SCRATCHS: Side-Channel Resistant Applications Through Co-designed Hardware/Software

Frédéric Besson [1] Pascal Cotret [2] Nicolas Gaudin [2] Guy Gogniat [2]
Jean-Loup Hatchikian-Houdot [1] Guillaume Hiet [3] Vianney Lapôtre [2]
Pierre Wilke [3]

Contact: pierre.wilke@inria.fr

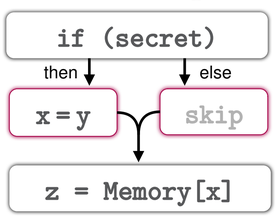
[1] EPICURE / IRISA / INRIA, Rennes
[2] Lab-STICC, UBS, Lorient / ENSTA Bretagne, Brest
[3] CIDRE / IRISA / INRIA, CentraleSupélec, Cesson-Sévigné

https://project.inria.fr/scratchs/

Context

Timing vulnerabilities caused by behavior depending on a secret

```
x = 0, y = 64
if (secret){
    x = y
}
z = Memory[x]
```



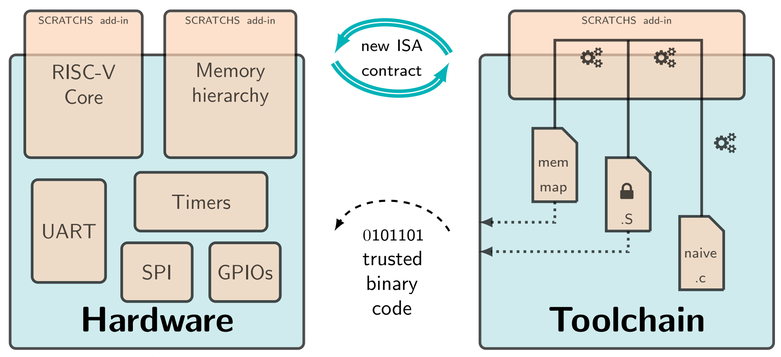
Timing differences :
- in branching
- in memory access

→ Secret exposed

Attacker: observes time ⇒ deduces `secret`

- Behavior duration depends on resource usage (like memory access).
- Timing is observable when resource usage is shared between the victim and the attacker.
- Countermeasures already exist (resources partitioning, Constant-Time programming), but are often costly.

SCRATCHS



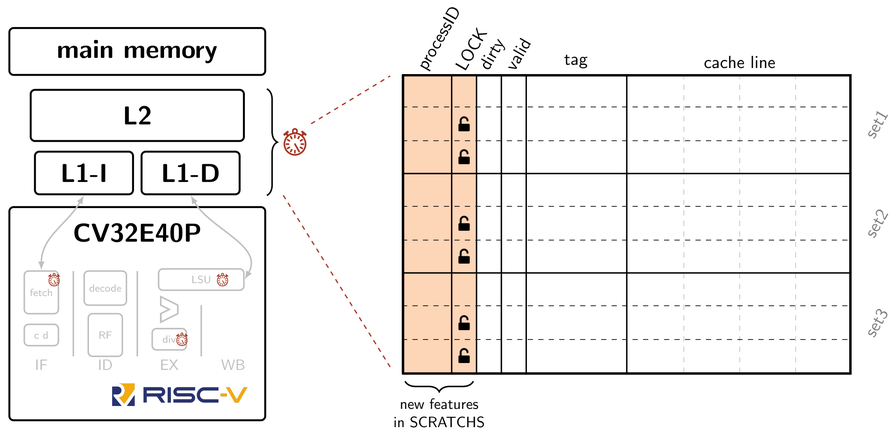
SCRATCHS's goal is to **co-design** a **RISC-V processor** and a **compiler toolchain**:

- Immune sensitive code to **timing side-channel attacks**.
- Minimal overhead on the micro-architecture.
- Considering a small-scale embedded system.

- **Hardware** implements **security mechanisms**.
- **Compiler** produces binaries able to use these mechanisms to be **side-channel resistant**.

`lock/unlock`

Memory hierarchy and some functional units temporal behaviors (e.g. ALU, **LSU**, **division** or branching) can leak **information** ⏱.

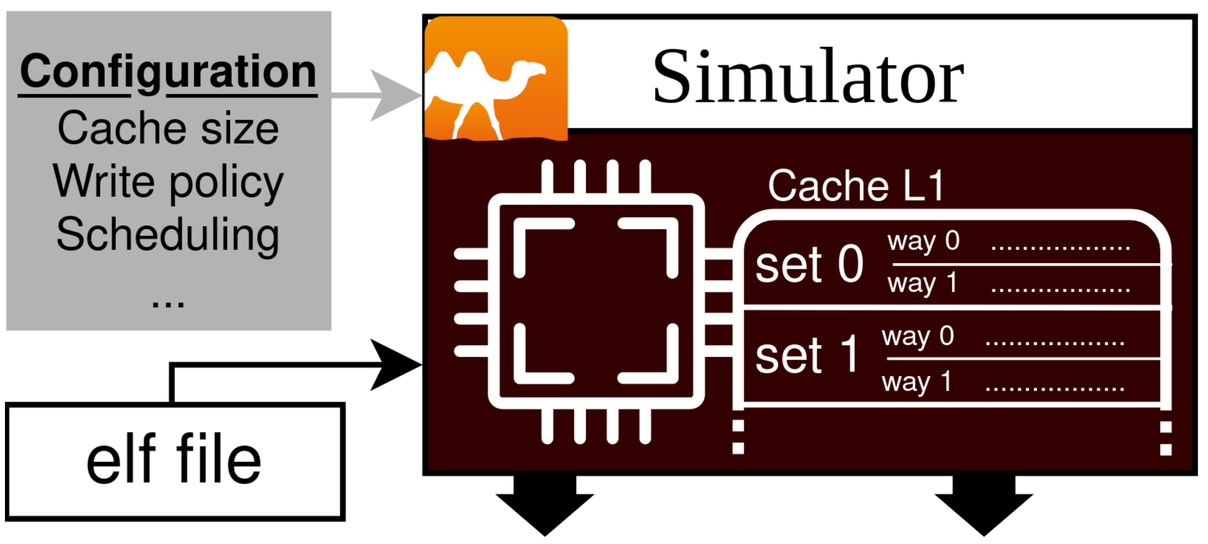


We identify three sources of leakage on the CV32E40P RISC-V processor:

Leak		Solutions
Division and modulo op.	→	Constant-time mode through a CSR register
Non-aligned data requests	→	Solved by compiler toolchain
Cache accesses (L1, L2, TLB...)	→	New `lock` and `unlock` instructions

`lock/unlock` mechanism:
- The cache line is locked in cache until the locking process issues an `unlock` operation
- At least one way of the cache is kept available to other processes' data
- Low overhead targeting FPGA (+4.7% on registers and +0.6% on LUTs)

Simulation and leakages



Program	Abstract leakage	Concrete leakage
int a = b + c;	[•]	[•]
int a = array[i];	cache_set(&array + i)	cache miss
lock(&array + i);	cache_set(&array + i)	cache hit
int a = array[i];	[•]	cache hit

Abstract leakage: What could be seen, depends on current program and inputs
Concrete leakage: What is observed. Depends on current instruction and cache state.

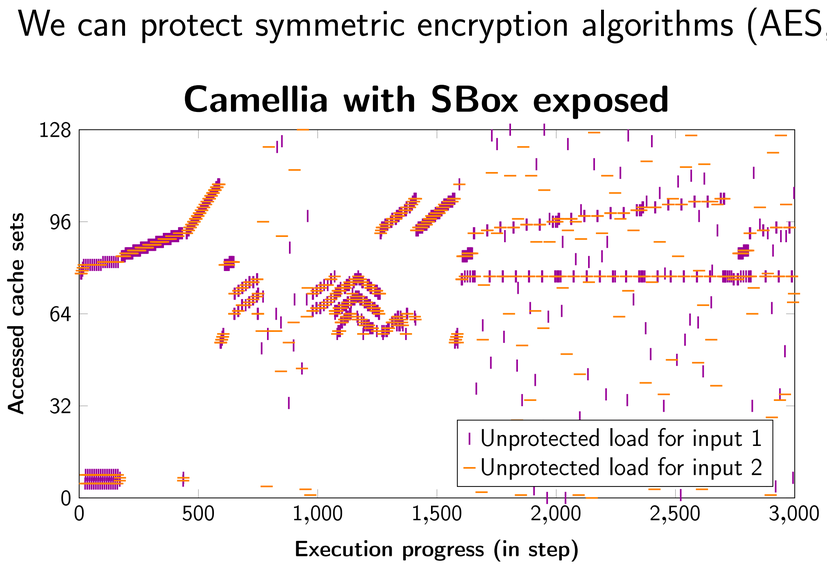
Our claim:

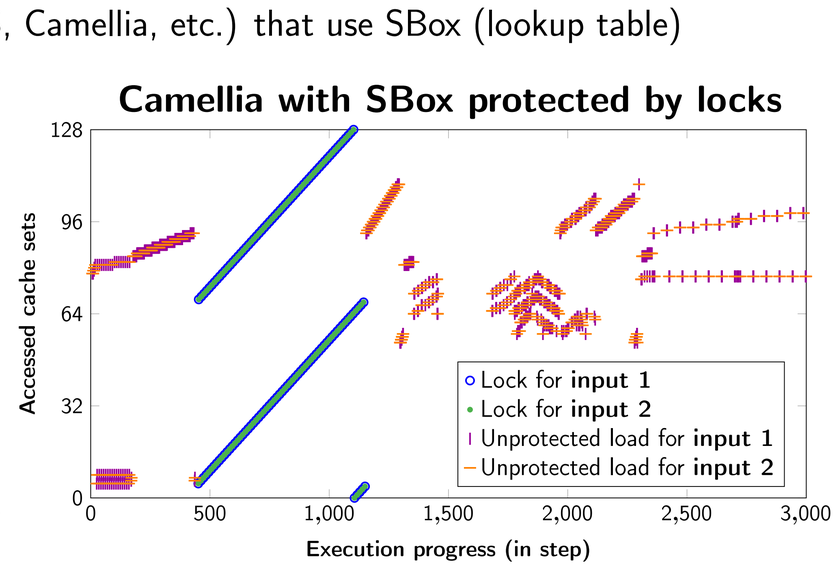
All information of **Concrete leakage** is deductible from **Abstract leakage**
⇒ Non interference of input in **Abstract leakage** means no timing attack possible
⇒ Security guarantee for a program possible (for a given input space)

Results

Security evaluation

We can protect symmetric encryption algorithms (AES, Camellia, etc.) that use SBox (lookup table)

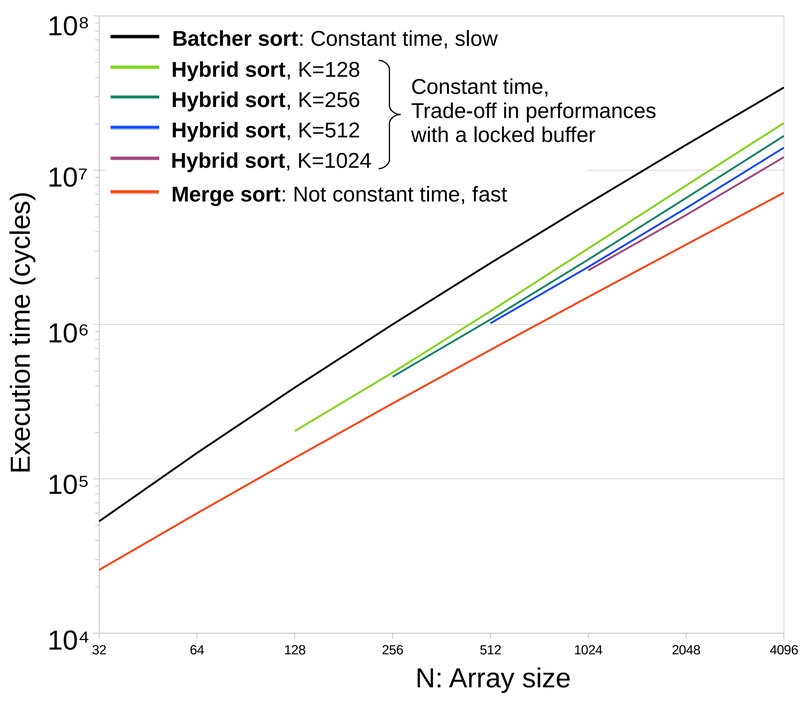




These figures display memory accesses that are visible to a potential attacker.
Variation of visible accesses depending on the input means the input is exposed to timing attacks.

Performance evaluation

Comparison between sorting algorithms



[1] N. Gaudin et al., "Work in Progress: Thwarting Timing Attacks in Microcontrollers using Fine-grained Hardware Protections," 2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), Delft, Netherlands, 2023, pp. 304-310, doi: 10.1109/EuroSPW59978.2023.00038.