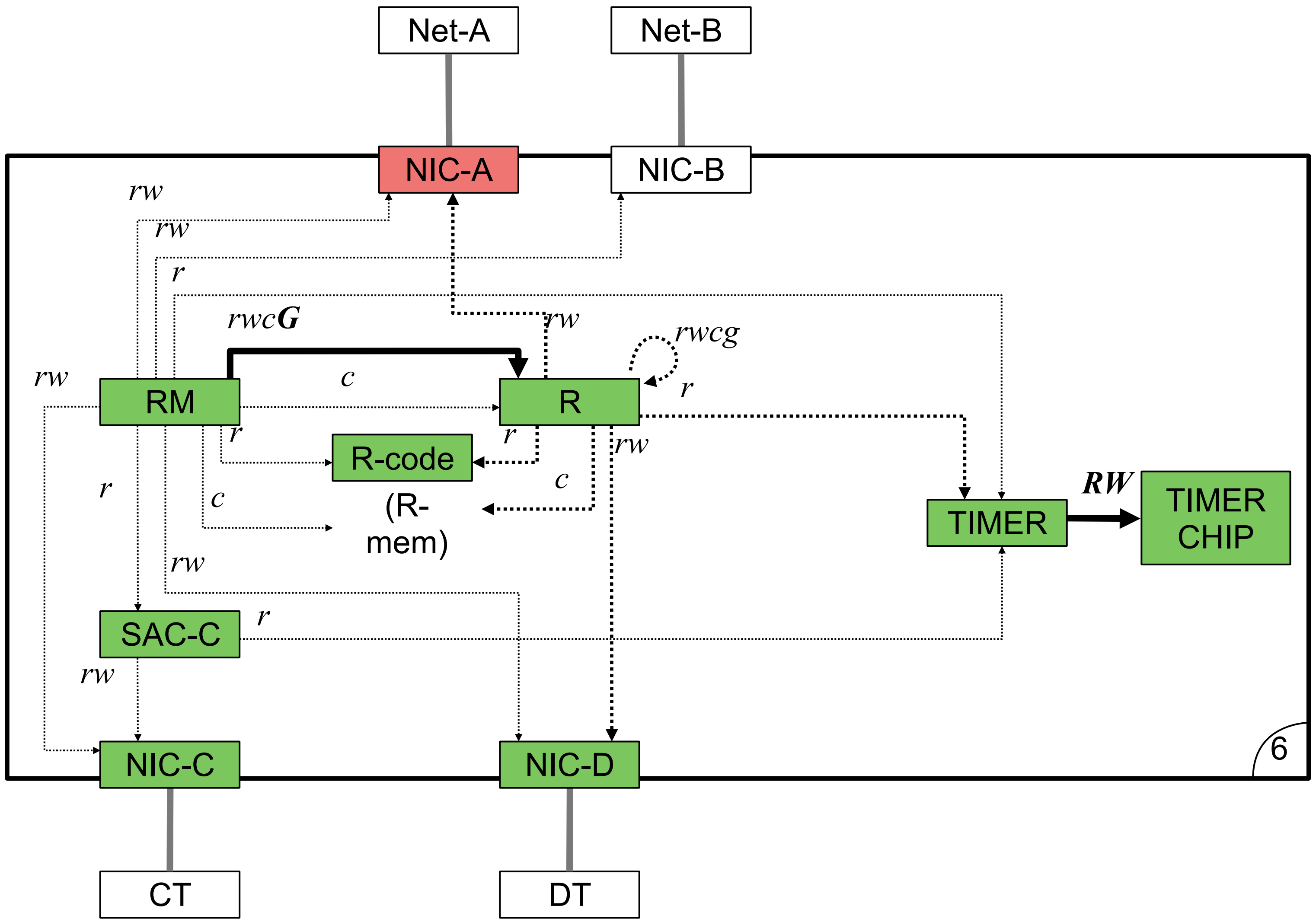


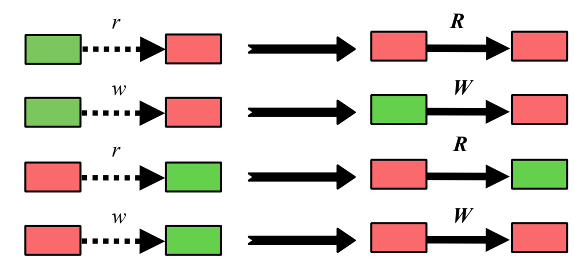
```
RM_id         -> Some ( {rw_to_NIC_A, rw_to_NIC_B, …, rwcg_to_R}, not_contaminated )
SAC_C_id      -> Some ( {rw_to_NIC_C, r_to_TIMER}, not_contaminated )
TIMER_id      -> Some ( {rw_to_TIMER_CHIP}, not_contaminated )
R_id          -> Some ( {}, not_contaminated  )
R_mem_id      -> None
NIC_A_id      -> Some ( {}, contaminated )
NIC_B_id      -> Some ( {}, not_contaminated )
NIC_C_id      -> Some ( {}, not_contaminated )
NIC_D_id      -> Some ( {}, not_contaminated )
R_code_id     -> Some ( {}, not_contaminated )
TIMER_CHIP_id -> Some ( {}, not_contaminated )
```

Life Cycle

RM grants to R its rights to NIC-A, NIC-D,

R-mem, R-code TIMER, and itself



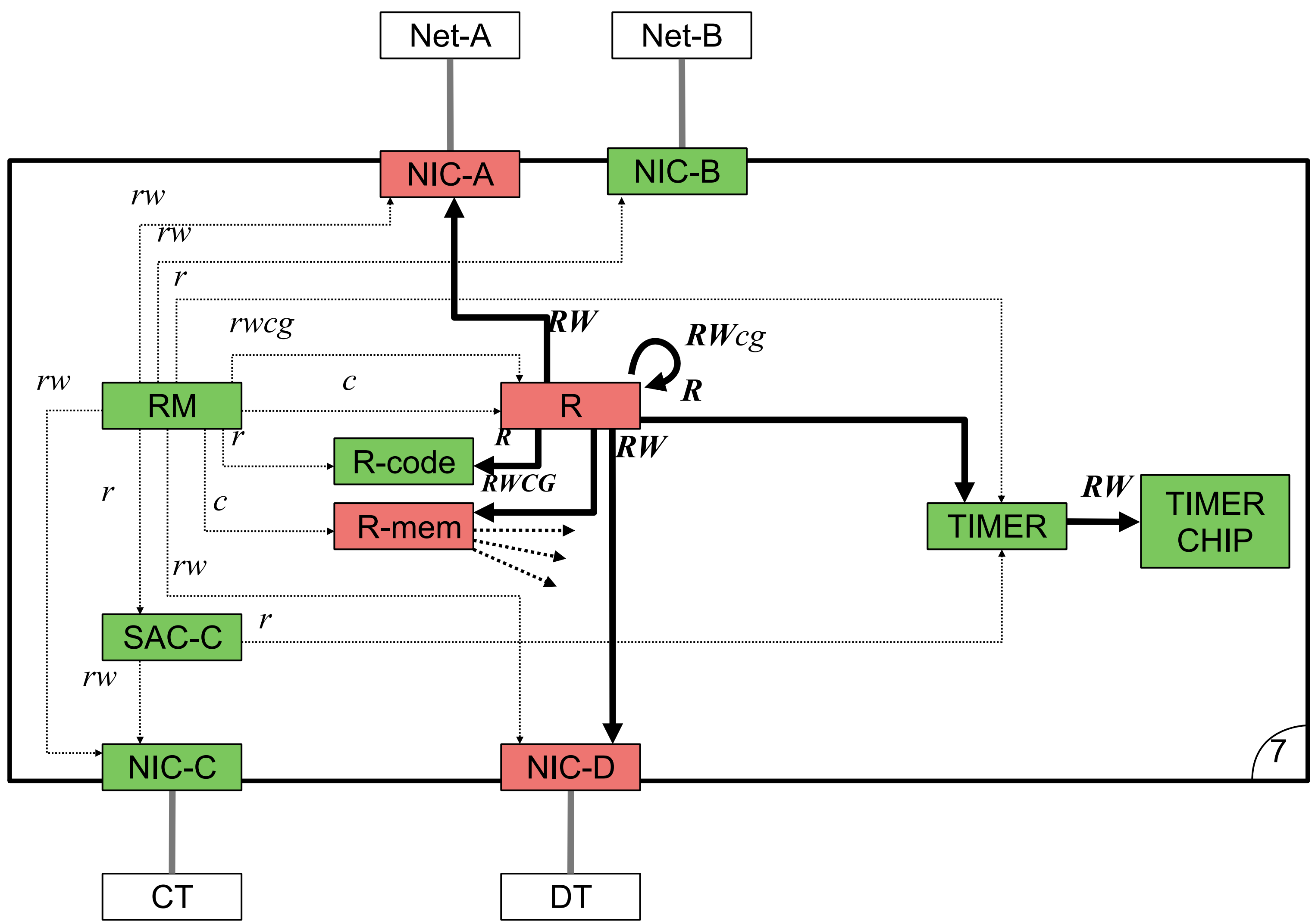


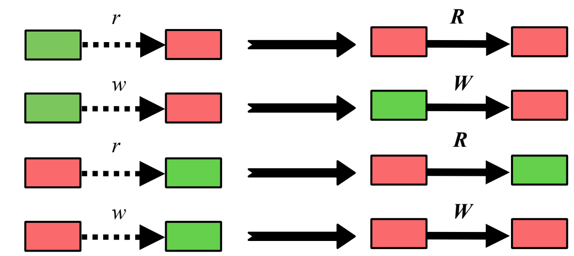

```
RM_id          -> Some ( {rw_to_NIC_A, rw_to_NIC_B, …, rwcg_to_R}, not_contaminated )
SAC_C_id       -> Some ( {rw_to_NIC_C, r_to_TIMER}, not_contaminated )
TIMER_id       -> Some ( {rw_to_TIMER_CHIP}, not_contaminated )
R_id           -> Some ( {rw_to_NIC_A, rw_to_NIC_B, …}, not_contaminated  )
R_mem_id       -> None
NIC_A_id       -> Some ( {}, contaminated )
NIC_B_id       -> Some ( {}, not_contaminated )
NIC_C_id       -> Some ( {}, not_contaminated )
NIC_D_id       -> Some ( {}, not_contaminated )
R_code_id      -> Some ( {}, not_contaminated )
TIMER_CHIP_id  -> Some ( {}, not_contaminated )
```

Life Cycle

DT starts to communicate with Net-A, through R.

This may imply the creation of new objects using R-mem and granting caps to them.





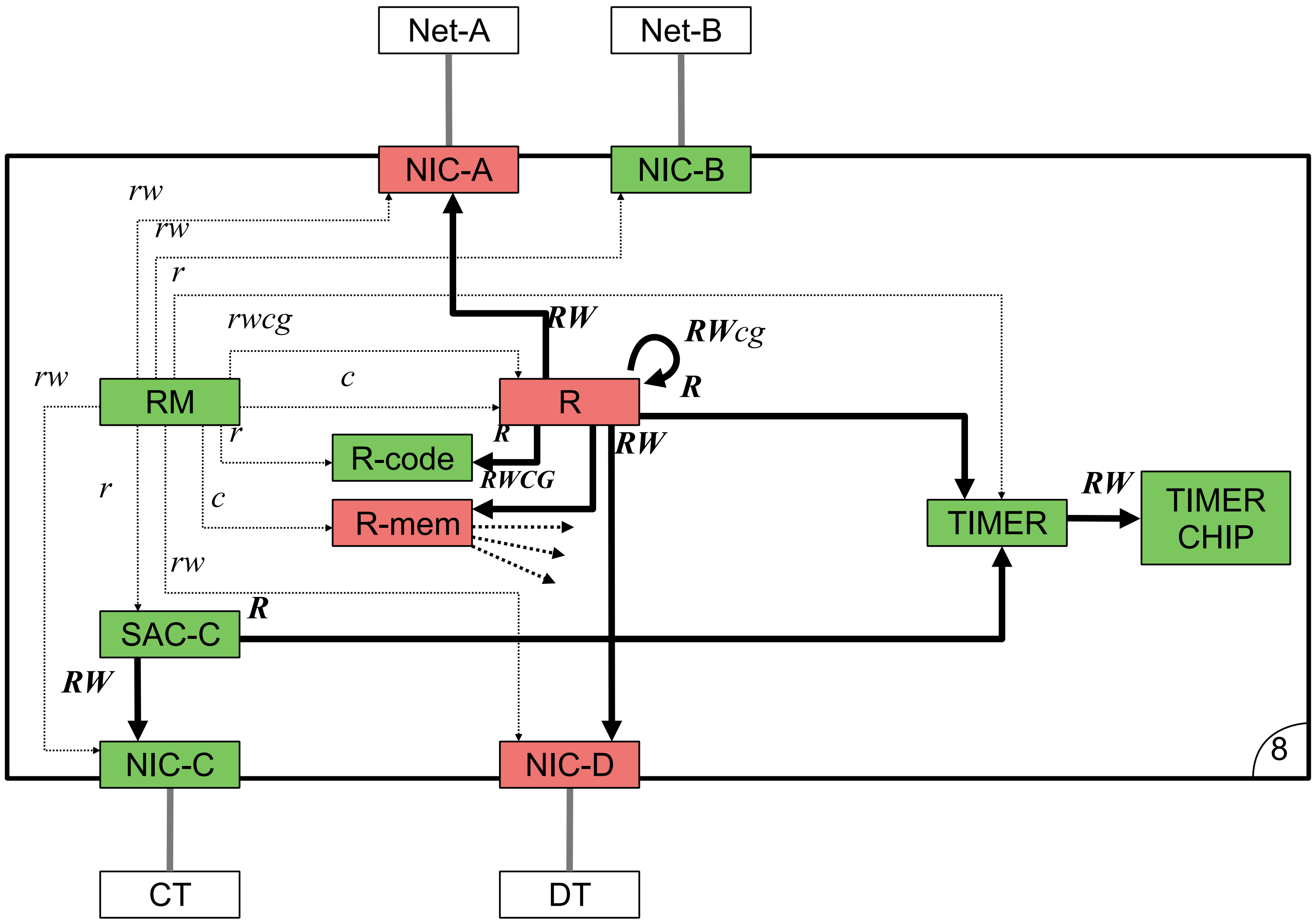

```
RM_id          -> Some ( {rw_to_NIC_A, rw_to_NIC_B, …, rwcg_to_R}, not_contaminated )
SAC_C_id       -> Some ( {rw_to_NIC_C, r_to_TIMER}, not_contaminated )
TIMER_id       -> Some ( {rw_to_TIMER_CHIP}, not_contaminated )
R_id           -> Some ( {rw_to_NIC_A, rw_to_NIC_B, …}, contaminated )
R_mem_id       -> Some ( {…}, contaminated  )
NIC_A_id       -> Some ( {}, contaminated )
NIC_B_id       -> Some ( {}, not_contaminated )
NIC_C_id       -> Some ( {}, not_contaminated )
NIC_D_id       -> Some ( {}, contaminated )
R_code_id      -> Some ( {}, not_contaminated )
TIMER_CHIP_id  -> Some ( {}, not_contaminated )
```

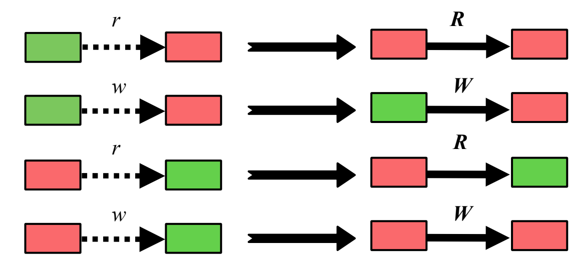
Life Cycle

NICTA

CT sends a request to switch to Net-B

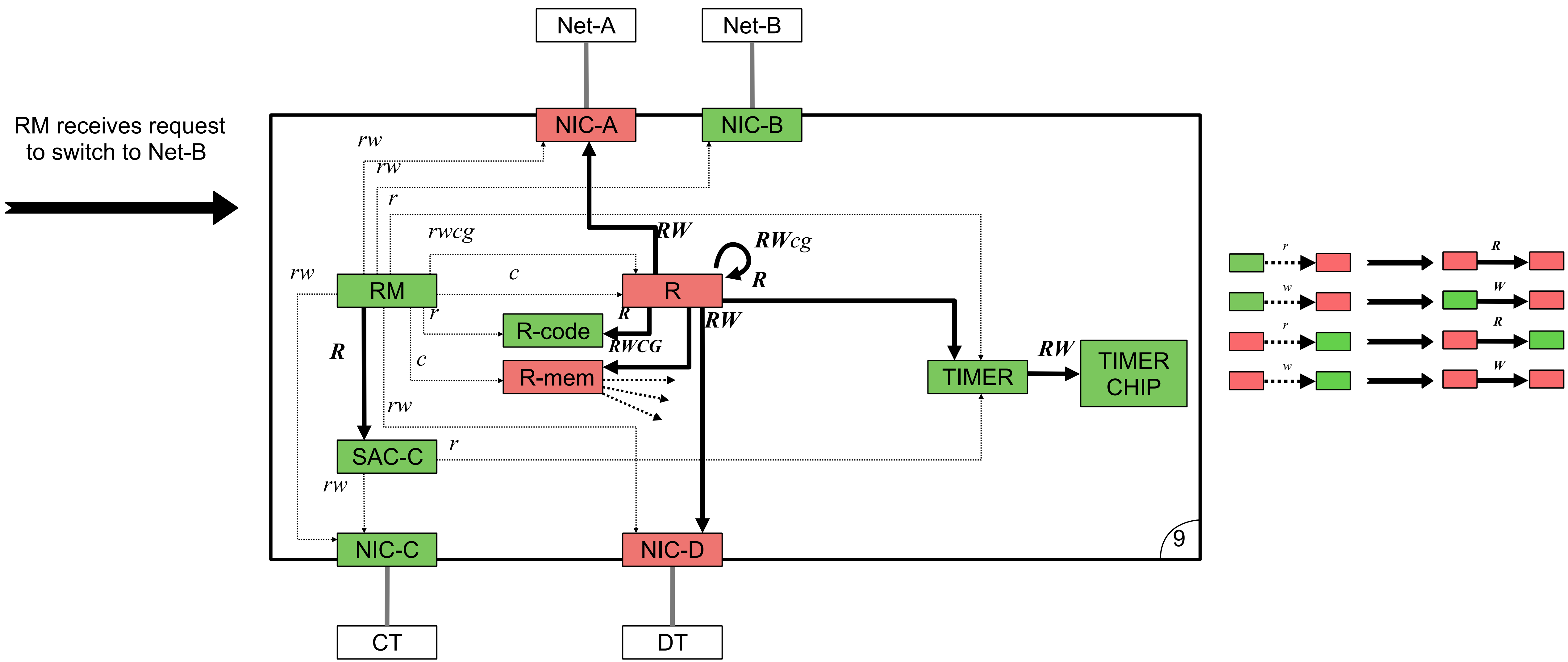
(while DT still communicates with Net-A through R)






```
RM_id          -> Some ( {rw_to_NIC_A, rw_to_NIC_B, …, rwcg_to_R}, not_contaminated )
SAC_C_id       -> Some ( {rw_to_NIC_C, r_to_TIMER}, not_contaminated )
TIMER_id       -> Some ( {rw_to_TIMER_CHIP}, not_contaminated )
R_id           -> Some ( {rw_to_NIC_A, rw_to_NIC_B, …}, contaminated )
R_mem_id       -> Some ( {…}, contaminated  )
NIC_A_id       -> Some ( {}, contaminated )
NIC_B_id       -> Some ( {}, not_contaminated )
NIC_C_id       -> Some ( {}, not_contaminated )
NIC_D_id       -> Some ( {}, contaminated )
R_code_id      -> Some ( {}, not_contaminated )
TIMER_CHIP_id  -> Some ( {}, not_contaminated )
```


Life Cycle

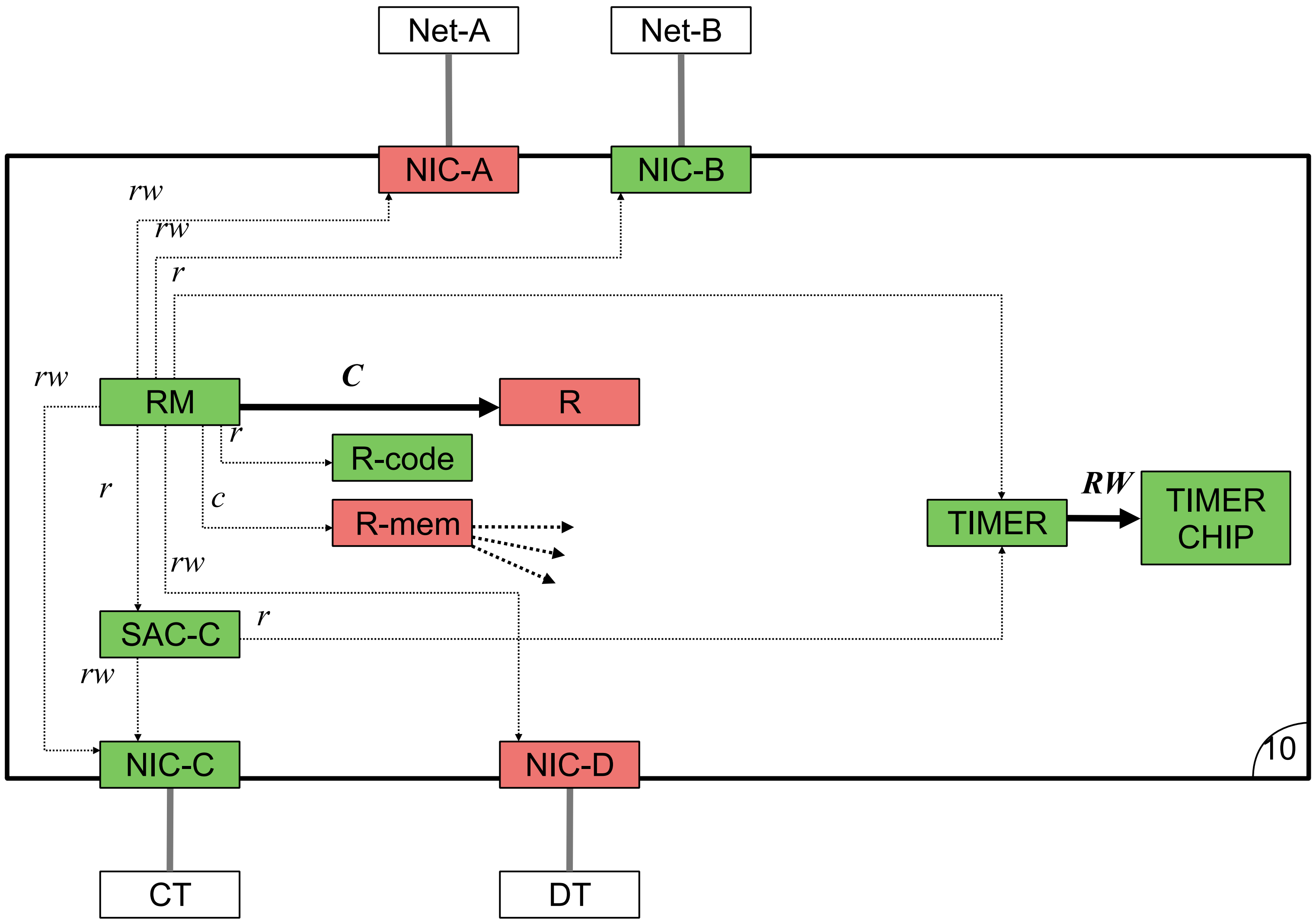


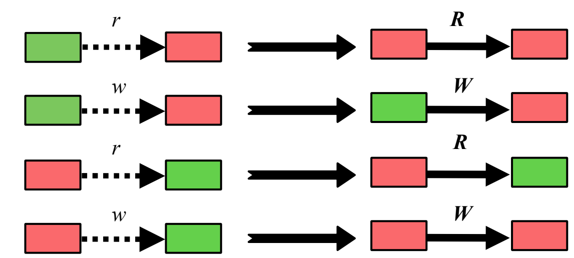

```
RM_id          -> Some ( {rw_to_NIC_A, rw_to_NIC_B, …, rwcg_to_R}, not_contaminated )
SAC_C_id       -> Some ( {rw_to_NIC_C, r_to_TIMER}, not_contaminated )
TIMER_id       -> Some ( {rw_to_TIMER_CHIP}, not_contaminated )
R_id           -> Some ( {rw_to_NIC_A, rw_to_NIC_B, …}, contaminated )
R_mem_id       -> Some ( {…}, contaminated  )
NIC_A_id       -> Some ( {}, contaminated )
NIC_B_id       -> Some ( {}, not_contaminated )
NIC_C_id       -> Some ( {}, not_contaminated )
NIC_D_id       -> Some ( {}, contaminated )
R_code_id      -> Some ( {}, not_contaminated )
TIMER_CHIP_id  -> Some ( {}, not_contaminated )
```

Life Cycle

RM revokes all caps given to R

(using the cap used to create R)



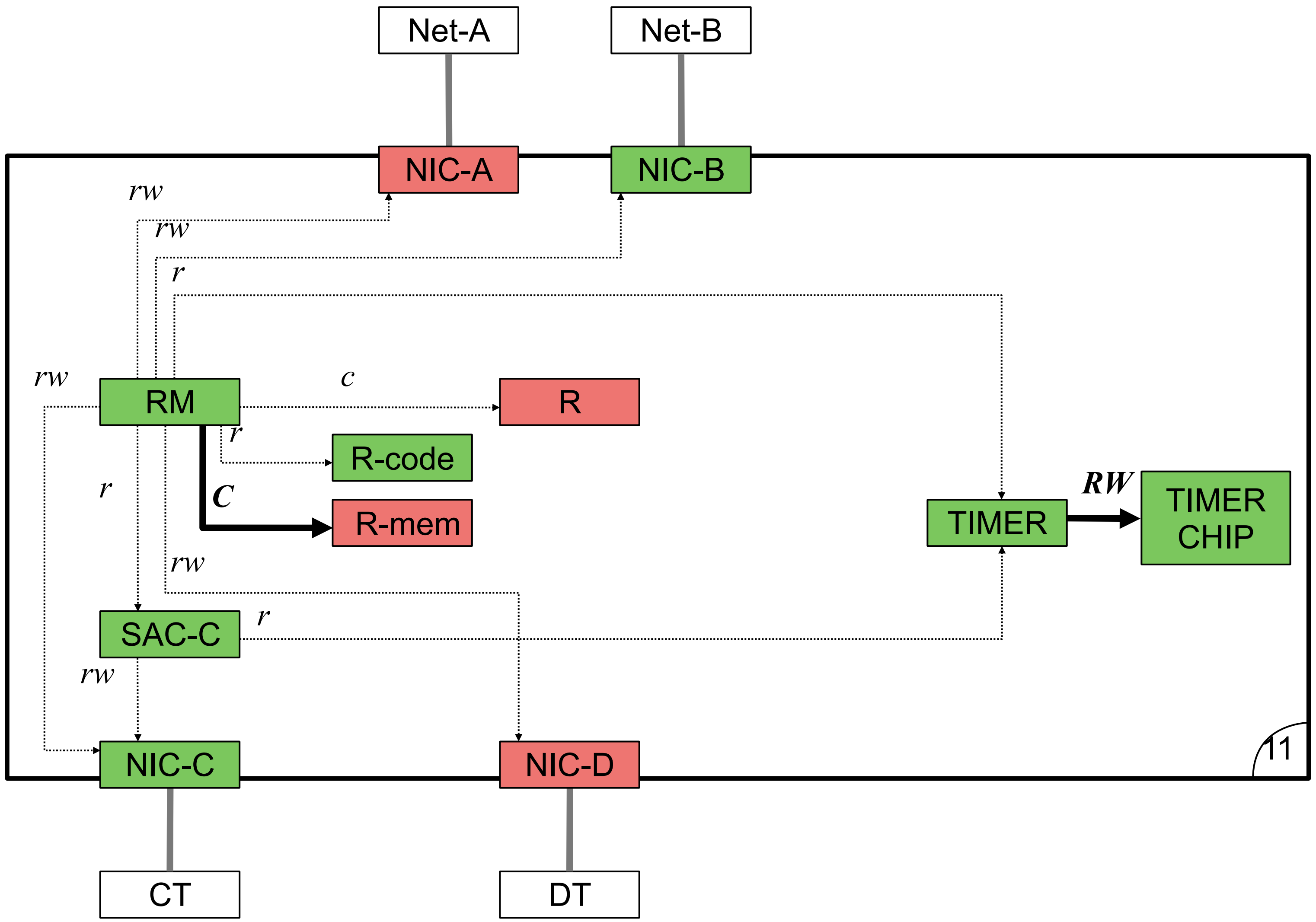


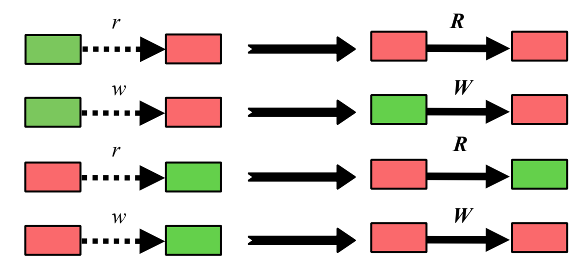

```
RM_id          -> Some ( {rw_to_NIC_A, rw_to_NIC_B, …, rwcg_to_R}, not_contaminated )
SAC_C_id       -> Some ( {rw_to_NIC_C, r_to_TIMER}, not_contaminated )
TIMER_id       -> Some ( {rw_to_TIMER_CHIP}, not_contaminated )
R_id           -> Some ( {}, contaminated )
R_mem_id       -> Some ( {…}, contaminated  )
NIC_A_id       -> Some ( {}, contaminated )
NIC_B_id       -> Some ( {}, not_contaminated )
NIC_C_id       -> Some ( {}, not_contaminated )
NIC_D_id       -> Some ( {}, contaminated )
R_code_id      -> Some ( {}, not_contaminated )
TIMER_CHIP_id  -> Some ( {}, not_contaminated )
```

Life Cycle

RM revokes caps of R-mem

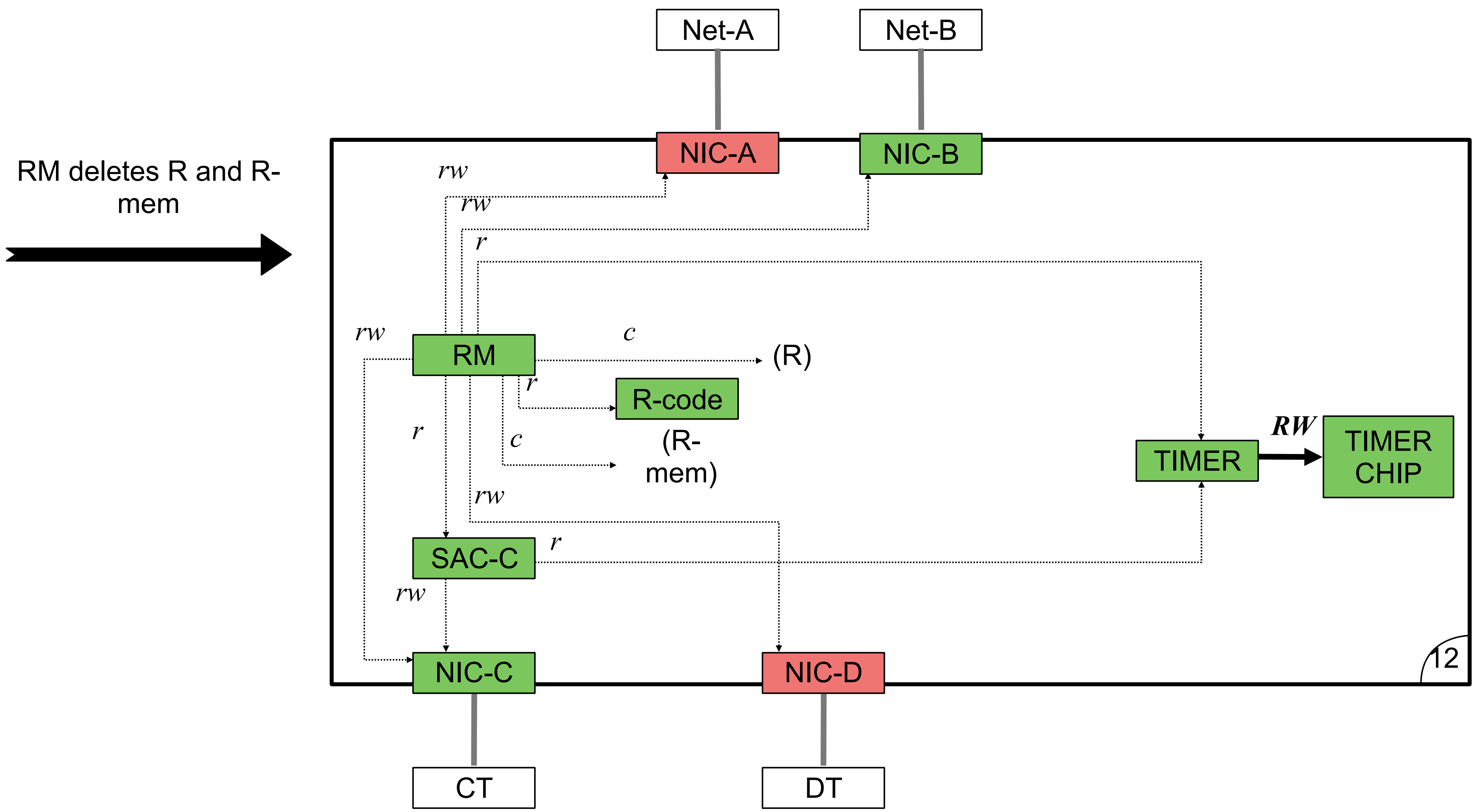
(using create cap to R-mem)

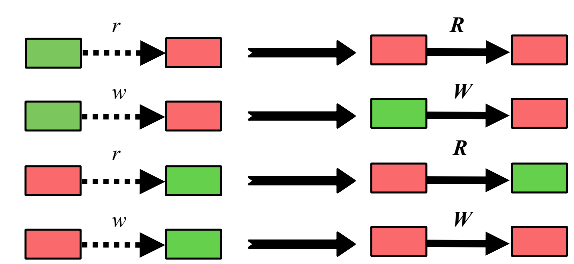





```
RM_id          -> Some ( {rw_to_NIC_A, rw_to_NIC_B, …, rwcg_to_R}, not_contaminated )
SAC_C_id       -> Some ( {rw_to_NIC_C, r_to_TIMER}, not_contaminated )
TIMER_id       -> Some ( {rw_to_TIMER_CHIP}, not_contaminated )
R_id           -> Some ( {}, contaminated )
R_mem_id       -> Some ( {}, contaminated  )
NIC_A_id       -> Some ( {}, contaminated )
NIC_B_id       -> Some ( {}, not_contaminated )
NIC_C_id       -> Some ( {}, not_contaminated )
NIC_D_id       -> Some ( {}, contaminated )
R_code_id      -> Some ( {}, not_contaminated )
TIMER_CHIP_id  -> Some ( {}, not_contaminated )
```

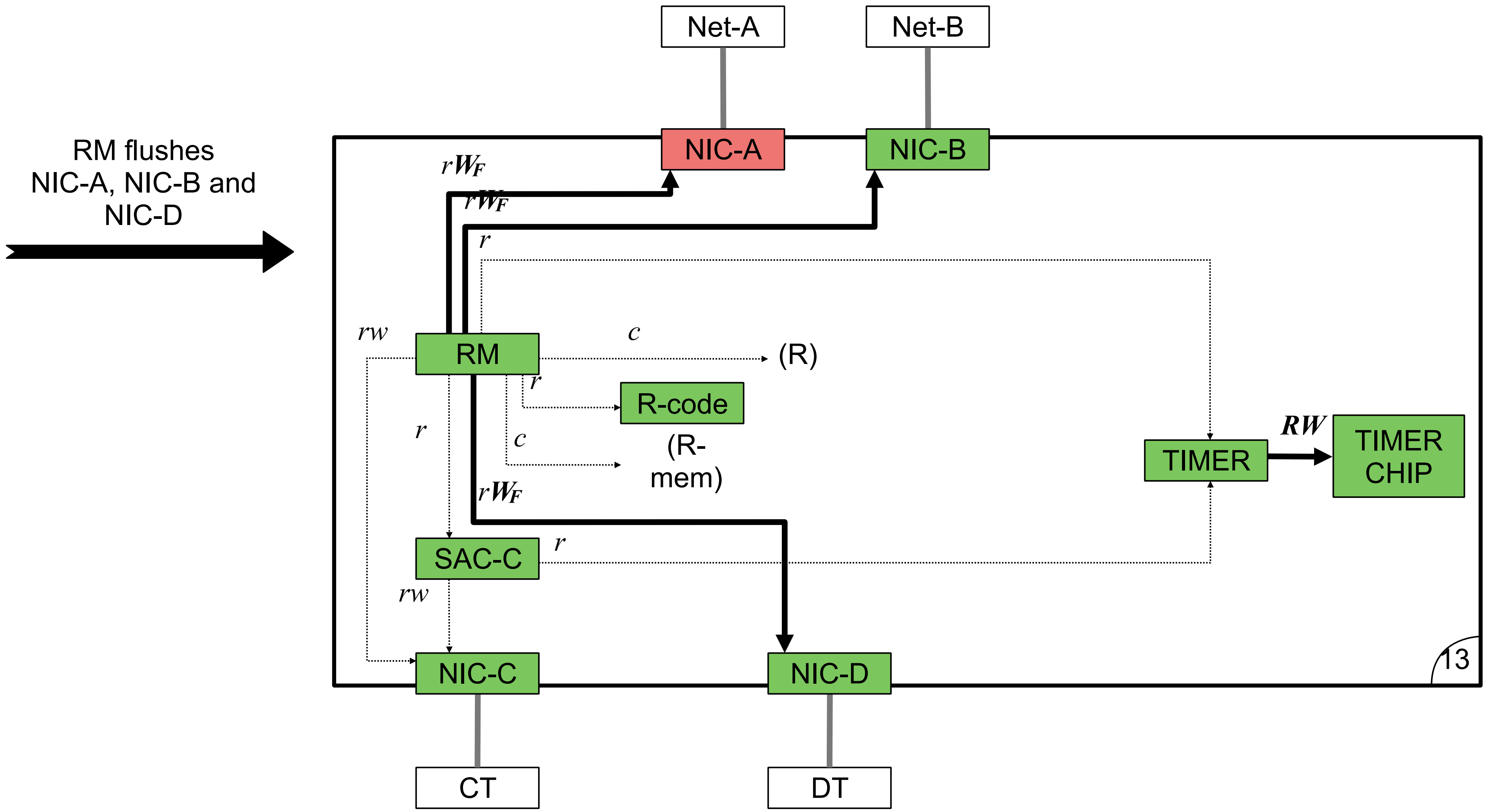
Life Cycle

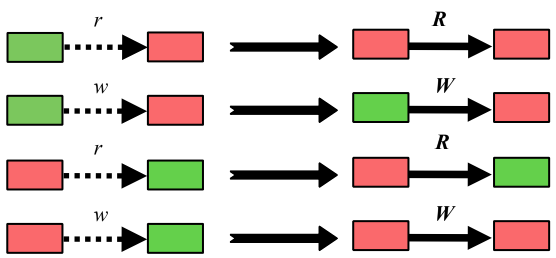





```
RM_id          -> Some ( {rw_to_NIC_A, rw_to_NIC_B, …}, not_contaminated )
SAC_C_id       -> Some ( {rw_to_NIC_C, r_to_TIMER}, not_contaminated )
TIMER_id       -> Some ( {rw_to_TIMER_CHIP}, not_contaminated )
R_id           -> None
R_mem_id       -> None
NIC_A_id       -> Some ( {}, contaminated )
NIC_B_id       -> Some ( {}, not_contaminated )
NIC_C_id       -> Some ( {}, not_contaminated )
NIC_D_id       -> Some ( {}, contaminated )
R_code_id      -> Some ( {}, not_contaminated )
TIMER_CHIP_id  -> Some ( {}, not_contaminated )
```

Life Cycle



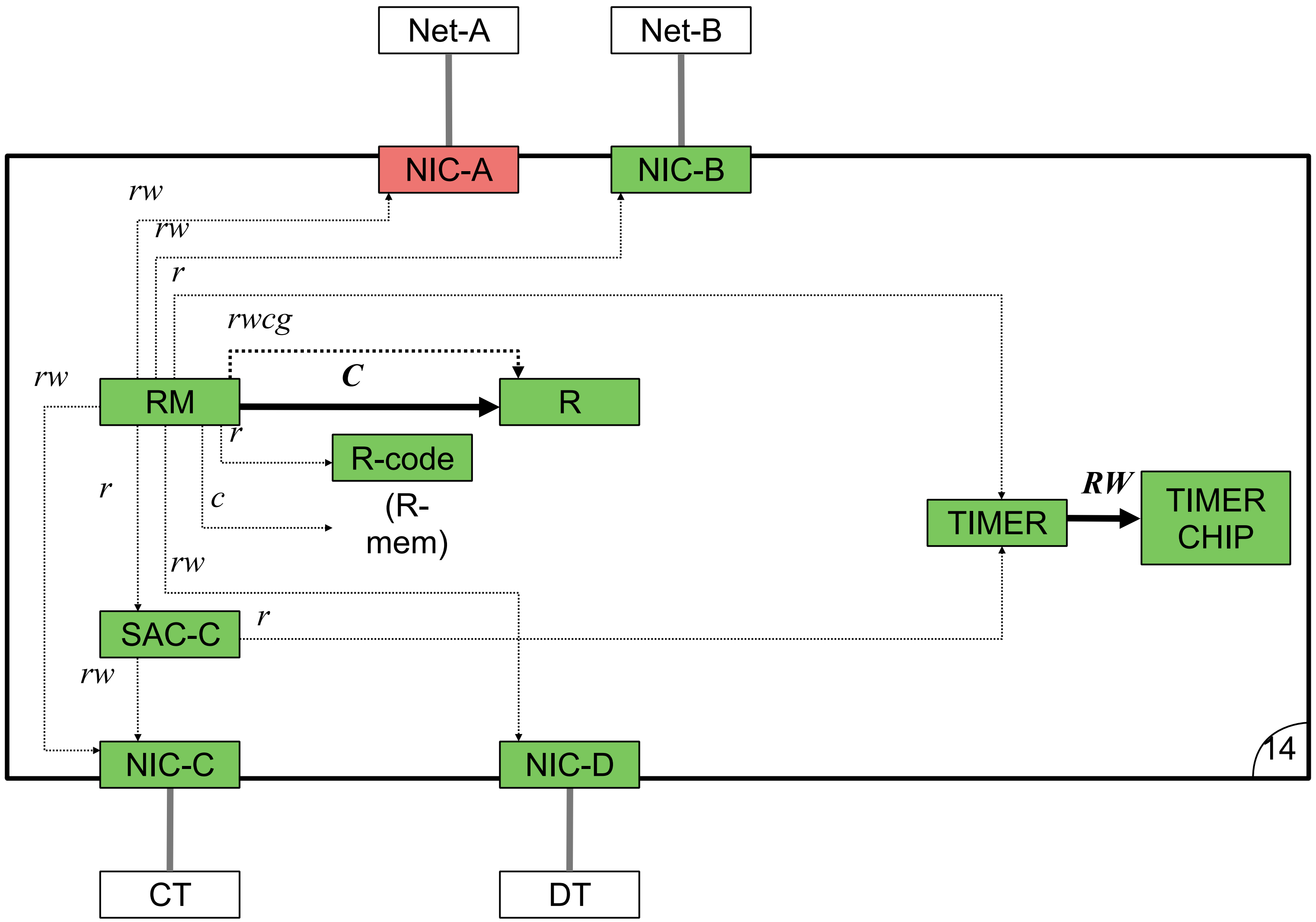


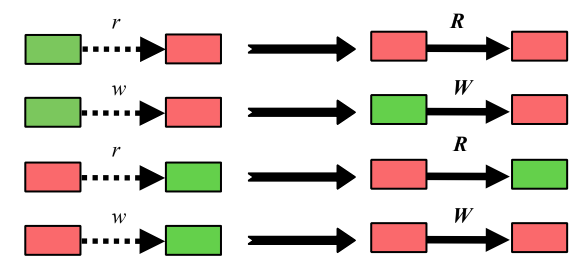

```
RM_id         -> Some ( {rw_to_NIC_A, rw_to_NIC_B, …}, not_contaminated )
SAC_C_id      -> Some ( {rw_to_NIC_C, r_to_TIMER}, not_contaminated )
TIMER_id      -> Some ( {rw_to_TIMER_CHIP}, not_contaminated )
R_id          -> None
R_mem_id      -> None
NIC_A_id      -> Some ( {}, contaminated )
NIC_B_id      -> Some ( {}, not_contaminated )
NIC_C_id      -> Some ( {}, not_contaminated )
NIC_D_id      -> Some ( {}, not_contaminated )
R_code_id     -> Some ( {}, not_contaminated )
TIMER_CHIP_id -> Some ( {}, not_contaminated )
```

Life Cycle

RM creates a new Router instance R

(and gets full rights to the newly created object)





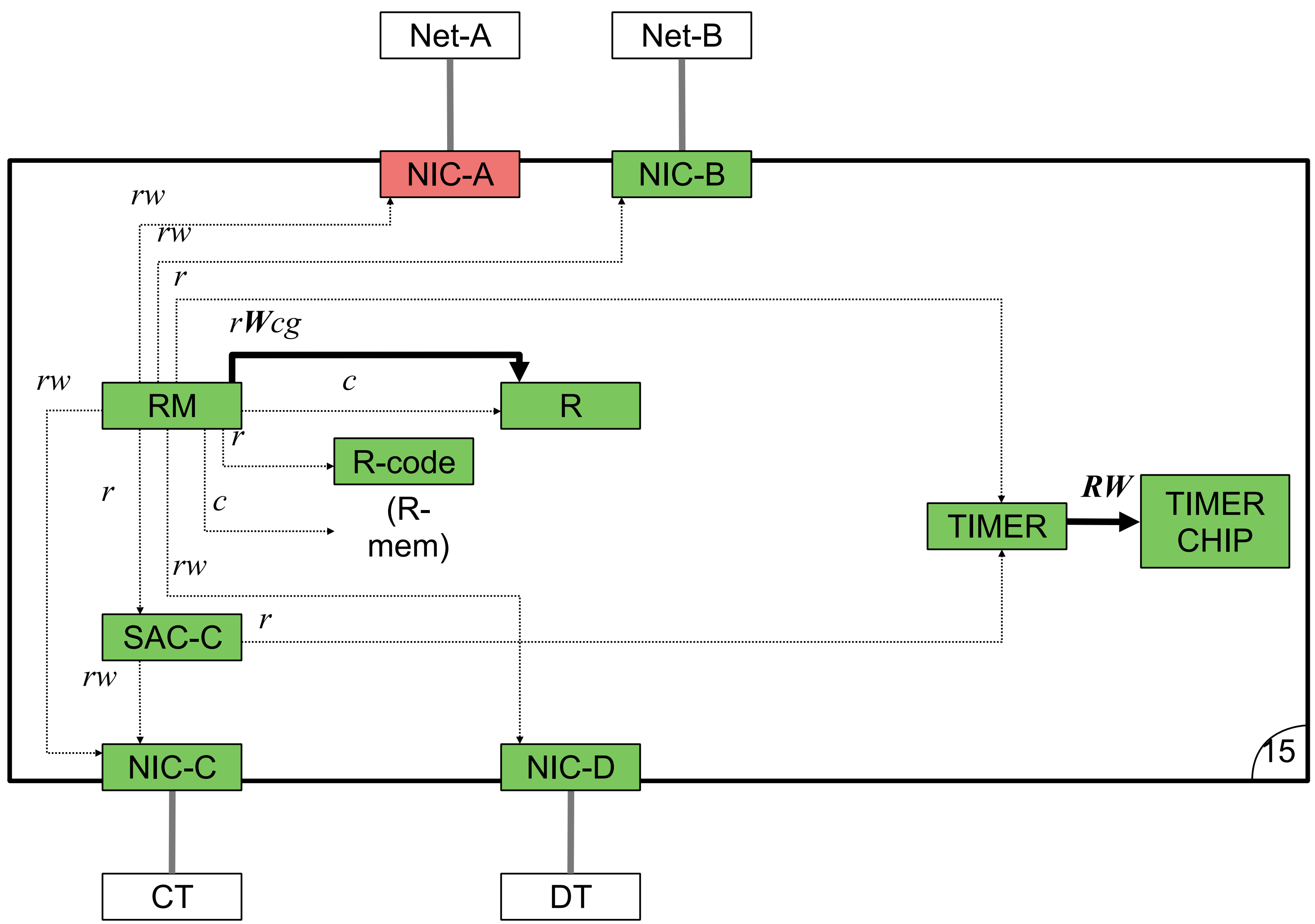

```
RM_id          -> Some ( {rw_to_NIC_A, rw_to_NIC_B, …, rwcg_to_R}, not_contaminated )
SAC_C_id      -> Some ( {rw_to_NIC_C, r_to_TIMER}, not_contaminated )
TIMER_id       -> Some ( {rw_to_TIMER_CHIP}, not_contaminated )
R_id           -> Some ( {}, not_contaminated )
R_mem_id       -> None
NIC_A_id      -> Some ( {}, contaminated )
NIC_B_id       -> Some ( {}, not_contaminated )
NIC_C_id       -> Some ( {}, not_contaminated )
NIC_D_id       -> Some ( {}, not_contaminated )
R_code_id     -> Some ( {}, not_contaminated )
TIMER_CHIP_id -> Some ( {}, not_contaminated )
```

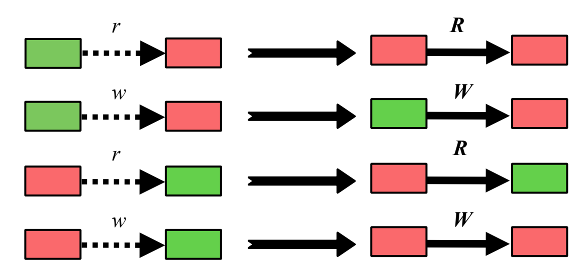
Life Cycle

NICTA

RM grants to R its rights to NIC-B, NIC-D,

R-mem, R-code, TIMER and itself

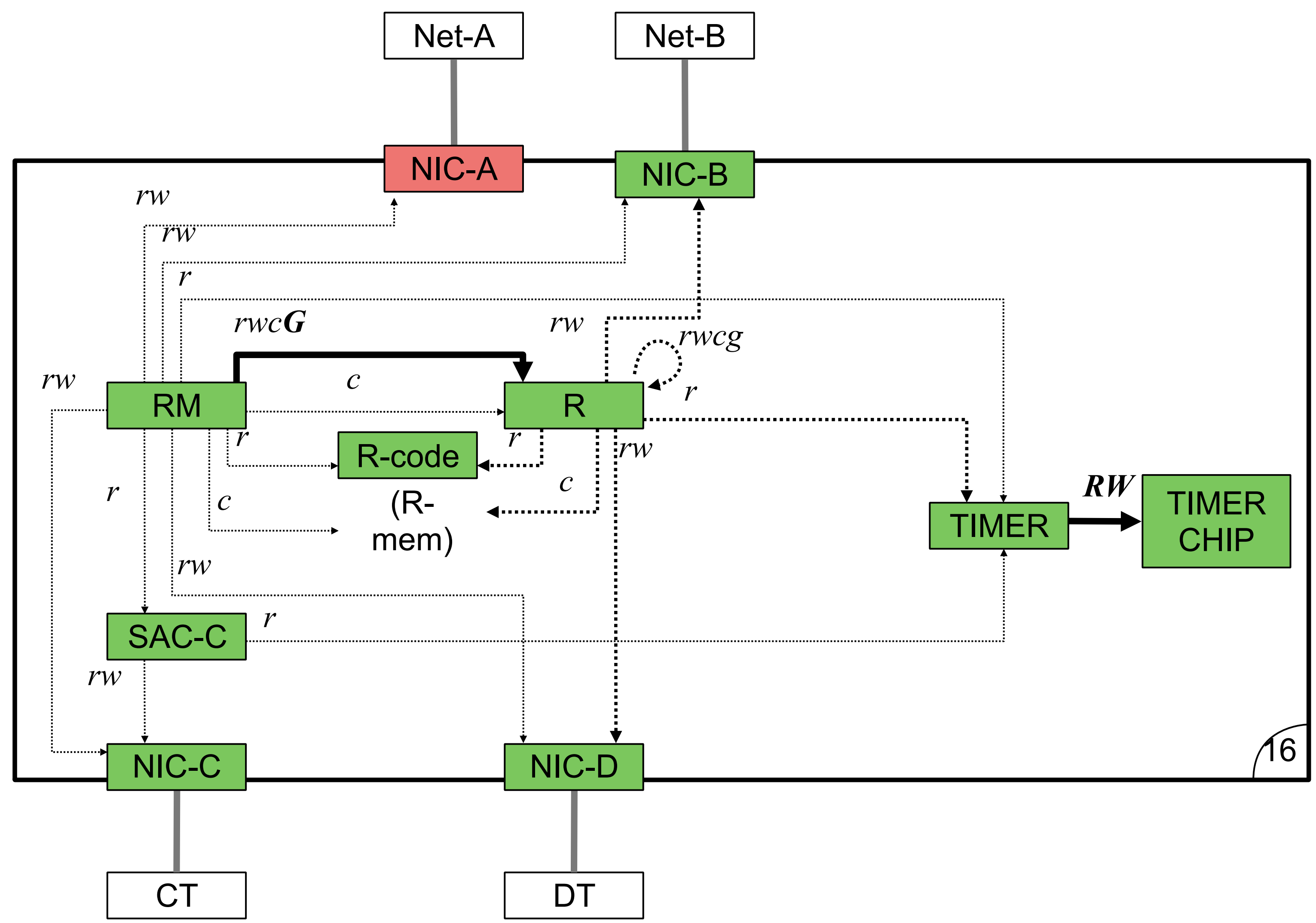


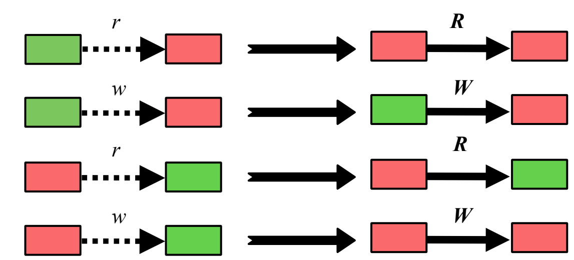



```
RM_id          -> Some ( {rw_to_NIC_A, rw_to_NIC_B, …, rwcg_to_R}, not_contaminated )
SAC_C_id       -> Some ( {rw_to_NIC_C, r_to_TIMER}, not_contaminated )
TIMER_id       -> Some ( {rw_to_TIMER_CHIP}, not_contaminated )
R_id           -> Some ( {}, not_contaminated )
R_mem_id       -> None
NIC_A_id       -> Some ( {}, contaminated )
NIC_B_id       -> Some ( {}, not_contaminated )
NIC_C_id       -> Some ( {}, not_contaminated )
NIC_D_id       -> Some ( {}, not_contaminated )
R_code_id      -> Some ( {}, not_contaminated )
TIMER_CHIP_id  -> Some ( {}, not_contaminated )
```

Life Cycle

RM grants to R its rights to NIC-B, NIC-D, R-mem, R-code, TIMER and itself



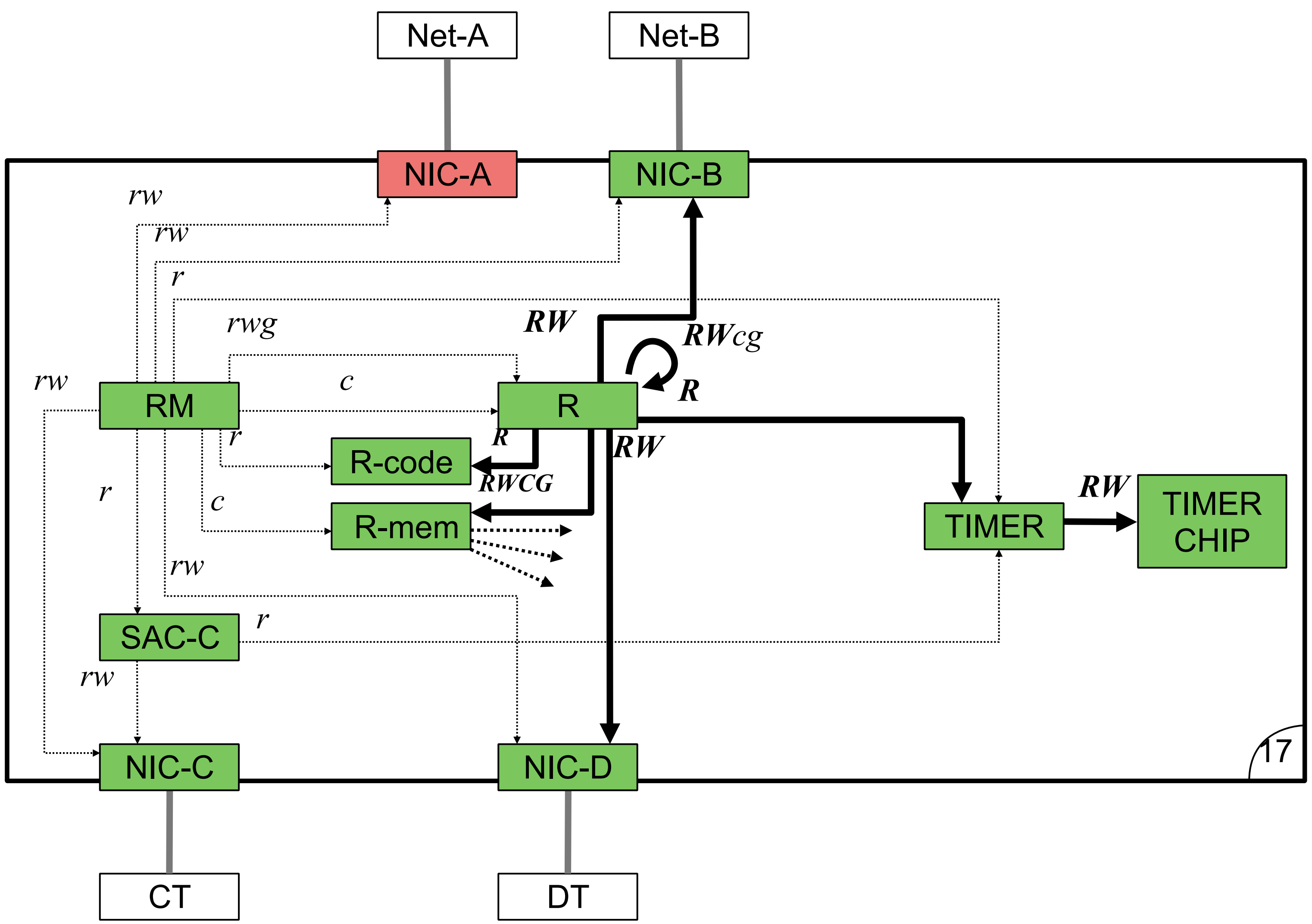


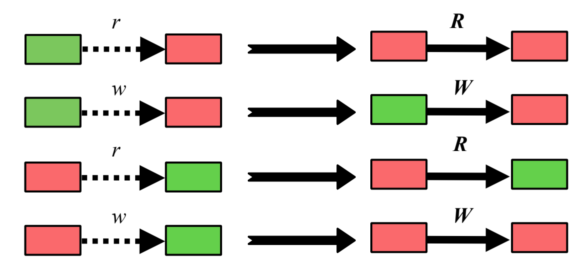

```
RM_id         -> Some ( {rw_to_NIC_A, rw_to_NIC_B, …, rwcg_to_R}, not_contaminated )
SAC_C_id      -> Some ( {rw_to_NIC_C, r_to_TIMER}, not_contaminated )
TIMER_id      -> Some ( {rw_to_TIMER_CHIP}, not_contaminated )
R_id          -> Some ( {rw_to_NIC_A, rw_to_NIC_B, …}, not_contaminated )
R_mem_id      -> None
NIC_A_id      -> Some ( {}, contaminated )
NIC_B_id      -> Some ( {}, not_contaminated )
NIC_C_id      -> Some ( {}, not_contaminated )
NIC_D_id      -> Some ( {}, not_contaminated )
R_code_id     -> Some ( {}, not_contaminated )
TIMER_CHIP_id -> Some ( {}, not_contaminated )
```

Life Cycle

DT starts to communicate with Net-B, through R.

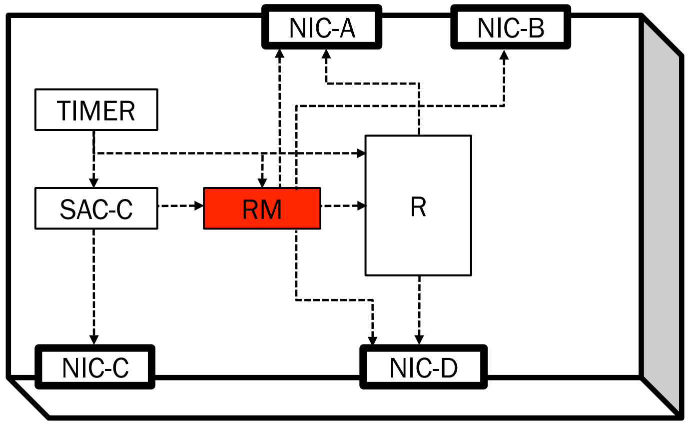
This may imply the creation of new objects using R-mem and granting some caps to them.





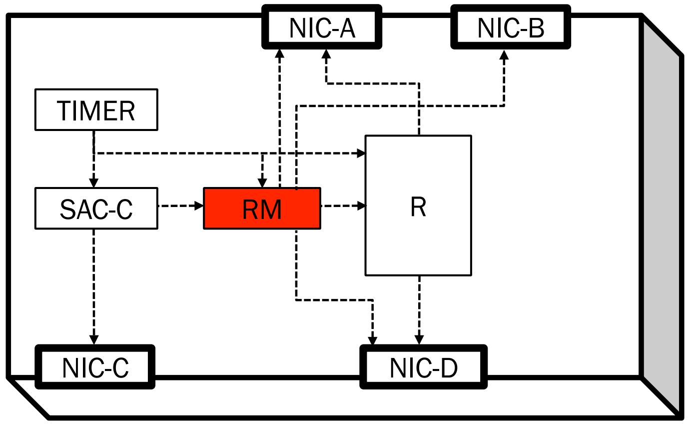

```
RM_id          -> Some ( {rw_to_NIC_A, rw_to_NIC_B, …, rwcg_to_R}, not_contaminated )
SAC_C_id       -> Some ( {rw_to_NIC_C, r_to_TIMER}, not_contaminated )
TIMER_id       -> Some ( {rw_to_TIMER_CHIP}, not_contaminated )
R_id           -> Some ( {rw_to_NIC_A, rw_to_NIC_B, …}, not_contaminated )
R_mem_id       -> Some ( {…}, not_contaminated )
NIC_A_id       -> Some ( {}, contaminated )
NIC_B_id       -> Some ( {}, not_contaminated )
NIC_C_id       -> Some ( {}, not_contaminated )
NIC_D_id       -> Some ( {}, not_contaminated )
R_code_id      -> Some ( {}, not_contaminated )
TIMER_CHIP_id  -> Some ( {}, not_contaminated )
```

So far



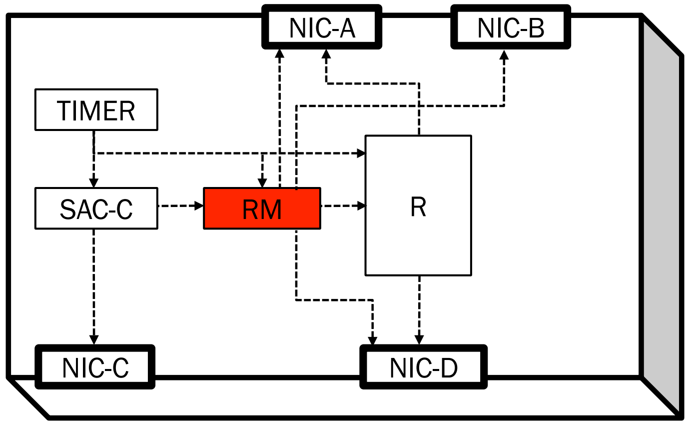
So far

- Can build systems with
  - large untrusted components
  - plus few small, trusted components
  - trusted = needs behaviour spec

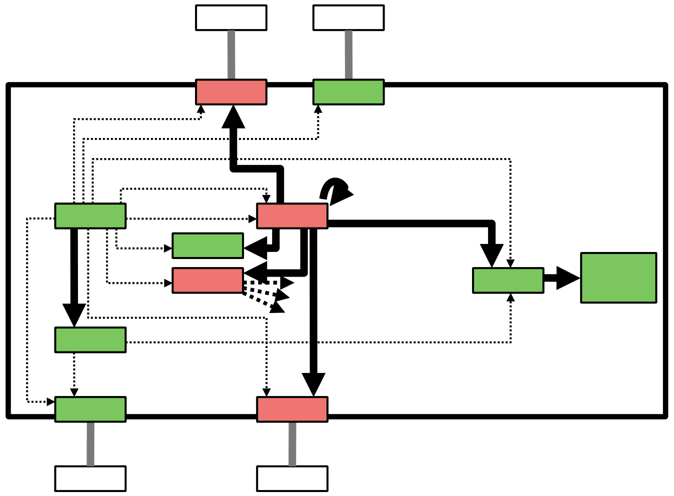


So far

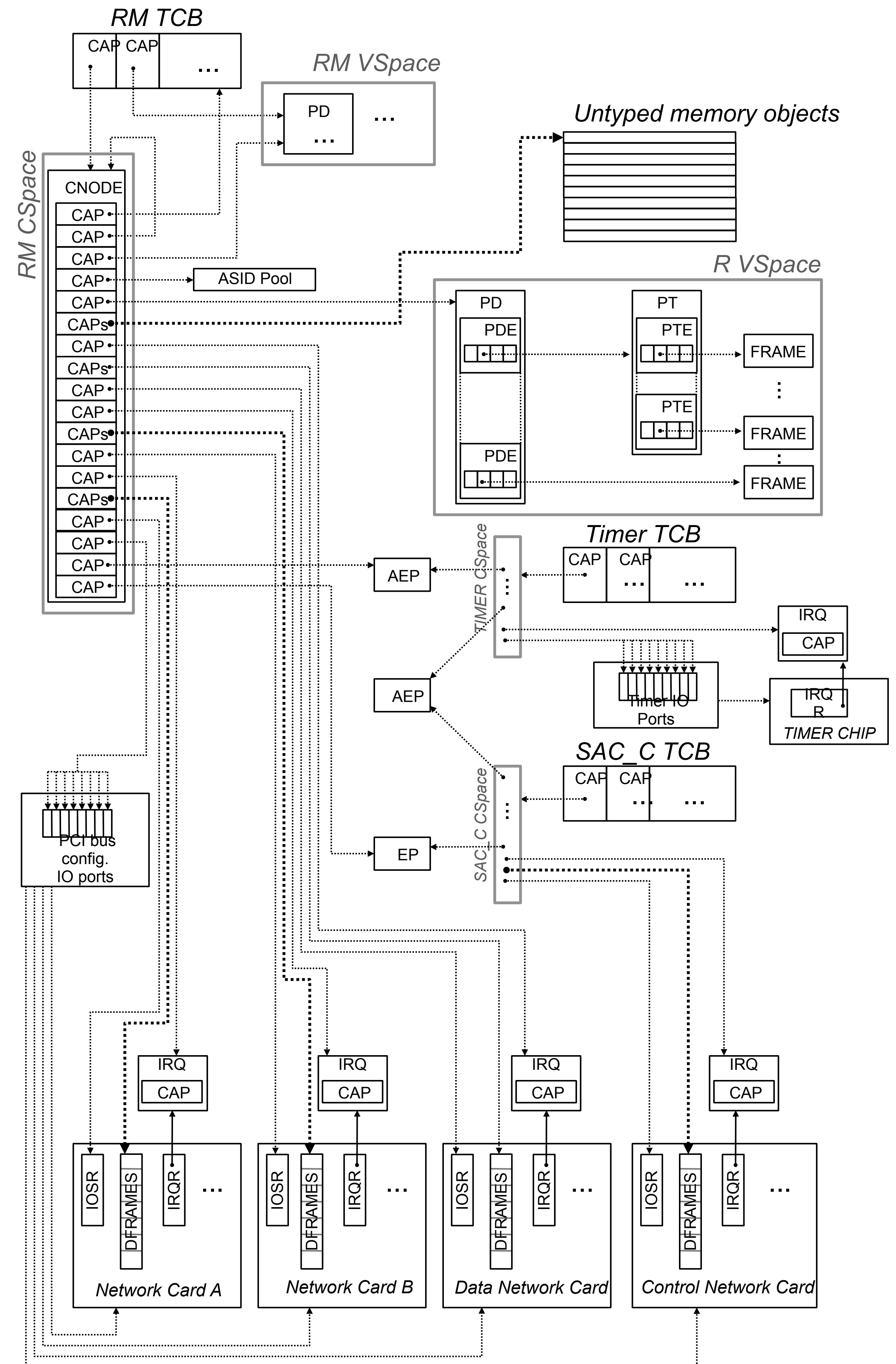
- Can build systems with
  - large untrusted components
  - plus few small, trusted components
  - trusted = needs behaviour spec



- Use take-grant to model security
  - can simulate system
  - modelling already finds bugs
  - high-level proof in Isabelle/HOL or SPIN
  - includes behaviour of trusted component



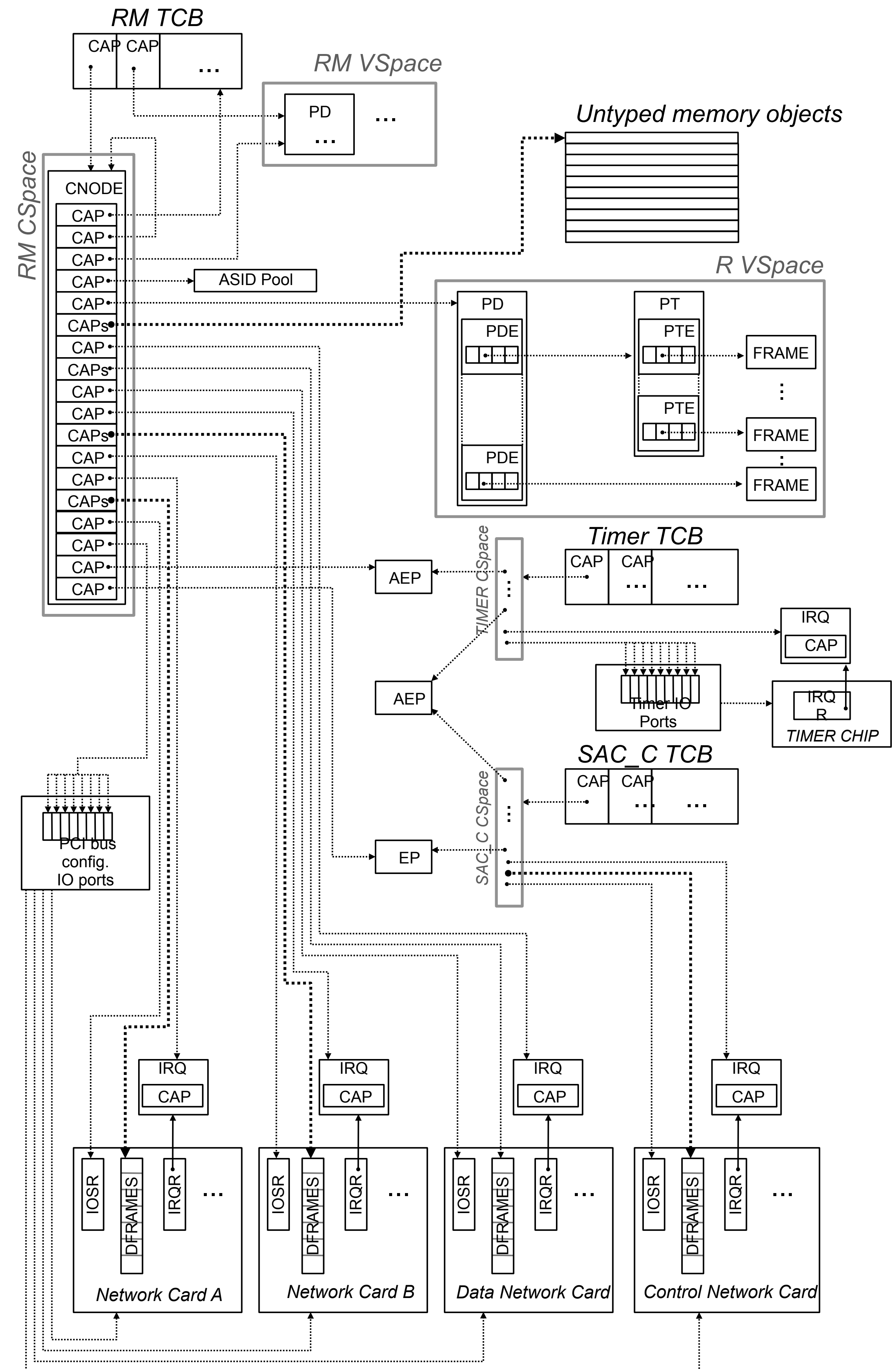
Future



IOSR = IOSpace Root Pointer
IRQR = IRQ register reference
DFRAMES = Device Frames

Future

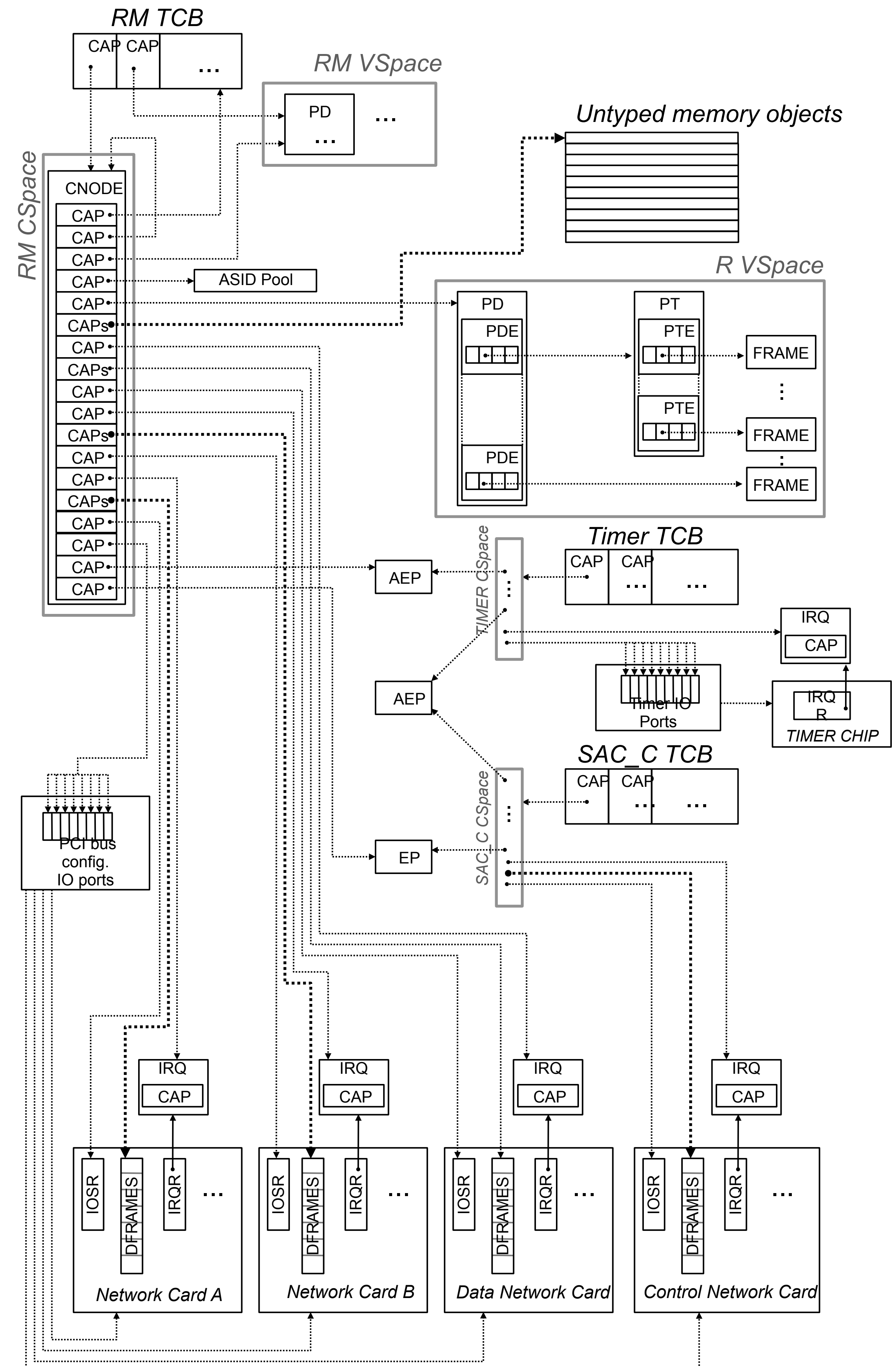
- Need to
  verify low-level design



IOSR = IOSpace Root Pointer
IRQR = IRQ register reference
DFRAMES = Device Frames

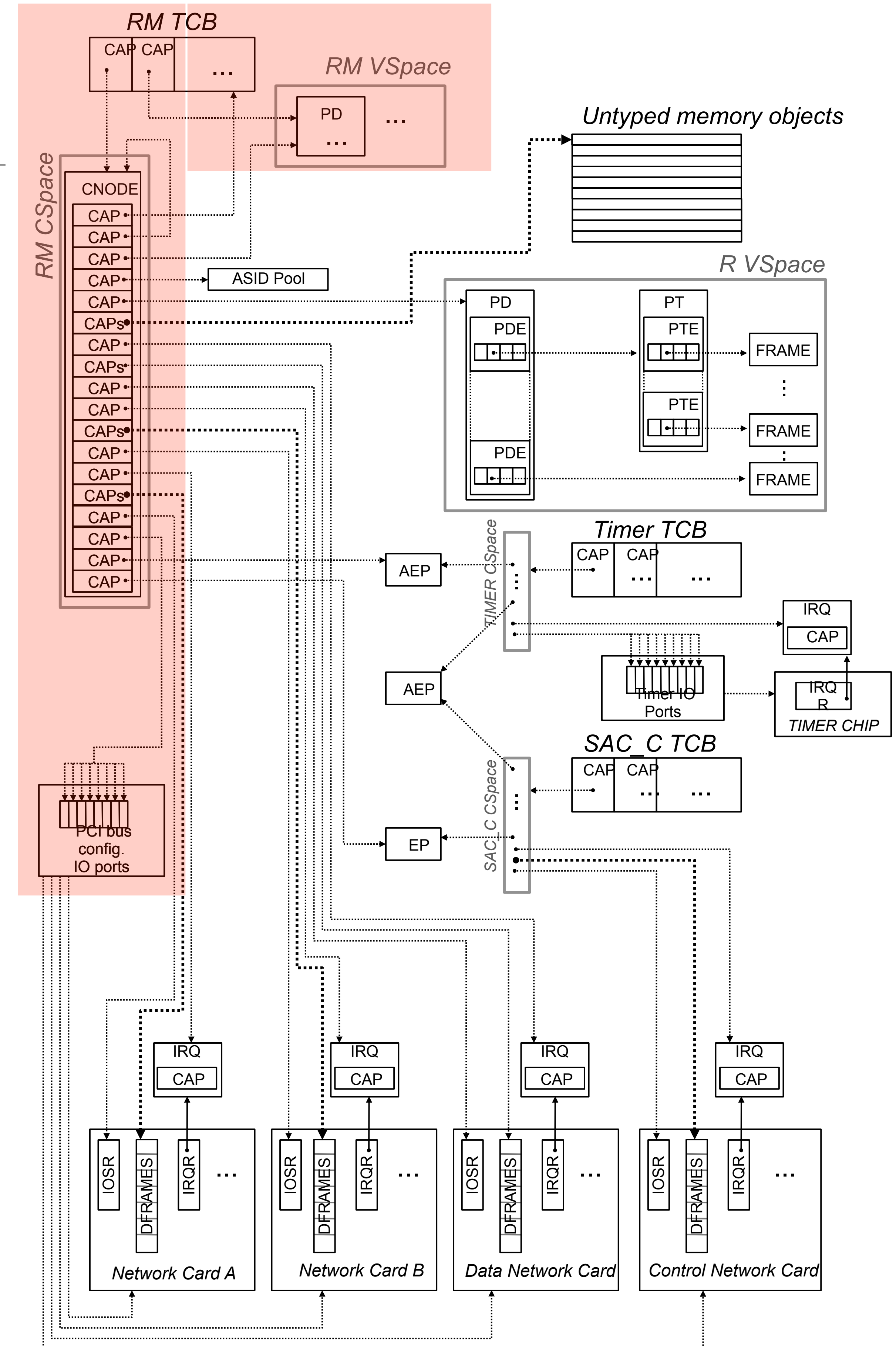
Future

- Need to verify low-level design
- Building tool-chain for:
  - describing cap layout (capDL)
  - generating booter
  - generating booter proof
  - abstraction to take-grant



IOSR = IOSpace Root Pointer
IRQR = IRQ register reference
DFRAMES = Device Frames

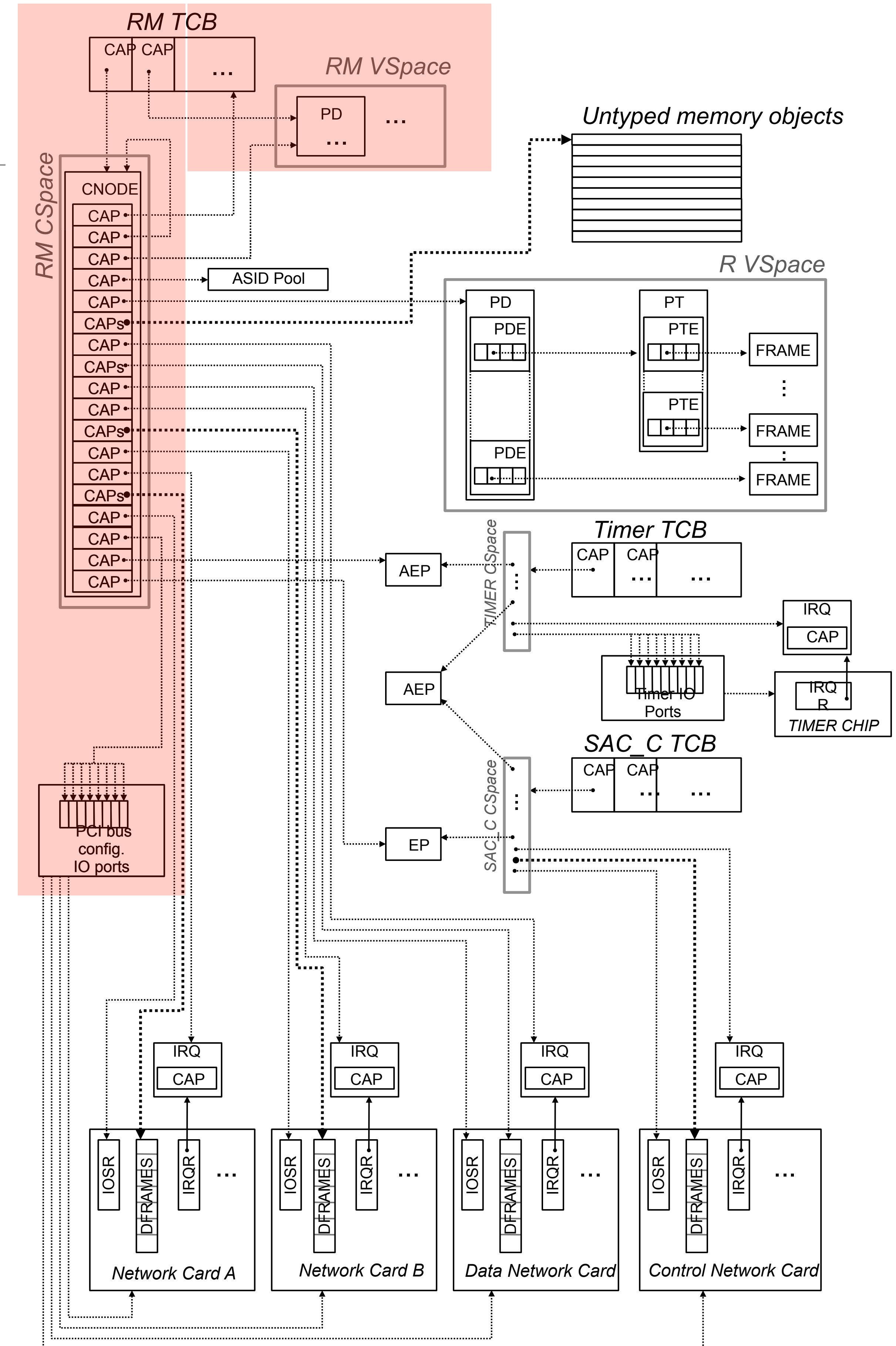
More Future



IOSR = IOSpace Root Pointer
IRQR = IRQ register reference
DFRAMES = Device Frames

More Future

- Verify
  Trusted Component

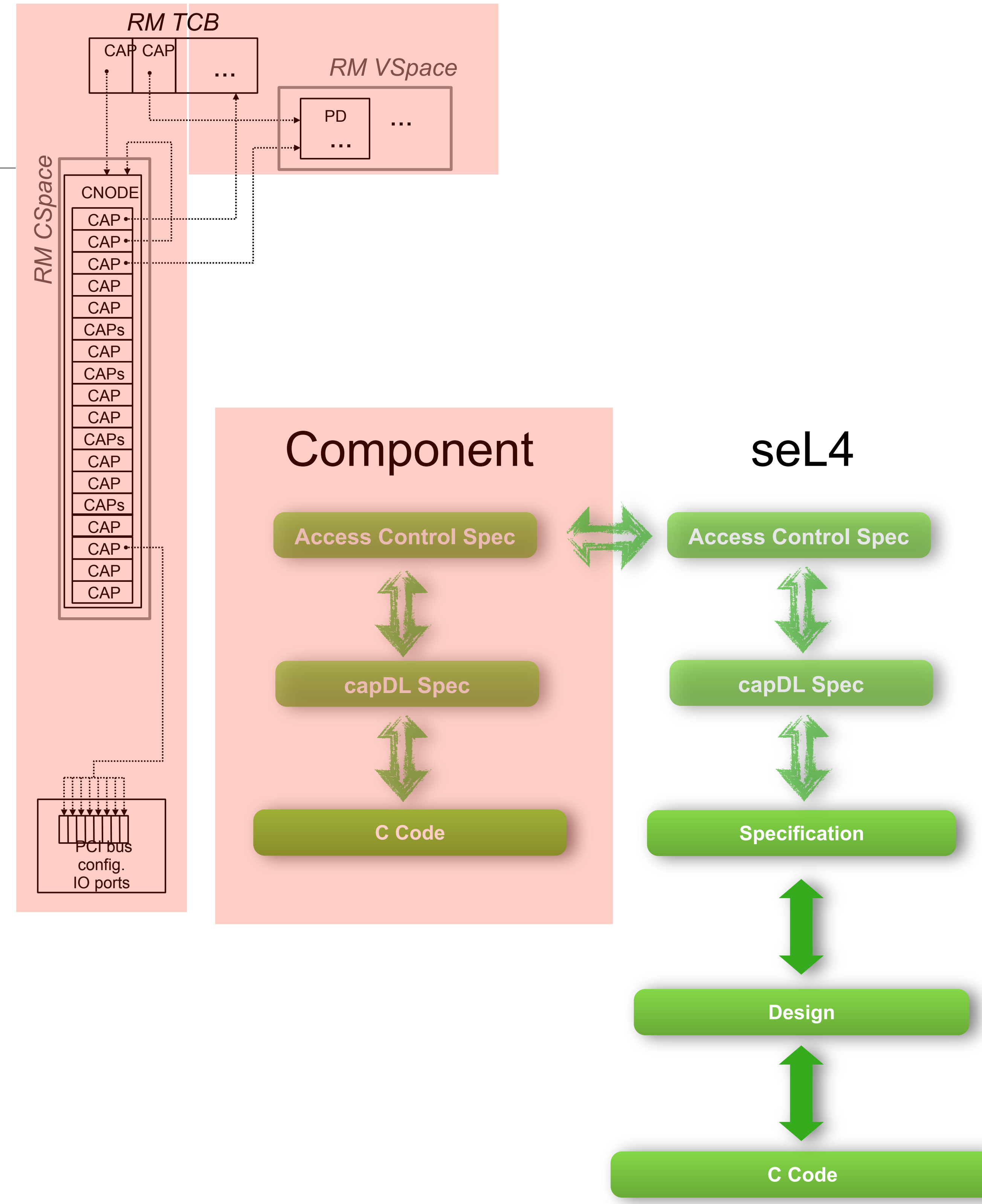


IOSR = IOSpace Root Pointer
IRQR = IRQ register reference
DFRAMES = Device Frames

More Future

- Verify
  Trusted Component

- Refine to C:
  – interface with kernel
  – use most abstract level possible
  – make sure sec property preserved by refinement



Summary

Summary

Summary

Formal proof all the way from spec to C.

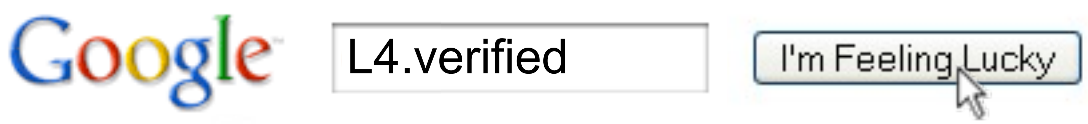
- **200kloc** handwritten, machine-checked proof
- ~**460** bugs (160 in C)
- Verification on **code**, **design**, and **spec**
- Systems with **trusted components**
- The future: formal proof for large systems down to code

Formal Code Verification up to 10kloc:

It works.
It's feasible.
It's cheaper.

(It's fun, too)

Thank You



NICTA