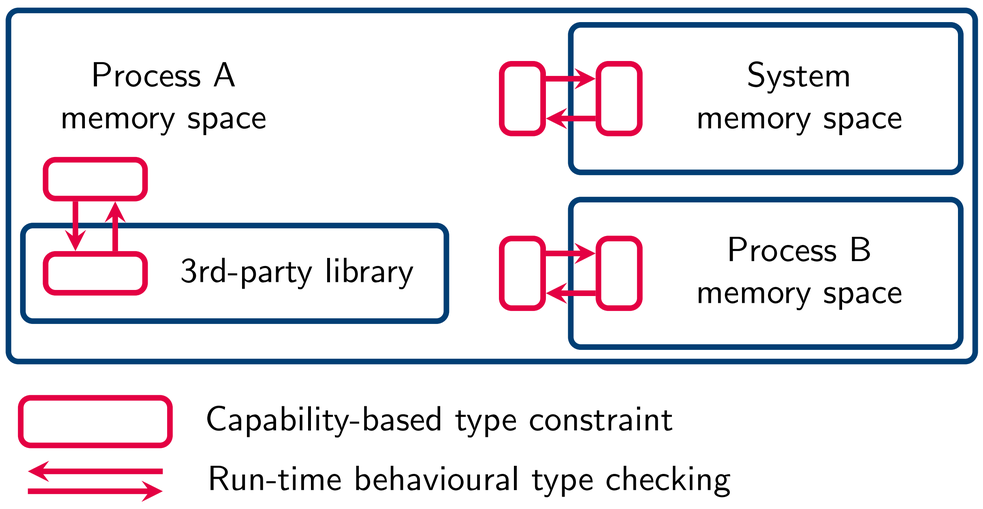
APPCONTROL: ENFORCING APPLICATION BEHAVIOUR THROUGH TYPE-BASED CONSTRAINTS

Wim Vanderbauwhede[1], José Cano[1], Jan de Muijnck-Hughes[1], Cristian Urlea[1], Nobuko Yoshida[2], Adam Barwell[2], Klaus McDonald-Maier[3], Xiaojun Zhai[3], Sangeet Saha[3], Ludovico Poli[3], Michal Borowski[3]

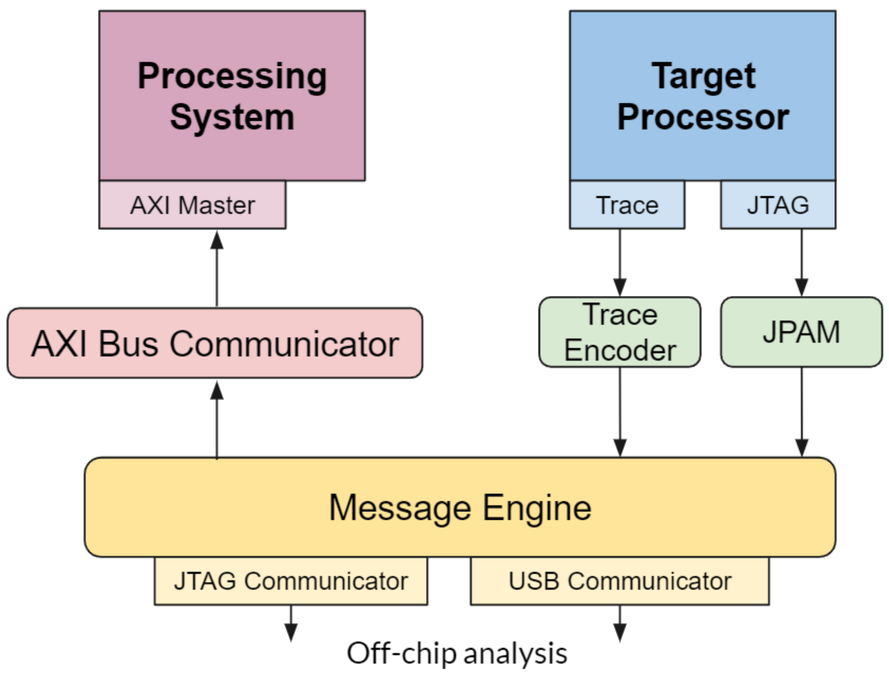
[1]University of Glasgow, [2]Imperial College London, [3]University of Essex

PROJECT OVERVIEW

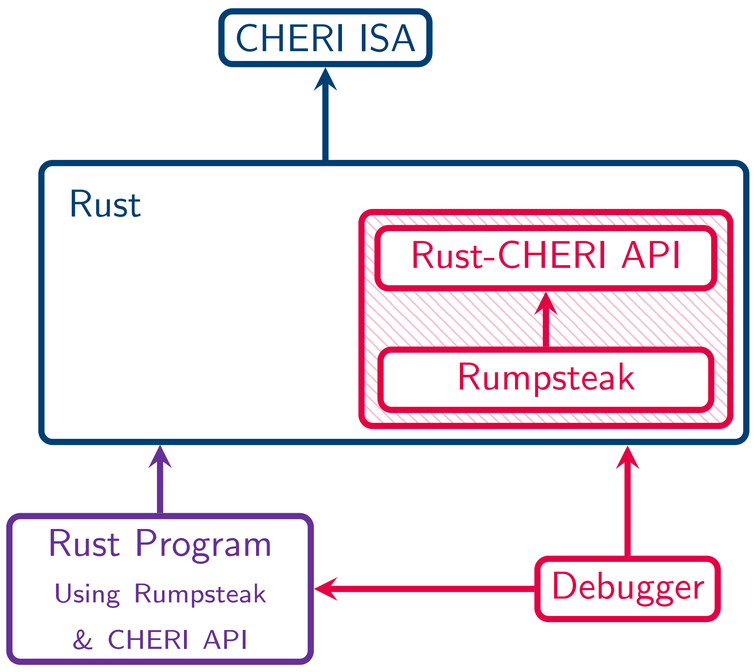
CHERI capabilities provide fine-grained memory protection, limiting access privileges of third-party applications. However, in order to **secure program interaction**, since capabilities say nothing about **program behaviour**, we use **Behavioural Types** to capture the behavioural structure of application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



Behavioural types will 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



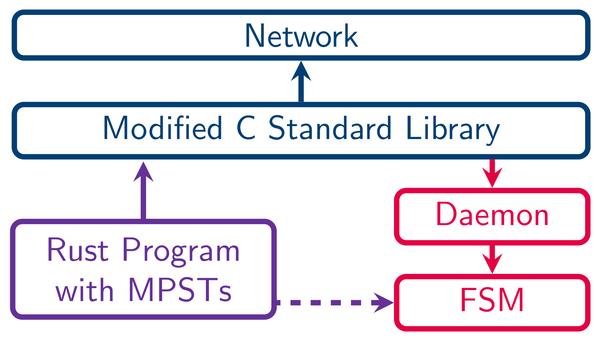
- Develop a **Rust API** enabling use of CHERI Capabilities.
- Develop **multiparty session type (MPST)** theories and tools to ensure capability-based behavioural properties in Rust.
- Develop a framework to enable the monitoring and **debugging** of capability-supporting Rust code.

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

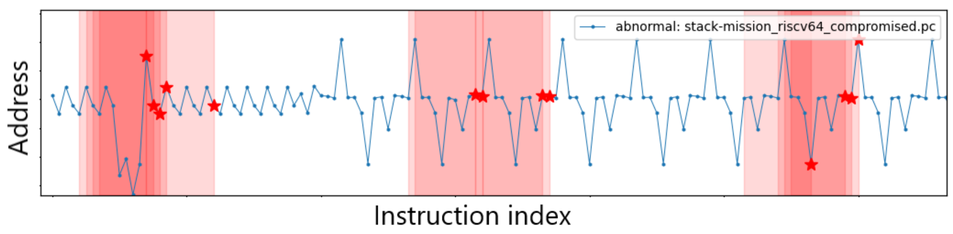
- Developed the Rumpsteak library, supporting asynchronous MPST in Rust with automatic message reordering.
- Communications between two processes or a 3rd party library can be **statically type-checked** for behavioural correctness, ensuring **deadlock-freedom** of protocols.
- We are currently adapting Rumpsteak for the Rust API.
- `https://github.com/zakcutner/rumpsteak`

A ring protocol in Rumpsteak:

```
1 type Source = Receive<T, Ready, Select<T, SourceChoice>>;
2 enum SourceChoice { Value(Value, Source),
3                     Stop(Stop, End) }
4 type Sink = Send<S, Ready, Branch<S, SinkChoice>>;
5 enum SinkChoice { Value(Value, Sink),
6                   Stop(Stop, End) }
```

DEBUGGER AND DIAGNOSTICS TOOLS

- Development of prototypes of non-intrusive, on-chip debug system for open-source processors (e.g. RISC-V).
- Extraction of program metrics from CHERI-RISC-V Qemu emulator.
- Performance comparison of conventional (e.g. FSA, n-gram) and novel (e.g. autoencoder-forest) abnormal behaviour detection methods.



KEY OUTSTANDING QUESTIONS

Rust API

- What is the optimal format for the FSM encoding?
- What are the possible attack surfaces for this model?
- Would Rust capability support need explicit libC capability support?

MPST in Rust

- What information in the capabilities should be exposed to the programmer in session types?
- What extensions to MPST do capabilities specifically enable?

Debugger

- What are the best metrics to extract for our model?
- To what extent can our models be applied at hardware level?
- What is the best way to integrate our model into the debugging toolchain?

PROJECT WEB SITE

`https://dsbd-appcontrol.github.io/`