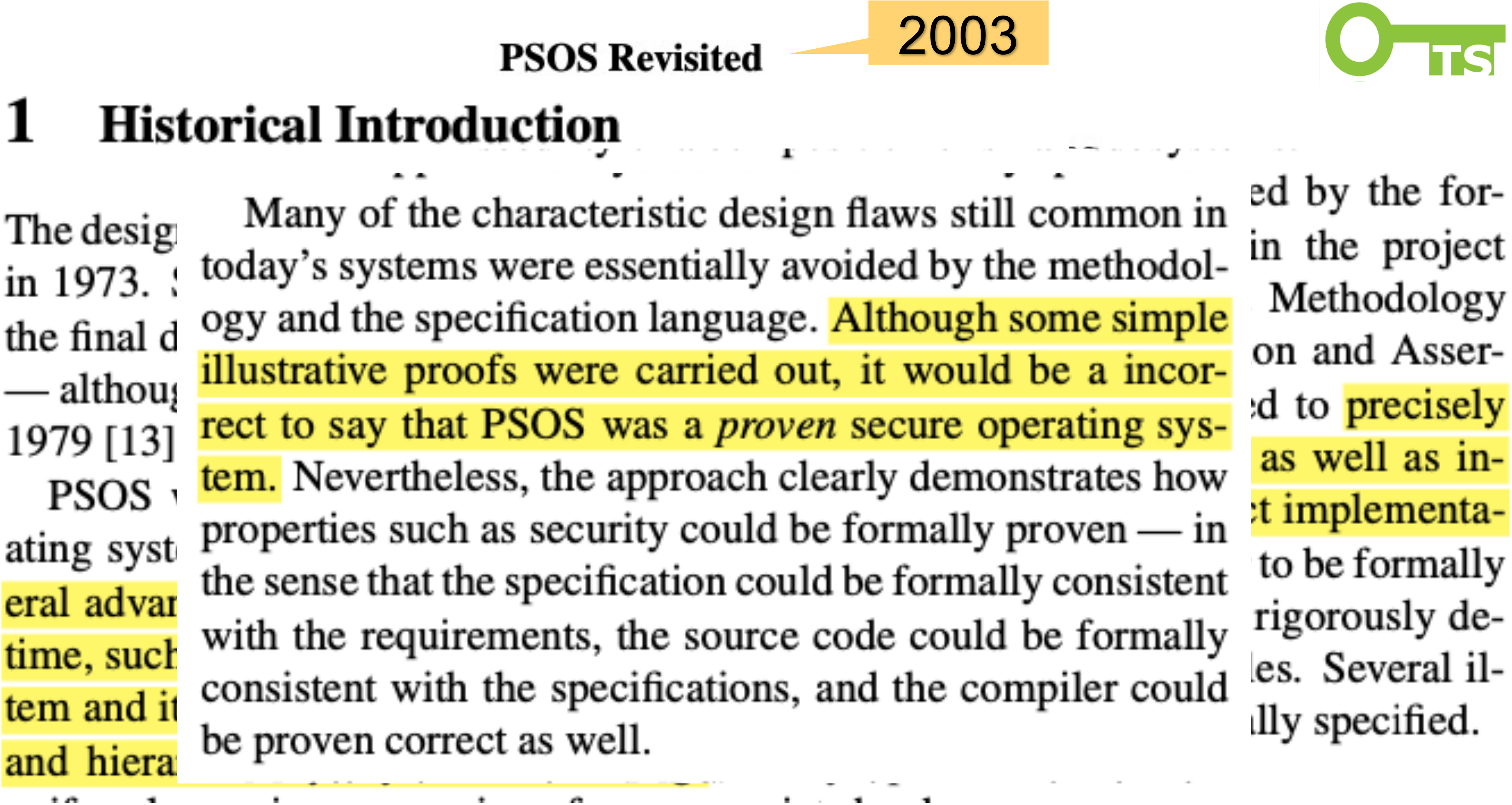
School of Computer Science & Engineering

Trustworthy Systems Group

Will we ever have truly secure operating systems?

Gernot Heiser

gernot@unsw.edu.au
@microkerneldude.bsky.social
https://microkerneldude.org/



PSOS Revisited

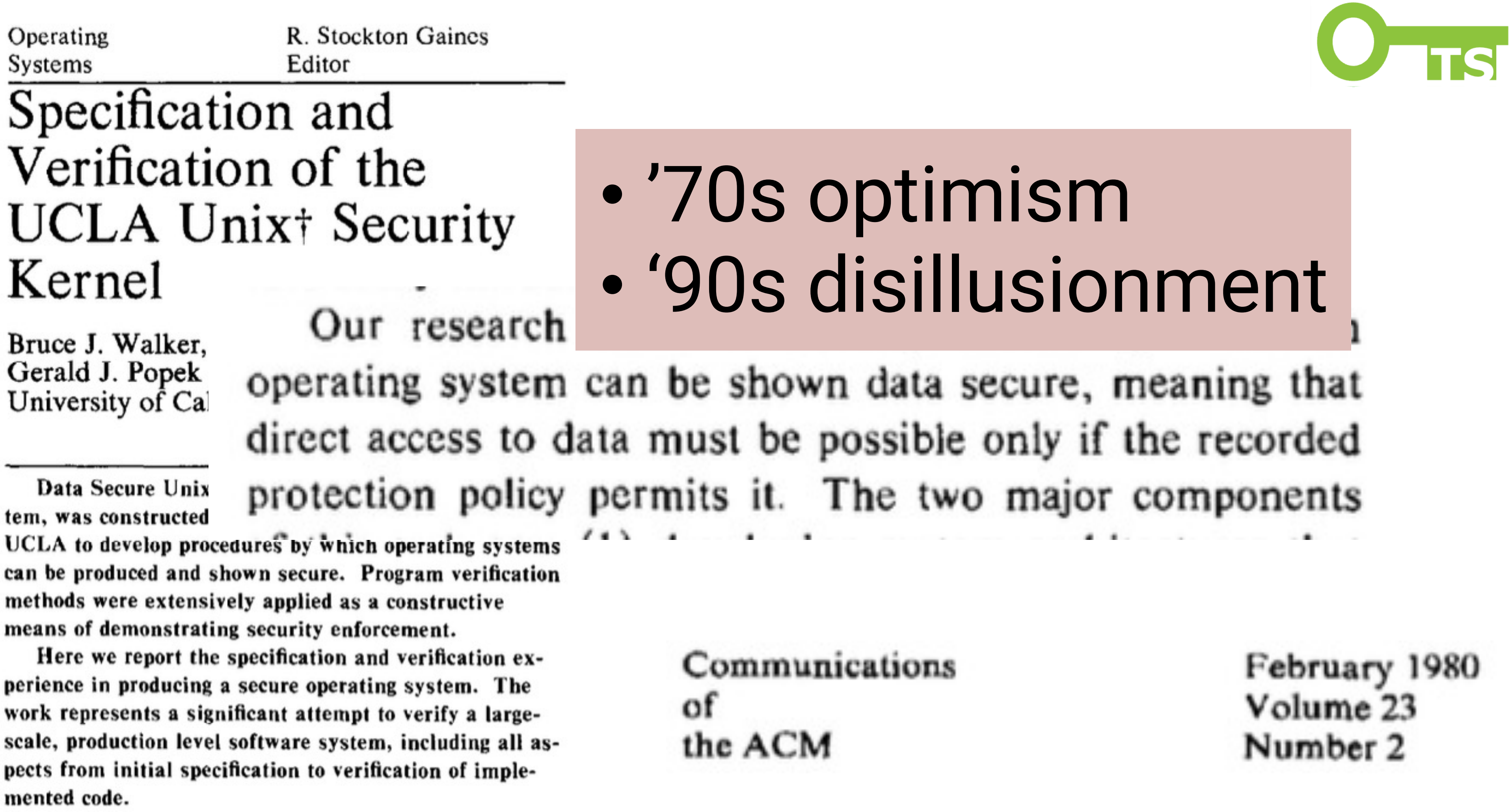
2003

1 Historical Introduction

The desig
in 1973.
the final d
— althoug
1979 [13]
PSOS
ating syst
eral advan
time, such
tem and it
and hiera

Many of the characteristic design flaws still common in today's systems were essentially avoided by the methodology and the specification language. Although some simple illustrative proofs were carried out, it would be a incorrect to say that PSOS was a *proven* secure operating system. Nevertheless, the approach clearly demonstrates how properties such as security could be formally proven — in the sense that the specification could be formally consistent with the requirements, the source code could be formally consistent with the specifications, and the compiler could be proven correct as well.

ed by the for-
in the project
Methodology
on and Asser-
ed to precisely
as well as in-
t implementa-
to be formally
rigorously de-
les. Several il-
lly specified.



Operating Systems

R. Stockton Gaines Editor

Specification and Verification of the UCLA Unix† Security Kernel

Bruce J. Walker, Gerald J. Popek University of Ca

Data Secure Unix tem, was constructed UCLA to develop procedures by which operating systems can be produced and shown secure. Program verification methods were extensively applied as a constructive means of demonstrating security enforcement.

Here we report the specification and verification experience in producing a secure operating system. The work represents a significant attempt to verify a large-scale, production level software system, including all aspects from initial specification to verification of implemented code.

Our research operating system can be shown data secure, meaning that direct access to data must be possible only if the recorded protection policy permits it. The two major components

Communications of the ACM

February 1980 Volume 23 Number 2

- ’70s optimism
- ‘90s disillusionment

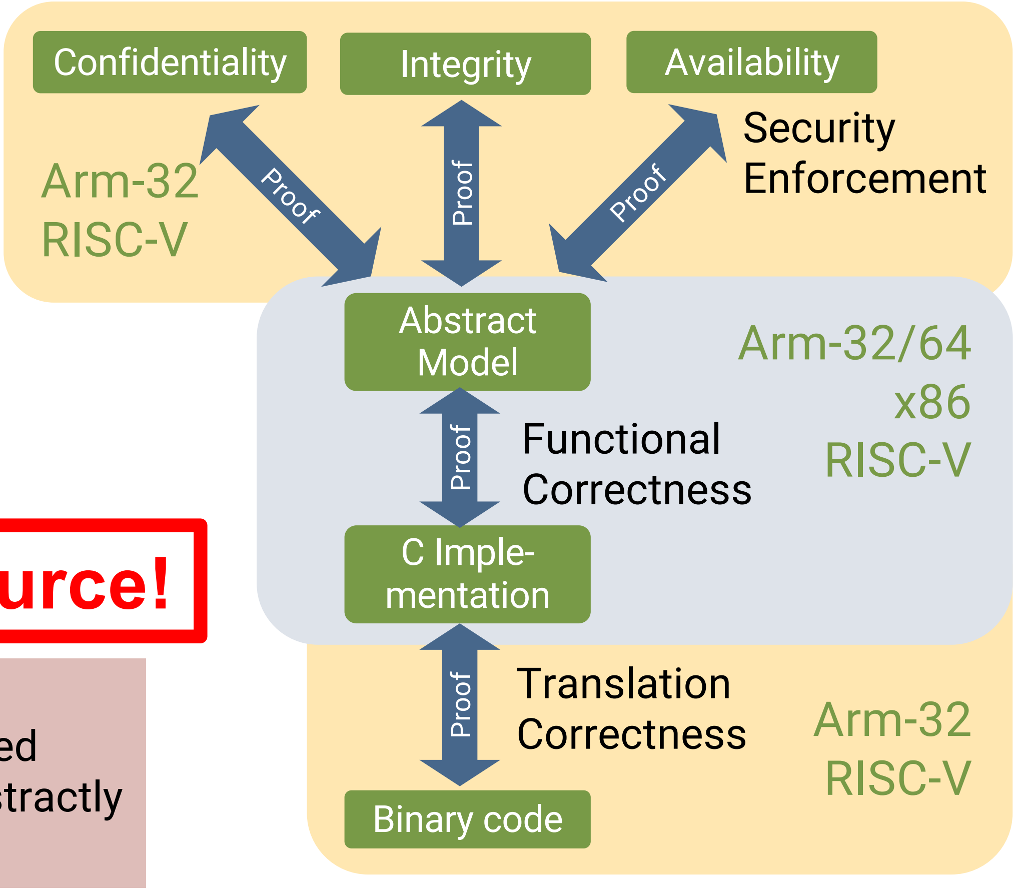
2009: Verification of a Microkernel

- World’s first OS kernel with correctness proof
- Most comprehensive verification
- Only verified OS with capability-based fine-grained protection
- Only protected-mode RTOS with sound and compete WCET analysis

Open Source!

Present limitations

- Initialisation code not verified
- MMU, caches modelled abstractly
- Multicore not yet verified



Confidentiality

Integrity

Availability

Security Enforcement

Arm-32
RISC-V

Proof

Proof

Proof

Abstract Model

Proof

Functional Correctness

Arm-32/64
x86
RISC-V

C Implementation

Proof

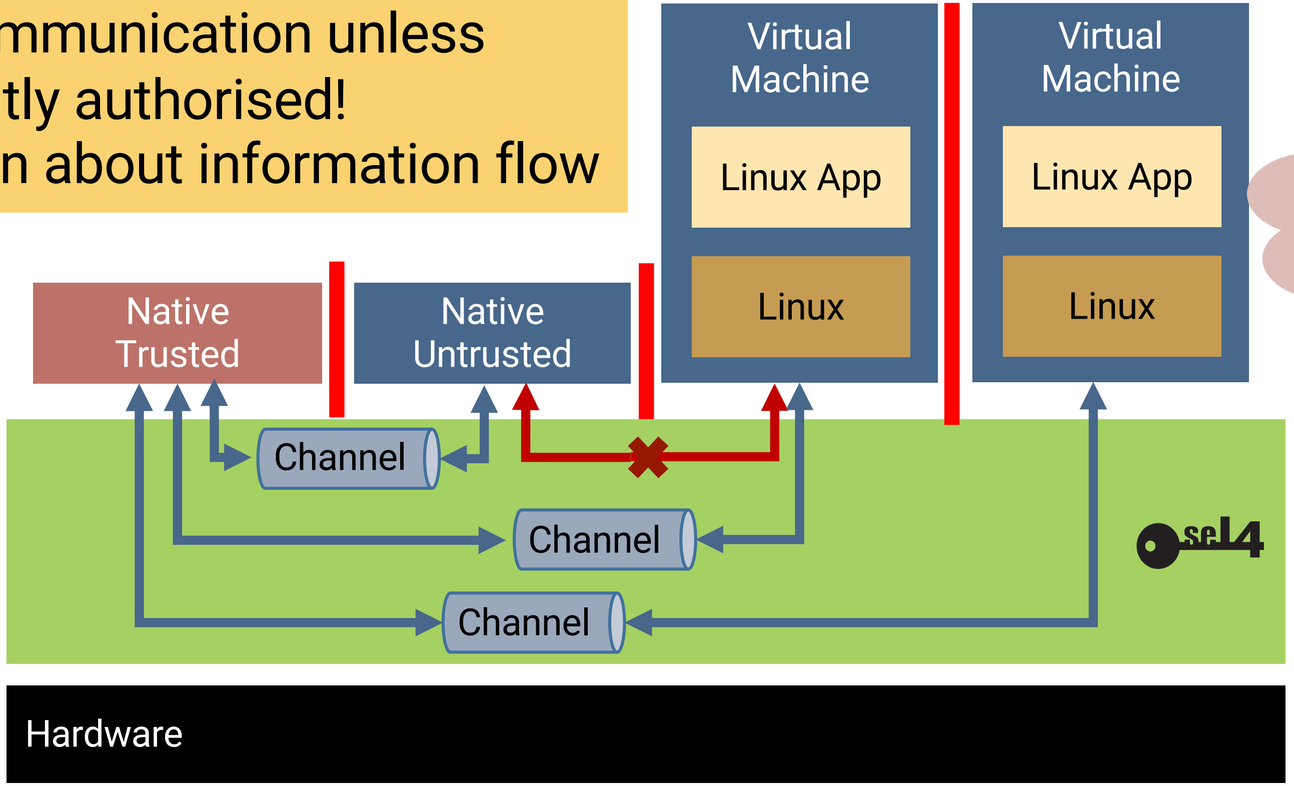
Translation Correctness

Arm-32
RISC-V

Binary code

Capabilities: Fine-Grained Protection

- Enforce *least privilege*
- No communication unless explicitly authorised!
- Reason about information flow



No capabilities? You're not serious about security!

The Benchmark for Performance

Smaller is better

Round-trip cross-address-space IPC on 64-bit Intel Skylake

	seL4	Fiasco.OC	Google
		aka L4Re	Zircon
Latency (cycles)	986	2717	8157
Mandatory HW cost* (cycles)	790	790	790
Overhead absolute (cycles)	196	1972	7367
Overhead relative	25%	240%	930%

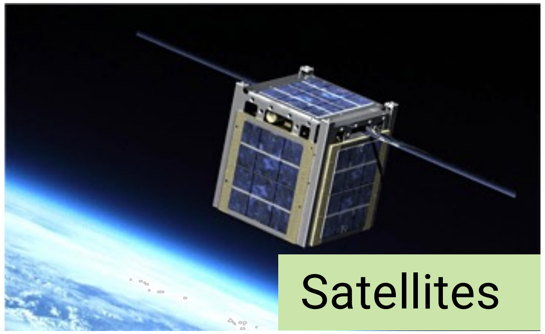
***:** The Cost of SYCALL + 2 × SWAPGS + SYSRET = 395 cycles, times 2 for round-trip

Source:

Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, Haibo Chen: “SkyBridge: Fast and Secure Inter-Process Communication for Microkernels”, EuroSys, April 2019

Used in Real-World Systems

Autonomous vehicles



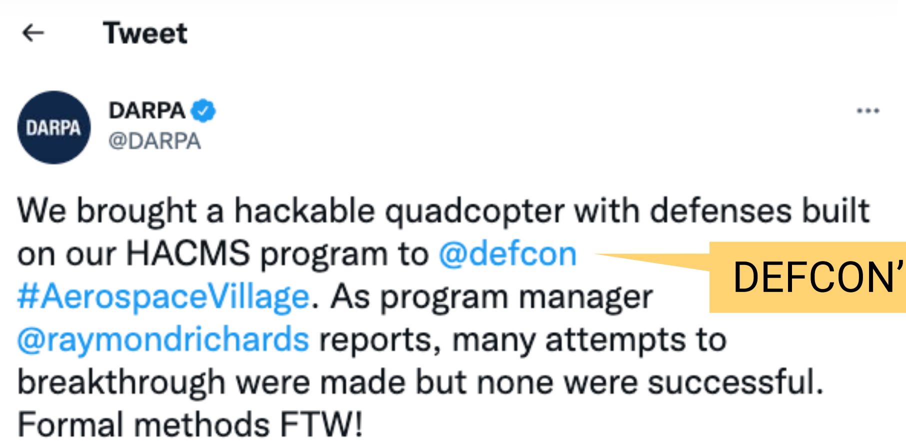
Satellites

Secure communication device
In use in multiple defense forces

Critical infrastructure protection

Cars

“World’s Most Secure Drone”



Tweet

DARPA
@DARPA

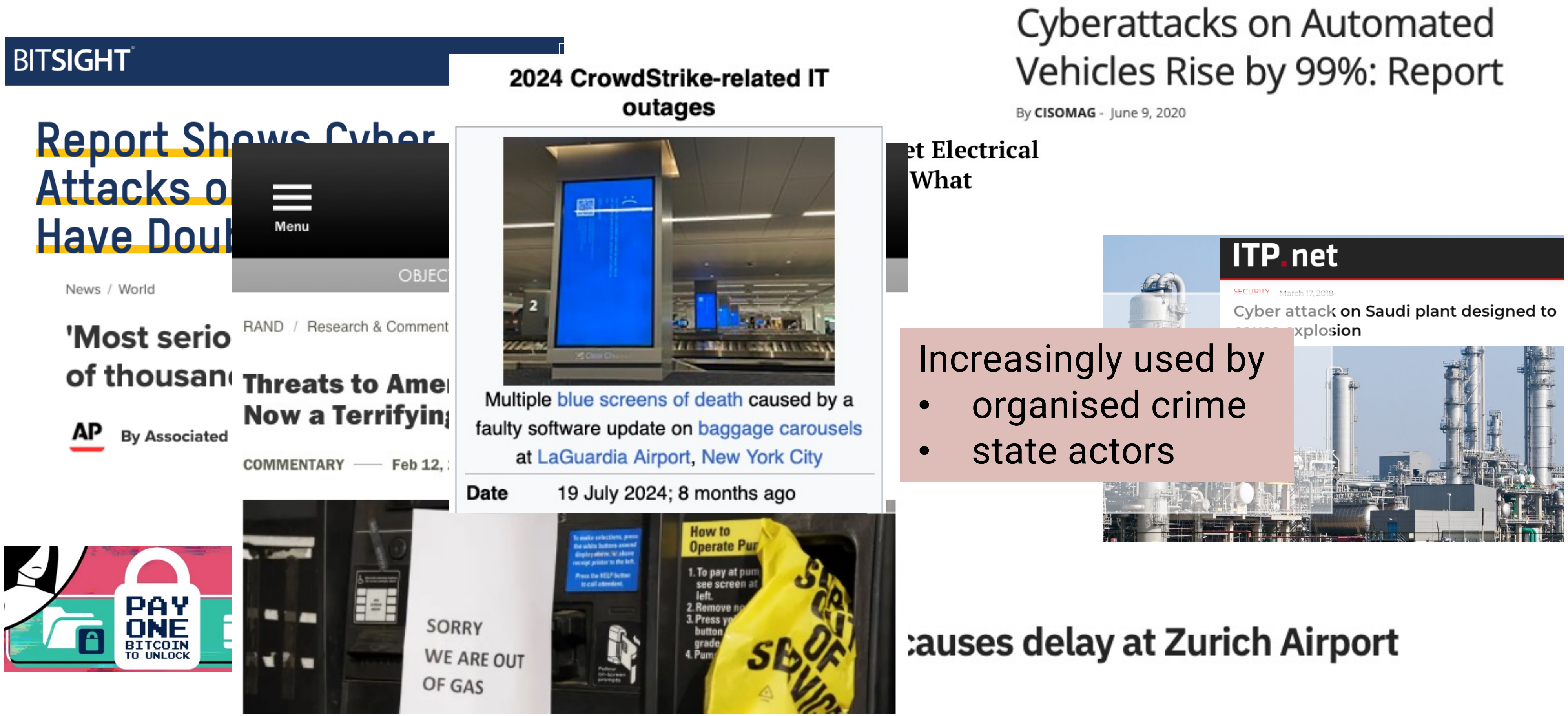
We brought a hackable quadcopter with defenses built on our HACMS program to @defcon #AerospaceVillage. As program manager @raymondrichards reports, many attempts to breakthrough were made but none were successful. Formal methods FTW!

DEFCON’22

seL4 Timeline

- July’09: Proof of implementation correctness (Arm-32)
- Aug’11: Proof of integrity enforcement
- Nov’11: Sound worst-case execution-time analysis
- May’13: Proof of confidentiality enforcement
- Jun’13: Proof of compilation correctness
- Jul’14: **seL4 open-sourced (GPL)**
- 2012–17: DARPA HACMS: seL4 in real-world systems
- 2018: x86 verification
- Jun’20: RISC-V verification
- Mar’24: Arm-64 verification
- Sep’24: Commercial electric car

Yet Security Failures Are Everywhere



BITSIGHT

Report Shows Cyber Attacks o Have Doul

News / World

'Most serio of thousan

AP By Associated

Menu

OBJEC

RAND / Research & Comment

Threats to Ame Now a Terrifyin

COMMENTARY —— Feb 12,

2024 CrowdStrike-related IT outages

Multiple blue screens of death caused by a faulty software update on baggage carousels at LaGuardia Airport, New York City

Date 19 July 2024; 8 months ago

et Electrical What

Cyberattacks on Automated Vehicles Rise by 99%: Report

By CISOMAG - June 9, 2020

ITP.net

SECURITY March 17, 2018

Cyber attack on Saudi plant designed to cause explosion

causes delay at Zurich Airport

Increasingly used by
- organised crime
- state actors

Why Still No Secure OS?

A Microkernel

Microkernel:

- OS code that must execute in privileged mode
- Everything else belongs in user mode servers
- Servers are subject to the microkernel's security enforcement!

Assembly language of operating systems

Consequence:

- Small: 10 kLOC
- Only fundamental, policy-free mechanisms
- No application-oriented services/abstractions
- **BYO file system, memory manager, device drivers**

Leave to community/ industry to build

seL4 Experience of the First 10+ Years

seL4's assurance and power are (still!) unrivalled

TS contributed poor designs too!

Good design on seL4 requires deep expertise

Rare beyond TS and ex-TSers

Arcane build system didn't help!

The world needs an OS that is:

- **secure**
- easy to use
- open source

Community did not deliver a secure OS!

LionsOS

Stop The Train Wrecks!

LionsOS Aims

Aim 1:
Practical, easy-to-use, open-source OS for wide range of *embedded/IoT/cyberphysical* use cases

Can use static architecture

Must be well designed!

Aim 2:
Uncompromising performance

Aim 3:
Most secure OS ever

Must be verified!

Overarching Design Principle: KISS!

LionsOS is what Posix/Linux isn't!

Helps development **and** verification!

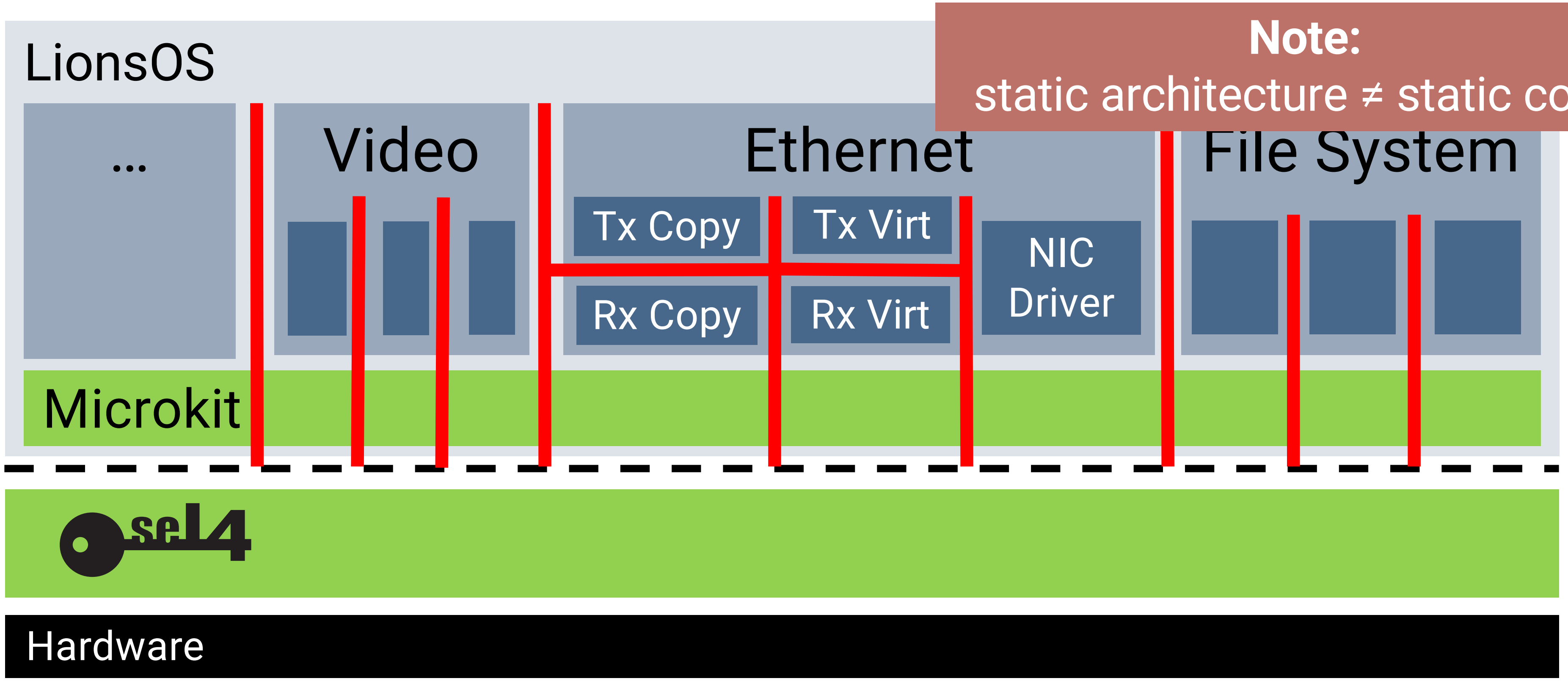
Radical simplicity:

- fine-grained modularity, strict separation of concerns
- event-driven programming model
- use-case-specific policies

… but we'll have Posix-like I/O wrappers

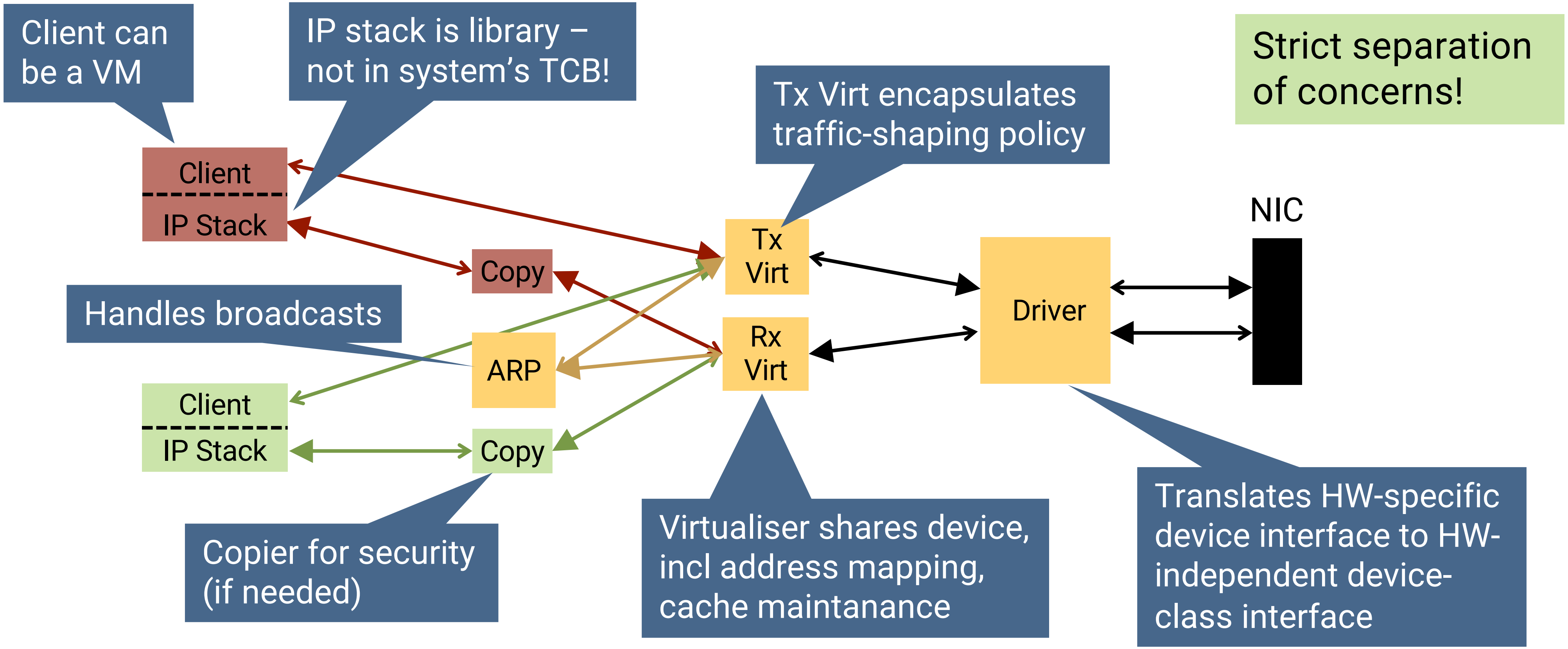
Use-case diversity by replacing components

LionsOS: Highly Modular System



Note:
static architecture ≠ static code!

Example: Networking Subsystem



Client can be a VM

IP stack is library – not in system's TCB!

Tx Virt encapsulates traffic-shaping policy

Strict separation of concerns!

Handles broadcasts

Copier for security (if needed)

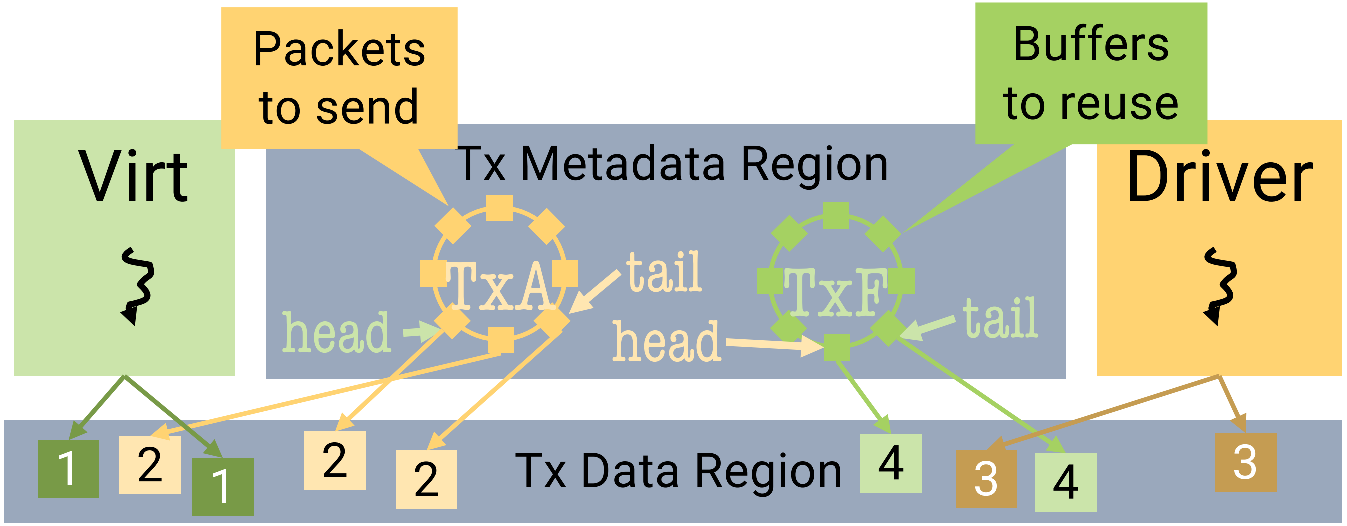
Virtualiser shares device, incl address mapping, cache maintanance

Translates HW-specific device interface to HW-independent device-class interface

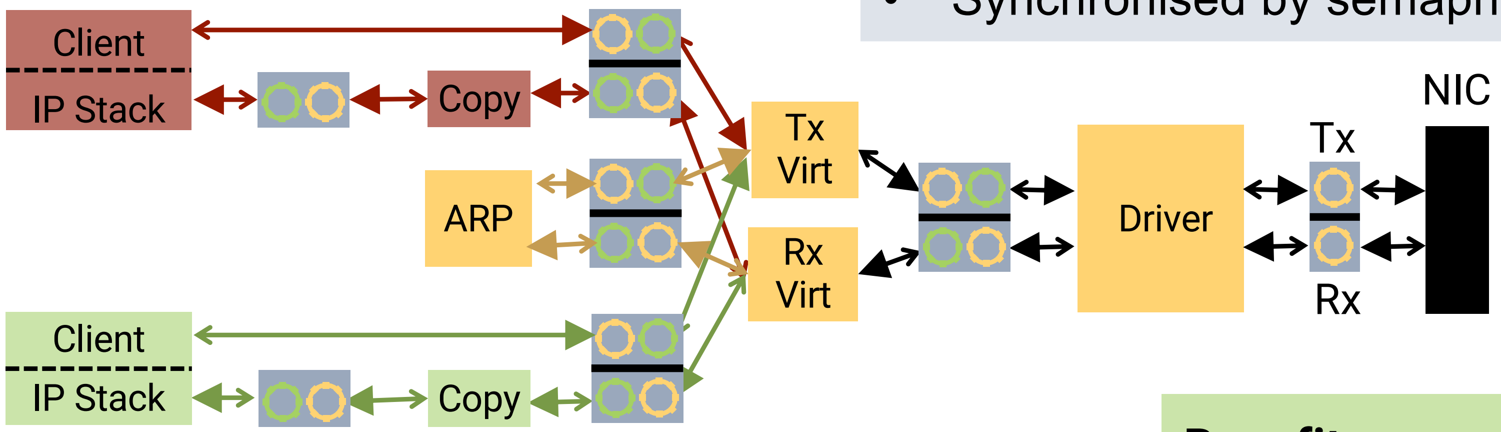
Zero-copy Data Transfer

Components are single-threaded – “Tamed” concurrency!

- Lock-free bounded queues
- Single producer, single consumer
- Similar to ring buffers used by NICs
- Synchronised by semaphores



Networking Detail



Zero-copy communication:

- Lock-free, single-producer, single-consumer, bounded queues
- Synchronised by semaphores

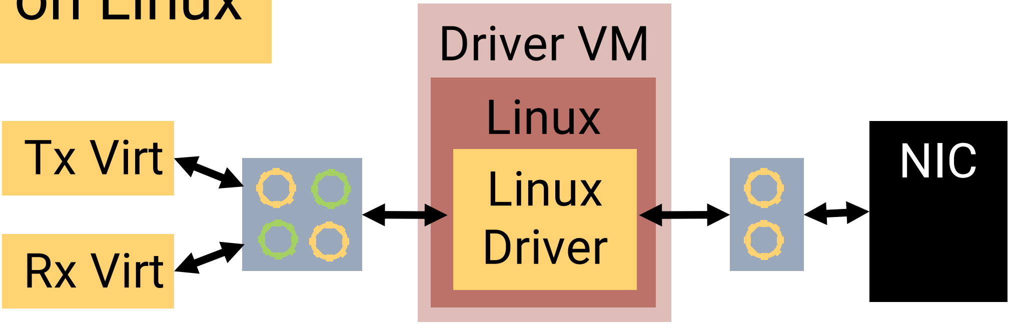
Benefits:

- simple components
- **location transparency**
- **suitable for verification**

Legacy Re-use: Driver VMs

Can re-use unmodified Linux drivers:

- Transparently use driver VM instead of native driver
- Linux app in VM uses UIO to communicate with in-kernel driver
- develop LionsOS components on Linux



Comparison to Linux on i.MX8M

Linux:
- NW driver: 3k lines
- NW system total: 1M lines

Performance?

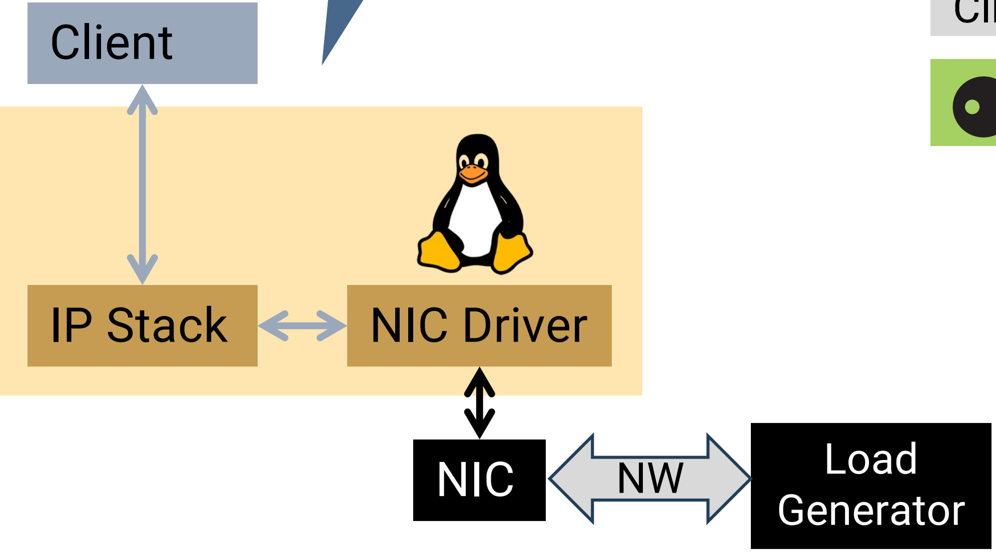
LionsOS:
- NW driver: 400 lines
- Virtualiser: 160 lines
- Copier: 80 lines
- IP stack: much simpler, client library
- shared NW system total < 1,000 lines

Written by second-year student!

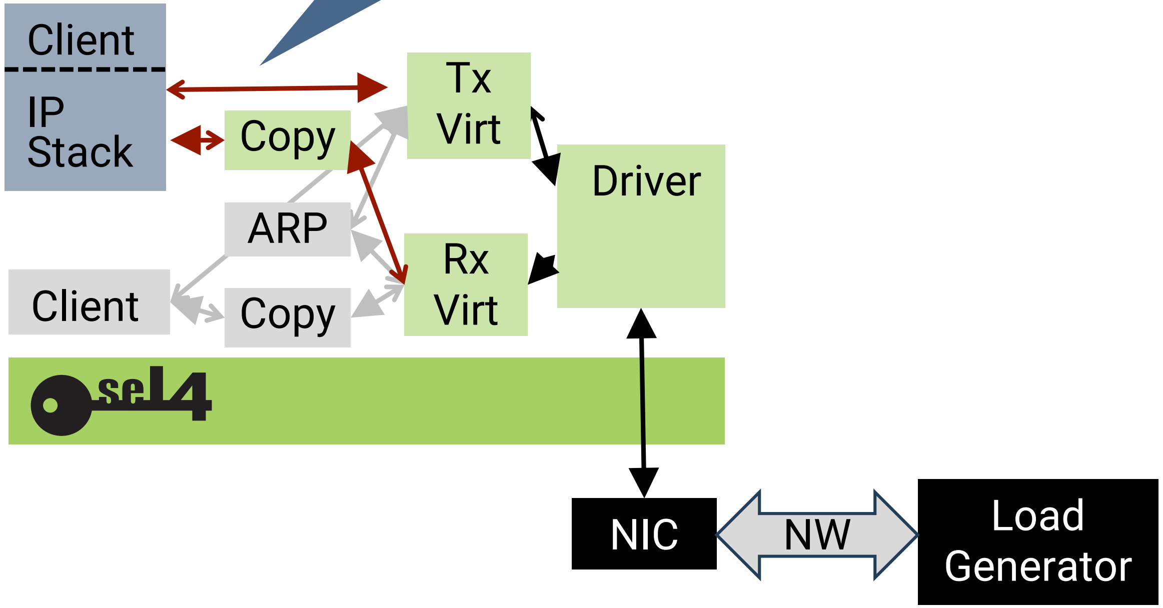
Presently use lwip

Evaluation Setup

Linux: 2 context switches per packet

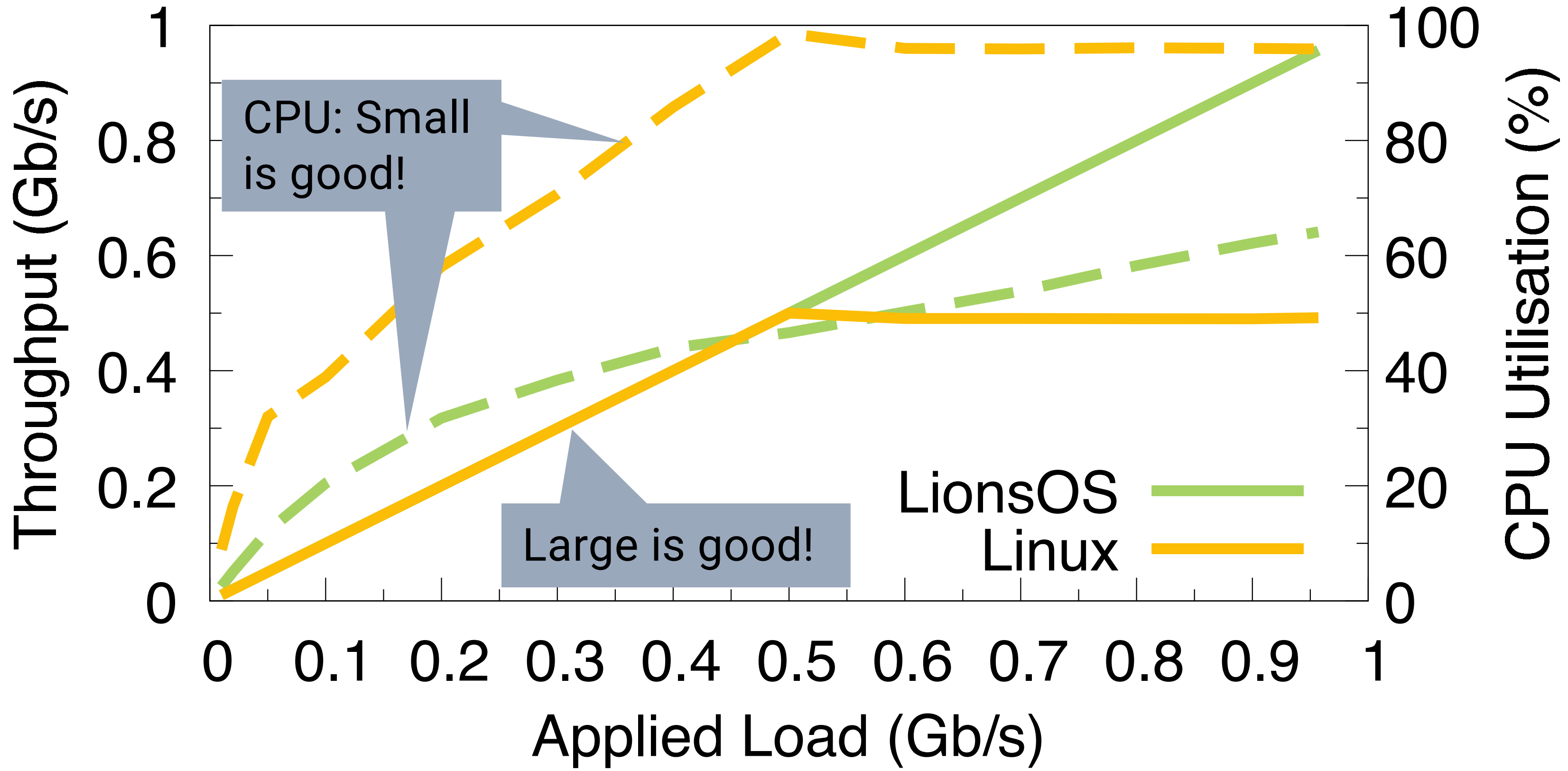


LionsOS: ~10 context switches per packet



- External load generator
- Measures throughput, latency
- Client echoes packets

Performance: i.MX8M, 1Gb/s Eth, UDP

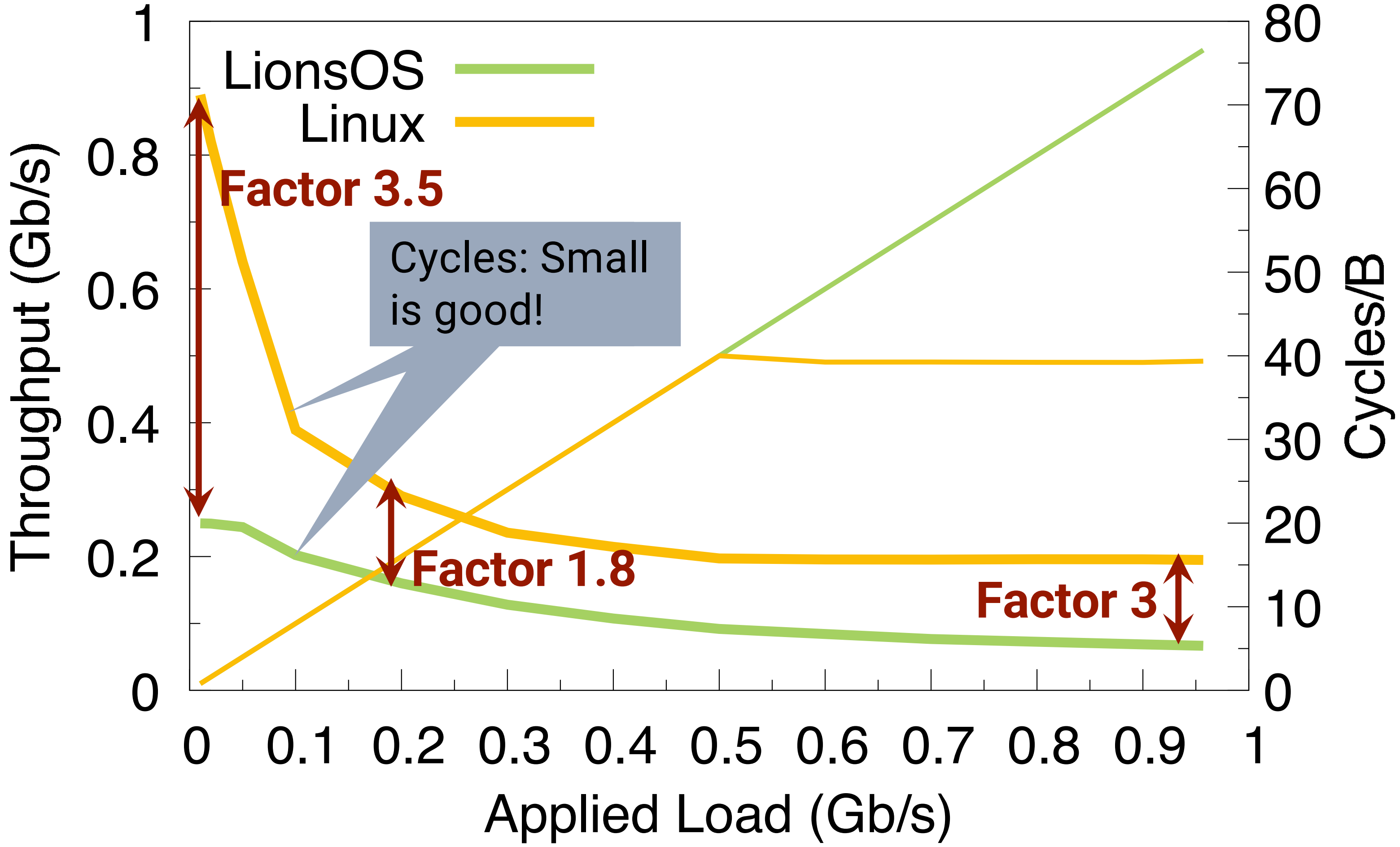


CPU: Small is good!

Large is good!

Single-core configuration

Performance: Processing Cost per Byte



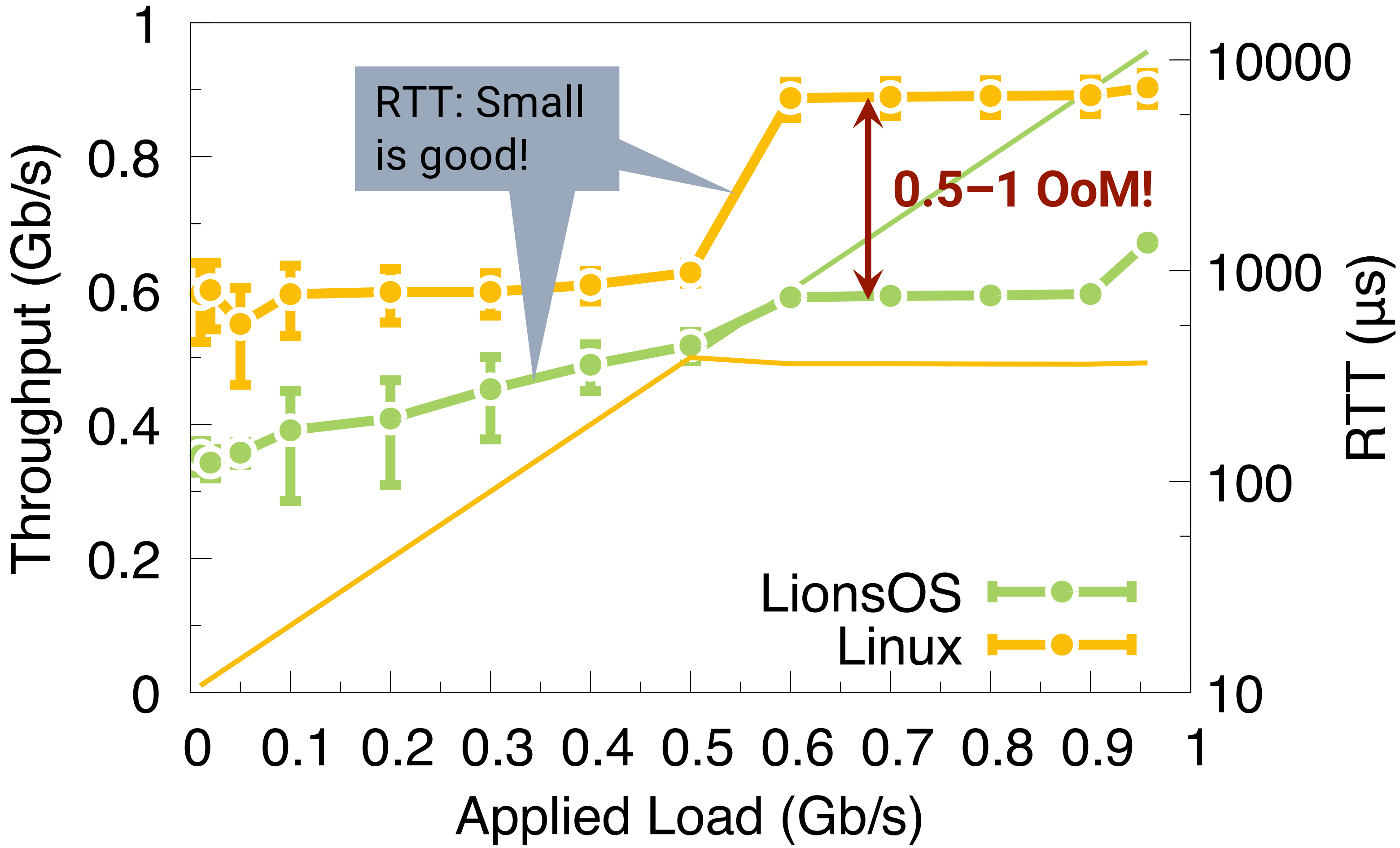
Factor 3.5

Cycles: Small is good!

Factor 1.8

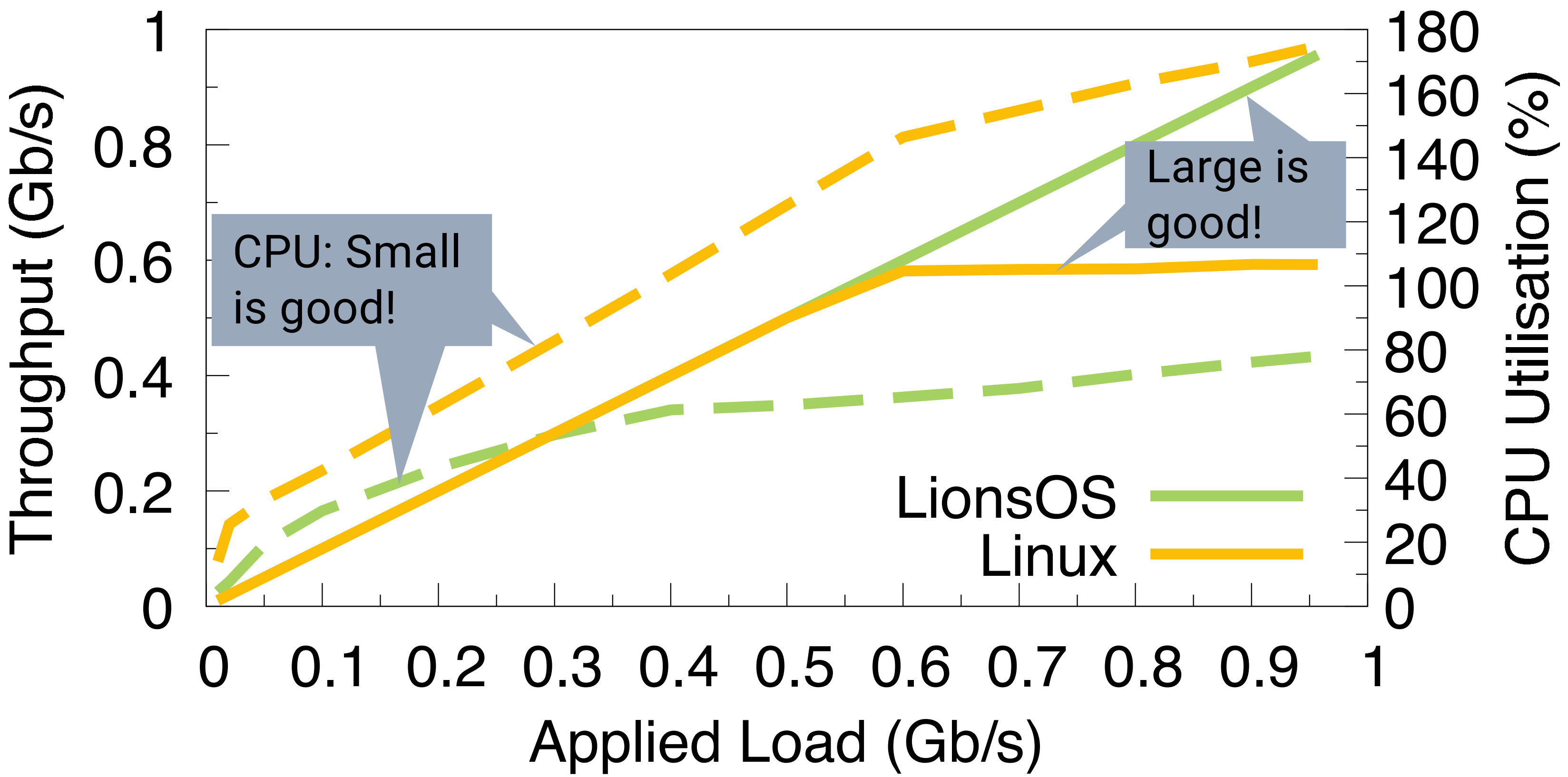
Factor 3

Performance: Round-Trip Times



RTT: Small is good!

Performance: i.MX8M, 1Gb/s Eth, UDP



CPU: Small is good!

Large is good!

Multicore configuration

Why This Difference?

Linux:
- NW driver: 3k lines
- NW system total: 1M lines

Simplicity Wins!

LionsOS executes less code!
➢ Direct consequence of use-case-specific policies!

LionsOS:
- NW driver: 400 lines
- Virtualiser: 160 lines
- Copier: 80 lines
- IP stack: much simpler, client library
- shared NW system total < 1,000 lines

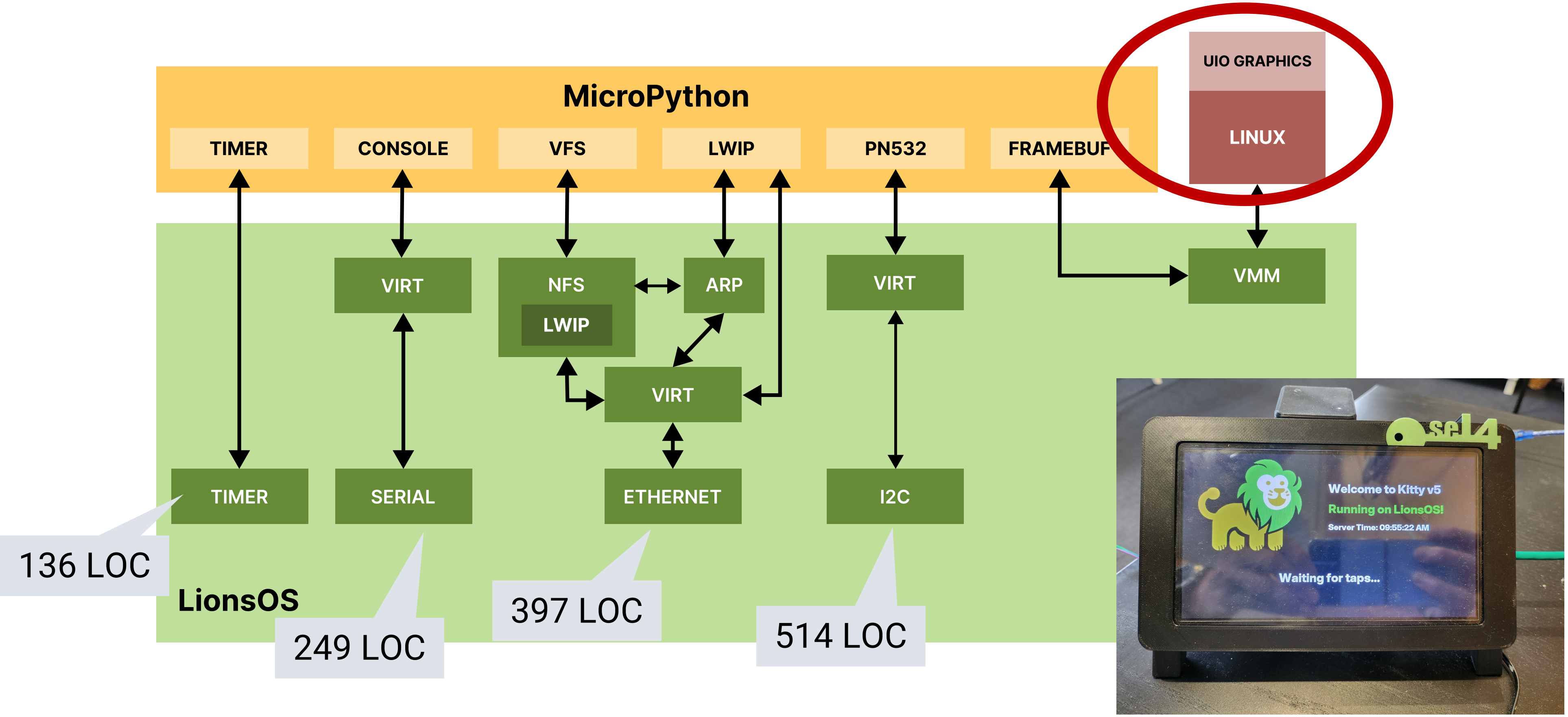
LionsOS Status

- Funding: DARPA, Cyberagentur
- Networking done
- Storage done-ish (per-client FAT file-system library)
- Framework for re-using Linux drivers (driver VMs)
- Sound, I^2C, hot-plugging close to merging
- Visual component editor & build tools
- Proof of concept of run-time policy replacement

To do:

- Run-time code-update framework
- Core management
- …

PoC: Point-of-Sale Terminal: “Kitty”



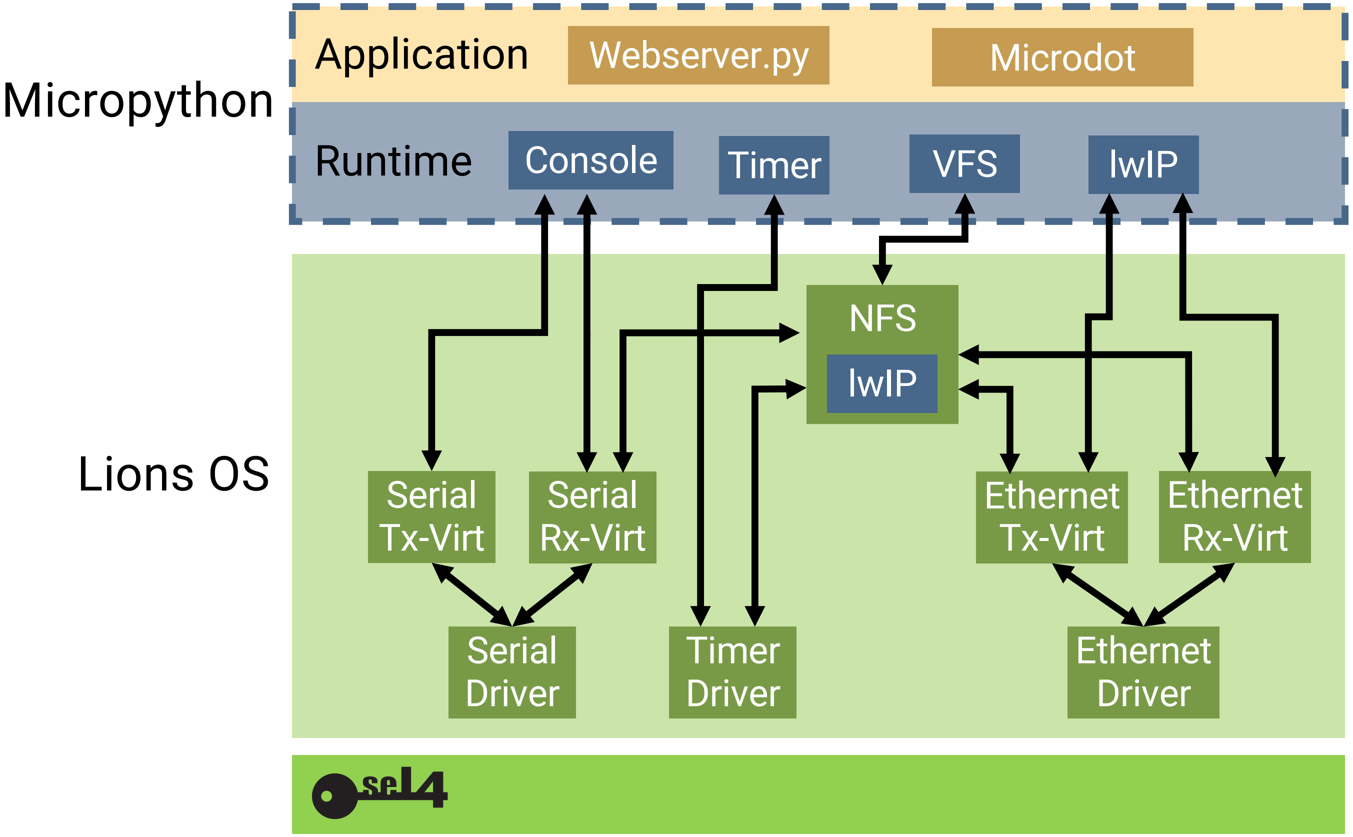
PoS LionsOS Code Sizes (all C)

Trusted:
- 15 modules/ libraries
- Av 210 LoC

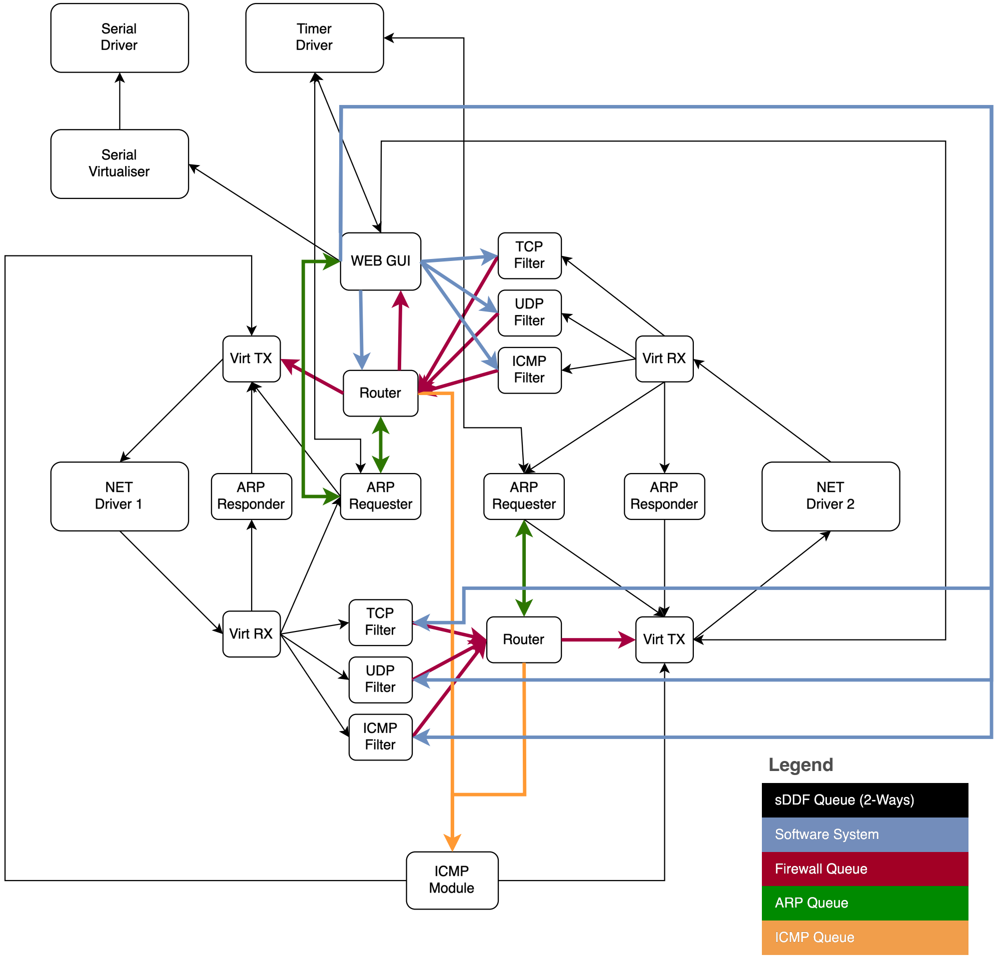
Component	LoC	Library	LoC
Serial Driver	249	Microkit	303
Serial Tx Virt	175	Serial queue	219
Serial Rx Virt	126	I^{2}C queue	101
I^{2}C Driver	514	Eth queue	140
I^{2}C Virt	154	Filesys queue & protocol	268
Timer Driver	136		
Eth Driver	397	Coroutines	848
Eth Tx Virt	122	**LWIP**	**16,280**
Eth Rx Virt	160	**NFS**	**45,707**
Eth Copier	79	**VMM**	**3,098**

Untrusted

Underneath https://sel4.systems/

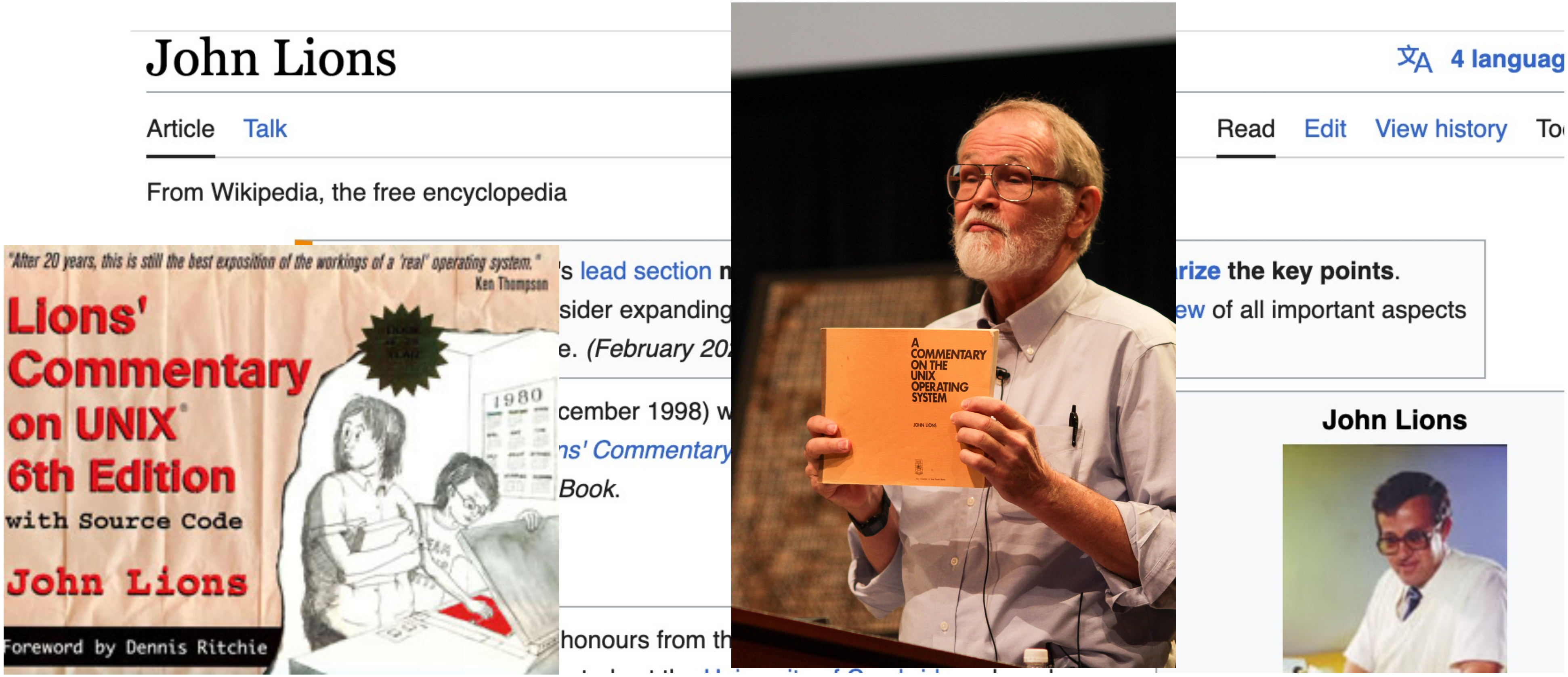


A Firewall


Serial Driver
Timer Driver
Serial Virtualiser
WEB GUI
TCP Filter
UDP Filter
ICMP Filter
Virt TX
Router
Virt RX
NET Driver 1
ARP Responder
ARP Requester
ARP Requester
ARP Responder
NET Driver 2
Virt RX
TCP Filter
UDP Filter
ICMP Filter
Router
Virt TX
ICMP Module
Legend
sDDF Queue (2-Ways)
Software System
Firewall Queue
ARP Queue
ICMP Queue

What’s In a Name?

- https://en.wikipedia.org/wiki/John_Lions



John Lions

文A 4 languag

Article Talk

Read Edit View history To

From Wikipedia, the free encyclopedia

's lead section m … rize the key points.

sider expanding … ew of all important aspects

e. (February 20

cember 1998) w

ns' Commentary

Book.

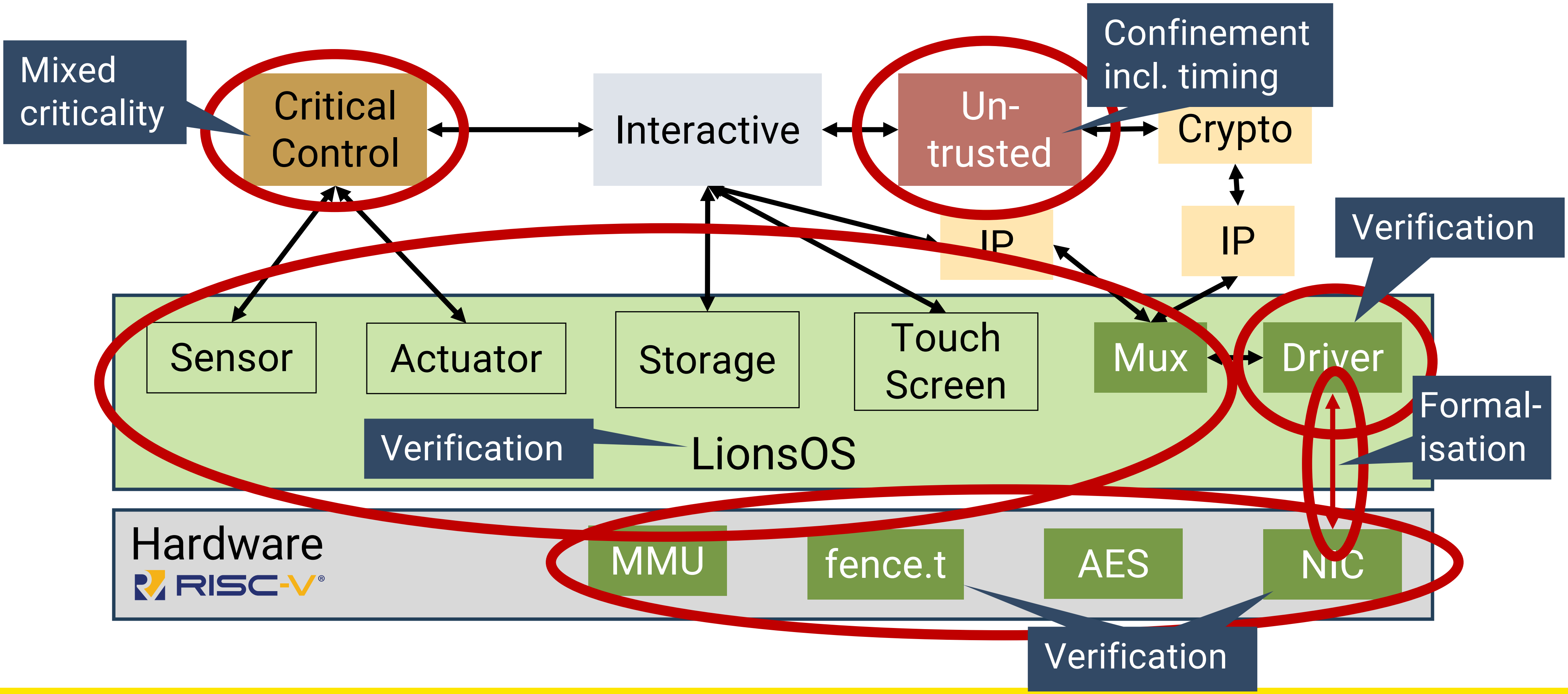
honours from th

John Lions

"After 20 years, this is still the best exposition of the workings of a 'real' operating system." Ken Thompson
Lions' Commentary on UNIX 6th Edition with Source Code
John Lions
Foreword by Dennis Ritchie
1980
A COMMENTARY ON THE UNIX OPERATING SYSTEM
JOHN LIONS

How About Verification?

Agenda for Next 3 Years



Verifying LionsOS – How?

- LionsOS programming model:
  - simple event handlers
  - strictly sequential code

Very little time spent on debugging component logic

Suitable for SMT solvers

- Fine-grained modularity:
  - concurrency by distribution, “tamed” concurrency
  - complex signalling protocols

Protocol bugs are mostly performance problems

Ideal for model checking!

Automatic proofs!

Challenge:
composition of proofs

Device Driver Dilemma

seL4 is one-off, justifies cost

High seL4 verification costs partially due to C language

Drivers are commodity, must be cheap!

Drivers are low-level, need C-like language

Better language would reduce cost

Idea:

1. Simplify drivers
2. Use verification-friendly systems language
3. Automate (part of) verification

LionsOS

- Well-defined semantics
- Memory-safe

- Verified compiler
- de-compilation

Verifying Systems Code

Problem:

- C semantics is complex and ambiguous
- Verifying C code is expensive

How about Rust?

- Strong type safety helps avoiding bugs

But:

- No agreed language semantics
- Huge trusted computing base
  - compiler
  - run time
- Interfacing to hardware needs “unsafe” escapes

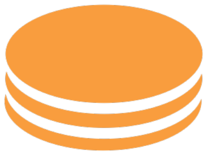
Rust is no help in achieving end-to-end verification!

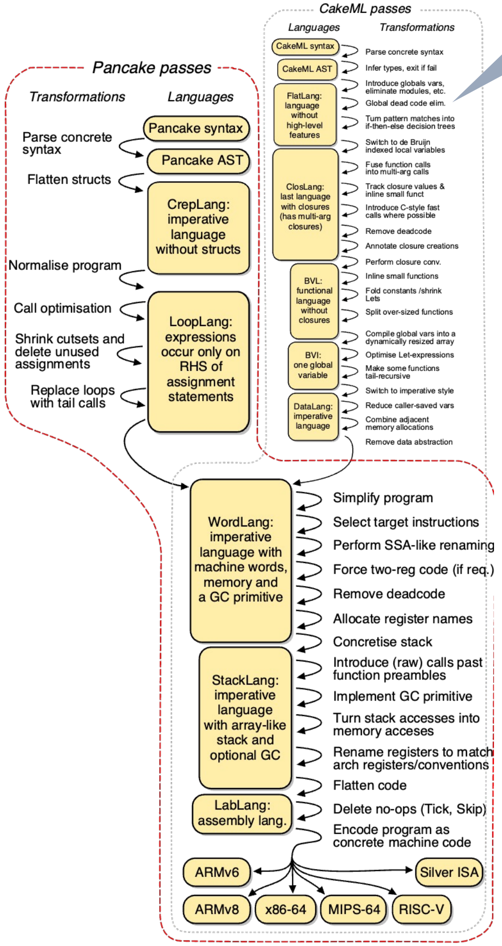
PANCAKE

A Language for Verified Systems Programming

Approach:

- Re-use lower part of CakeML compiler stack
- Get verified Pancake compiler quickly
- Retain mature framework/ecosystem





Pancake passes
Transformations
Languages
Pancake syntax
Parse concrete syntax
Pancake AST
Flatten structs
CrepLang: imperative language without structs
Normalise program
Call optimisation
Shrink cutsets and delete unused assignments
Replace loops with tail calls
LoopLang: expressions occur only on RHS of assignment statements
CakeML passes
Languages
Transformations
CakeML syntax
Parse concrete syntax
CakeML AST
Infer types, exit if fail
Introduce globals vars, eliminate modules, etc.
FlatLang: language without high-level features
Global dead code elim.
Turn pattern matches into if-then-else decision trees
Switch to de Bruijn indexed local variables
ClosLang: last language with closures (has multi-arg closures)
Fuse function calls into multi-arg calls
Track closure values & inline small funct
Introduce C-style fast calls where possible
Remove deadcode
Annotate closure creations
Perform closure conv.
BVL: functional language without closures
Inline small functions
Fold constants /shrink Lets
Split over-sized functions
Compile global vars into a dynamically resized array
BVI: one global variable
Optimise Let-expressions
Make some functions tail-recursive
Switch to imperative style
DataLang: imperative language
Reduce caller-saved vars
Combine adjacent memory allocations
Remove data abstraction
WordLang: imperative language with machine words, memory and a GC primitive
Simplify program
Select target instructions
Perform SSA-like renaming
Force two-reg code (if req.)
Remove deadcode
Allocate register names
Concretise stack
Introduce (raw) calls past function preambles
StackLang: imperative language with array-like stack and optional GC
Implement GC primitive
Turn stack accesses into memory acceses
Rename registers to match arch registers/conventions
Flatten code
LabLang: assembly lang.
Delete no-ops (Tick, Skip)
Encode program as concrete machine code
ARMv6
Silver ISA
ARMv8
x86-64
MIPS-64
RISC-V

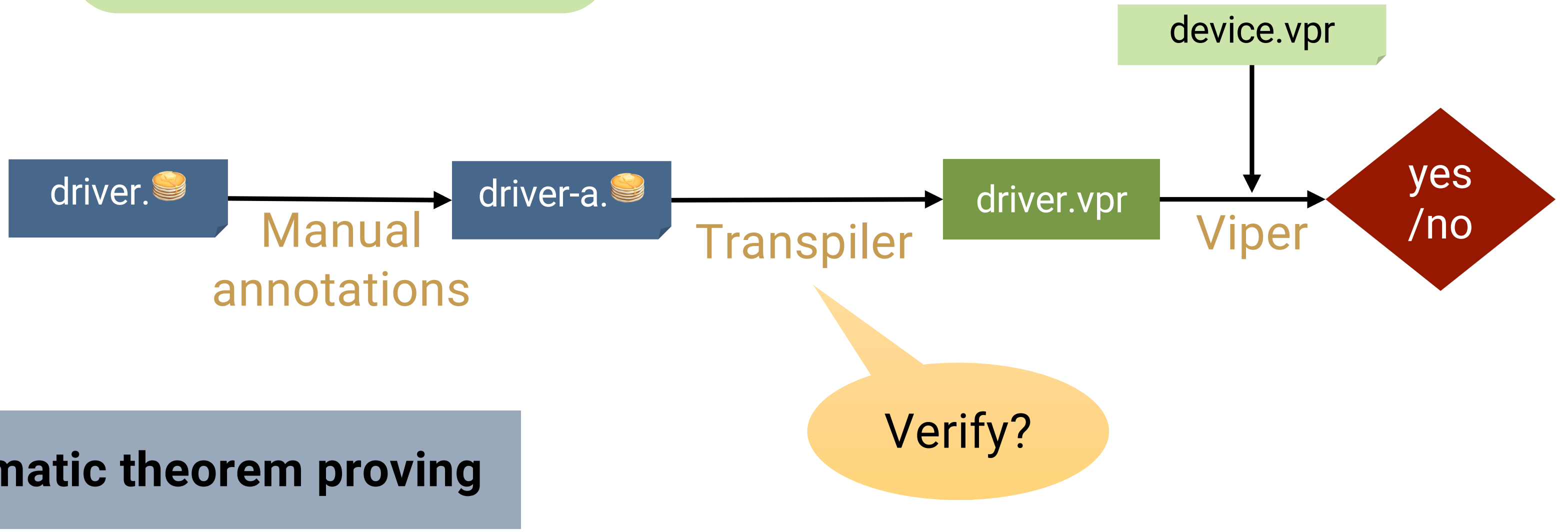
CakeML

Verifying OS Components with Pancake

First success:
High-performance
Ethernet driver
verified!

Also looking at:

- Verified de-compilation to higher level
- plus interactive theorem proving



device.vpr

driver.🥞

Manual annotations

driver-a.🥞

Transpiler

driver.vpr

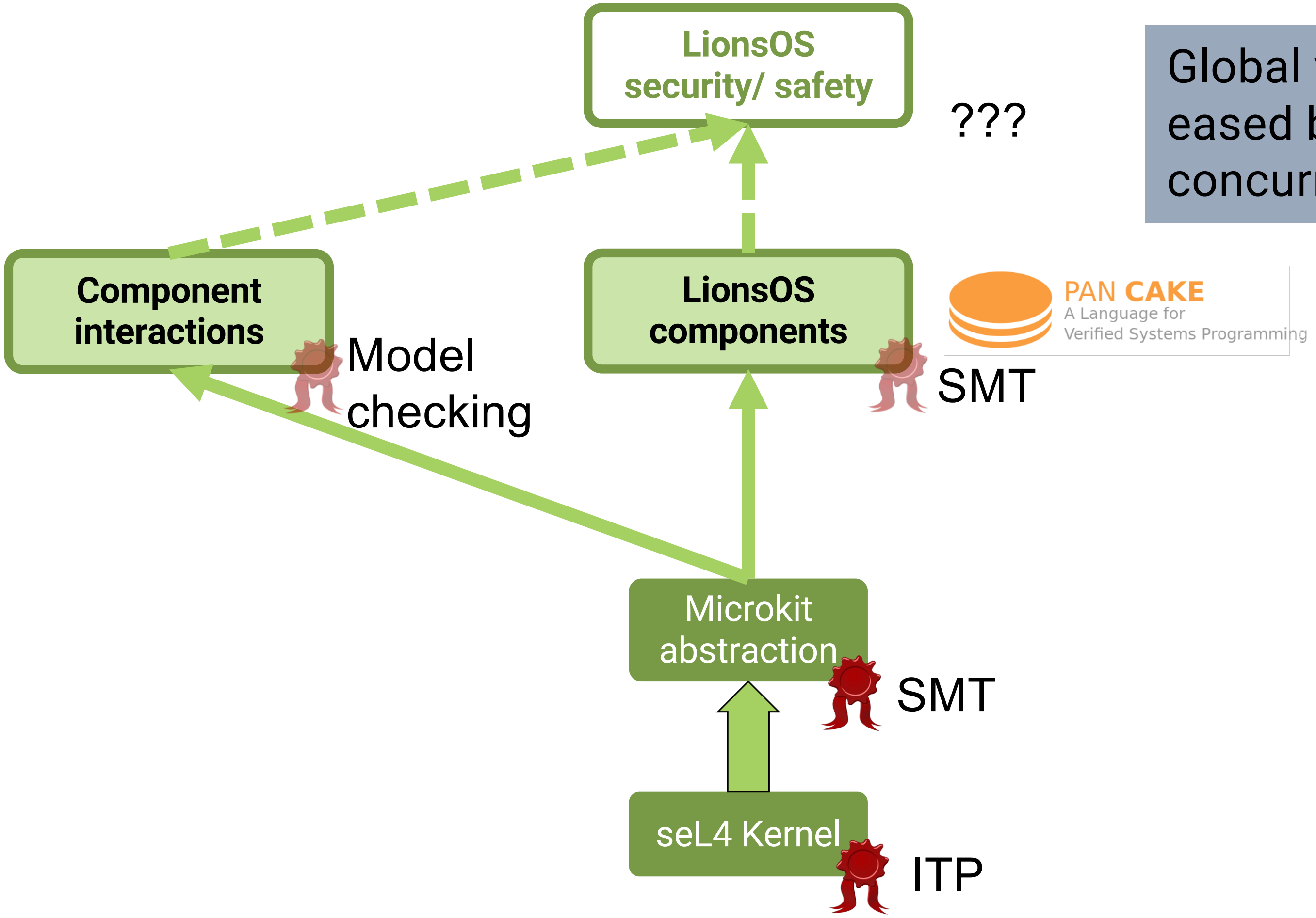
Viper

yes /no

Verify?

Automatic theorem proving

Verifying Lions OS

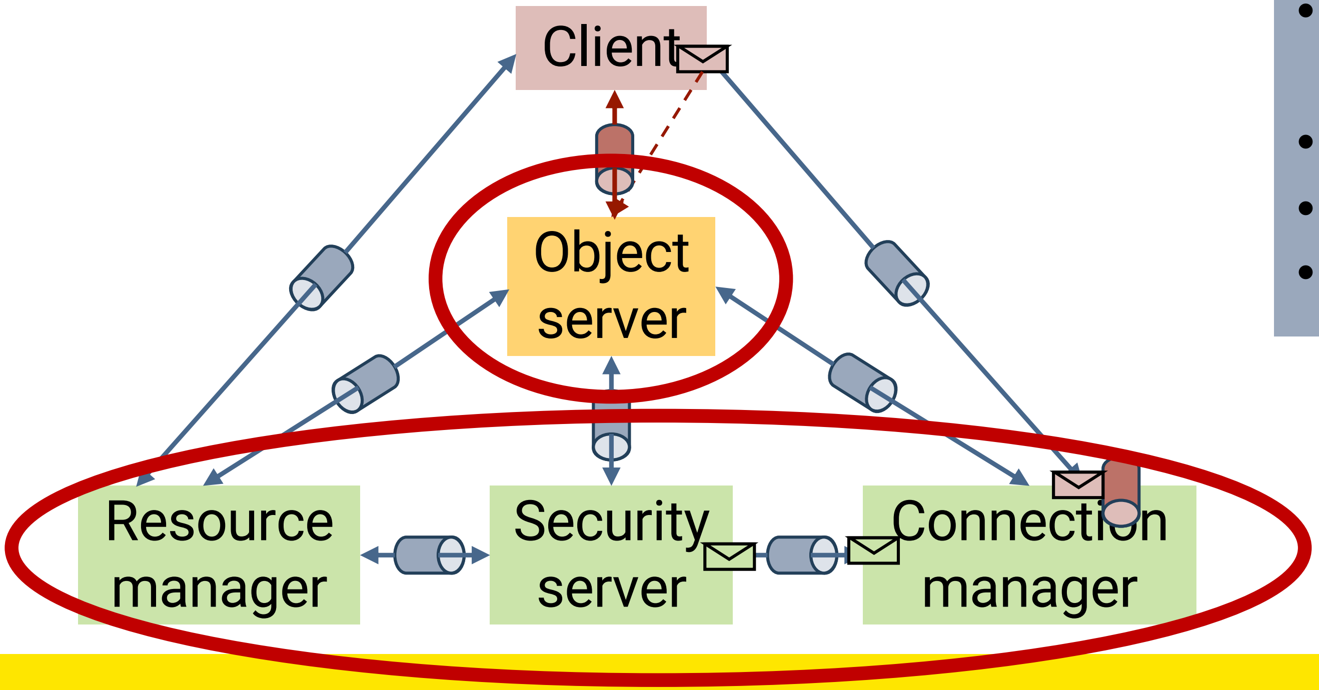


Global verification eased by “tamed” concurrency

Looking ahead:
Provably secure general-purpose OS

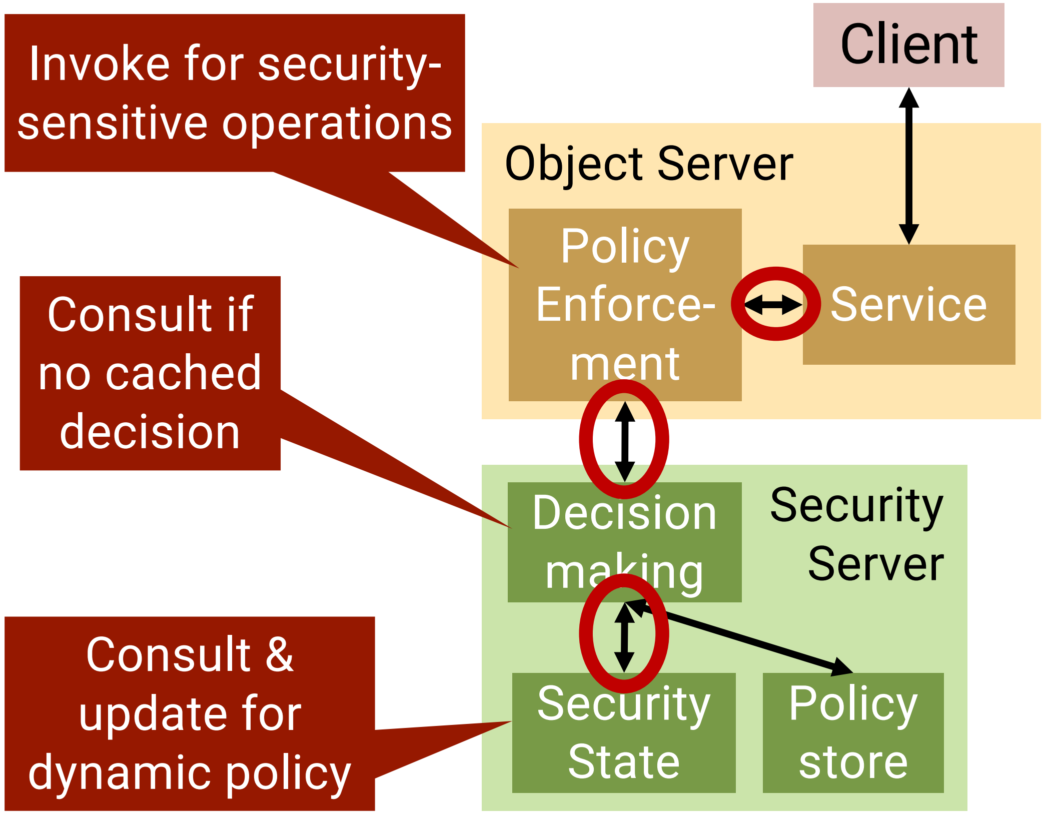
Beyond LionsOS: General Purpose OS

Aim: General-purpose OS that **provably** enforces a general security policy

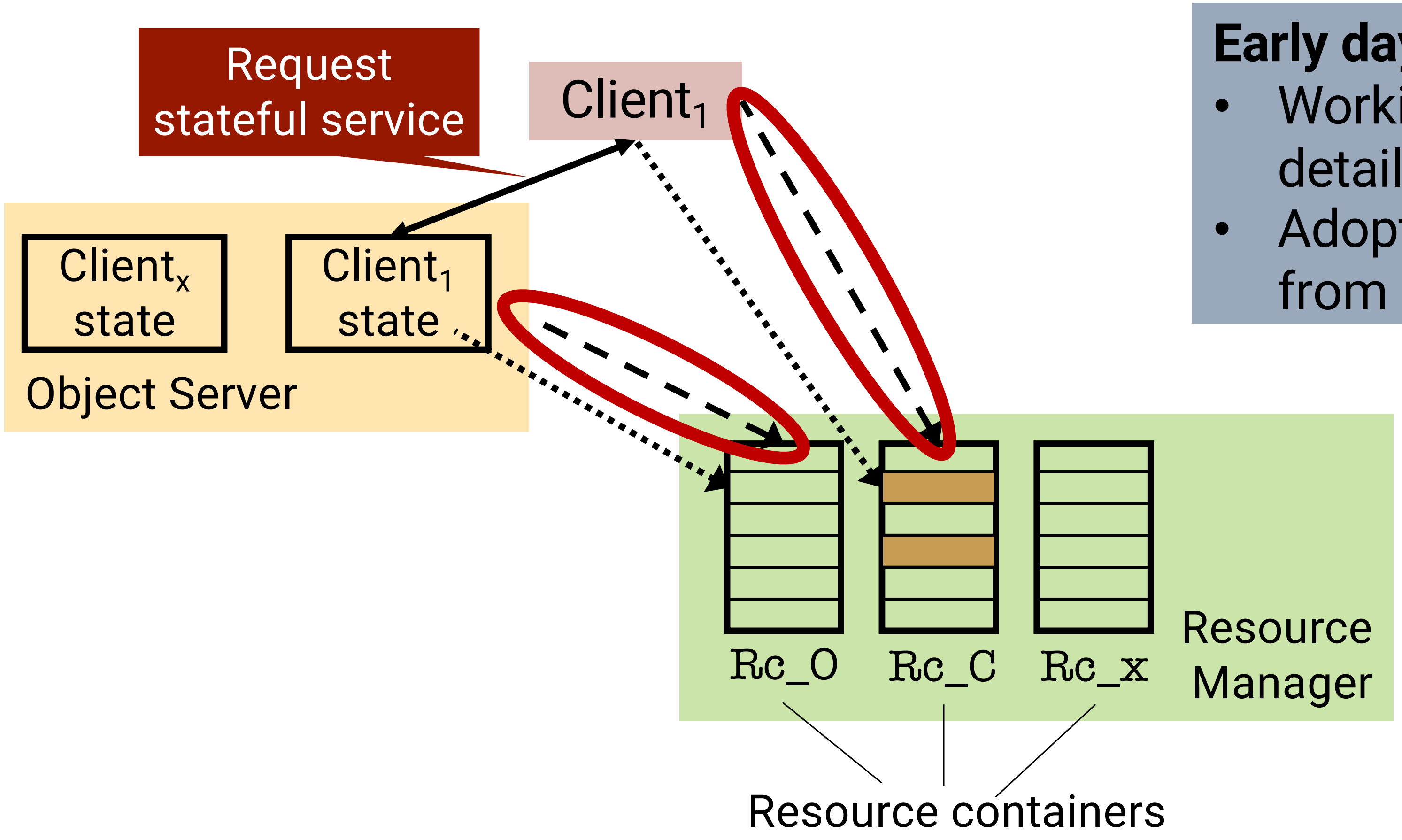


Requires:
- mandatory security-policy enforcement
- Security-policy diversity
- minimal TCB
- low-overhead enforcement

Core Ideas: Dynamic Enforcement


Invoke for security-sensitive operations
Consult if no cached decision
Consult & update for dynamic policy
Client
Object Server
Policy Enforce-ment
Service
Decision making
Security Server
Security State
Policy store

Core Ideas: Resource Donation



Early days:

- Working on framework, details of model
- Adopt components from LionsOS

Truly Secure OSes – Finally Happening?

LionsOS

LionsOS:

- Highly performant
- First components verified
- 3-Year plan for end-end proofs
- Limited to static architectures

General-purpose OS:

- Very early days
- … but optimism from LionsOS experience

Security is no excuse for bad performance!

https://trustworthy.systems

We’re hiring!
Operating-systems researchers