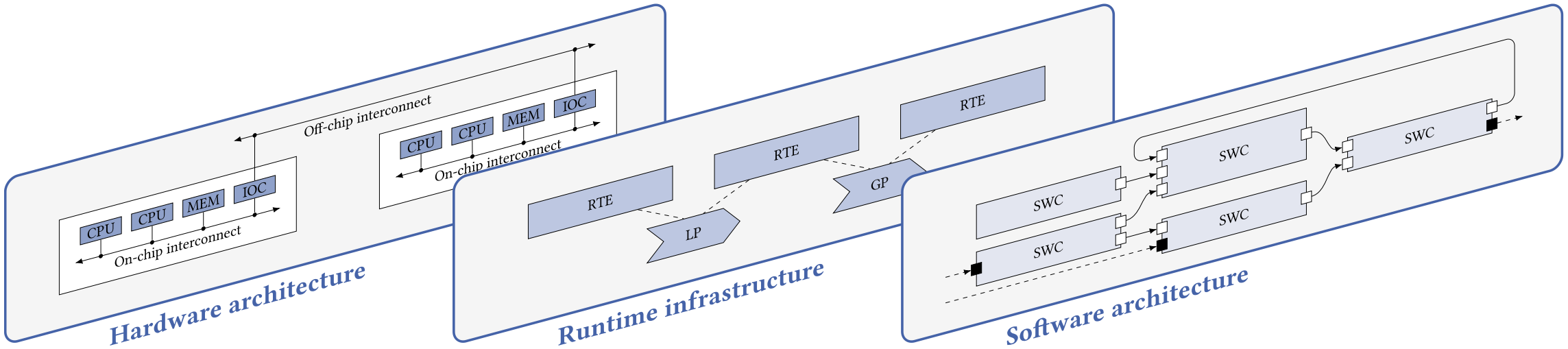
This work is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957210.

Karlsruhe Institute of Technology

Pattern-Based Information Flow Control for Safety-Critical On-Chip Systems

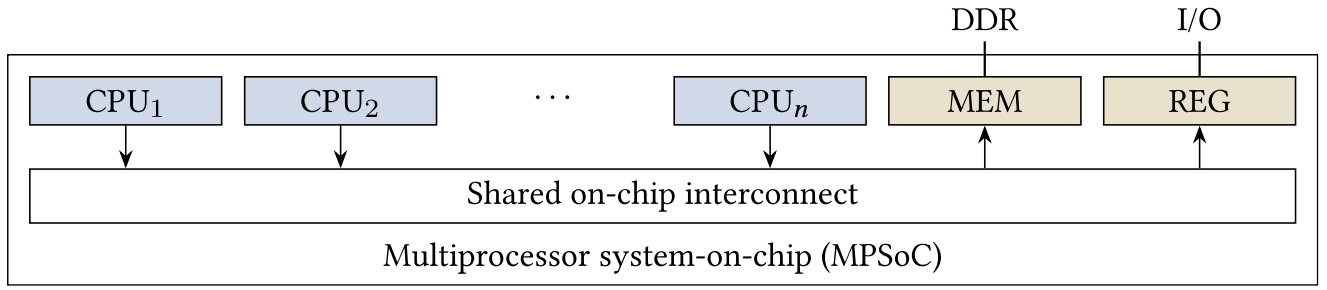
Tobias Dörr, Florian Schade, and Jürgen Becker

SafeComp 2023, Toulouse · September 21, 2023



Motivation

- On-chip integration comes with benefits in terms of cost, weight, performance, …
- **Multiprocessor system-on-chip (MPSoC)** devices are appealing target platforms [1]:



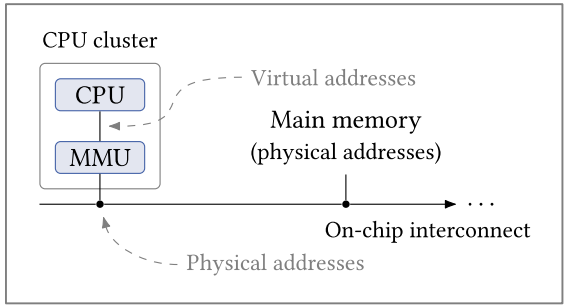
- Safety challenge: **unacceptable interferences** between integrated applications
- Goal: ensure that explicit interferences do not violate safety requirements

Timing-related interferences are not the focus of this work!

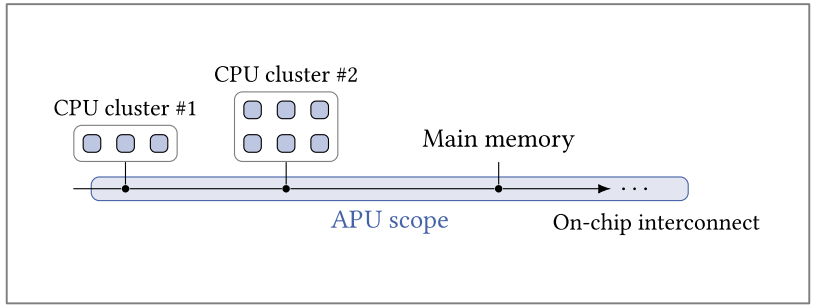
[1] M. Hassan. **Heterogeneous MPSoCs for Mixed-Criticality Systems: Challenges and Opportunities**. IEEE Design & Test, 35(4), August 2018.

State of the art

- MPSoCs provide **logical isolation mechanisms** for on-chip transactions
  - *Example I: Memory Management Units (MMUs) of processors*
  - *Example II: Access Protection Units (APUs) in the sense of [2]*



(I) Memory Management Unit



(II) Access Protection Unit

- Usage of such mechanisms contributes to the fulfillment of safety requirements, e.g.:
  - *ISO 26262-11 [3]: "Techniques such as hypervisors can help to achieve software partitioning […]"*
- However: they **control local transactions** instead of end-to-end flows!

[2] T. Nojiri et al. **Domain Partitioning Technology for Embedded Multicore Processors**. IEEE Micro, 29(6), 2009.
[3] International Organization for Standardization (ISO). **ISO 26262-11:2018: Road vehicles — Functional safety — Guideline on application of ISO 26262 to semiconductors**. Geneva, 2018.

Scientific context

- **X-by-Construction (XbC) paradigm** [4]:
  - Auto-generate software system implementations
  - Ensure that they meet non-functional properties by construction

- **XANDAR project** [5] funded by the European Union:
  - Address safety and security requirements via automated XbC patterns
  - Key input into the XANDAR toolchain is a software architecture model

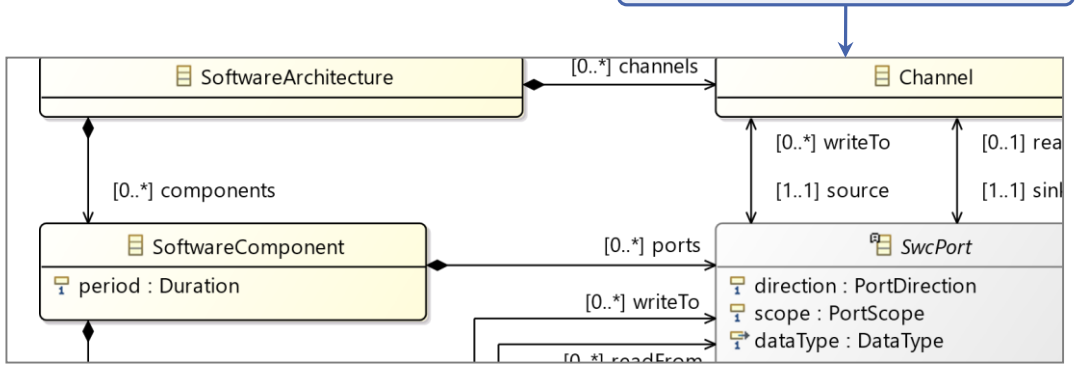
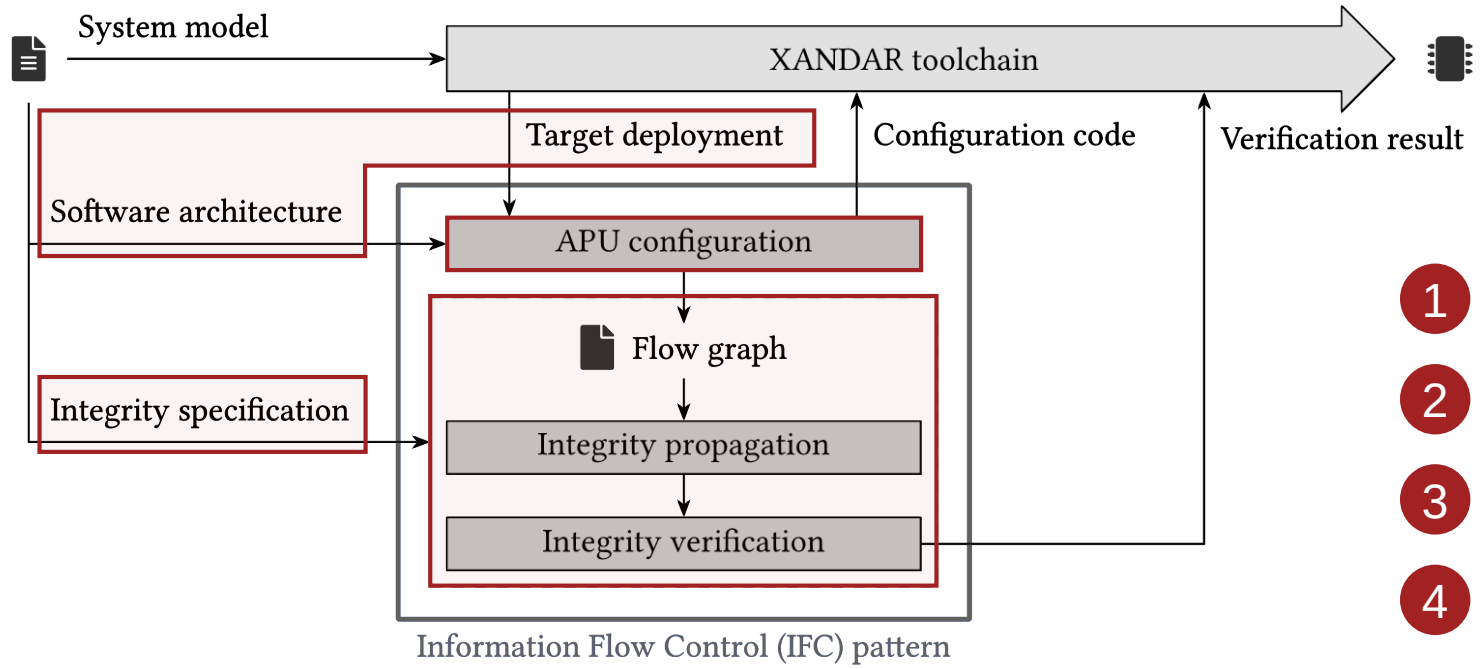


Image taken from [6]

[4] M.H. ter Beek, L. Cleophas, et al. **X-by-Construction**. ISoLA '18, Oct. 2018.
[5] L. Masing, T. Dörr, et al. **XANDAR: Exploiting the X-by-Construction Paradigm in Model-based Development of Safety-critical Systems**. DATE '22, March 2022.
[6] XANDAR project website (https://xandar-project.eu, visited on 09/16/2023).

Big picture

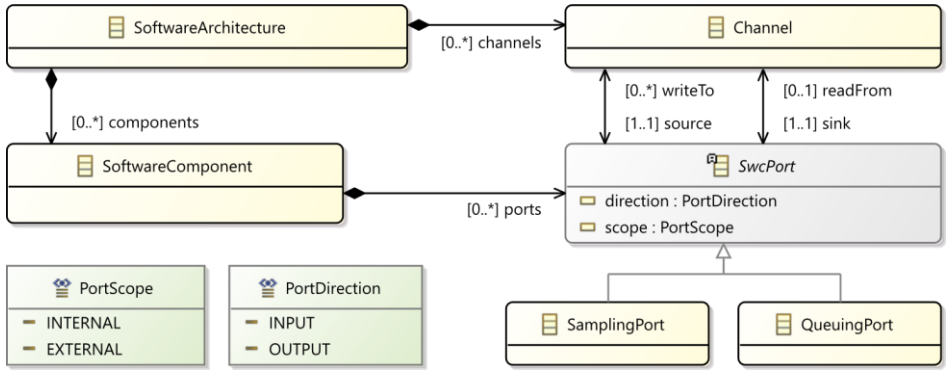
- Proposed: **Information Flow Control (IFC) pattern** to ensure integrity of end-to-end flows



1. Input model
2. APU configuration
3. Integrity specification
4. Integrity analysis

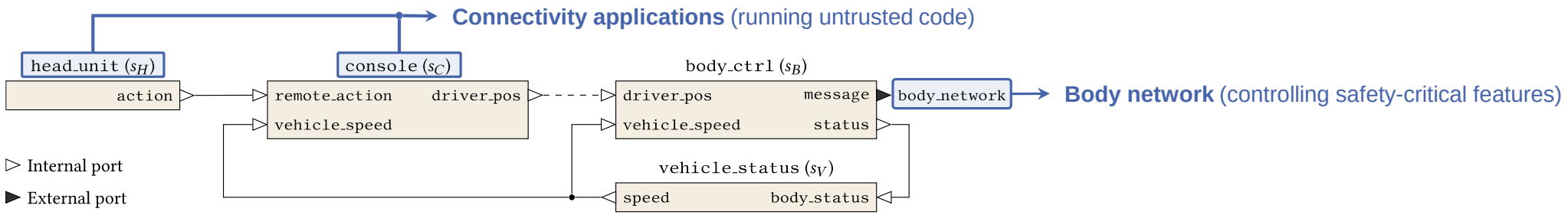
Input model: software architecture

- Excerpt of the **software architecture metamodel** from the XANDAR project:



Formalization → $(S, P, C, \varphi_0, \varphi_s, \varphi_p)$

- *Sample software architecture from the automotive domain:*



Input model: target deployment

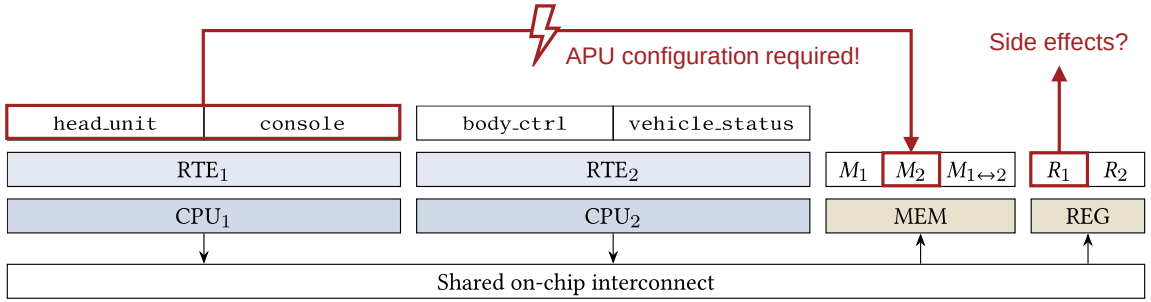
- **Automatic deployment** by the XANDAR toolchain:

Definition 1 (deployment strategy). *The deployment strategy maps a software architecture to a platform with core clusters* $\{CPU_1, \ldots, CPU_n\}$ *such that:*

1. *Every core cluster* CPU_i *with* $i \in \{1, \ldots, n\}$ *executes a runtime environment* RTE_i *such as a bare-metal hypervisor.*
2. *Each specified SWC is executed by exactly one runtime environment.*
3. *To each* RTE_i *with* $i \in \{1, \ldots, n\}$*, a dedicated memory region* M_i *and a dedicated set of memory-mapped registers* R_i *is assigned.*
4. *For each pair of runtime environments* $\{RTE_i, RTE_j\}$ *with* $i, j \in \{1, \ldots, n\}$

Formalization → $(\Omega, \lambda, \Psi_{M1}, \Psi_{M2}, \Psi_R, \mu, \mu')$

- *Sample deployment for the previous car server example:*

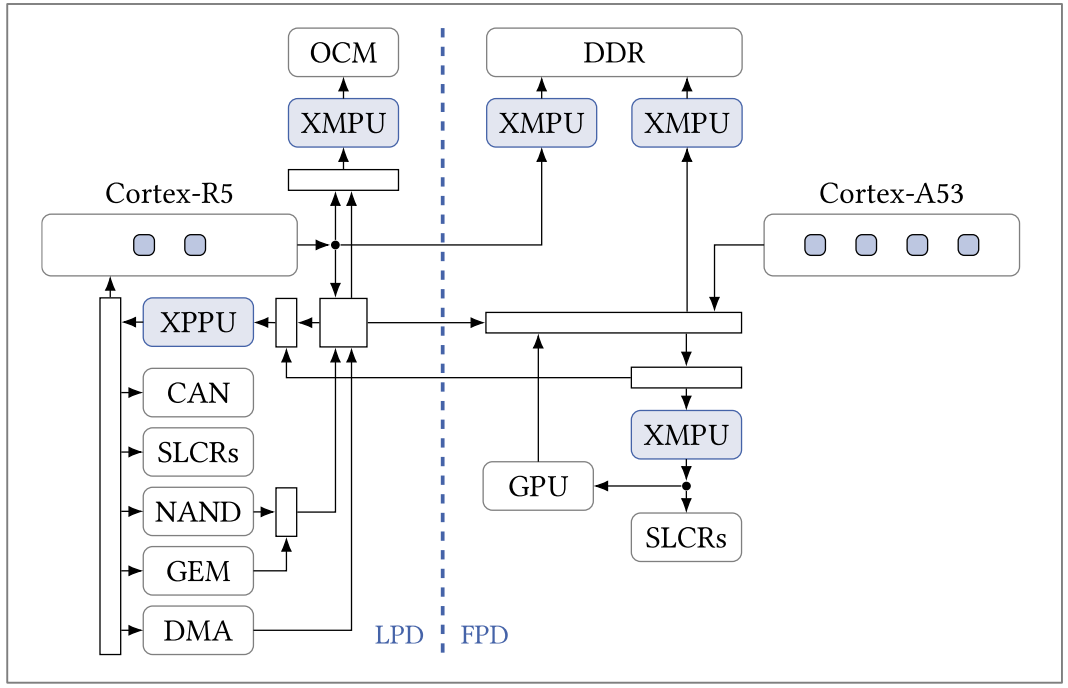


APU configuration

- **Currently supported APUs** are those on the Xilinx Zynq UltraScale+ MPSoC:
  - Xilinx Peripheral Protection Unit (XPPU)
  - Xilinx Memory Protection Unit (XMPU)

- Configurations are automatically derived from:
  - Private on-chip resources: Ψ_{M1} + Ψ_{R}
  - Shared on-chip resources: Ψ_{M2}
  - $\mu : (\Psi_{M1} \cup \Psi_{R}) \rightarrow (\Omega \cup S)$ with $\lambda : S \rightarrow \Omega$
  - $\mu' : \Psi_{M2} \rightarrow \{\{\omega_1, \omega_2\} \mid \omega_1, \omega_2 \in \Omega\}$

- Output: **C code** to be executed by any CPU
  - Writes the complete configuration to all APUs
  - Finally, locks this configuration in place



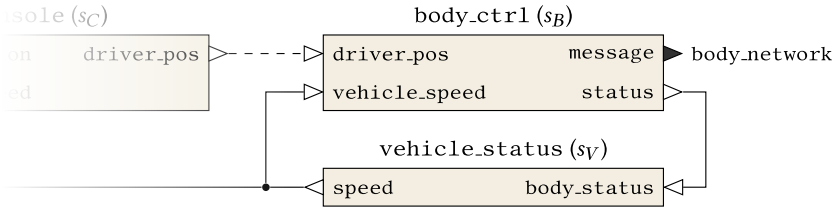
XPPU and four XMPUs of the Xilinx Zynq UltraScale+ MPSoC

Integrity specification

- Specification of accepted end-to-end flows based on a lattice-based model [7]
  - Label **critical sinks** (e.g. environment writes) with a <u>required</u> integrity level $\ell_R(v)$
  - Label **interference sources** (e.g. RTEs) with a <u>provided</u> integrity level $\ell_P(v)$
  - In addition: declare **flow barriers** between inputs and outputs of a SWC

- **Integrity lattice** = bounded lower semilattice of integrity levels $(L, \leq)$, e.g. $L = \{0, 1, 2, 3\} \subseteq \mathbb{N}_0$

```
lattice: "demo_lattice",
flow_barriers: [
    { swc: "body_ctrl", output: "message", input: "driver_pos" },
    { swc: "body_ctrl", output: "status", input: "driver_pos" },
],
provided_integrity: [
    { provider: "mpsoc", level: 3 },
    { provider: "port", name: "vehicle_status.speed", level: 1 },
    { provider: "port", name: "body_ctrl.message", level: 3 },
    // 5 entries hidden for brevity...
],
required_integrity: [
    { receiver: "env", port: "body_ctrl.message", level: 1 },
],
```



[7] K. J. Biba. **Integrity Considerations for Secure Computer Systems**. MITRE Corporation, Technical Report, Jun. 1975.

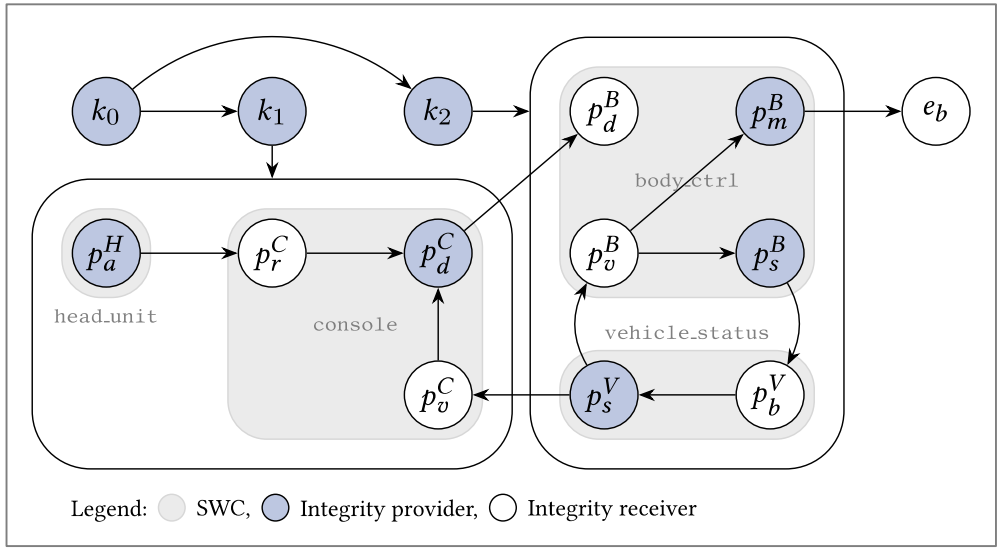
Integrity analysis: flow graph creation

- **Flow graph**: directed graph $G = (V, E)$
  - Auto-generated after (1) APU configuration and (2) flow barrier specification
  - Captures remaining interference paths

- Formal creation procedure:

Algorithm 1 Creation of the flow graph $G = (V, E)$

1: $V \leftarrow V_P \cup V_R, E \leftarrow C$ ▷ Vertices and explicit channels
2: $E \leftarrow E \cup \{(k_0, k(\omega)) : \omega \in \Omega\}$ ▷ MPSoC to all RTEs
3: $E \leftarrow E \cup \{(k(\omega), p) : p \in P,\ \omega = \lambda(\varphi_0(p))\}$ ▷ RTE to all of its ports
4: $E \leftarrow E \cup \{(q, r) \in P_{\text{IN}} \times P_{\text{OUT}} : \varphi_0(q) = \varphi_0(r)\} \setminus B$ ▷ SWC-internal flows
5: $E \leftarrow E \cup \{(v, e(v)) : v \in V_I\} \cup \{(e(v), v) : v \in V_O\}$ ▷ External inputs/outputs
6: **for all** $x \in \Psi_R$ **do** ▷ Implicit paths via registers
7: $\Delta \leftarrow \text{ReachableUnits}(x)$
8: **if** $|\Delta| \geq 0 \wedge \mu(x) \notin \Omega$ **then return** null
9: $E \leftarrow E \cup \{(k(\mu(x)), \delta) : \delta \in \Delta\}$
10: **return** (V, E)



Flow graph for the car server example

Integrity analysis: propagation + verification

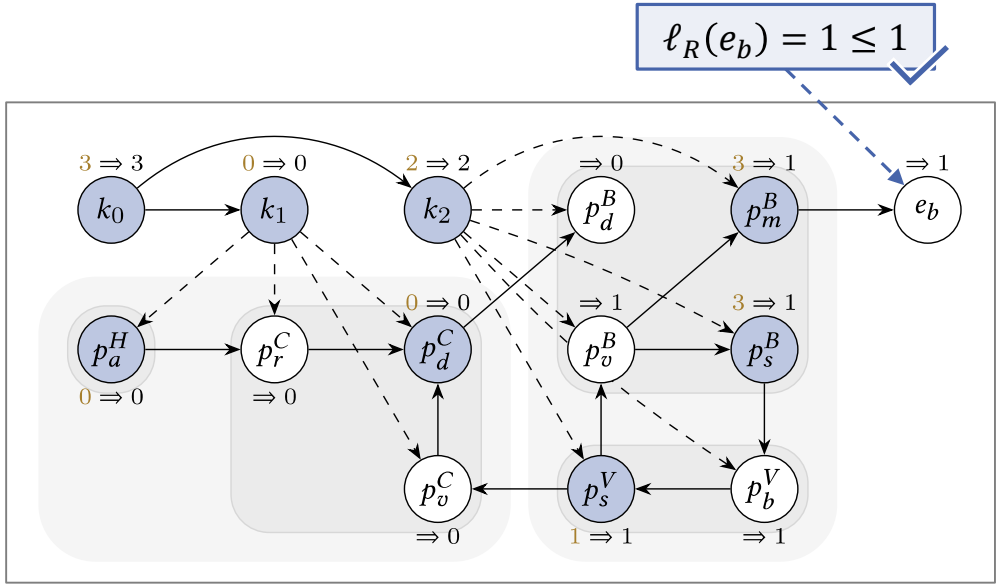
- **Integrity propagation:** determine the effective integrity $\ell'(v)$ for all vertices $v \in V$
- **Integrity verification:** ensure that $\ell_R(v) \leq \ell'(v)$ holds whenever $\ell_R(v)$ is defined

Required integrity level of $v \in V$

- Formal propagation procedure:

Algorithm 2 Integrity propagation via flow graph edges

```
1: for all v ∈ V do ℓ'(v) ← if v ∈ V_P then ℓ_P(v) else ⊤
2: Q ← Queue(V_P)                                   ▷ Queue of vertices in V
3: while NotEmpty(Q) do
4:     u ← Dequeue(Q)                               ▷ Get the next vertex to handle
5:     for v ∈ V : (u, v) ∈ E do
6:         ℓ'_0 ← ℓ'(v)
7:         ℓ'(v) ← ℓ'(u) ∧ ℓ'(v)                    ▷ Propagate its ℓ' to successor v
8:         if ℓ'(v) ≠ ℓ'_0 then Enqueue(v)          ▷ Enqueue v if its ℓ' value changed
```



Integrity propagation result for the car server example

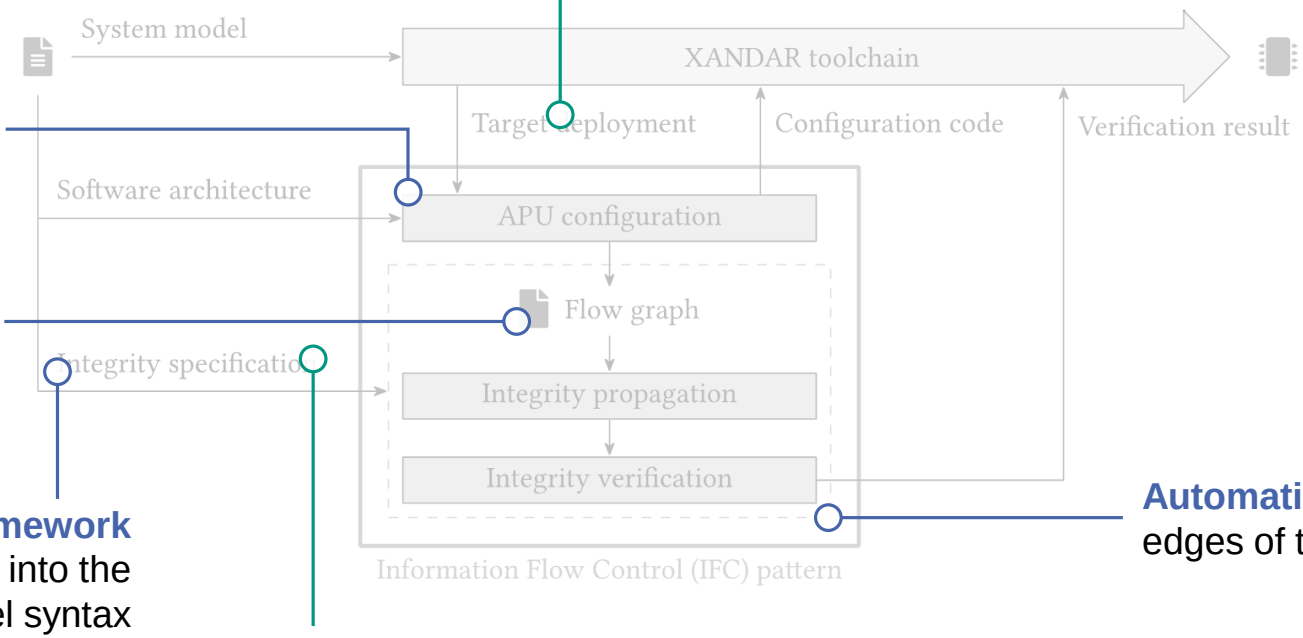
Summary and future work

Support for **additional deployment strategies** is future work (hierarchical RTEs, …)

Code generation for APUs on the Zynq UltraScale+ MPSoC

Automatic determination of interference paths after logical isolation

Lattice-based integrity framework seamlessly integrated into the XANDAR model syntax



Automatic integrity propagation via edges of the flow graph

Tool-supported integration with related **functional safety standards** (e.g., ISO 26262) is another topic for future research

Backup: related work

- Related technique from the computer security domain [8]:

> ***"Information flow tracking (IFT) is a fundamental computer security technique used to understand how information moves through a computing system."***

- IFT has been applied to secure computing systems at various abstraction levels
  - *Example: language + source code [9-10], program execution [11-12], hardware design [13-14]*

- Perspective: application of **static IFT** at the program execution level to control end-to-end flows
  - Analyze and generate configurations of logical isolation units (MMUs, APUs, …)
  - Enforce safety-related integrity requirements

[8] W. Hu, A. Ardeshiricham, R. Kastner. **Hardware Information Flow Tracking**. ACM Computing Surveys, 54(4), 2022.
[9] G. Le Guernic. **Automaton-based Confidentiality Monitoring of Concurrent Programs**. CSF '07, Venice, July 2007.
[10] T. Runge, A. Knüppel, T. Thüm, I. Schaefer. **Lattice-Based Information Flow Control-by-Construction for Security-by-Design**. FormaliSE '20, Seoul, May 2020.
[11] P. Pieper, V. Herdt, D. Große, R. Drechsler. **Dynamic Information Flow Tracking for Embedded Binaries using SystemC-based Virtual Prototypes**. DAC '20, San Francisco, July 2020.
[12] M. Hassan, V. Herdt, H. M. Le, D. Große, R. Drechsler. **Early SoC security validation by VP-based static information flow analysis**. ICCAD '17, Irvine, Nov. 2017.
[13] C. Pilato, K. Wu, S. Garg, R. Karri, F. Regazzoni. **TaintHLS: High-Level Synthesis for Dynamic Information Flow Tracking**. IEEE Trans. Comput.-Aid. Des. Integr. Circuits Syst., 38(5), 2019.
[14] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, R. Kastner. **Information flow isolation in I2C and USB**. DAC '11, June 2011.