

**Georg Lauterbach**

Faculty of Computer Science. Institute of System Architecture. Chair of Operating Systems.

# Enforcing Integrity and Software Fault Isolation in Microkernels with CHERI

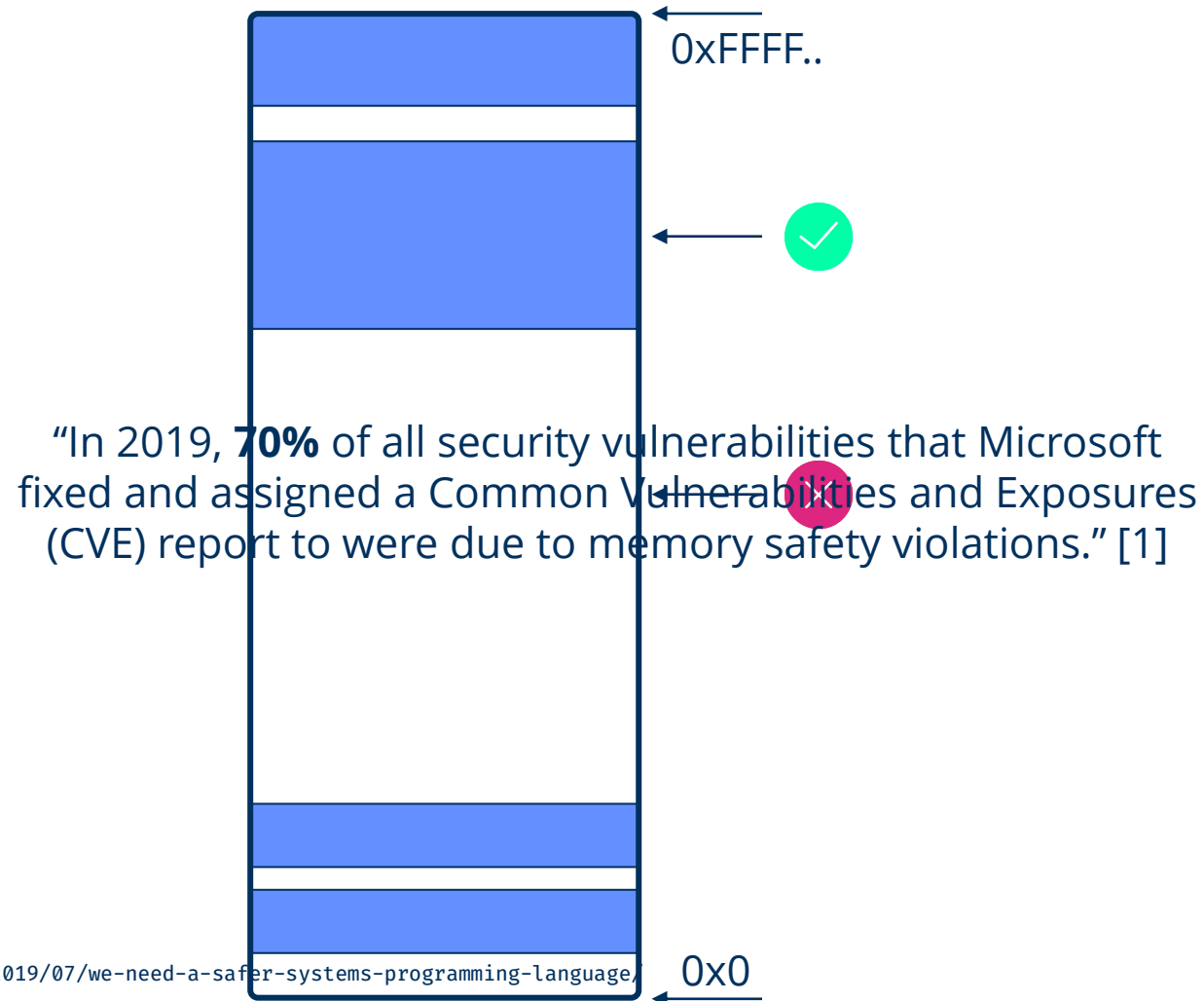
Reviewer Prof. Dr.-Ing. Horst Schirmeier [TUD] & Dr.-Ing. Nils Asmussen [TUD]

Supervisor Bohdan Trach (Ph.D.) [HUAWEI DRC]

Date & Time Dresden // 27 September 2023

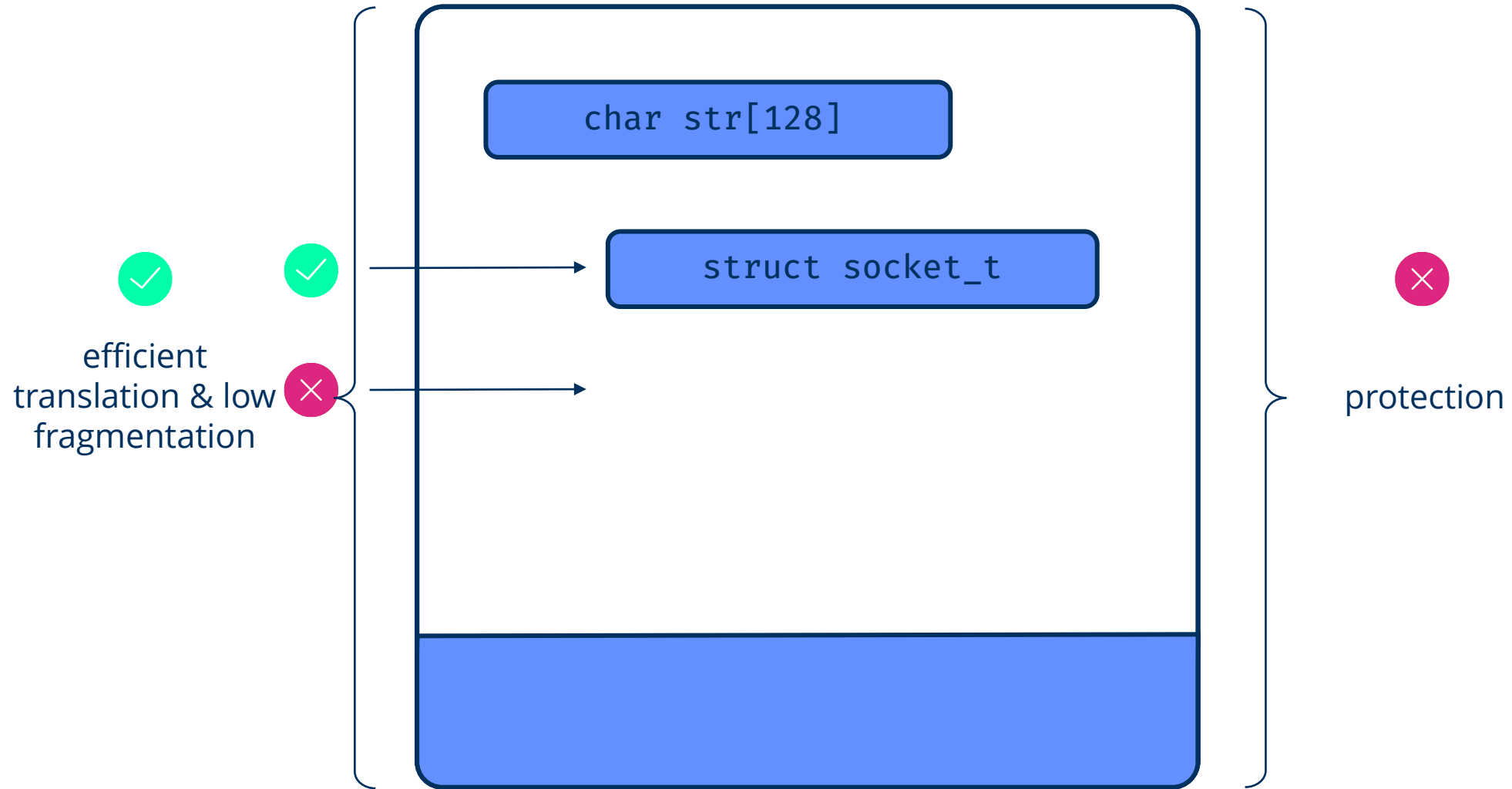
# Motivation and Technical Background

# Motivation and Technical Background

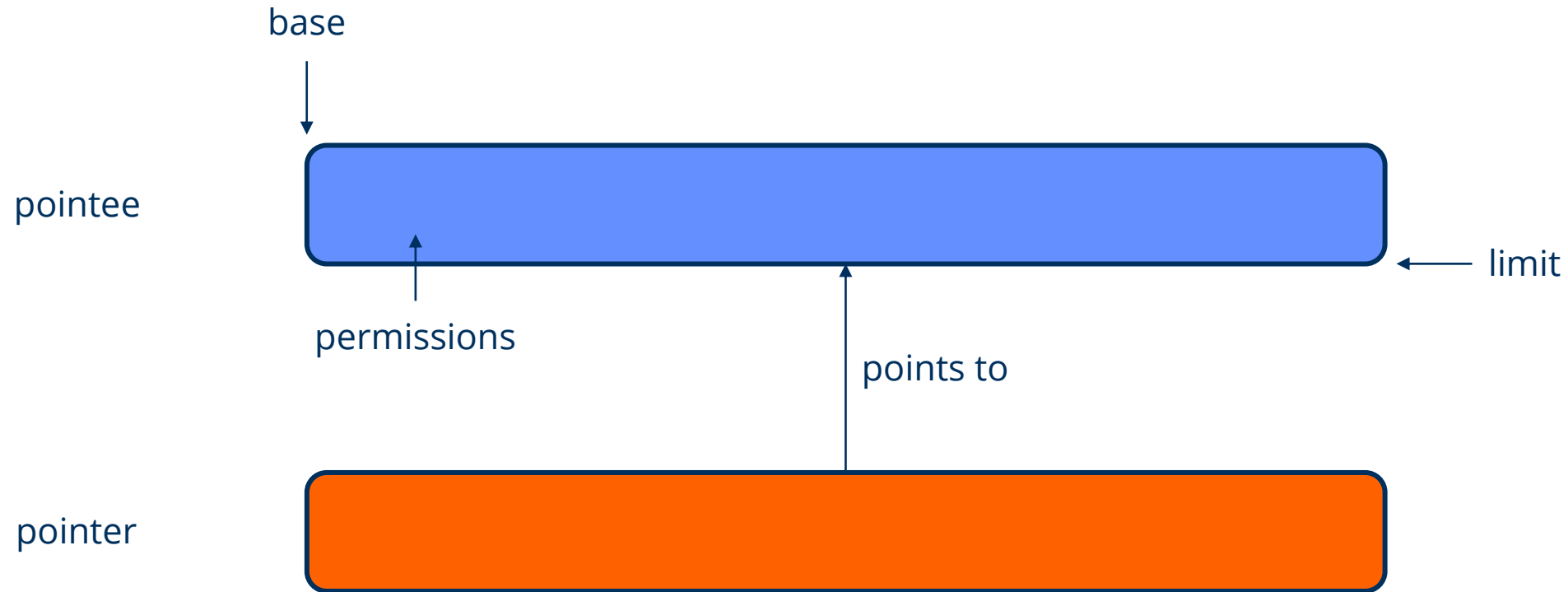


[1] <https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language>

# Motivation and Technical Background



# Motivation and Technical Background



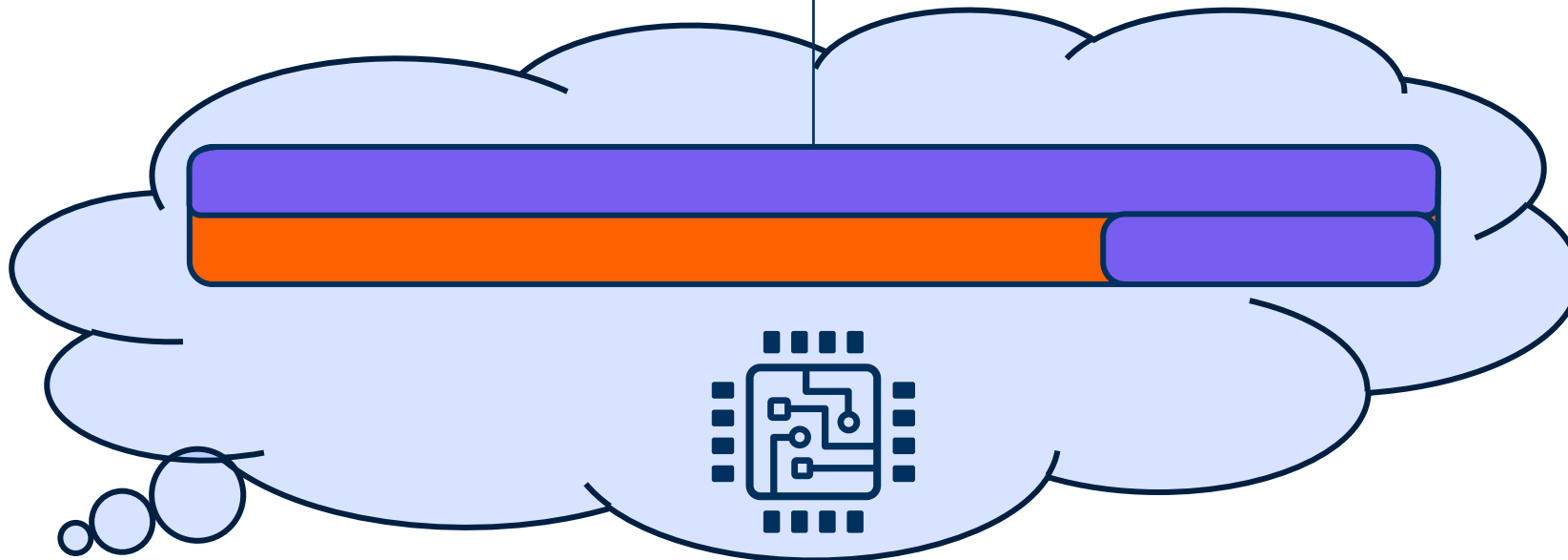
# Motivation and Technical Background

**provenance:** consistency, only data from trusted sources  
**monotonicity:** only data from trusted sources  
**integrity:** only data from trusted sources  
**valid capabilities:** only data from trusted sources

pointee

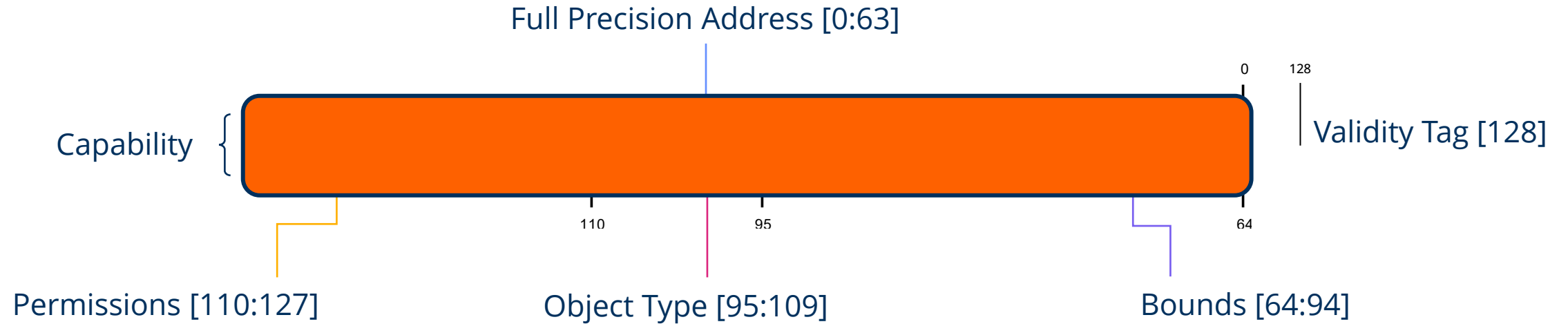


pointer



# Motivation and Technical Background

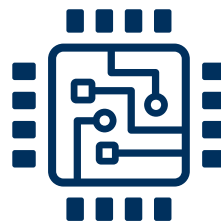
## CHERI



Capability usage rules:

1. Monotonicity
2. Integrity
3. Provenance

enforced in



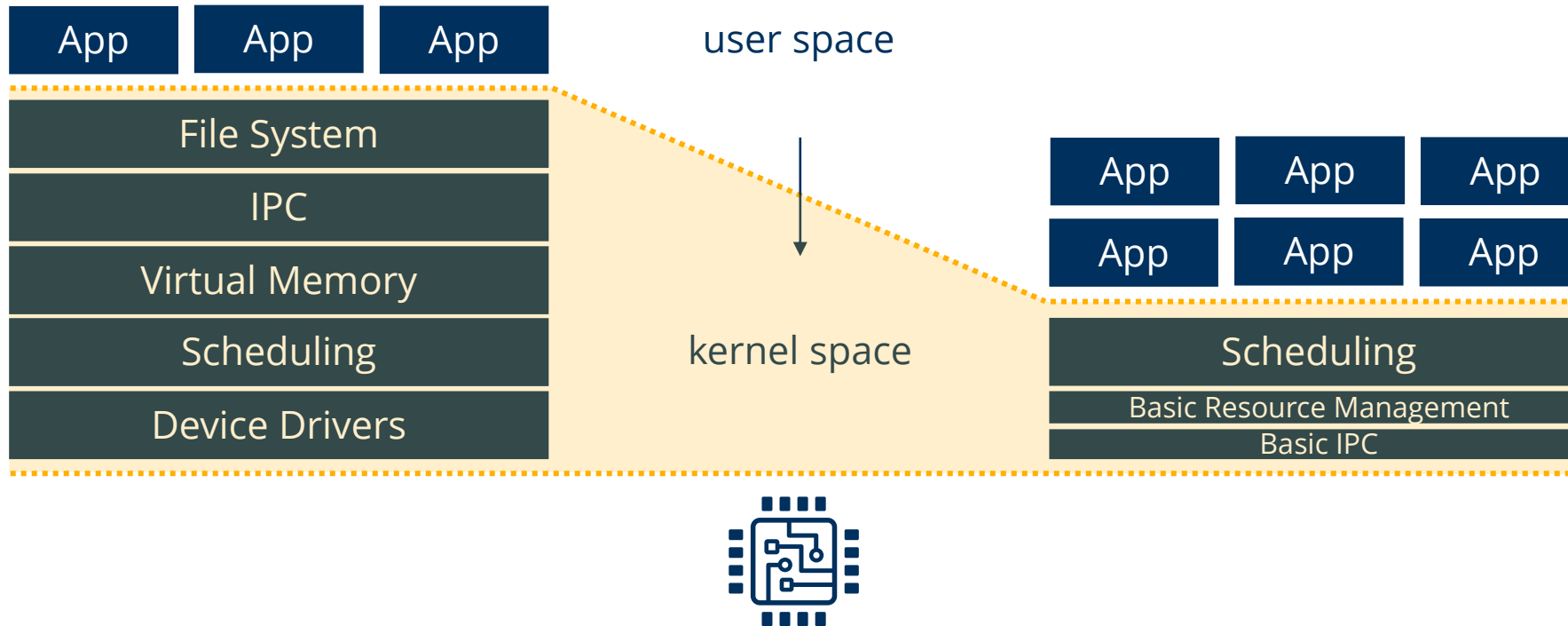
on top of



# Motivation and Technical Background

Monolithic Kernel

Microkernel





Motivation and Technical Background

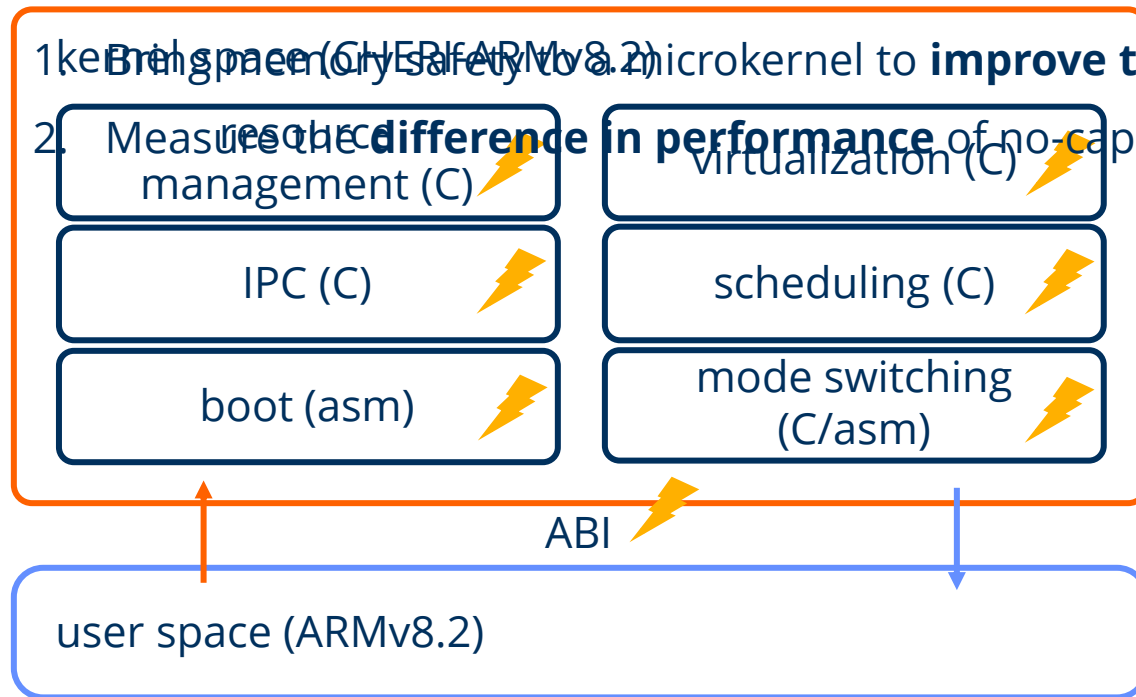
# Goals, Concept and Implementation

Evaluation

Conclusion and Discussion

Summary

# Goals and Concept



specific assembly

capability and capability extended registers

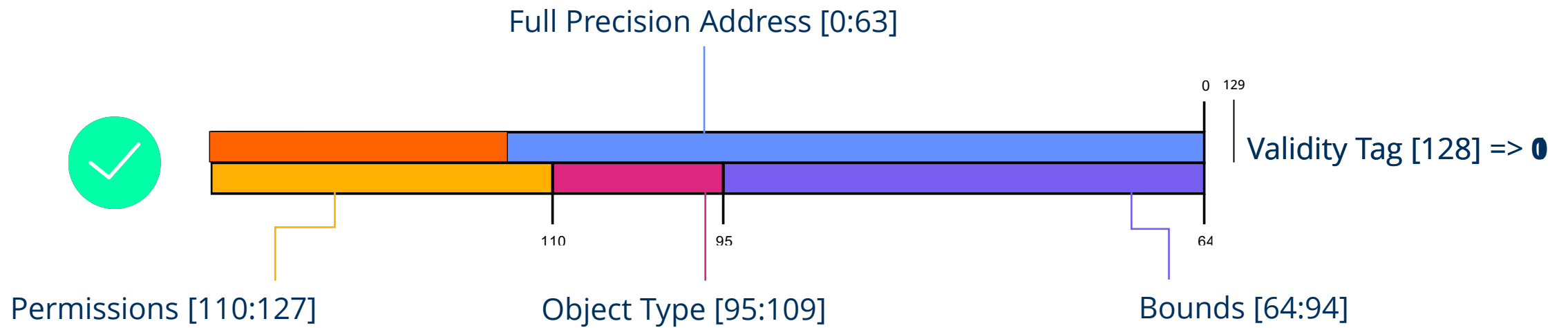
- ABI incompatibility
- types
- alignment
- offsets
- re-derivations (data structures not composing well with CHERI)

Special purpose registers: e.g. Stack Pointer  $SP \rightarrow R13$  Capability Stack Pointer  $CSP$

General purpose registers  $\times 32 \rightarrow$  Capability general purpose registers  $C0..C31$

New instructions => e.g. `scbnds` for setting `address_of(void *) = ptraddr_t == size_t`

# Re-Derivations



# Setting the Kernel Stack Pointer – A Comparison

```
ldr    X19, =__entry_stack_base
```

```
mov    SP, X19
```

---

[V] [bounds] X19

[V] [bounds] X20

```
ldr    X19, =__entry_stack_base
```

[0] [no] \_\_entry\_stack\_base

```
mov    X20, KERN_STACK_SIZE
```

[0] [no] \_\_entry\_stack\_base

[0] [no] KERN\_STACK\_SIZE

```
cvtd   C19, X19
```

[1] [no] \_\_entry\_stack\_base

[0] [no] KERN\_STACK\_SIZE

```
sub    X19, X19, X20
```

[1] [no] \_\_entry\_stack\_base - KERN\_STACK\_SIZE [0] [no] KERN\_STACK\_SIZE

```
scbnds C19, C19, X20
```

[1] [yes] \_\_entry\_stack\_base - KERN\_STACK\_SIZE [0] [no] KERN\_STACK\_SIZE

```
add    CSP, C19, X20
```

[1] [yes] \_\_entry\_stack\_base - KERN\_STACK\_SIZE [0] [no] KERN\_STACK\_SIZE

Motivation and Technical Background

Goals, Concept and Implementation

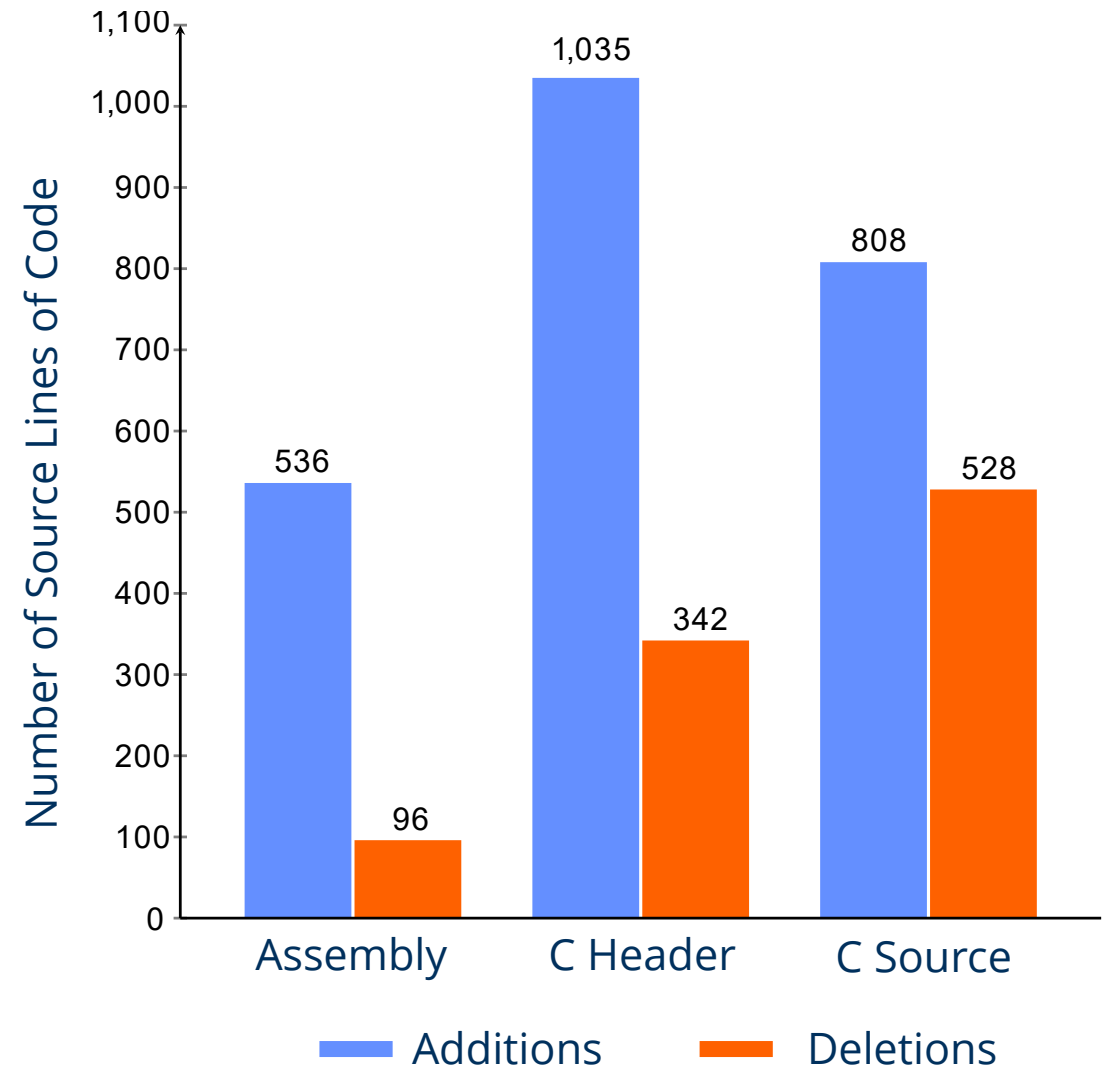
# Evaluation

Conclusion and Discussion

Summary

# The Porting Effort


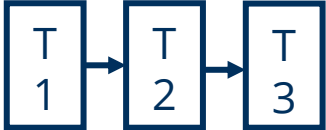

- **many trivial** changes (type issues, casts)
- **few difficult** issues (offsets in assembly, re-derivations for MMIO)
- modest changes in terms of SLoC: **4% changed** | **6% added**
- certain core microkernel services **disproportionally affected**
- intensive debugging required
- density of changes depend on **idiosyncrasy** (correct semantics for pointer types) of code



# New Fault Isolation Properties

Property	ARMv8.2-A	CHERI-ARMv8.2-A	Both
hardware-enforced memory safety	none	<b>spatial &amp; referential</b>	no temporal
compartmentalization of code	none	at level of individual C-language objects	
provoking faults unrelated to memory safety			not caught
provoking memory safety-related faults	may raise exceptions, mostly unrelatable to actual error	always raised exceptions	
miscellaneous	Rust?	<b>escape hatches exist</b> (due to re-derivations), but could be removed	

# Performance Measurements

- real hardware called *Morello*: Neoverse N1 (7nm) @ 2.5GHz running ARMv8.2-A (aarch64 only)
- two benchmark configurations
  - No-Capability: ARMv8.2 without CHERI capabilities
  - Capability: CHERI-ARMv8.2 with all pointers being CHERI capabilities
- three **main micro-benchmarks**
  1. IPC: sending “ping-pong” messages between two tasks 
  2. Resource management: delegating memory recursively between multiple “threads” 
  3. Scheduling: run many “threads” and (re-)scheduled them while they are working 
- in-depth investigations for IPC
  1. Mode switching: going to the kernel and back again
  2. Extended IPC benchmark: evaluating overhead of re-derivations

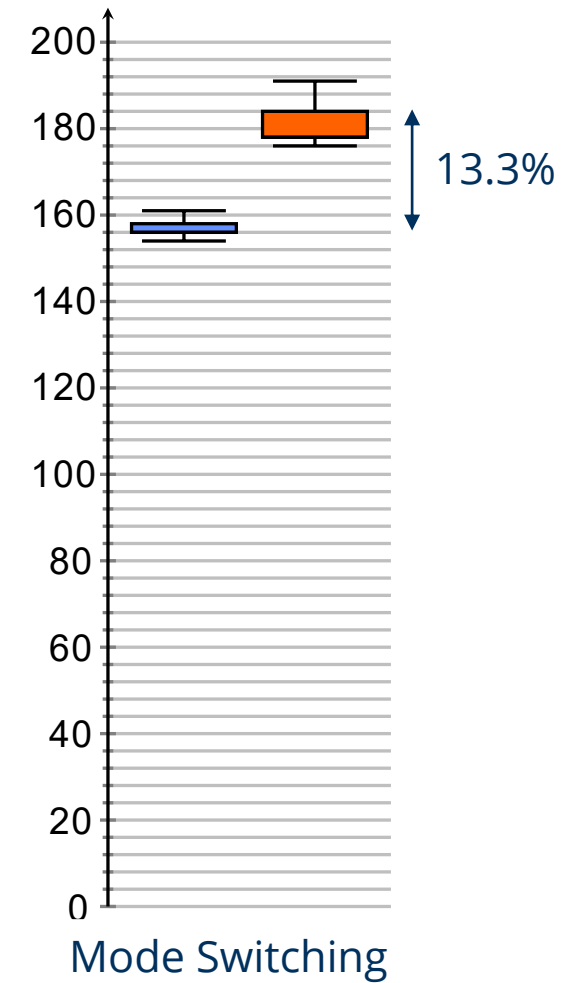
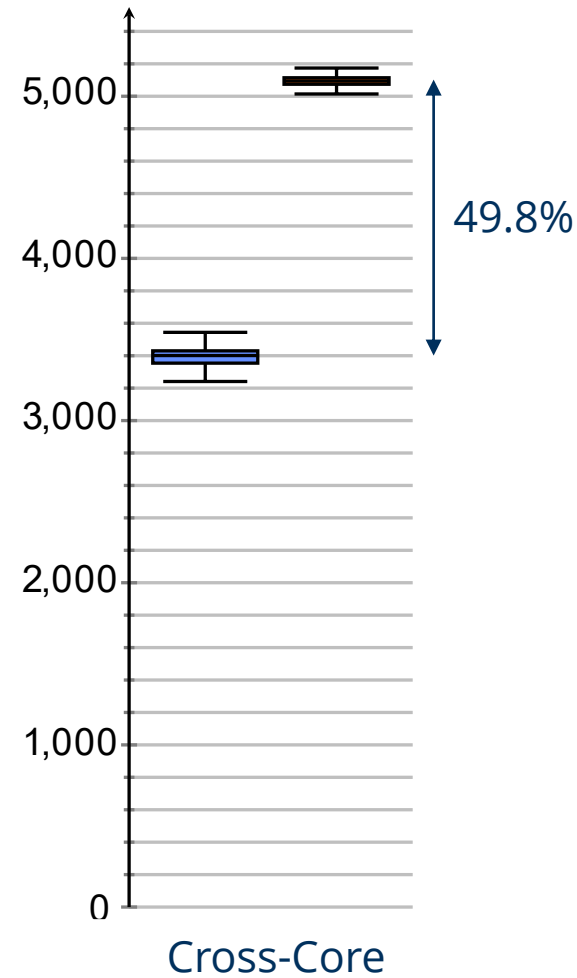
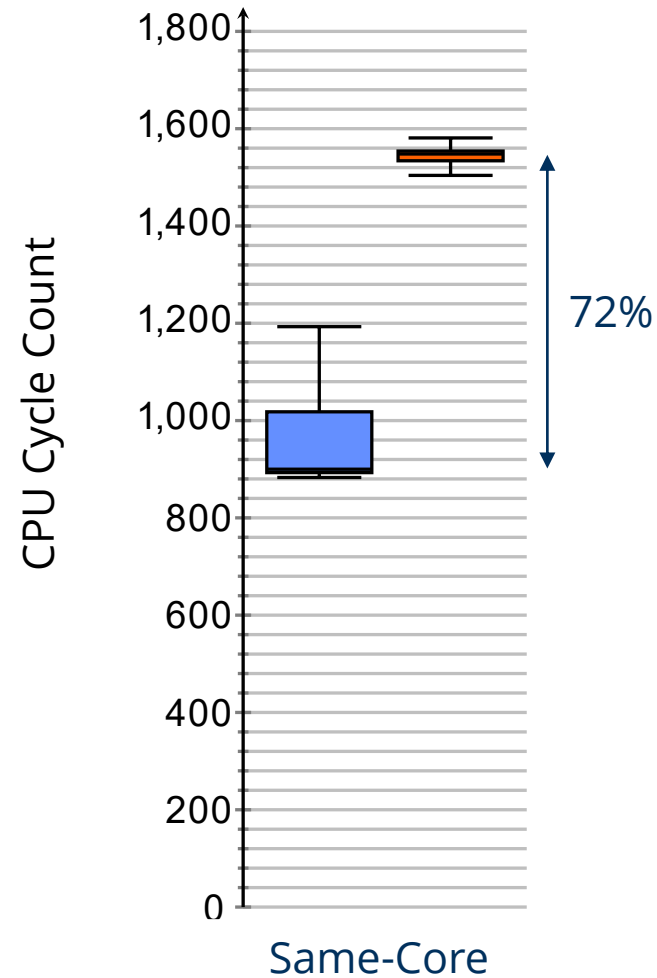


# IPC Benchmark Results

— No-Capability

— Capability

CPU Cycle Count: lower is better

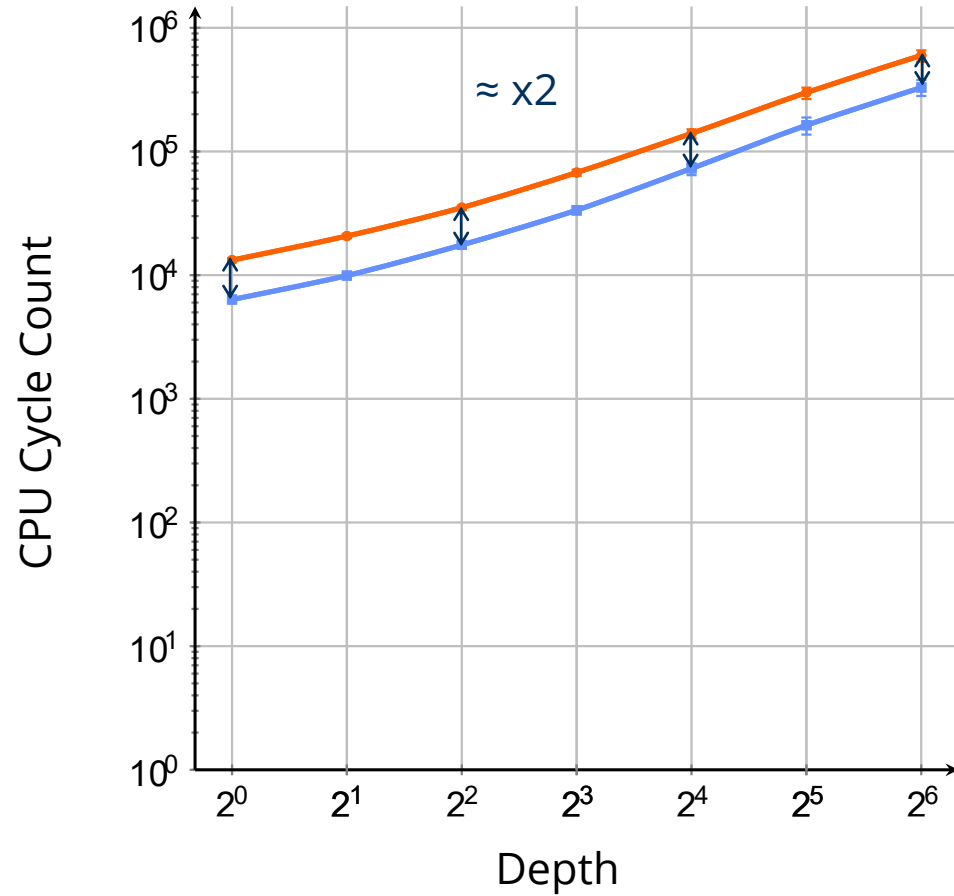


# Resource Management Benchmark Results

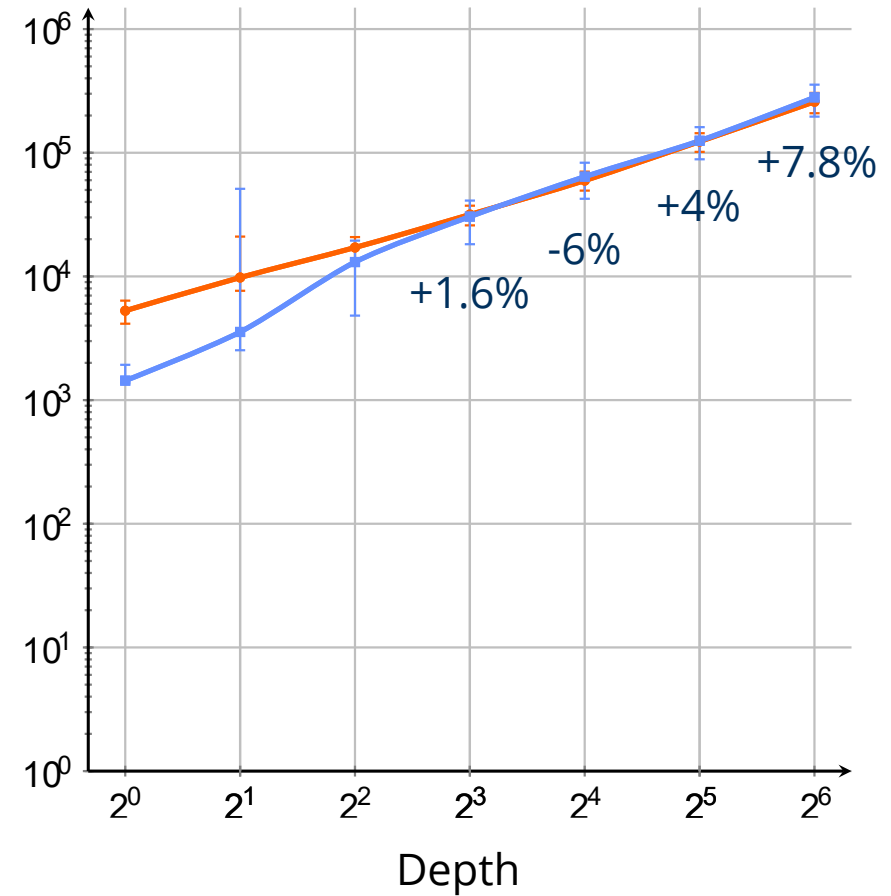
— No-Capability

— Capability

CPU Cycle Count: lower is better



Delegation



Revocation

Motivation and Technical Background

Goals, Concept and Implementation

Evaluation

# Conclusion and Discussion

Summary

# Conclusion and Discussion

- adoption of CHERI is **feasible** for microkernels but still **requires non-trivial efforts**
  - certain code disproportionately affected
  - documentation is scattered, examples are scarce
  - main factors for porting: data types and idiosyncrasy of code
- **notable performance degradations** experienced
  - no production-grade optimization of Morello microarchitecture yet
  - likely when structures do not compose well with CHERI
  - further investigations required
  - measure other overheads too (memory, energy consumption, etc.)
  - future work involving object capabilities
- fault isolation **significantly improved**
  - memory-safety related faults are caught
  - unit and integration tests still required

Motivation and Technical Background

Goals, Concept and Implementation

Evaluation

Conclusion and Discussion

# Summary

# Summary

1. Adoption for microkernels is **feasible** but requires **non-trivial efforts**
2. Fault isolation properties **significantly improved**
3. Currently and under specific circumstances:  
**notable performance degradations**
4. There is a lot of future work

Motivation and Technical Background

Goals, Concept and Implementation

Evaluation

Conclusion and Discussion

Summary

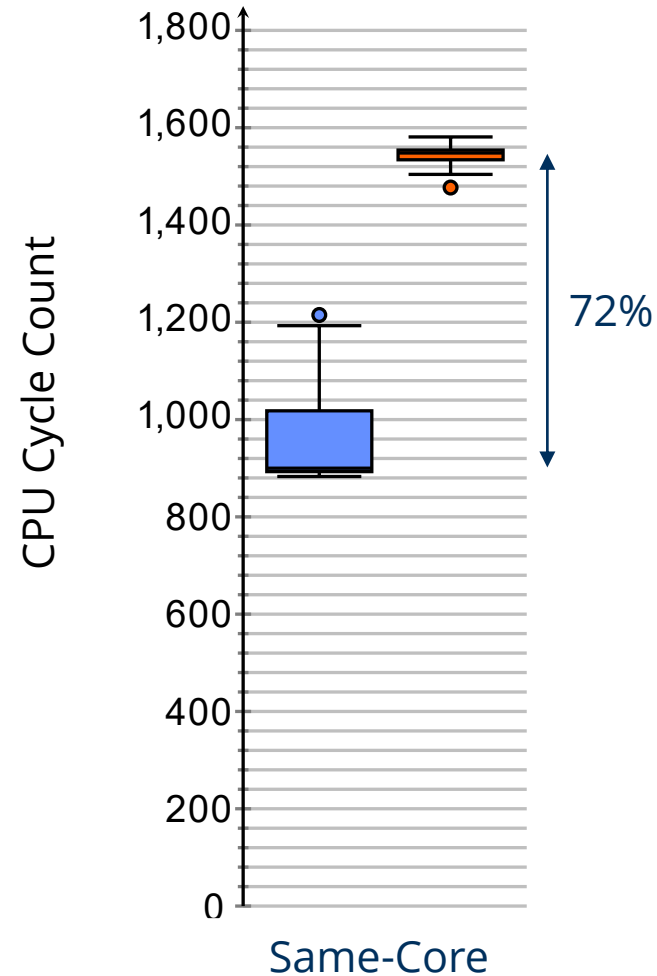
# Backup Slides

# Detailed IPC Benchmark Results

— No-Capability

— Capability

CPU Cycle Count: lower is better



- No-Capability
  - 2.4% (23) of cycles for capability-lookups
  - CPI: 0.73
- Capability
  - 17.8% (280) cycles for capability-lookup
  - 2 re-derivations per lookup on average
  - CPI: 1.04



# Other Approaches

1. **Mondrian** supplements page tables by word-granular in-memory “protection tables”, which contain permissions managed by a supervisor. Mondrian requires no user space ISA changes but instead relies on a supervisor mode to maintain the protection tables, which, in turn, requires a domain switch for each allocation and free event.
2. **Hardbound** is a hardware-assisted fat-pointer model that is rooted in software bounds-checking. But Hardbound's pointers are forgeable.
3. **Intel MPX** provides hardware-assisted bounds checking similar to Hardbound, but with important differences: bounds are atomically propagated, there is no compression, the tables are hierarchical, and transactional memory is required.
4. The **M-Machine** is a 64bit tagged-memory capability system design using guarded pointers to implement fine-grained memory protection for memory safety with almost zero ABI compatibility.
5. **Singularity** is an OS developed by Microsoft Research employing so-called Software Isolated Processes.
6. (K)ASan: address sanitizers are mostly debugging tool and probabilistic.

# A More Complicated Example

```
#include <arch/cheri/generic.h>

typedef void (*fn_any_t)(void);

static void *THE_ALMIGHTY_CAP = ...;

void * __attribute__((always_inline)) _cheri_unseal_sentry_cap(fn_any_t *const fn) {
    return cheri_unseal(fn, cheri_address_set(THE_ALMIGHTY_CAP, 0x1));
}

extern fn_any_t exception_vector_table_el1;

void exception_vector_init(void) {
    void *vector_table_el1_ptr = _cheri_unseal_sentry_cap(&exception_vector_table_el1);
    CHERI_REDERIVE(vector_table_el1_ptr, 0xffff00000000)
    write_to_reg__vbar_el1(vector_table_el1_ptr);
}
```

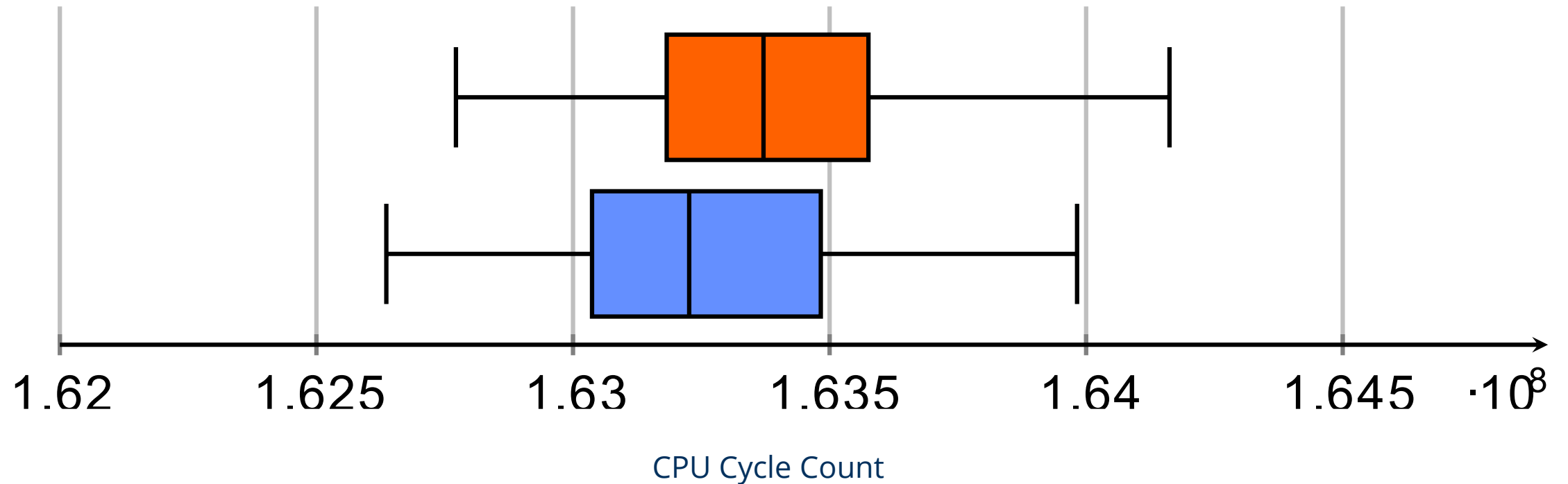
---

# Scheduling Benchmark Results

— No-Capability

— Capability

CPU Cycle Count: lower ( ← ) is better



# Future Work

1. Improved capability fault handling
2. User space code in kernel space
3. Capability-aware user space
4. Logical separation of kernel subsystems
5. Extensive analysis of performance penalties
6. Comprehensive benchmarking
7. Comprehensive fault injections