Enforcing Safety and Security Through Non-Intrusive Runtime Verification

Inês Gouveia and José Rufino

igouveia@lasige.di.fc.ul.pt jmrufino@ciencias.ulisboa.pt

LaSIGE, Faculdade de Ciências,
Universidade de Lisboa, Portugal

Ciências ULisboa
Faculdade de Ciências da Universidade de Lisboa

Navigators

Cyber-Physical Systems have been applied to **key areas** such as:

- Smart Grids
- Autonomous Mobile Systems
- Medical Monitoring
- Robotics

But are target of multiple **concerns**, namely:

- **Safety**
- **Security**
- Timeliness

Contribution: initial ideas to enforce safety and security in zero-knowledge environments

allowing

common attacks

and

unintentional faults

to be prevented, detected and dealt with

by means of an **observer**

"***Buffer overflow*** *vulnerabilities continue to be the primary attack method, accounting for* ***25% of the attacks***" - 2015 Dell Security Annual Threat Report

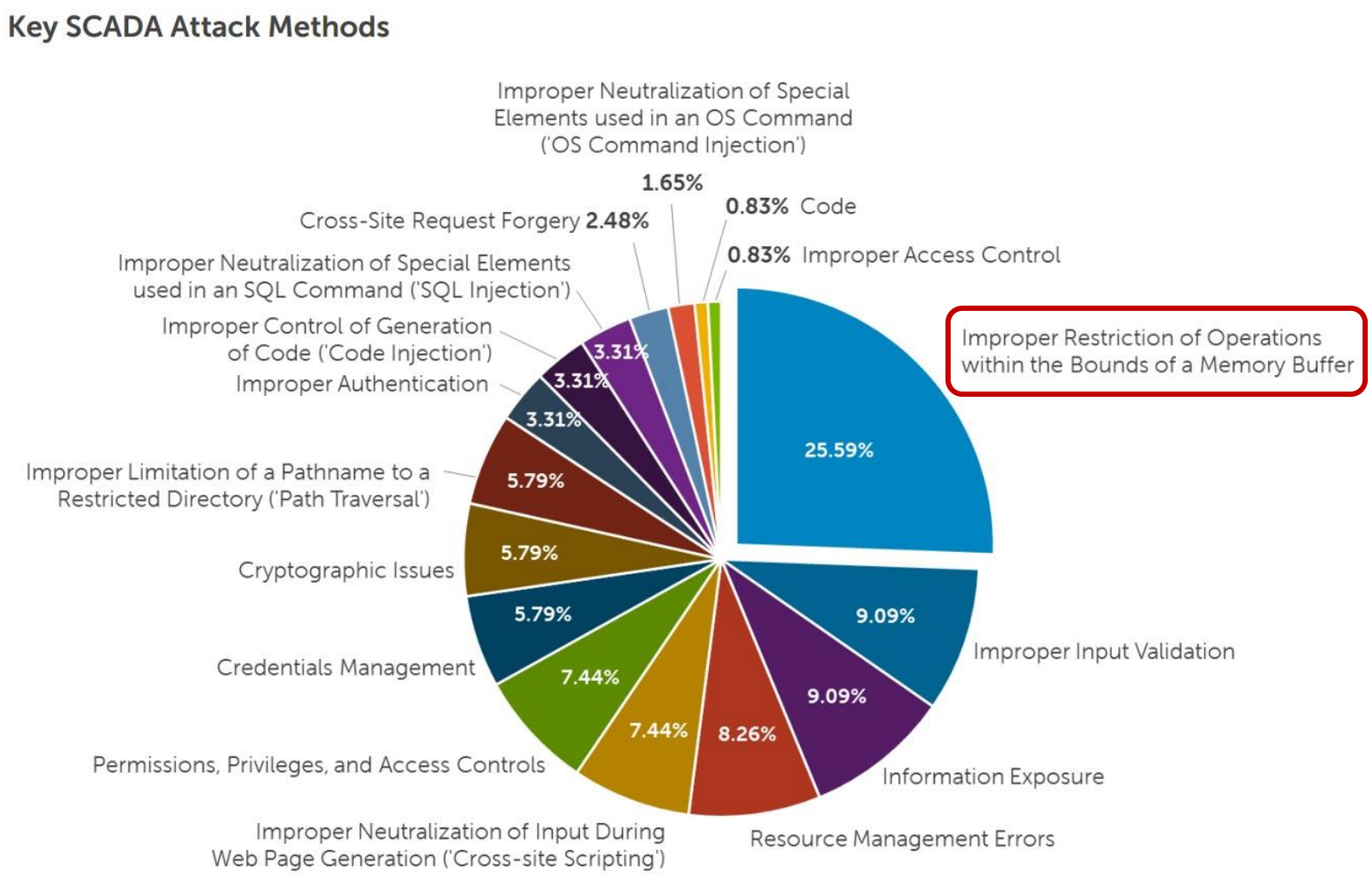


Fig. 1 - SCADA Attack Methods [source: 2015 Dell Security Annual Threat Report]

Naive approach! We:

Assume applications are made up of function calls

Verify if functions do not read/write memory zones they **should NOT access**

Provide **function-grained** memory protection and **denial of service** protection

Perform Non-Intrusive Runtime Verification (NIRV): **Observation** + **Monitoring**

Assumptions:

- Applications consist of **function calls**
- An observer can be **inserted** in the target platform and **connected** to the system bus (or between the cache and the processor)
- The observer is inserted within a **zero-knowledge** environment
- The **cache** is **write-through**
- **ELF** (**E**xecutable and **L**inkable **F**ormat) for binary files

Fig. 2 – ELF File

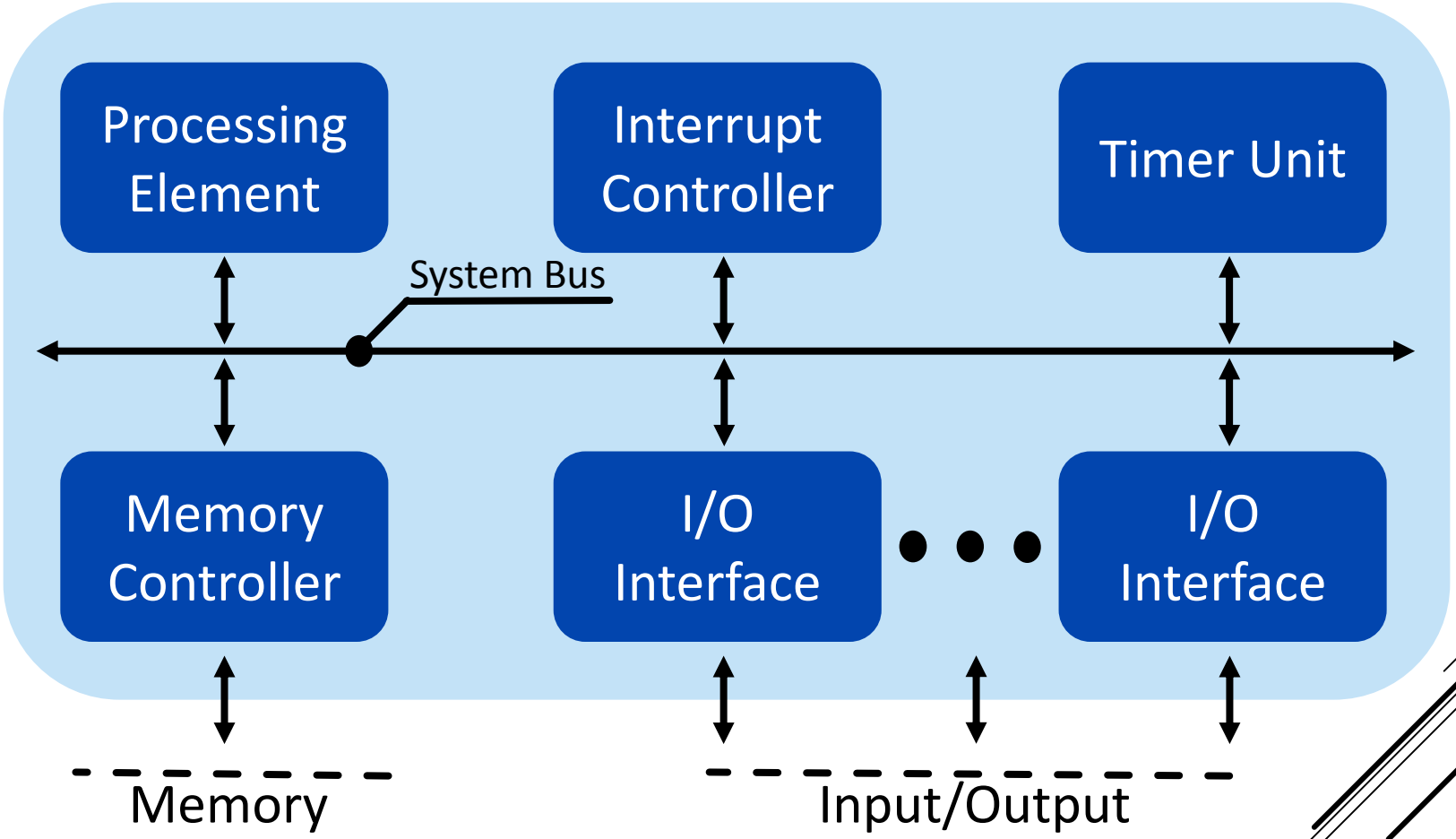


Fig. 3 - Generic System-on-a-Chip Architecture

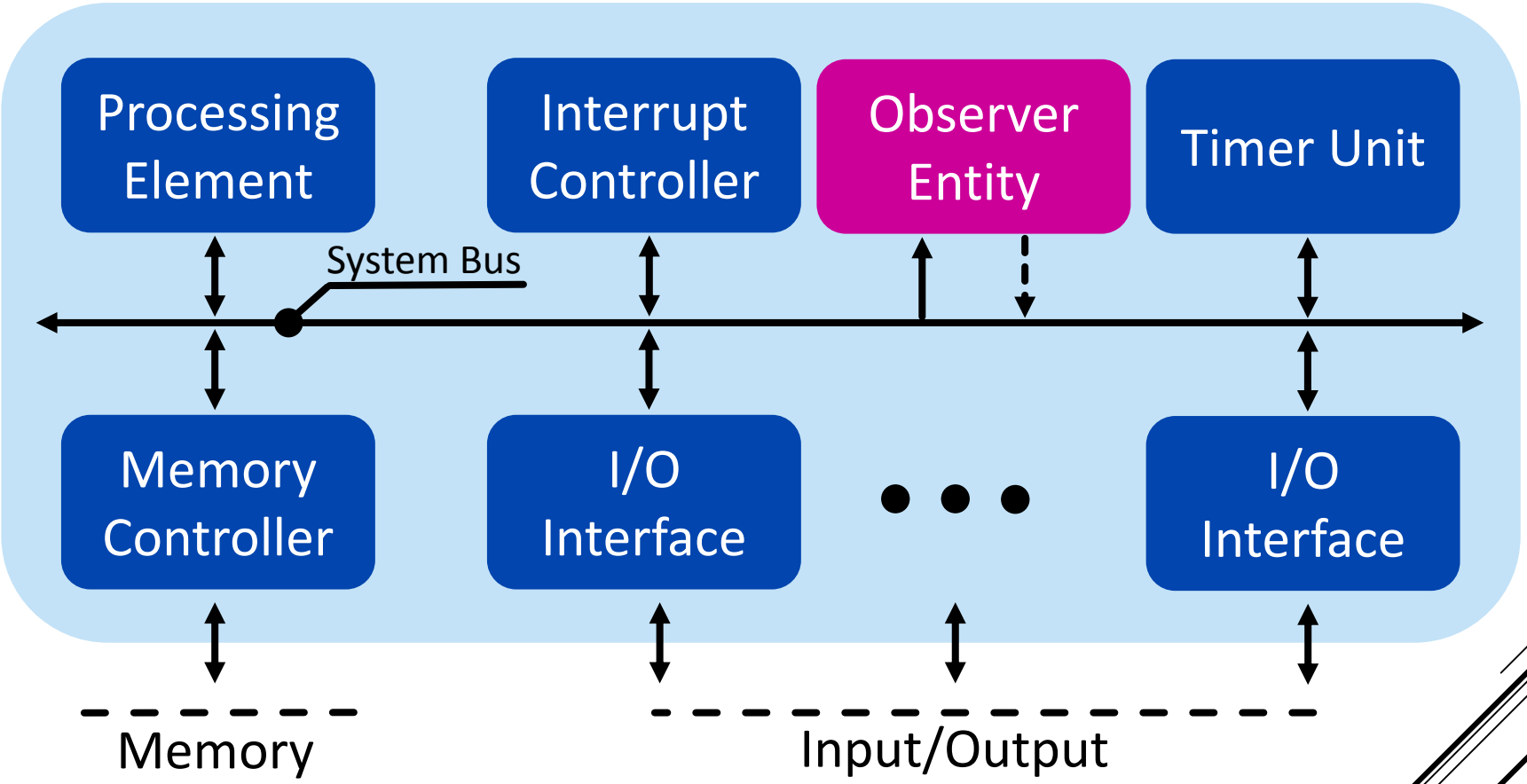


Fig. 4 - System-on-a-Chip Architecture including the Observer Entity

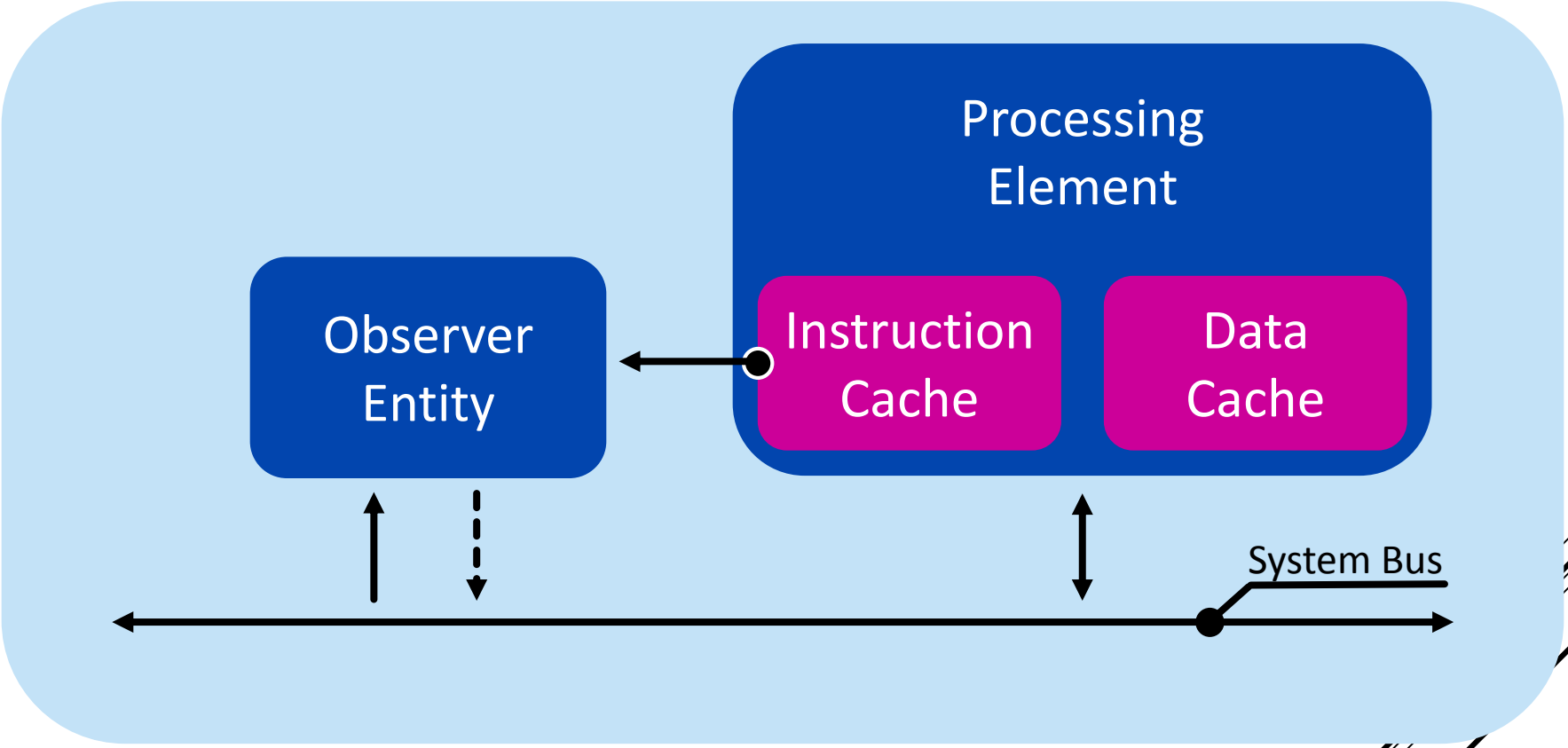


Fig. 5 – SoC platform with LEON3 processor

Why 2 approaches?

- Access to **instructions being fetched**
- Can't always turn off cache
- Additional data = more **detectable faults and attacks**

Observer Entity – brief description:

- **Black Box** → extra security layer (prevents hijacking)
- Specified in **VHDL** (**V**HSIC **H**ardware **D**escription **L**anguage)
- Only addresses apps running natively on the OS → **virtualization for future work**

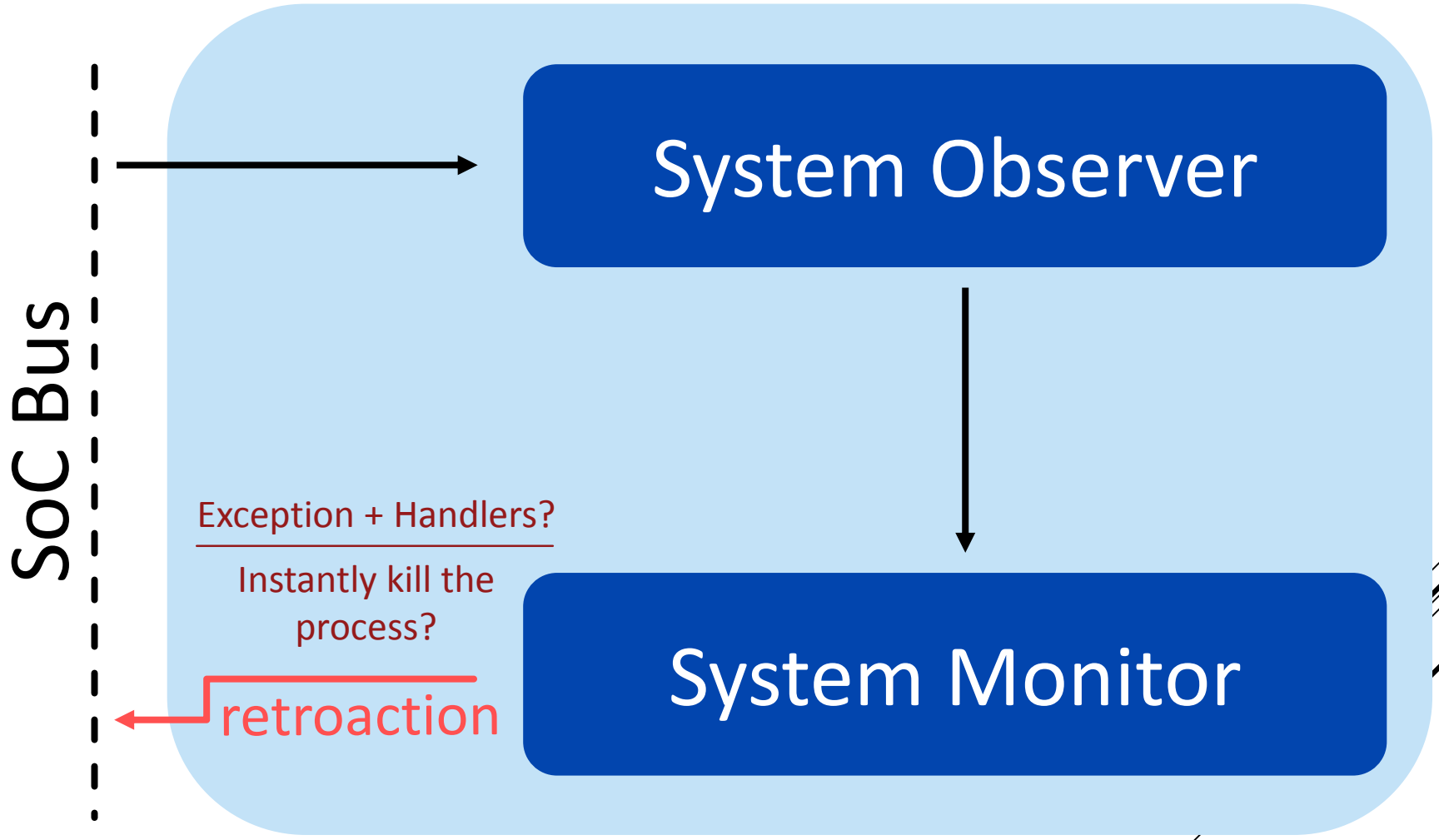


Fig. 6 - Simplified Observer Entity Architecture

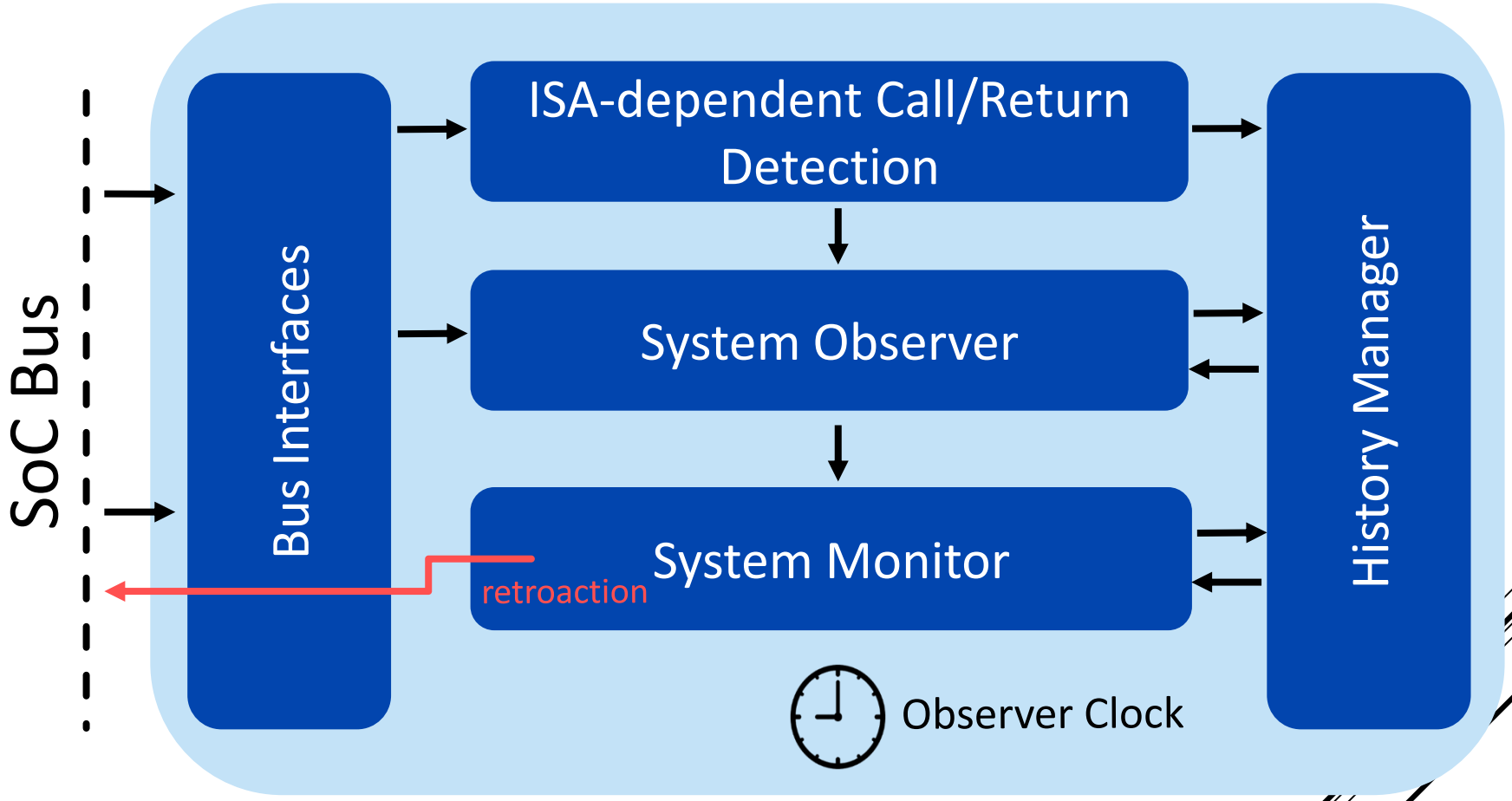
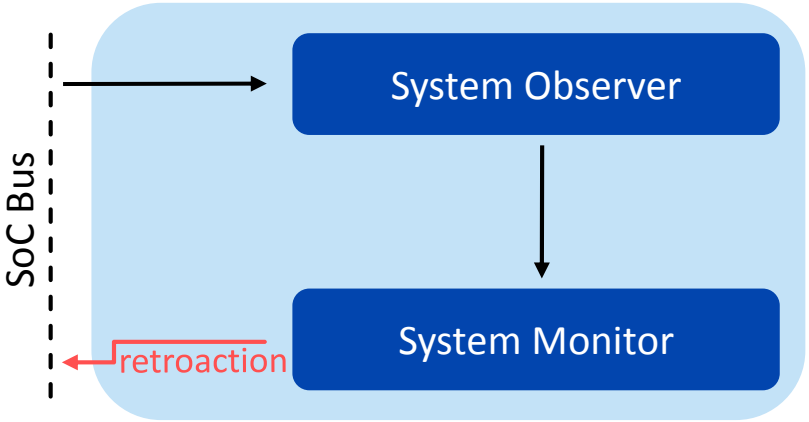
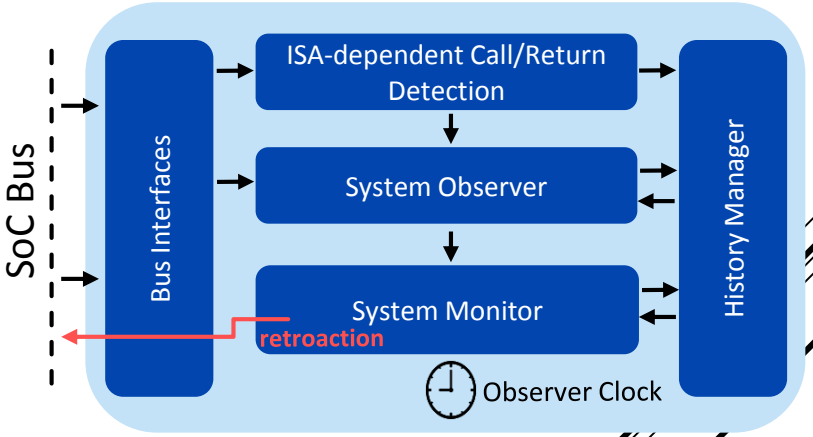


Fig. 7 - Complete Representation of the Observer Entity Architecture

Schematically, the **observer entity** is comprised of:

- **System Observer**
  - System Observer
  - ISA-dependent Call/Return Detection
- **System Monitor**
  - System Monitor
  - History Manager
- **Bus Interfaces**
- **Observer Clock**





Monitoring and Fault Detection:

Faults can be tackled at a coarse or fine grained degree depending on the knowledge level

- Huge level of detail
- Access to source code and binaries
- Familiarity with the types of input and output

} unrealistic scenario

that is why

We assume a zero-knowledge environment!

Monitoring and Fault Detection:

The observer will be able to detect **anomalies** such as:

- Unauthorized writing:
  - to **read-only memory**
  - outside of the **space reserved** for an application
  - outside the **data** or **bss section**
  - outside **function-grained stack frames**
- Alteration of data on the registers
  - return pointer substitutions
- **D**enial **o**f **S**ervice (DoS) attacks

We consider:

- **SPARC V8 (Gaisler)**
- **ARM (with CoreSight)**
- **Intel x86**

Through analysis of these architectures, we concluded

Not all of these architectures are technologically ready for the observer!

SPARC V8:

- 32-bit RISC architecture
- Mostly makes use of **registers** for:
  - Passing parameters
  - Storing global and local variables
- **8** global registers
- **24** registers in a **register window** (locals, ins and outs)
- Number of windows depends on implementation (2 to 32)

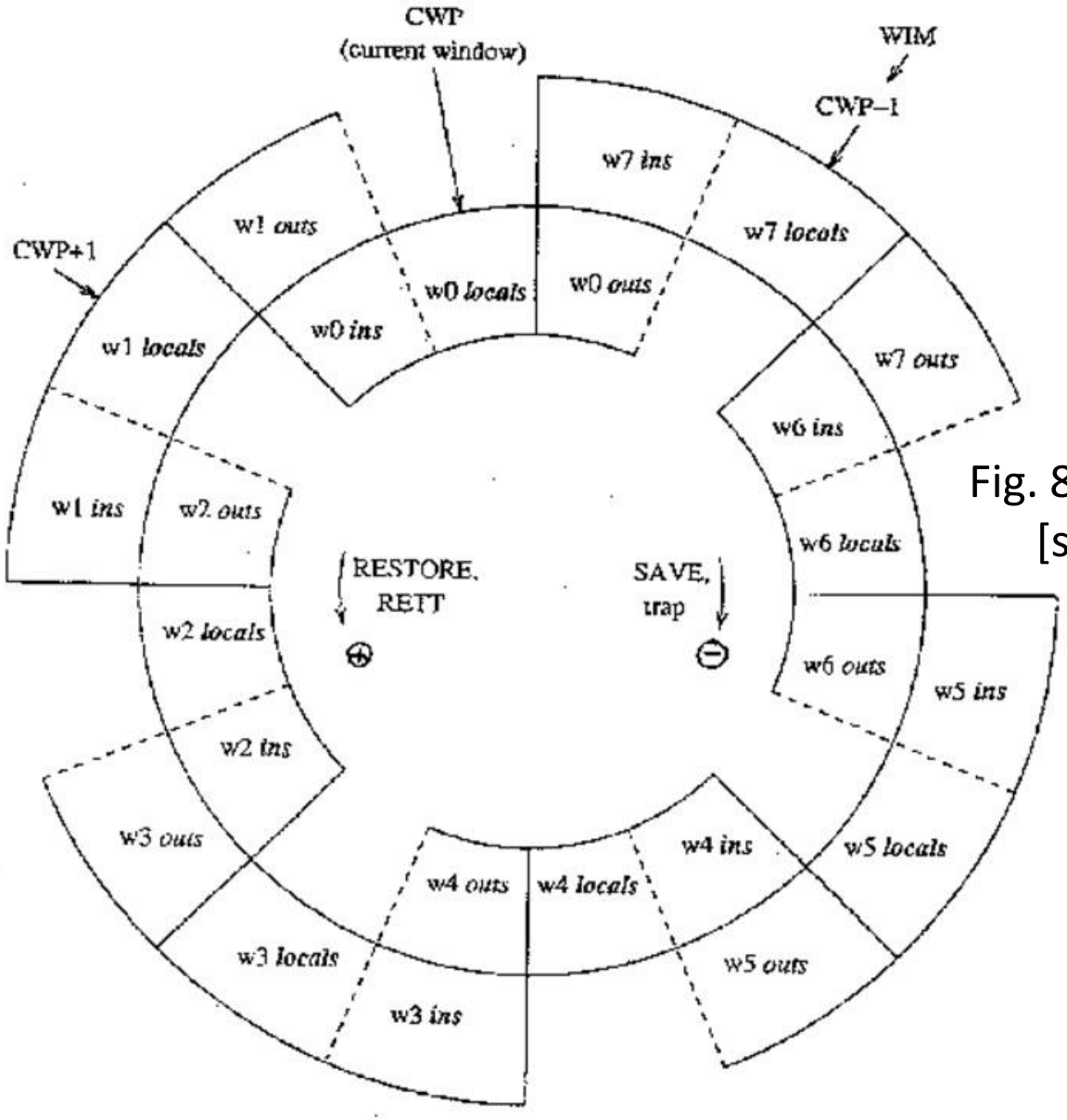


Fig. 8 – Register Window Rotation
[source: SPARC V8 Manual]

SPARC V8:

- **Arguments** and **variables** stored on the **stack** when out of **registers**
- Register window written on the stack when **window overflow**
- Stack organized in **stack frames**

call → saves current value of PC in %o7

```
save %sp, -x, %sp
...
...
...
restore
ret   =   jmpl %i7 + 8
```

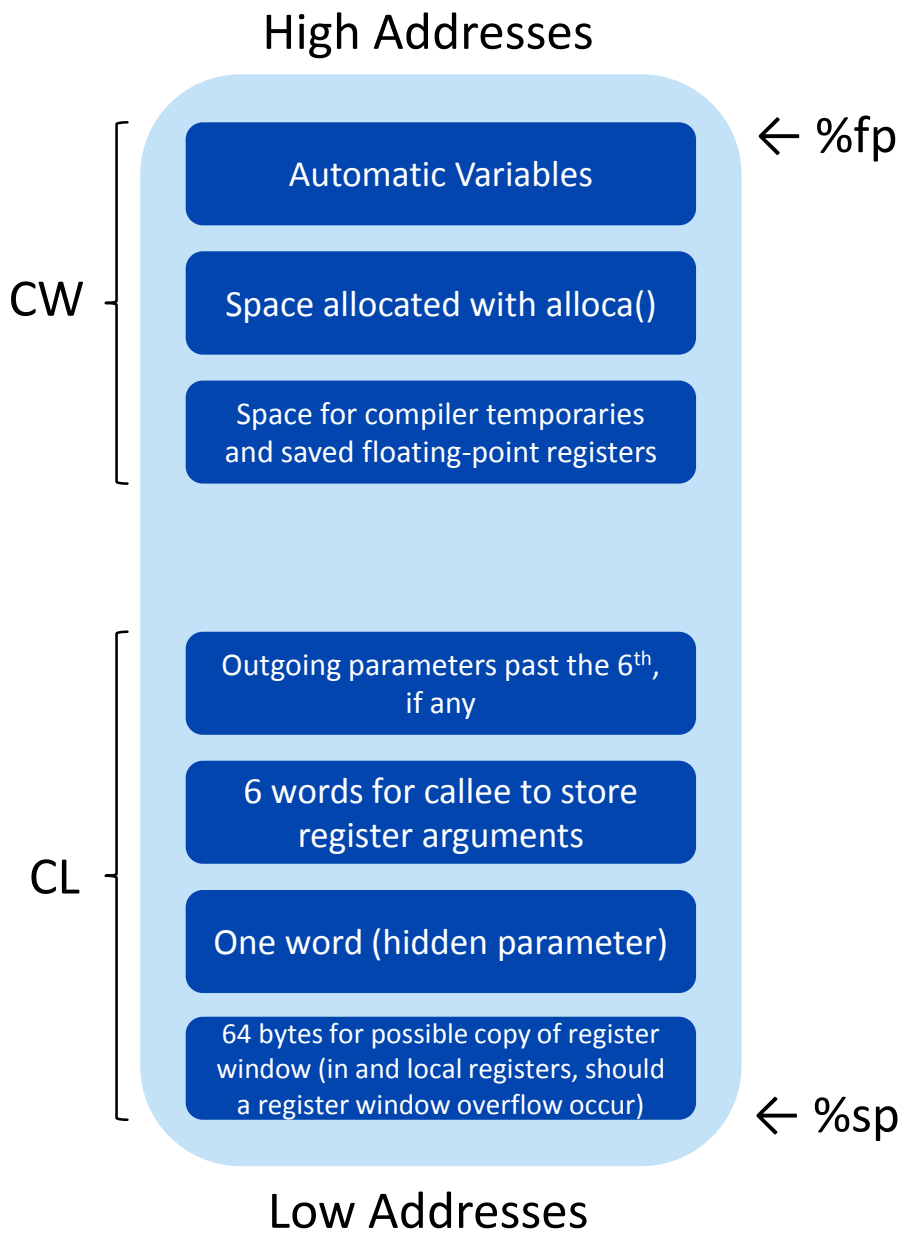


Fig. 9 – Stack Frame Organization

Observer connected to System Bus vs. Cache, an example:

Return Pointer Substitution

- System Bus:
  - **instructions being fetched** ✗
  - know when an instruction writes to a register ✗
- Cache:
  - **instructions being fetched** ✓
  - ex: know when an instruction writes to %i7 other than CALL ✓

return pointer register

Full-knowledge vulnerabilities

What we could know:

- Ex: **Location** and **sizes** of global and local variables

What we could solve:

- Ex: More **specific** types of buffer overflows
  - Variable-grained memory protection

Global variables' **addresses** and **sizes** could be directly extracted from the **binaries' symbol tables**.

Local variables, however, require binaries to be compiled with the right **debug options** so that their description can be accessed.

<u>If greater details were known:</u>

- Detect incorrect input and output
- Format string attacks
- Path traversal attacks
- Other common attacks and faults

Important: Safety + Security + Timeliness

Zero knowledge presents some challenges: types of **faults** and **attacks** to detect

→ which we address

Full knowledge observation and monitoring is also possible

The observer captures and reacts to information

And functions as a **non-intrusive black box**

Provides an **extra layer of security**

Questions?

http://www.navigators.di.fc.ul.pt