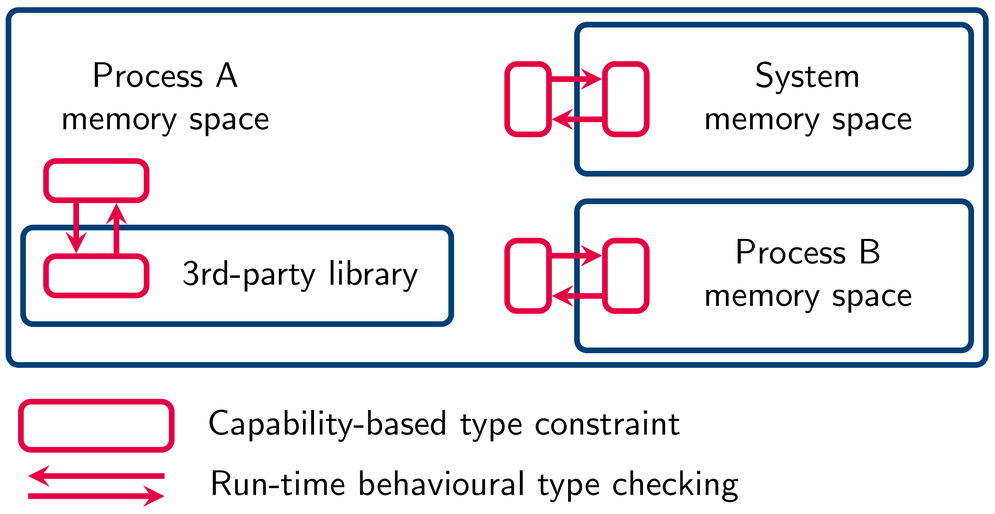
APPCONTROL: ENFORCING APPLICATION BEHAVIOUR THROUGH TYPE-BASED CONSTRAINTS

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]

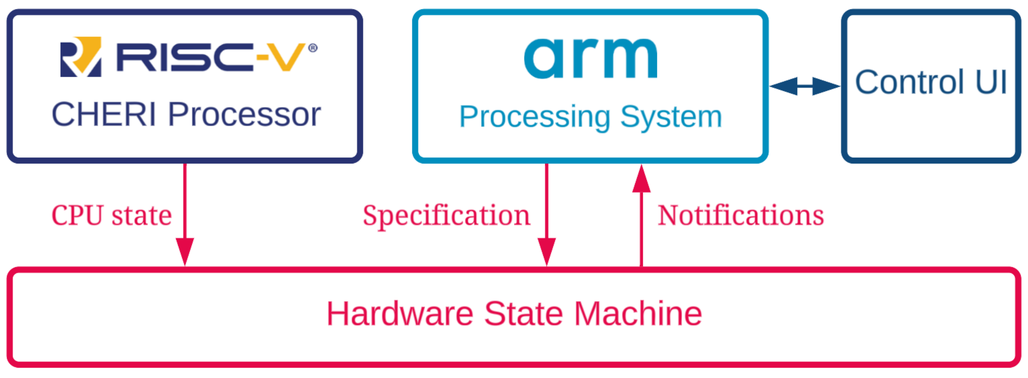
[1]University of Glasgow, [2]University of Oxford, [3]University of Essex

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

- 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.

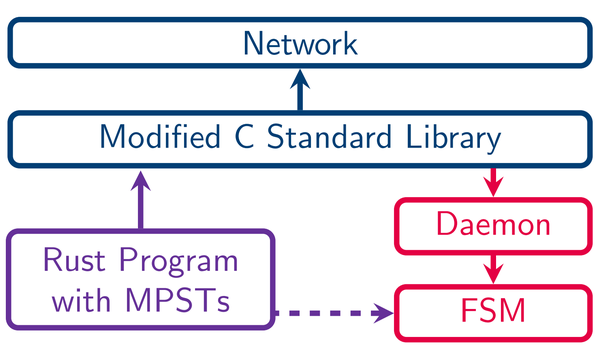
CAPABLE LANGUAGE

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

Capable is a barebones 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.

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 { ... }
```

BEHAVIOUR TRACING OF CHERI-ENABLED PROCESSOR ARCHITECTURE

Please see the additional poster.

MORELLO-HAT: HIGH-LEVEL API AND TOOLING

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

The **Morello-HAT** project intends to create a common API that can be used by compiler developers as well as programmers of higher-level languages, to allow them to leverage Morello's HW capabilities to improve memory security and type safety, spatial as well as temporal, of their language and programs.

The project consists of three work packages, one to develop the API using C++, Rust, Go and Dart, one to demonstrate the usability and effectiveness of the API on a series of example applications by ML-based detection of vulnerabilities and assessment of the effectiveness of mitigation through the use of the API and one to use HW capabilities to enhance the debug infrastructure.

MORELLO-MAURAUDER

Morello-Maurader is an exploration of the mechanisms that can be used to constrain program behaviour on the Morello/CheriBSD platform. Allowed system calls are shimmed by loading a dedicated shared library and their parameters are filtered using a simulation of refinement types.

Disallowed system calls are filtered out using the capsicum implementation in CheriBSD.

Future work will look at an equivalent mechanism on Morello-Linux and integration into the Rust ecosystem, allowing for the implementation of a behaviourally-typed memory allocator demonstration.

LEARN MORE

ACKNOWLEDGEMENTS

The authors thank the UKRI Digital Security by Design (DSbD) Programme for funding the AppControl project through grant EP/V000462/1 and Morello-Hat through grant EP/X015955/1.