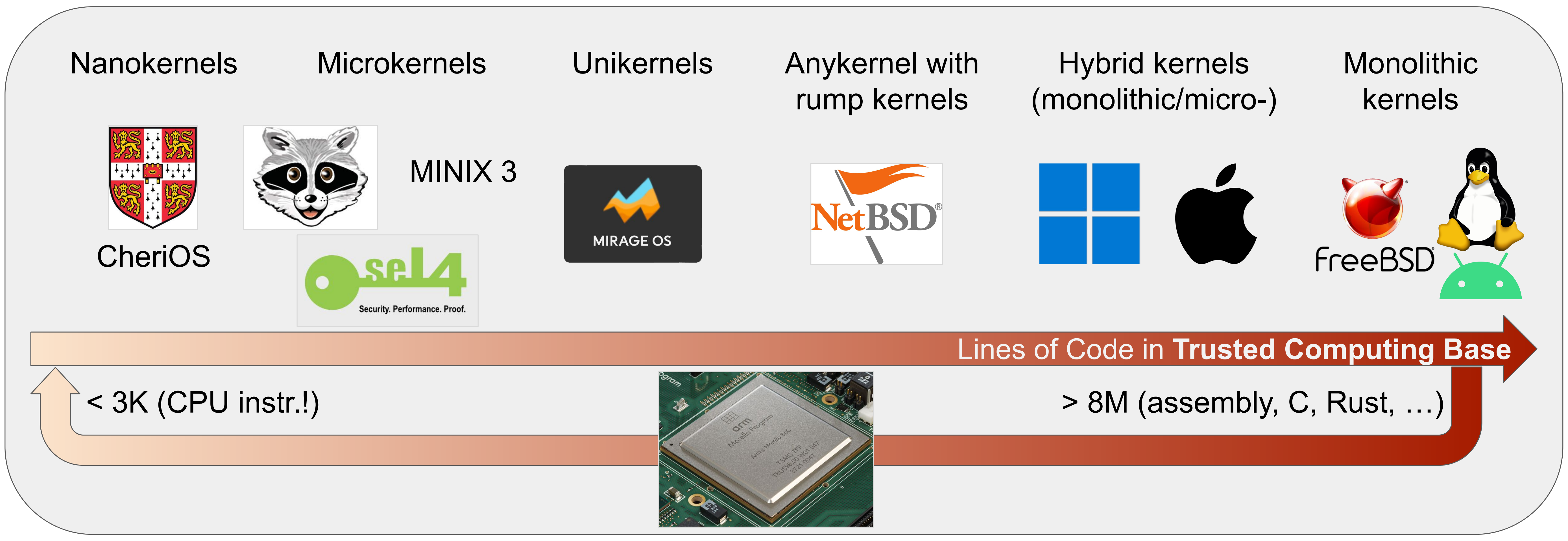
Kernel compartmentalization in CheriBSD

Konrad Witaszczyk, Department of Computer Science and Technology, University of Cambridge

Single-address-space operating systems like CheriOS and CompartOS show the potential for kernel compartmentalization using CHERI. However, techniques used in them can only be applied to kernels if you build them from scratch. In this work, I explore compartmentalization strategies in an example mainstream monolithic kernel – CheriBSD based on FreeBSD – consisting of millions of trusted lines of code.


Nanokernels
Microkernels
Unikernels
Anykernel with rump kernels
Hybrid kernels (monolithic/micro-)
Monolithic kernels
CheriOS
MINIX 3
seL4
MIRAGE OS
NetBSD
FreeBSD
Lines of Code in Trusted Computing Base
< 3K (CPU instr.!)
> 8M (assembly, C, Rust, …)

Hypothesis: Hardware-assisted compartmentalization with CHERI can significantly reduce the risk of vulnerabilities being exploited in a monolithic kernel with minor implementation and performance costs.

Research approach

As described in "Towards a theory of application compartmentalisation" by Robert N. M. Watson et al., there are multiple approaches to application decomposition for compartmentalization. In the context of monolithic kernels, the kernel code (case a in Figure 1) can be reorganized into separate entities with specific responsibilities (case b), separate data-specific instances of the kernel code can be instantiated (case c), or the kernel routines can be associated with data specific to them (case d).

I plan to explore some of these strategies and evaluate their security guarantees and performance overheads using different ISA features in Arm Morello and CHERI-RISC-V.

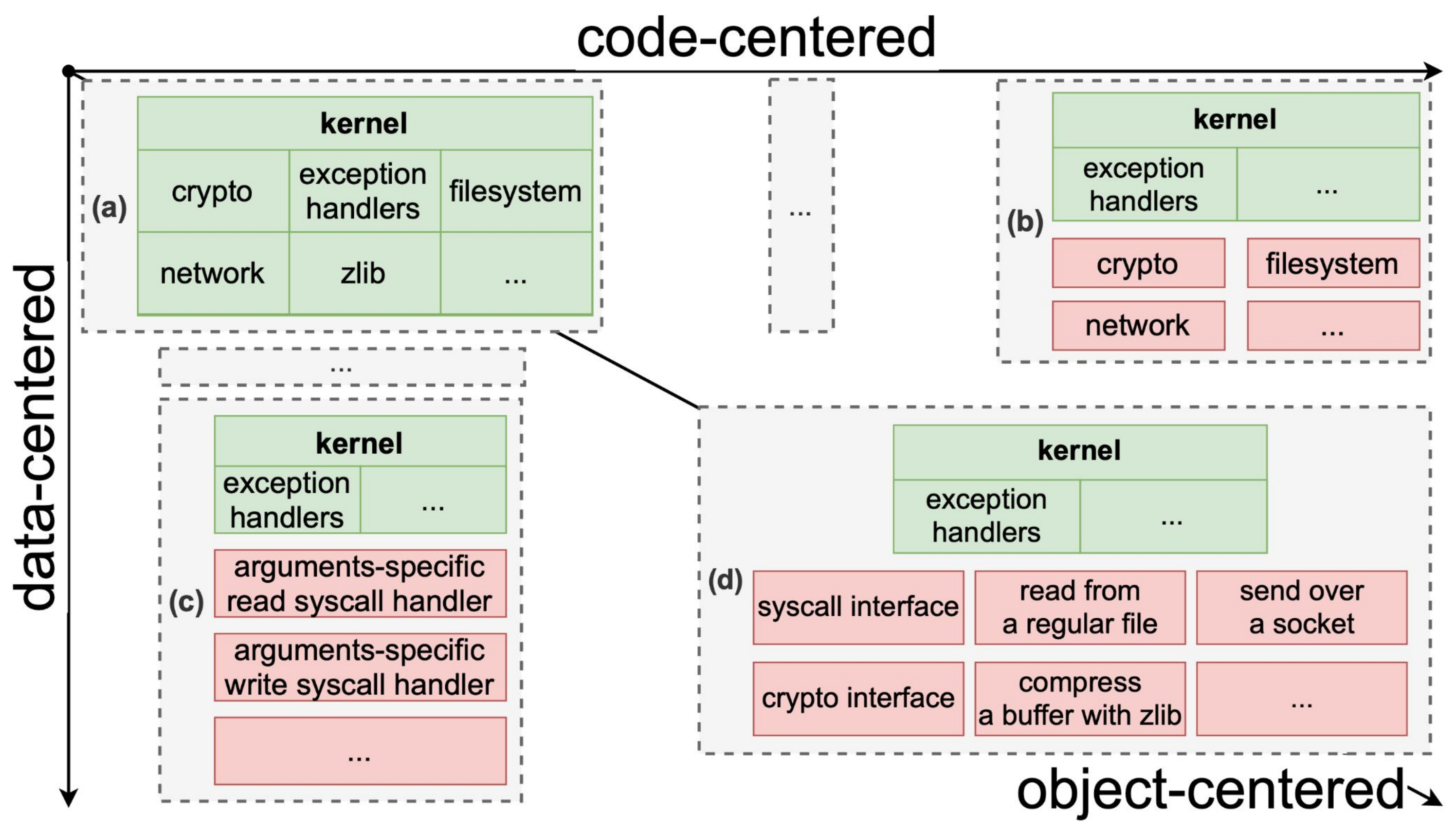


Figure 1. Potential approaches towards compartmentalization in a monolithic kernel

Kernel linker-based compartmentalization

In this code-centered model, kernel module code is executed in a compartment specific to a kernel thread. The kernel linker is responsible for constructing compartment entries when relocating kernel modules' symbols. The domain transition mechanism (Figure 2) uses the Executive and Restricted modes in Arm Morello to protect capability registers from being manipulated by an untrusted compartment.

- **Compartment code:** code in the kernel or a kernel module.
- **Compartment entry:** a sealed entry capability function pointer to a trampoline wrapping a target function pointer to compartment's code.
- **Compartment:** a kernel thread running compartment's code after calling a compartment entry. The compartment has a separate kernel thread stack allocated.

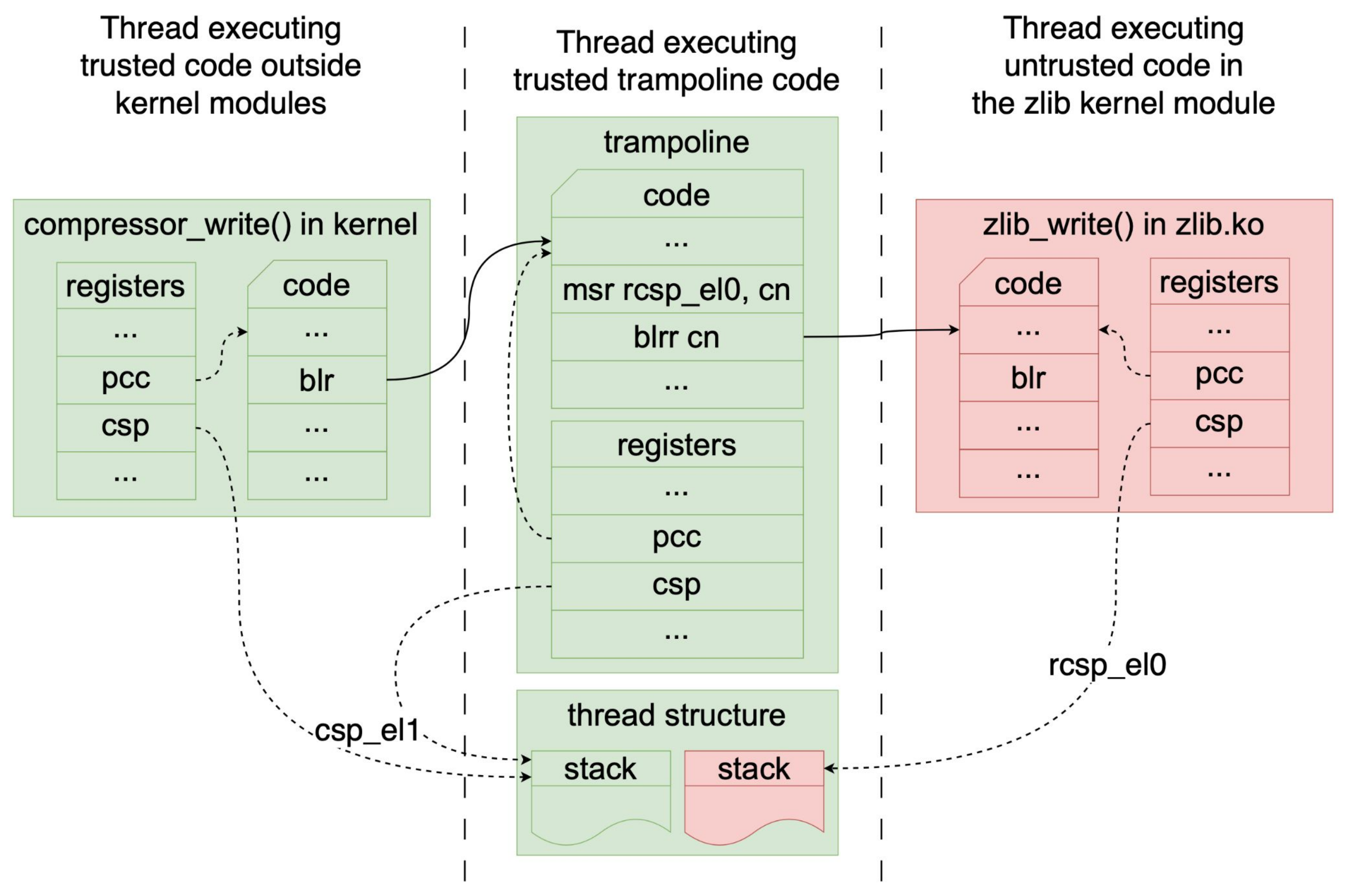


Figure 2. An example domain transition between compartments using a trampoline. The `blrr` instruction branches into the Restricted mode using a capability without the Executive permission bit set making the `csp` register refer to `rcsp_el0` instead of `csp_el1`

Timeline

This research project started in October, 2022 as part of my PhD. During the first year, I developed a prototype of the kernel linker-based compartmentalization model. In the second year, I plan to extend this implementation to other kernel modules, improve the trampoline code to prevent capability leaks in unused registers and implement a compartmentalization policy mechanism. After that, I plan to explore the object-centered approach in compartmentalization.

Contact Konrad.Witaszczyk@cl.cam.ac.uk

UNIVERSITY OF CAMBRIDGE