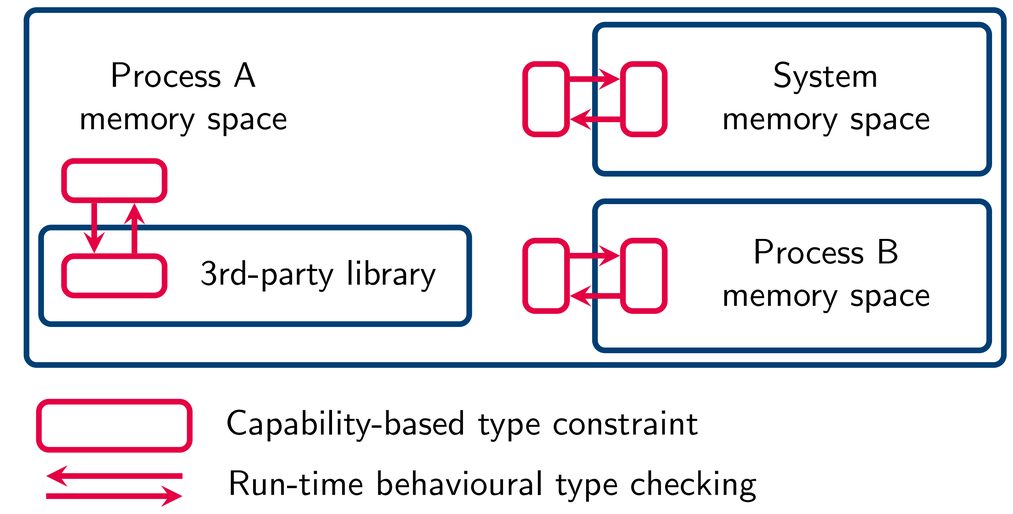
APPCONTROL: ENFORCING APPLICATION BEHAVIOUR THROUGH TYPE-BASED CONSTRAINTS

Wim Vanderbauwhede[1], José Cano[1], Laura Voinea[1], Nobuko Yoshida[2], Martin Vassor[2], Klaus McDonald-Maier[3], Xiaojun Zhai[3], Ludovico Poli[3], Michal Borowski[3], Chandrajit Pal[3]

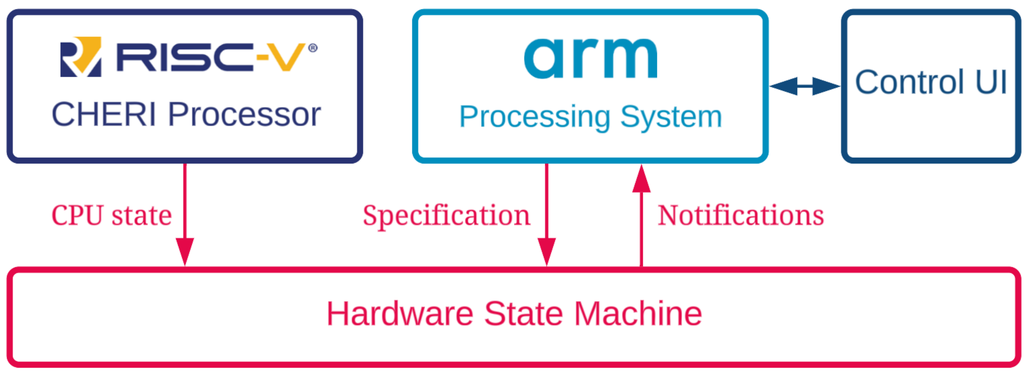
[1]University of Glasgow, [2]University of Oxford, [3]University of Essex

PROJECT OVERVIEW

CHERI capabilities provide fine-grained memory protection, limiting access privileges of third-party applications. However, in order to **secure program interaction**, since capabilities say nothing about **program behaviour**, we use **Behavioural Types** to capture the behavioural structure of application interfaces.



Behavioural types ensure correctness of behaviour, provided that the specification is correct. Debugging a specification-based system demands the ability to **debug the specification at run-time**.



OUR APPROACH

Behavioural typing supports **compile-time** checking of program behaviour when its implementation is known, and **runtime** checking of program behaviour when it is not known.

- Develop a **Rust API** enabling use of CHERI Capabilities.
- Develop **multiparty session type (MPST)** theories and tools to ensure capability-based behavioural properties in Rust.
- Develop a framework to enable the monitoring and **debugging** of capability-supporting Rust code.
- Use capabilities to support the unsafe parts of Rust (e.g. using FFIs in session types)

CAPABLE LANGUAGE

We are interested in how CHERI-Style Capabilities interplay with session types in imperative languages such as Rust/C.

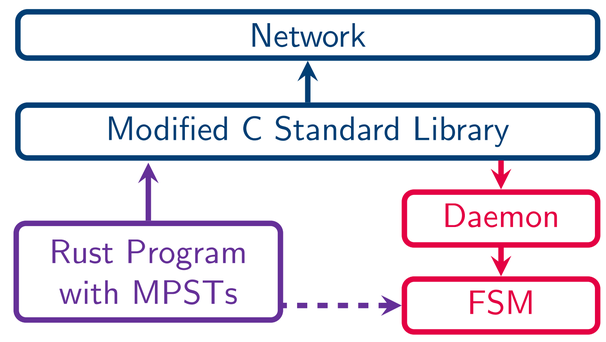
Capable is a bare-bones imperative language with ML-style references written in Idris2 as a intrinsically-scoped/typed EDSL. Capable is our experimental language to see how the type-system should work.

ACKNOWLEDGEMENTS

The authors thank the UKRI Digital Security by Design (DSbD) Programme for funding the AppControl project through grant EP/V000462/1.

RUST API

The Rust API will connect MPSTs with CHERI's memory control capabilities. We identify socket layer system calls to be the most important use case for monitoring adherence to the specification. As Rust's POSIX socket library is built on top of the C standard library, we can address this by modifying the libC to communicate these calls to a listening daemon. The daemon can maintain a finite-state machine (FSM) determined by the specification.



MPST IN RUST

- Developed Rumpsteak, a library for asynchronous MPST in Rust
- Support for refined protocols.
- Protocols generated from 3rd party library **statically type-checked** for behavioural correctness, ensuring **deadlock-freedom**.
- We are currently adapting Rumpsteak for the Rust API, using capabilities in refinements.

An example of refined protocol (excerpt):

```
choice at B{
    More(x : int {x < n}) from B to C;
    continue Loop;
} or {
    Less(x : int {x > n}) from B to C;
    continue Loop;
} or {
    Correct(x : int {x = n}) from B to C;
    continue Loop;
}
```

BEHAVIOURAL PROFILING AND ANOMALY DETECTION

- Extraction of program metrics from the custom implementation of CHERI Flute based SoC.
- ZC706 board implementation of PYNQ wrapper for CHERI-RISC-V Flute processor, utilizing Continuous Monitoring System (CMS) hardware module for baremetal programs.
- Real-time oversight and control of CMS through GUI desktop app.
- Development of intelligent analytics framework (IAF).
- Verification framework for quantitative performance evaluation of anomaly detection methods using adapted EEMBC Automotive 1.1 benchmark.

For more details please see the additional poster.

LEARN MORE

AppControl

Capable

Rumpsteak