Fine-grained dynamic partitioning against cache-based side channel attacks

Nicolas GAUDIN

SemSecuElec Seminar - June 27, 2025

Summary

- Context
- State of the art
- Fine-grained locking mechanism
- Implementation with an N-way set associative cache
- Conclusion

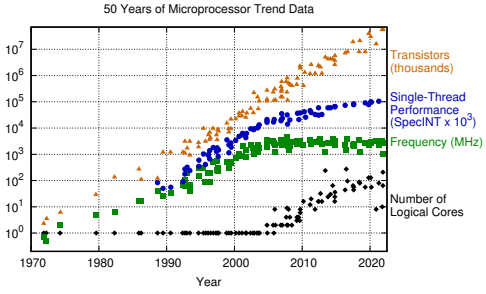
Summary

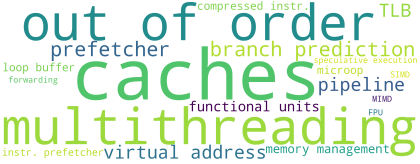
Context

- Number of IoT devices soars
- Wide usage
- Critical applications
  - Health
  - Bank
  - Industry
  - Transport
  - Logistic

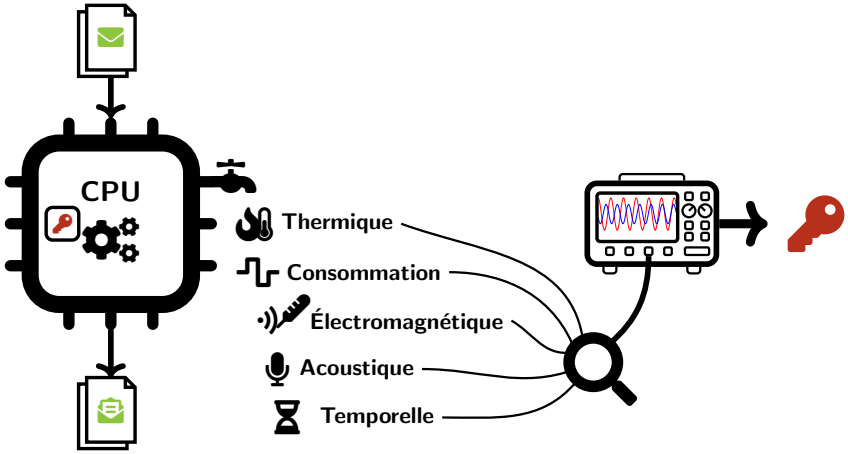
Context

- Number of IoT devices soars
- Wide usage
- Critical applications
  - Health
  - Bank
  - Industry
  - Transport
  - Logistic

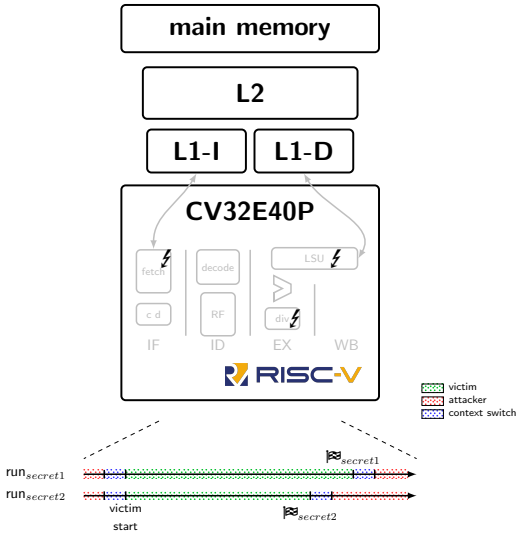




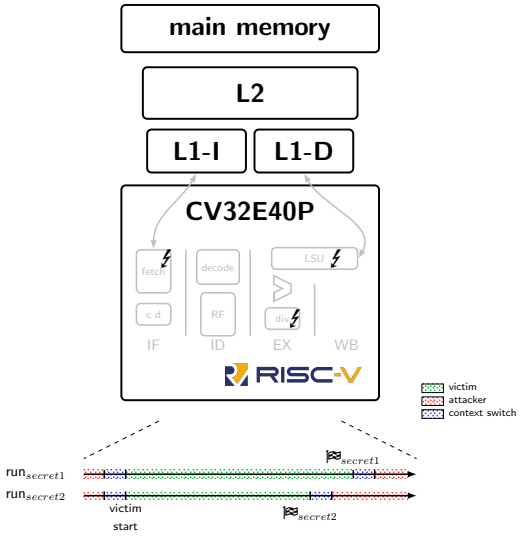
Side Channel Attacks (SCA)



Timing leakage



Timing leakage



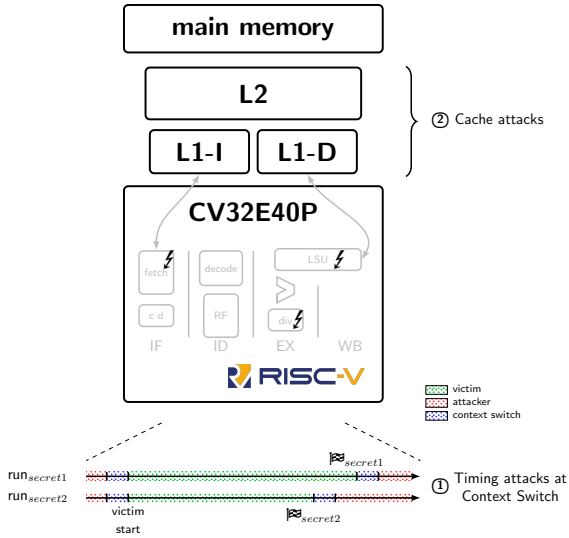
Sources of leakages exploited by :

Branching
if (condition(**secret**))

Operation with variable execution time
dividend/**secret**;

Index for Memory access
array[**secret**];

Timing leakage



Sources of leakages exploited by :

Branching

if (condition(**secret**))

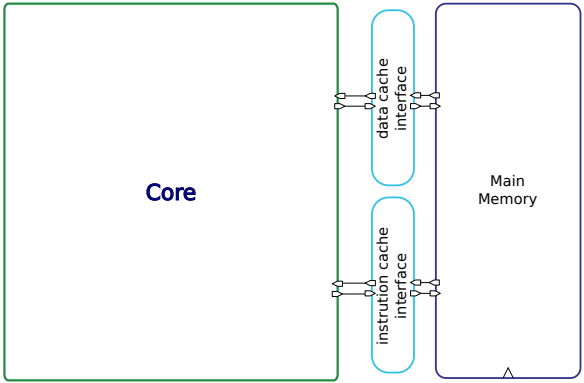
Operation with variable execution time

dividend/**secret**;

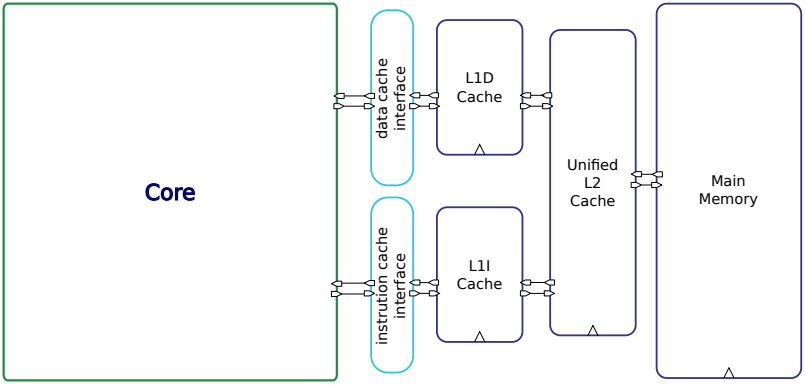
Index for Memory access

array[**secret**];

Cache Memories



Cache Memories

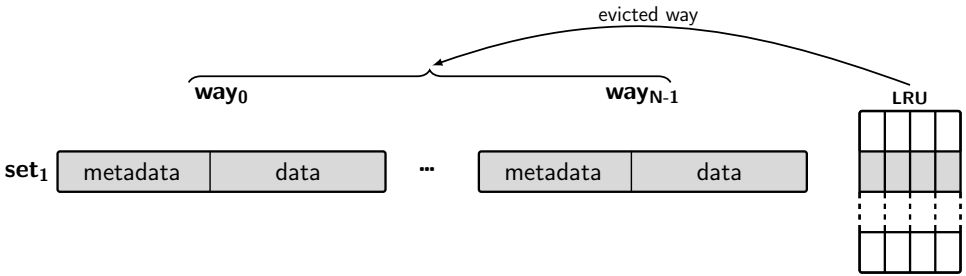


Cache Memories - N-way set associative cache

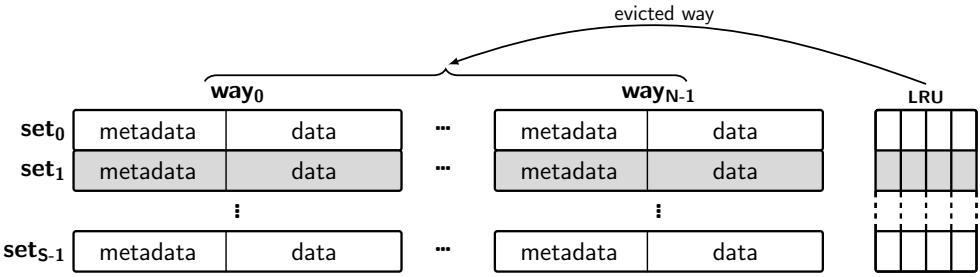
Cache Memories - N-way set associative cache

Cache Memories - N-way set associative cache

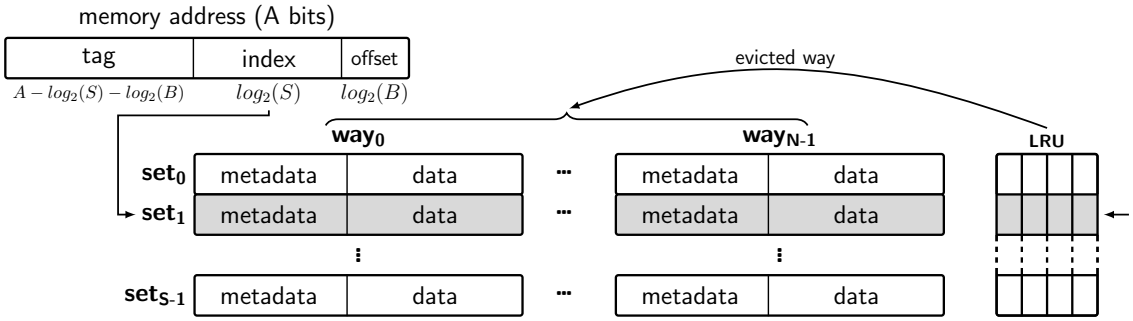
Cache Memories - N-way set associative cache



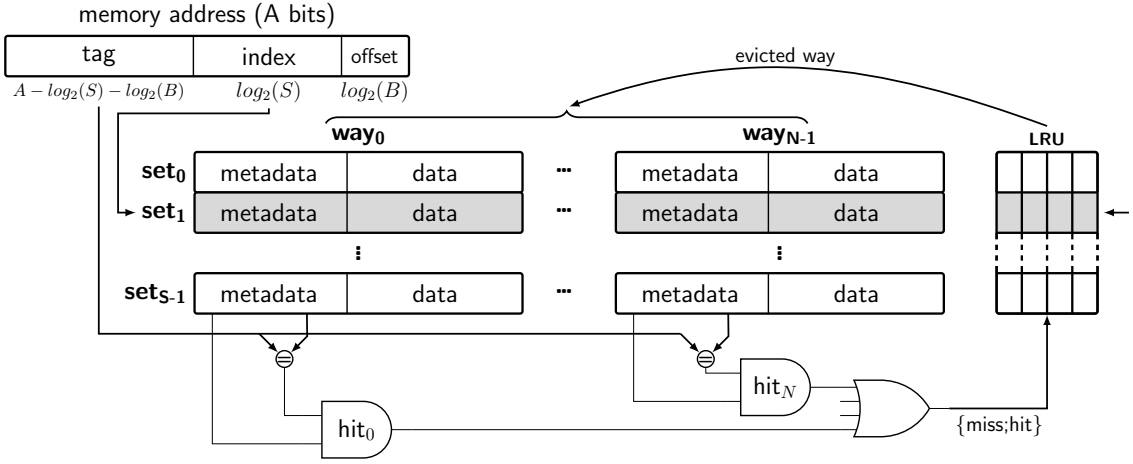
Cache Memories - N-way set associative cache



Cache Memories - N-way set associative cache



Cache Memories - N-way set associative cache



Prime+Probe attack

Objectives :

- Infer which cache set(s) is accessed by the victim
- Observe behavior of the attacker memory accesses

Prime+Probe attack

Objectives :

- Infer which cache set(s) is accessed by the victim
- Observe behavior of the attacker memory accesses

Requirements :

- Eviction Set matching with victim addresses
- Shared cache resource
- (High) Precision Timer

Prime+Probe attack

Objectives :

- Infer which cache set(s) is accessed by the victim
- Observe behavior of the attacker memory accesses

Requirements :

- Eviction Set matching with victim addresses
- Shared cache resource
- (High) Precision Timer

How ? :

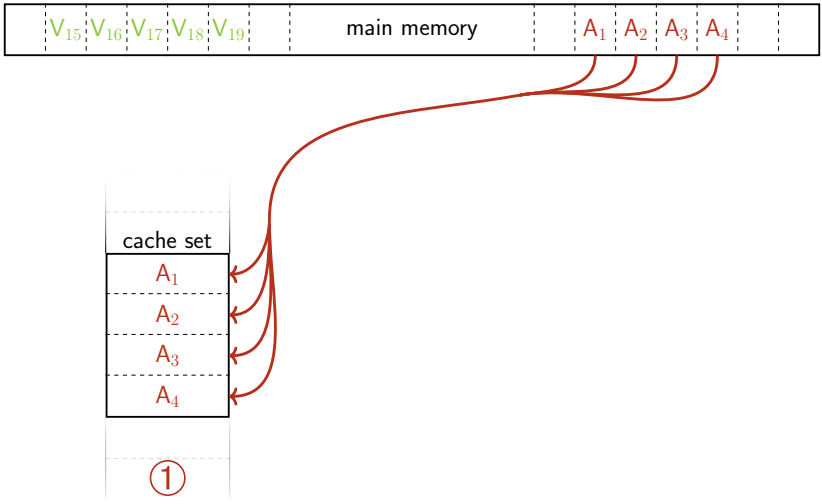
- with a 3-step attack

① fill a cache set using the eviction set

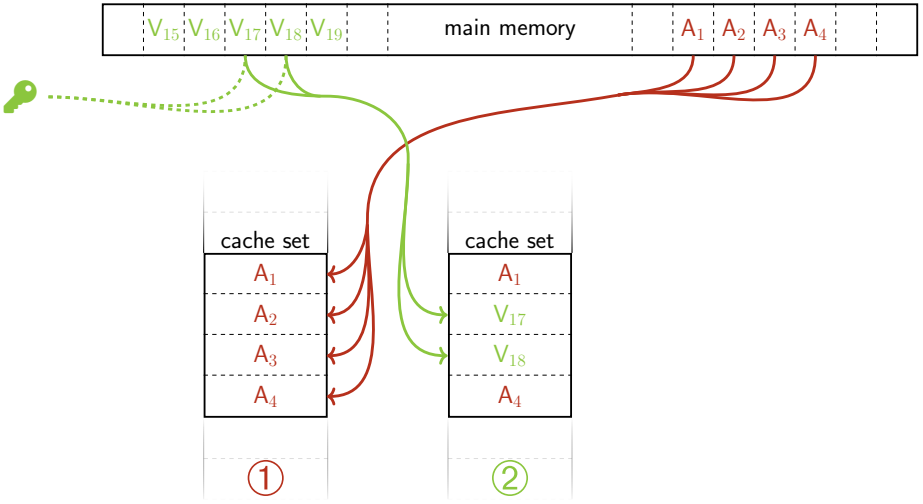
② let the victim execute

③ access and time each address of the eviction set

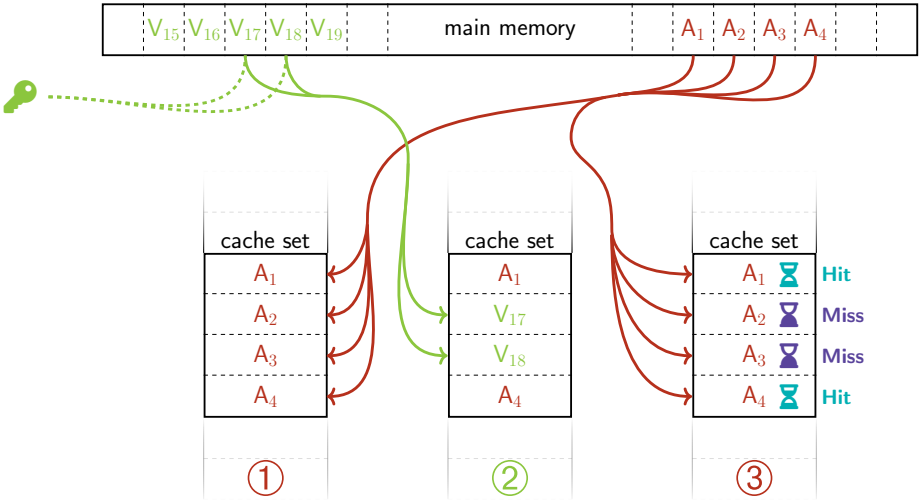
Prime+Probe attack



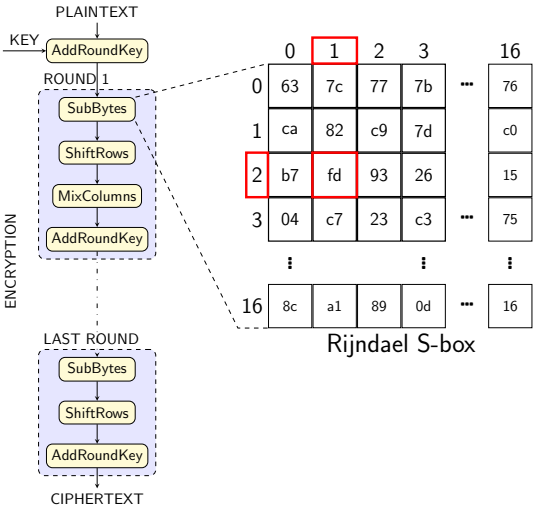
Prime+Probe attack



Prime+Probe attack

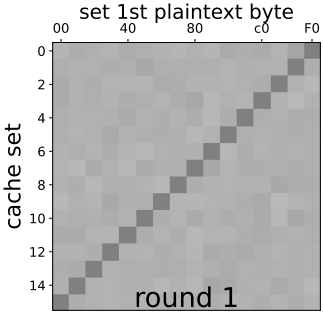


Considered attack on AES-128

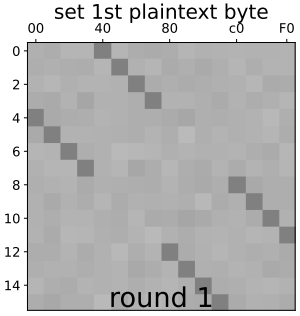


- SubBytes step accesses $\mathrm{SBOX}[P_i \oplus K_i]$
- Known plaintext attack

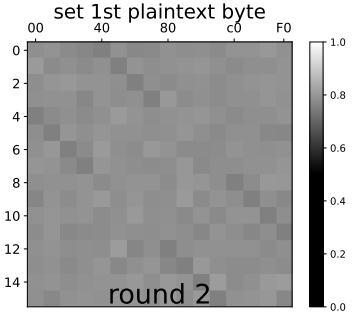
Example with Prime+Probe on AES-128



key = 0xFF



key = 0x42

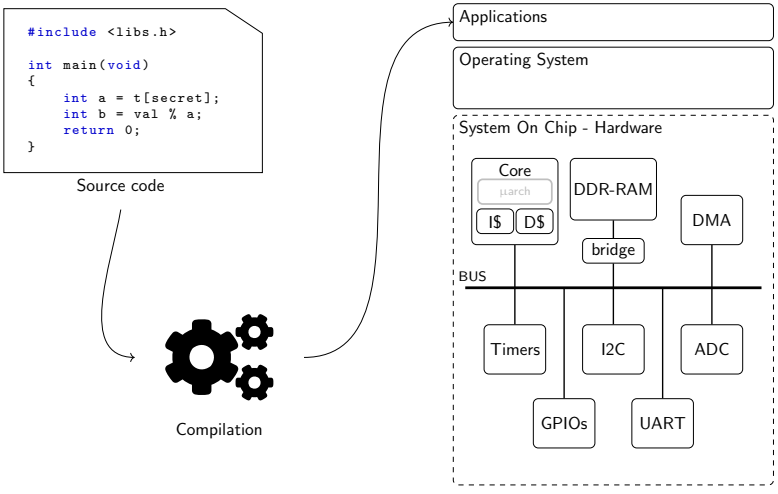


key = 0x42

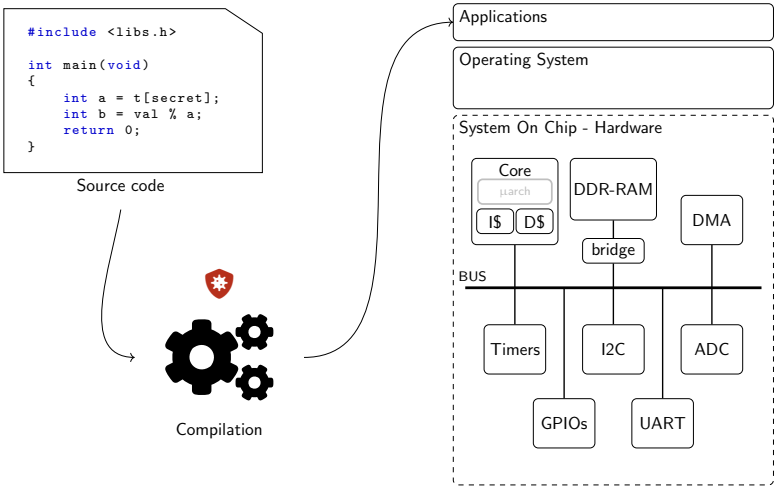
Summary

- Context
- State of the art
  - Detection-based
  - Randomization-based
  - Partitioning-based
- Fine-grained locking mechanism
- Implementation with an N-way set associative cache
- Conclusion

Detection-based countermeasures

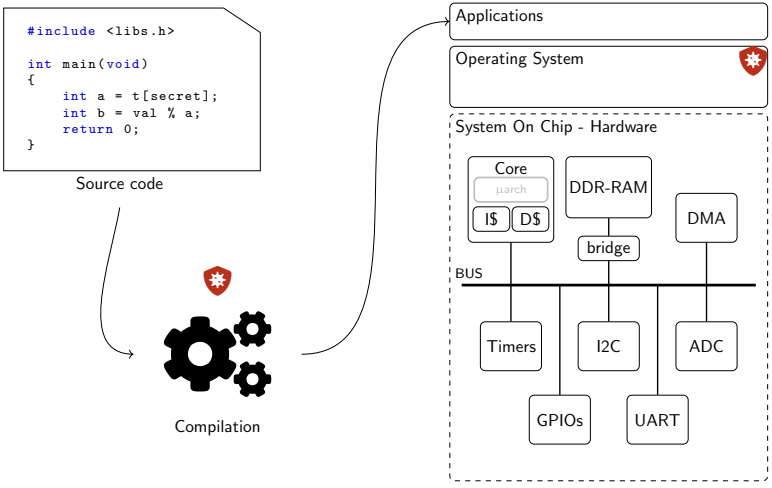


Detection-based countermeasures



- Winderix *et al.* [1]
- CacheBar [2]

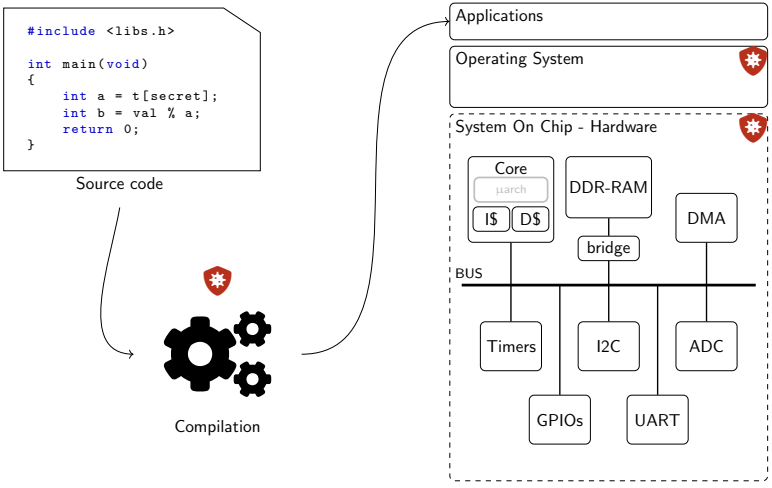
Detection-based countermeasures



- Winderix *et al.* [1]
- CacheBar [2]

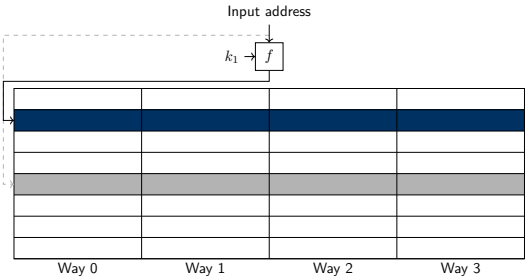
- Nights-Watch [3]
- WHISPER [4]

Detection-based countermeasures



- Winderix *et al.* [1]
- CacheBar [2]
- Nights-Watch [3]
- WHISPER [4]
- täkō [5]

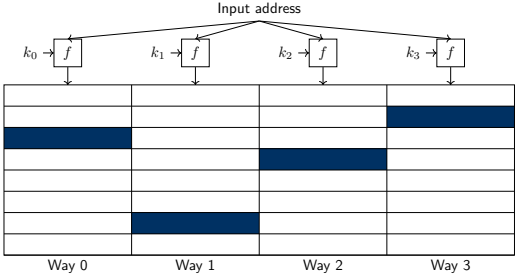
Randomization-based countermeasures - *Associative ways*



- Randomized the accessed cache set
  - Permutation table
  - Index Derivation Function

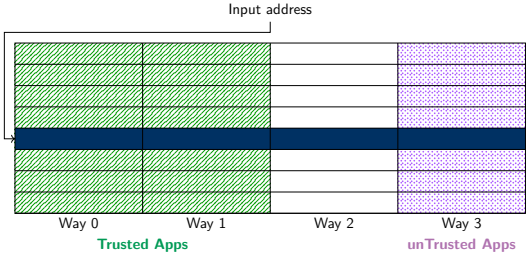
- RPcache [6]
- CEASER [7]
- ScrambleCache [8]
  - Easy to bypass this randomness [9]
  - Need to update primitives frequently
  - Maintain security costs performances

Randomization-based countermeasures - *Skewed ways*



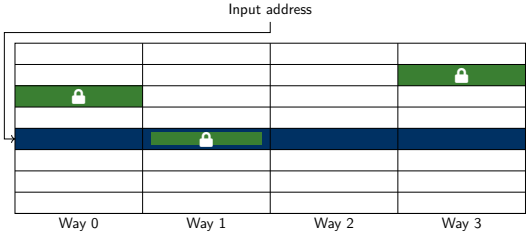
- Randomized accessed cache lines dissociating cache set
  - IDF for each way
- CEASER-S [10]
- ScatterCache [11]
  - Disruptive attack allows to find eviction set
  - Prime+Prune+Probe [9]
- ClepsydraCache [12]
  - Analog/digital costly design

Partitioning-based countermeasures - *coarse-grained*



- Partition resource to avoid conflicts
- Software-based :
  - COLORIS [13]
  - COTSknight [14]
- Hardware-based :
  - NoMoCache [15]
  - SecDCP [16]
    - Partitions do not meet needs

Partitioning-based countermeasures - *fine-grained*



- Partition resource to avoid conflicts
- Fine-grained :
  - Vantage [17]
  - PLcache [6]
    - Avoid conflict based attacks but leaks on LRU update

State of the Art Synthesis

- **Randomization-based**
  -  Autonomous
  -  Need to frequently update security (and so invalidate the cache)
- **Coarse Grained Partitionning**
  -  Provide a security support for OS/applications
  -  Can widely affect performances

State of the Art Synthesis

- **Randomization-based**
  - Autonomous
  - Need to frequently update security (and so invalidate the cache)
- **Coarse Grained Partitionning**
  - Provide a security support for OS/applications
  - Can widely affect performances

- Countermeasures are designed considering complex system
  - Doesn't fit with embedded systems requirements/possibilities
  - Often focus on Last Level Cache (large shared cache)
- Not directly compatible with embedded systems

State of the Art Synthesis

- **Fine Grained Partitionning**
  -  Provide a security support for OS/applications
  -  Affect slightly performances

State of the Art Synthesis

- **Fine Grained Partitionning**
  - Provide a security support for OS/applications
  - Affect slightly performances

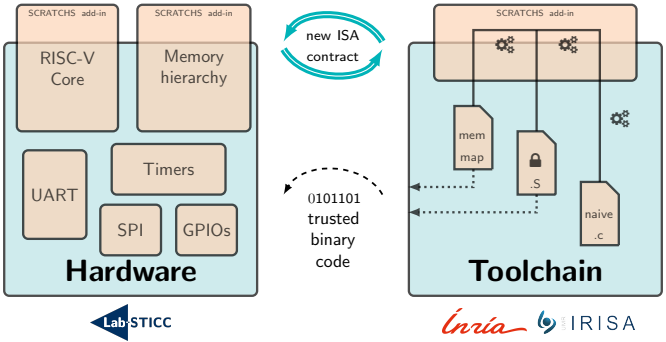
- PLcache is a candidate for our security mechanism
  - Introduce new instructions to reserve cache lines
  - Fit with embedded systems requirement and limitations

Summary

- Context
- State of the art
- Fine-grained locking mechanism
- Implementation with an N-way set associative cache
- Conclusion

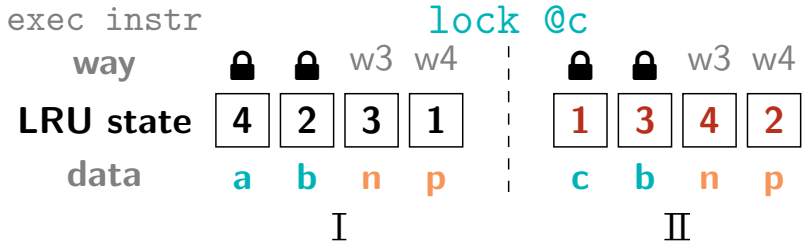
SCRATCHS project

Side-**C**hannel **R**esistant **A**pplications **T**hrough **C**o-designed **H**ardware/**S**oftware



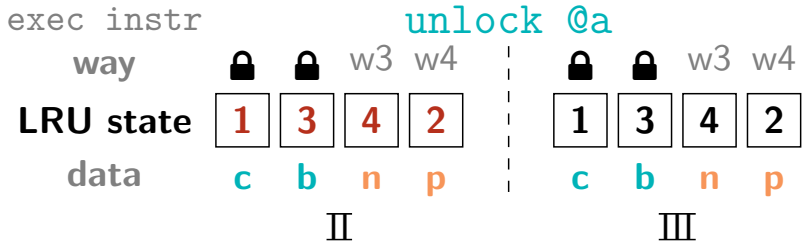
Ensure **efficient** and **on-demand** constant-time execution

PLcache [6] issues



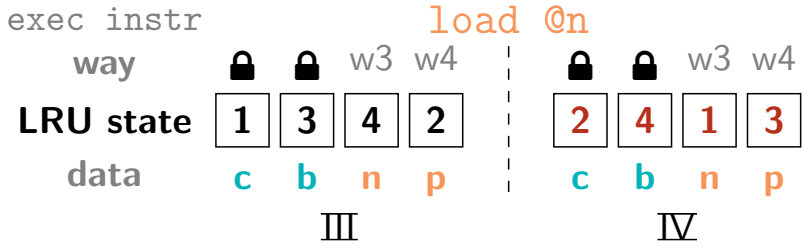
- No contant time accesses on locked data

PLcache [6] issues



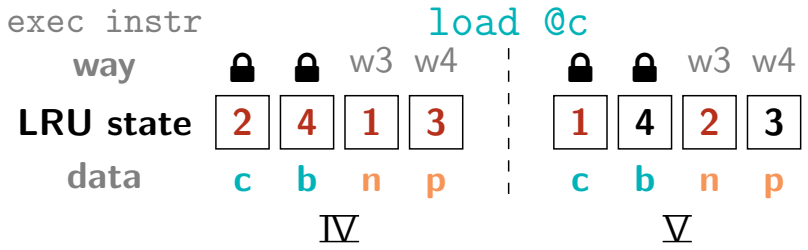
- No contant time accesses on locked data

PLcache [6] issues



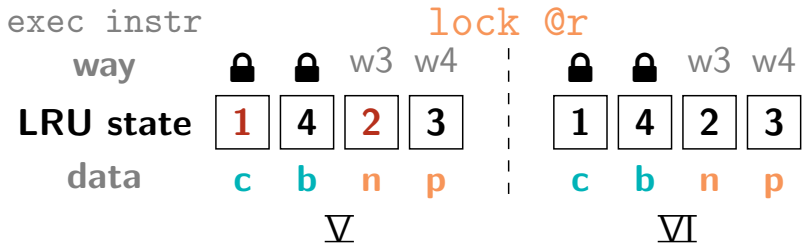
- No contant time accesses on locked data
- Shared LRU state among processes [18]

PLcache [6] issues



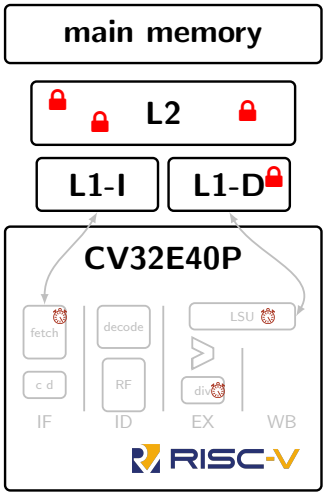
- No contant time accesses on locked data
- Shared LRU state among processes [18]

PLcache [6] issues



- No contant time accesses on locked data
- Shared LRU state among processes [18]
- Accesses can circumvent cache

Our `lock` Mechanism

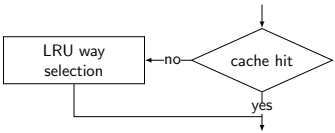


- Extend the Instruction Set Architecture
  - `lock` and `unlock` instructions

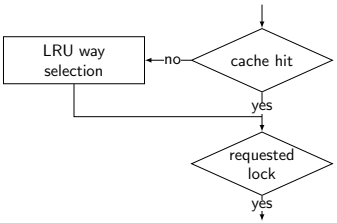
🔒 `lock` instr. keeps data cache line in cache
  - guarantee constant time access
  - locked cache line cannot be evicted
  - mitigate Evict+Time and Prime+Probe

🔓 `unlock` instr. releases locked cache line
  - data can be evicted

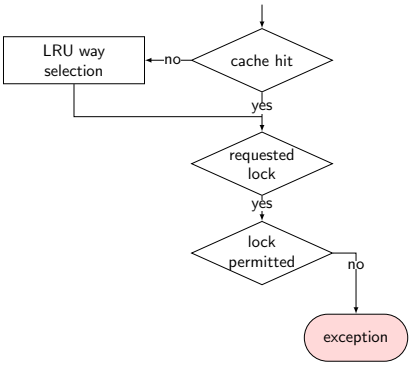
Cache access procedure with locking mechanism



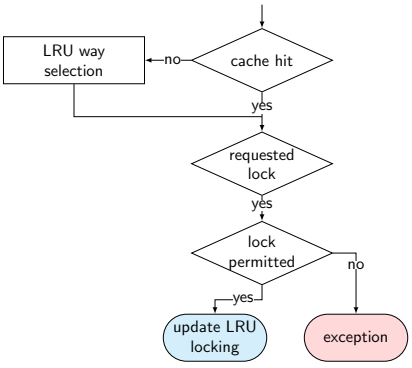
Cache access procedure with locking mechanism



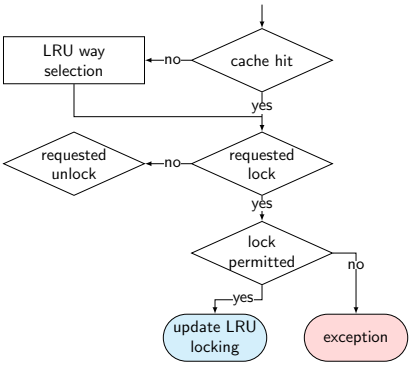
Cache access procedure with locking mechanism



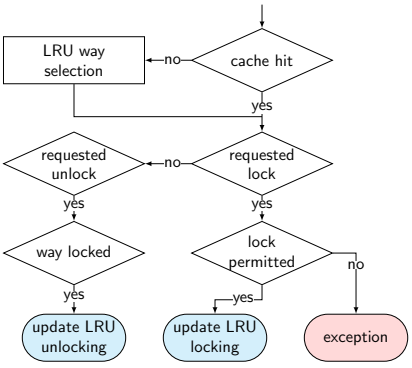
Cache access procedure with locking mechanism



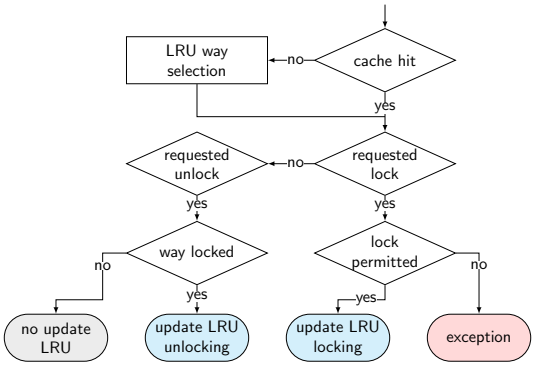
Cache access procedure with locking mechanism



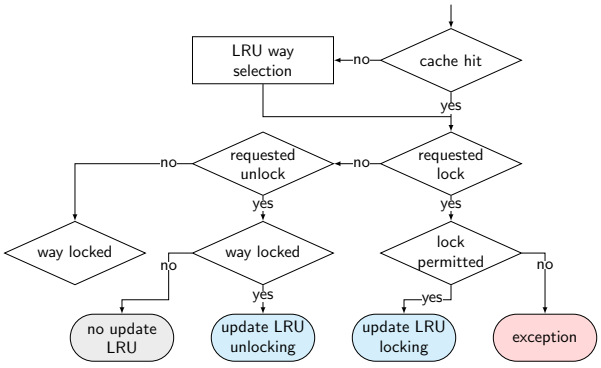
Cache access procedure with locking mechanism



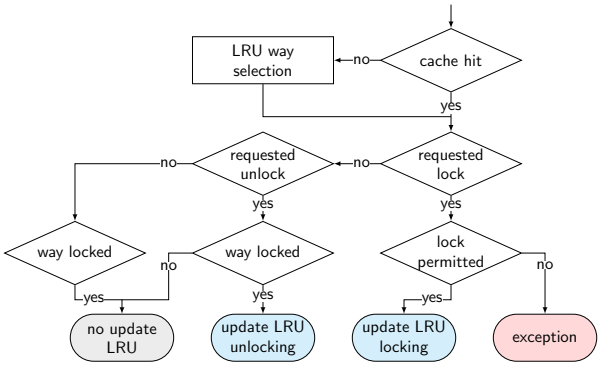
Cache access procedure with locking mechanism



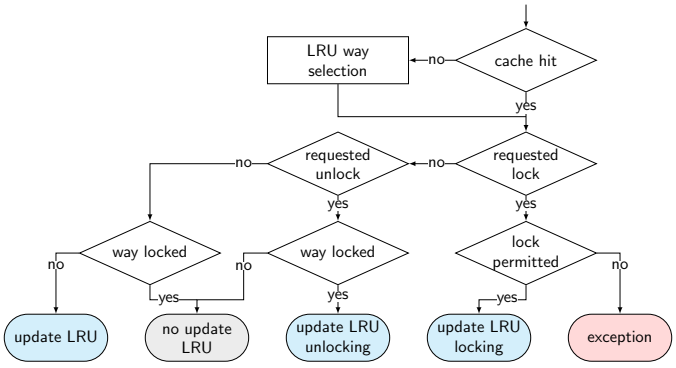
Cache access procedure with locking mechanism



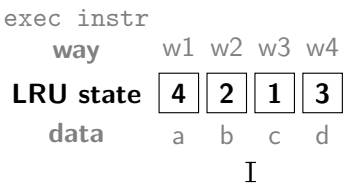
Cache access procedure with locking mechanism



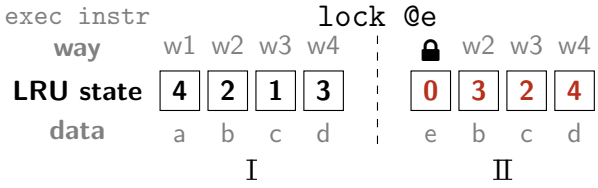
Cache access procedure with locking mechanism



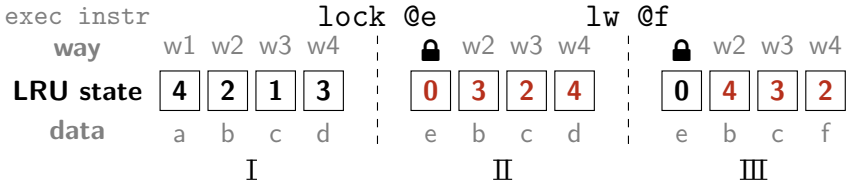
Replacement policy update



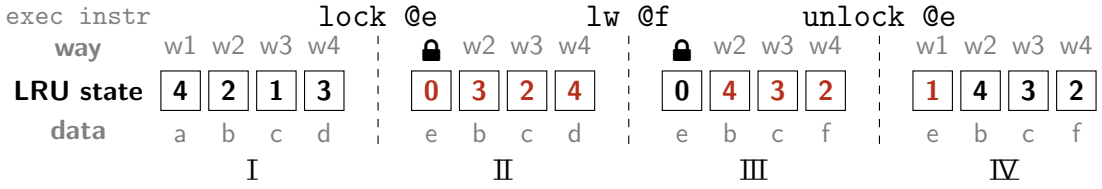
Replacement policy update



Replacement policy update



Replacement policy update



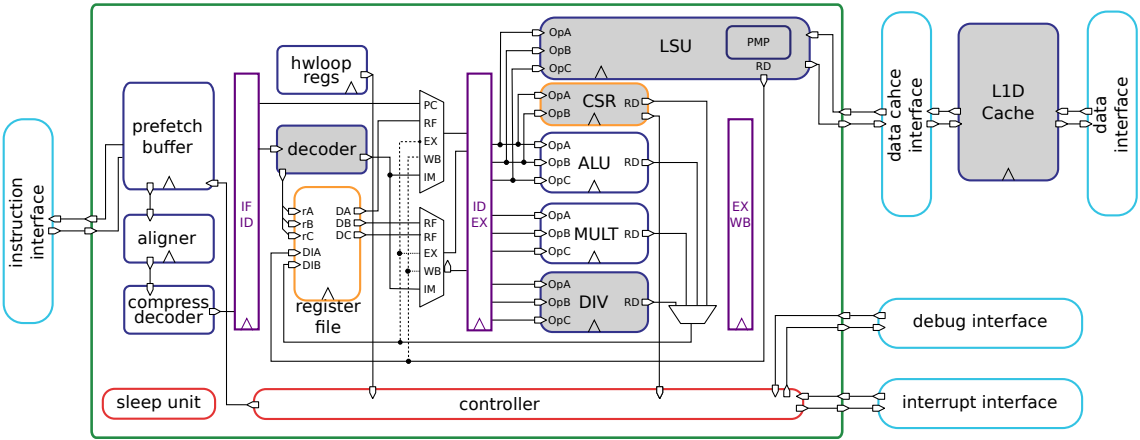
Summary

- Context
- State of the art
- Fine-grained locking mechanism
- Implementation with an N-way set associative cache
- Conclusion

Goal of this implementation

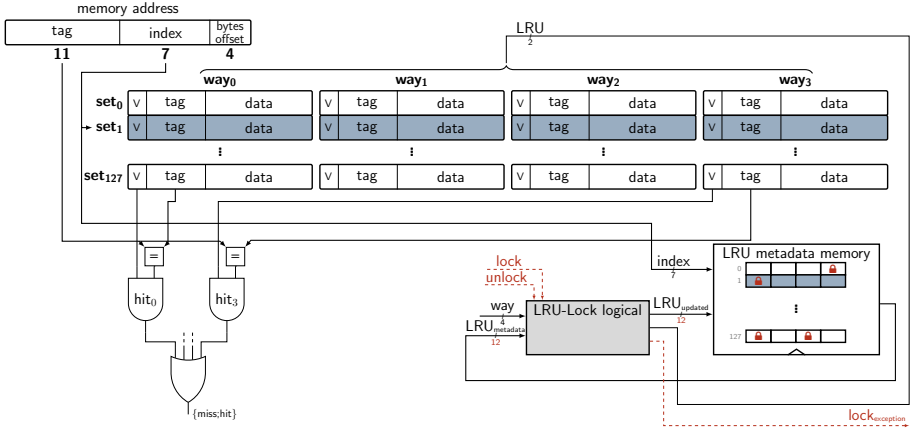
- Evaluate the **security effectiveness** of `lock`
- Evaluate **impact on performances** on cache sized for embedded systems
  - on processes relying on locking mechanism
  - on processes while other apps relies on locking mechanism

Implementation - *core level*

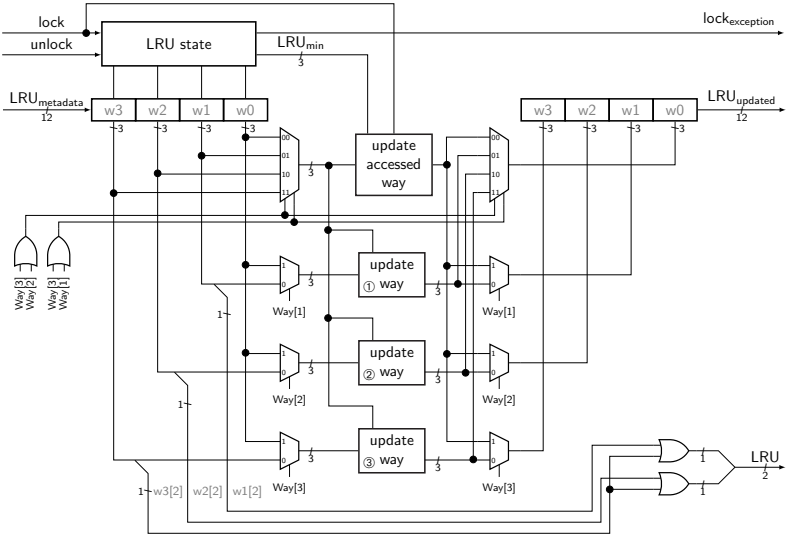


based on the OpenHWgroup CV32E40P core

Implementation - *cache level*



Implementation - *replacement policy level*



Impact on resource utilization

Post implementation area result

	without lock			with lock			Overhead	
	LUT	FF	BRAM	LUT	FF	BRAM	LUT	FF
↪ LRU	26	24	0,5	50	34	0,5	+ 92,31%	+ 41,67%
↪ Cache	980	1 065	8,5	1 007	1 077	8,5	**+ 2,76%**	**+ 1,1%**
↪ Core	4 669	2 233	0	4 666	2 235	0	- 0,06%	+ 0,09%
CPU	5 661	3 467	8,5	5 683	3 481	8,5	**+ 0,39%**	**+ 0,40%**

Considering AMD/Xilinx Kintex-7 FPGA family with Vivado 2022.2

Impact on resource utilization

Post implementation area result

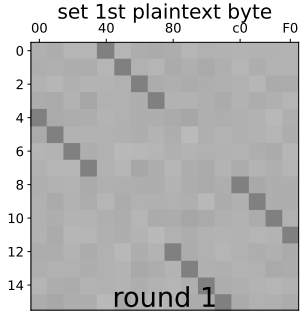
	without lock			with lock			Overhead	
	LUT	FF	BRAM	LUT	FF	BRAM	LUT	FF
↪ LRU	26	24	0,5	50	34	0,5	+ 92,31%	+ 41,67%
↪ Cache	980	1 065	8,5	1 007	1 077	8,5	**+ 2,76%**	**+ 1,1%**
↪ Core	4 669	2 233	0	4 666	2 235	0	- 0,06%	+ 0,09%
CPU	5 661	3 467	8,5	5 683	3 481	8,5	**+ 0,39%**	**+ 0,40%**

Considering AMD/Xilinx Kintex-7 FPGA family with Vivado 2022.2

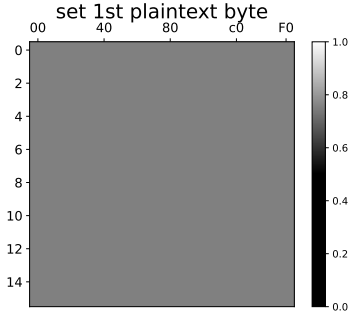
BRAM' Bits overhead

At cache level, baseline stores **72,704 bits** (cache lines + LRU metadata)
+ **512 bits** to implement our lock mechanism ▸ overhead of 0.7%.

Impact on Security - *considering* Prime+Probe *on AES-128*



Unprotected



using `lock`

Takeaway

The number of locked cache lines does not have to depend on secret.

Impact on Performances - *using `lock`*

Binary code size overhead

- AES-128 => **0,28%**
- Camellia => **0,23%**

Overhead induced by the insertion of `lock` and `unlock` instructions.

Impact on Performances - *using `lock`*

Binary code size overhead

- AES-128 => **0,28%**
- Camellia => **0,23%**

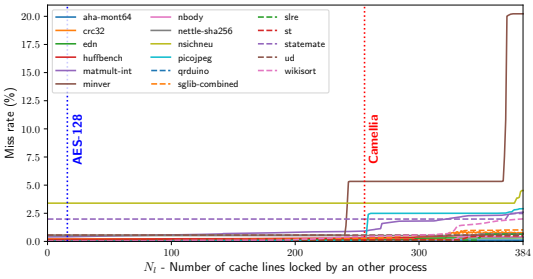
Overhead induced by the insertion of `lock` and `unlock` instructions.

Execution time overhead (+%)

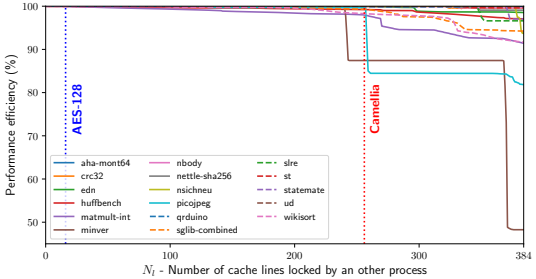
N_{Blocks}	1	4	8	16	64	128	512	1024
Camellia	367,7	99,6	54,22	28,88	7,62	3,85	0,97	0,48
AES-128	2,77	0,71	0,35	0,18	0,04	0,02	-	-

Camellia is an AES-like algorithm. Camellia is a **fast** encryption using **four tables** (4 kio of SBOXs).

Impact on Performances - *on Embench-IoT benchmark*



Miss rate



Execution time

Takeaway

AES-128 and Camellia have a **negligeable impact** on Embench-IoT.

Implementation Synthesis

At a glance :

- Locking mechanism implementation implies a **low area** overhead (<3% on cache)
- **Low** (negligeable) **impact** on overall performance

- Locking mechanism provides a **fine-grained** efficient and **on-demand security** against timing SCAs

Implementation Synthesis

At a glance :

- Locking mechanism implementation implies a **low area** overhead (<3% on cache)
- **Low** (negligeable) **impact** on overall performance

- Locking mechanism provides a **fine-grained** efficient and **on-demand security** against timing SCAs

Can a combination with a random skewed cache push back the limits ?

- Identify the exact number of locked cache lines
- Limit usage of lock
- Renew keys frequently to maintain security

Summary

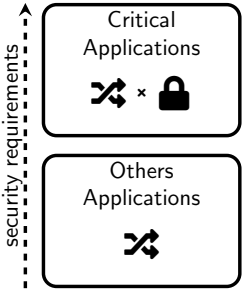
- Context
- State of the art
- Fine-grained locking mechanism
- Implementation with an N-way set associative cache
- Conclusion

Going further with our approach

- **Prevent** from a **new disruptive attack** to build eviction set (as Prime+Prune+Probe releases)

Going further with our approach

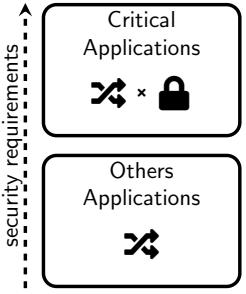
- **Prevent** from a **new disruptive attack** to build eviction set (as Prime+Prune+Probe releases)
- Provide security guarantees while an eviction set is up
  - keep critical applications unvulnerable
  - avoid attacker infering number of locked cache lines



🔀 maintains security for 6.8 M of memory accesses against P+P+P [19]

Going further with our approach

- **Prevent** from a **new disruptive attack** to build eviction set (as Prime+Prune+Probe releases)
- Provide security guarantees while an eviction set is up
  - keep critical applications unvulnerable
  - avoid attacker infering number of locked cache lines
- Evaluate **hardware resource overhead** to implement such a system



🔀 maintains security for 6.8 M of memory accesses against P+P+P [19]

Conclusion

- We extend the RISC-V ISA to :
  - **guarantee cache hit** for memory accesses on locked data
  - mitigate Evict + Time, Prime+Probe

- The locking mechanism implementation implies a **low area** overhead (<3%)
- **Low** (negligeable) **impact** on overall performance

- The locking mechanism provides a **fine-grained** efficient and **on-demand security** against timing SCAs

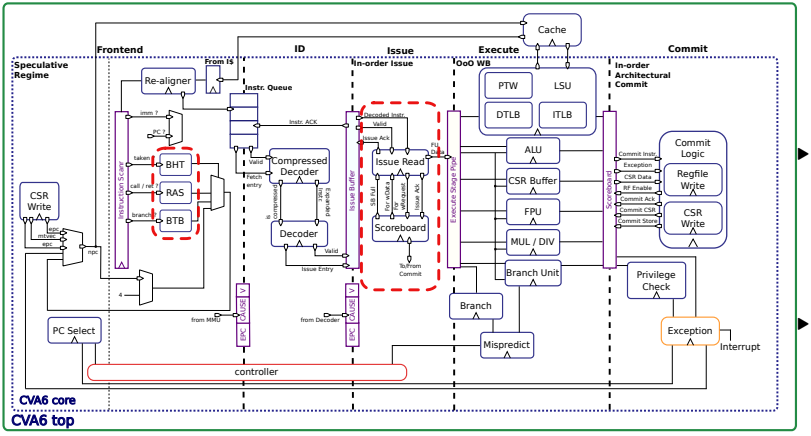
Fine-grained dynamic partitioning against cache-based side channel attacks

Nicolas GAUDIN

SemSecuElec Seminar - June 27, 2025

Perspectives - Axe ①

- Study impact of the microarchitecture

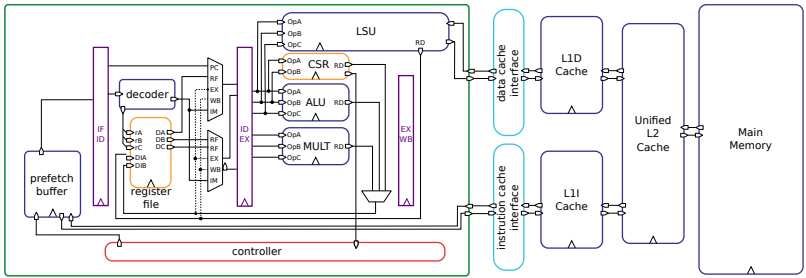


CVA6 Block diagram - *sources: CVA6 & Kévin Q.*

- Force the use of `fence` instruction on out of order
- Speculative execution of `unlock` on locked data

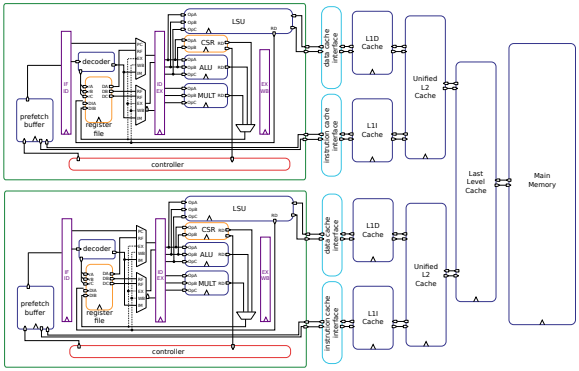
Perspectives - Axe ②

- Manage locking mechanism on more complex CPU
  - Extend the cache memory hierarchy



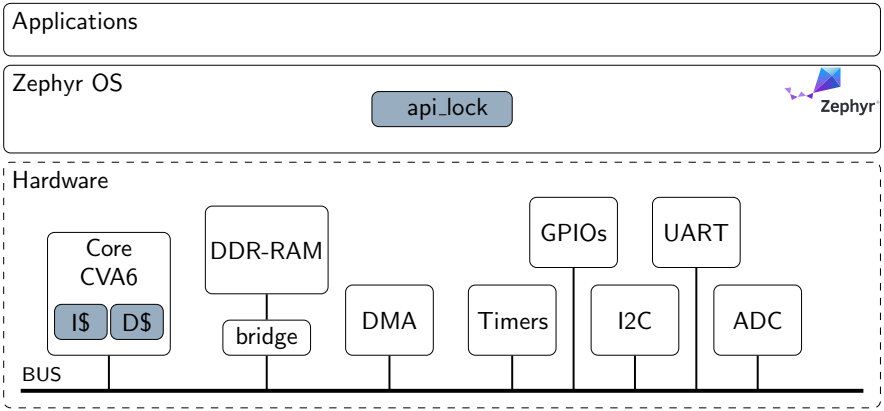
Perspectives - Axe ②

- Manage locking mechanism on more complex CPU
  - Extend the cache memory hierarchy
  - Add cores



Perspectives - Axe ③

- Operating system support
  - LockOS project

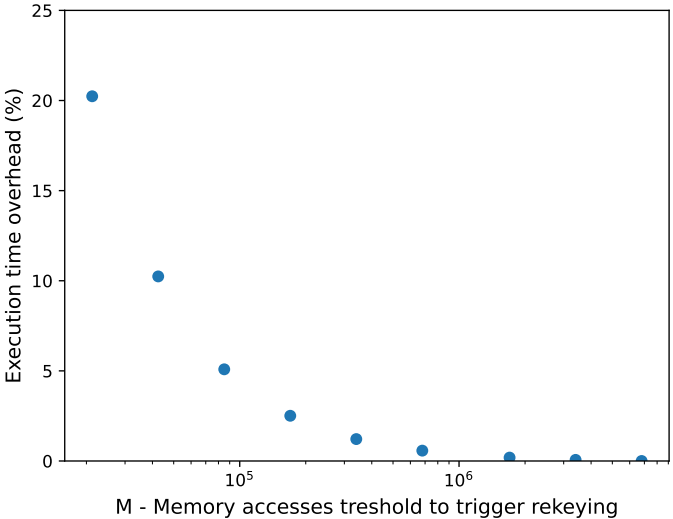


List of publications in conferences

- N. Gaudin, J.-L. Hatchikian-Houdot, F. Besson, P. Cotret, G. Gogniat, G. Hiet, V. Lapotre, and P. Wilke, "Work in progress: Thwarting timing attacks in microcontrollers using fine-grained hardware protections," in *Proc. IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, Jul. 2023. doi: 10.1109/EuroSPW59978.2023.00038
- M. Peters, N. Gaudin, J. P. Thoma, V. Lapôtre, P. Cotret, G. Gogniat, and T. Güneysu, "On the effect of replacement policies on the security of randomized cache architectures," in *Proc. ACM Asia Conference on Computer and Communications Security (ASIA CCS)*, 2024, pp. 483–497. doi: 10.1145/3634737.3637677
- N. Gaudin, P. Cotret, G. Gogniat, and V. Lapôtre, "A fine-grained dynamic partitioning against cache-based timing attacks via cache locking," in *Proc. IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2024. doi: 10.1109/ISVLSI61997.2024.00041

Impact on Performance when tuning security margin

Résultat préliminaire sur EmbenchIoT



Software Implementation

```
1  void fct(int* sensitive_table, int* input){
2    //lock phase
3    for(int i=0; i<sizeof(sensitive_table); i+=16)
4      __LOCK(&sensitive_table, i);
5
6    //algo accessing table depending on secret
7    algo(sensitive_table, input);
8
9    //unlock phase
10   for(int i=0; i<sizeof(sensitive_table); i+=16)
11     __UNLOCK(&sensitive_table, i);
12 }
```

```
c.mv   t4,a4
c.mv   t5,a5
c.add  t4,t5
lock   x0,0(t4)
```

__LOCK macro

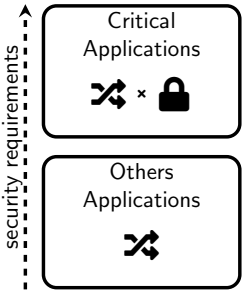
Listing 1: Example of use of the cache locking mechanism.

Goal of this implementation

- **Prevent** from a **new disruptive attack** to build eviction set (as Prime+Prune+Probe releases)

Goal of this implementation

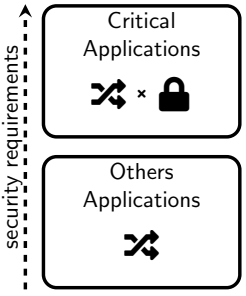
- **Prevent** from a **new disruptive attack** to build eviction set (as Prime+Prune+Probe releases)
- Provide security guarantees while an eviction set is up
  - keep critical applications unvulnerable
  - avoid attacker infering number of locked cache lines



🔀 maintains security for 6.8 M of memory accesses against P+P+P [19]

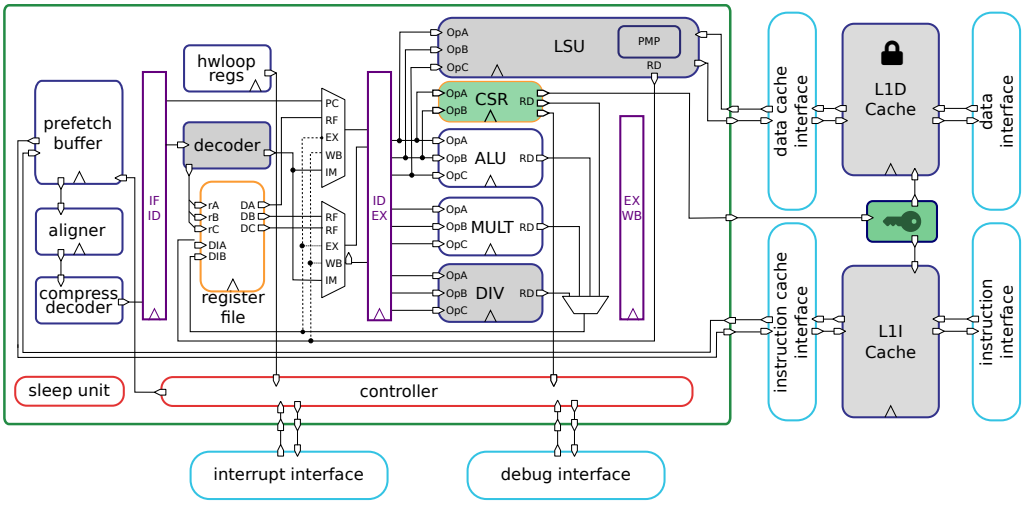
Goal of this implementation

- **Prevent** from a **new disruptive attack** to build eviction set (as Prime+Prune+Probe releases)
- Provide security guarantees while an eviction set is up
  - keep critical applications unvulnerable
  - avoid attacker infering number of locked cache lines
- Evaluate **hardware resource overhead** to implement such a system



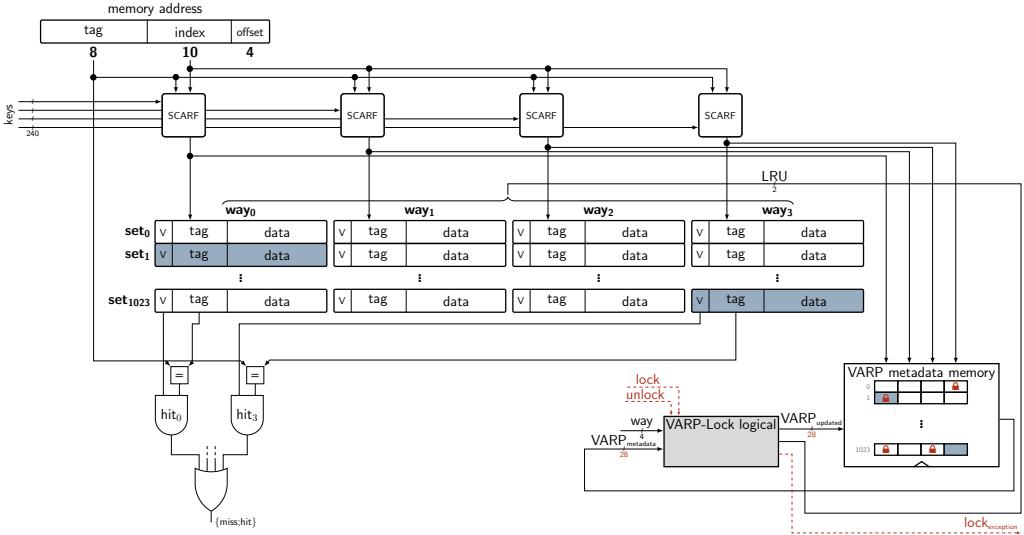
🔀 maintains security for 6.8 M of memory accesses against P+P+P [19]

Implementation - *core level*

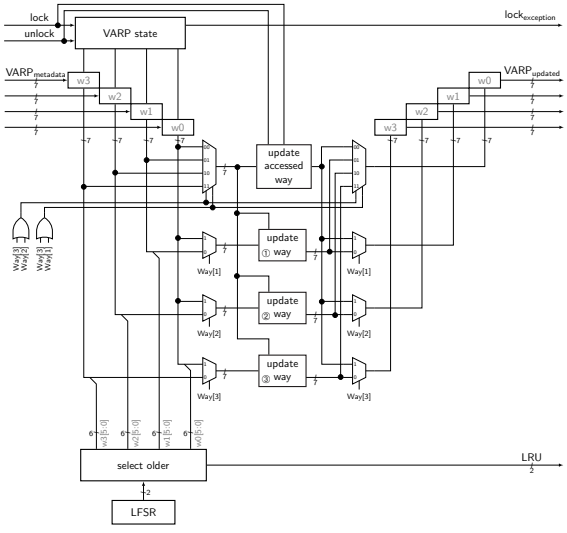


based on the OpenHWgroup CV32E40P core

Implementation - *cache level*



Implementation - *replacement policy level*



Upgrade of this implementation compared with the thesis

Integration of locking mechanism on such a cache system :

- **forbid lock on the last way**
  - keep the cache usable
  - avoid bypass

Integration of **keys renewing module** :

- multiple LFSRs to generate keys
- manage instruction and data caches keys
- invalidate and unlock the whole cache

Post implementation area results

	skewed alone			**skewed with lock**			Overhead	
	LUT	FF	BRAM	LUT	FF	BRAM	LUT	FF
↪ VARP-64	88	85	2	135	138	2	+ 53,41%	+ 62,35%
↪ Data Cache	4 264	2 287	18	4 313	2 342	18	+ 1,15%	+ 2,30%
↪ VARP-64	100	85	2	99	85	2	- 1,0%	0,00%
↪ Instr Cache	3 214	2 178	18	3 212	2 178	18	- 0,06%	0,00%
↪ Key update	628	3 490	0	628	3 490	0	0,00%	0,00%
↪ Core	4 884	2 310	0	4 862	2 310	0	- 0,45%	0,00%
CPU	13 044	10 561	36	13 070	10 616	36	+ 0,20%	+ 0,52%

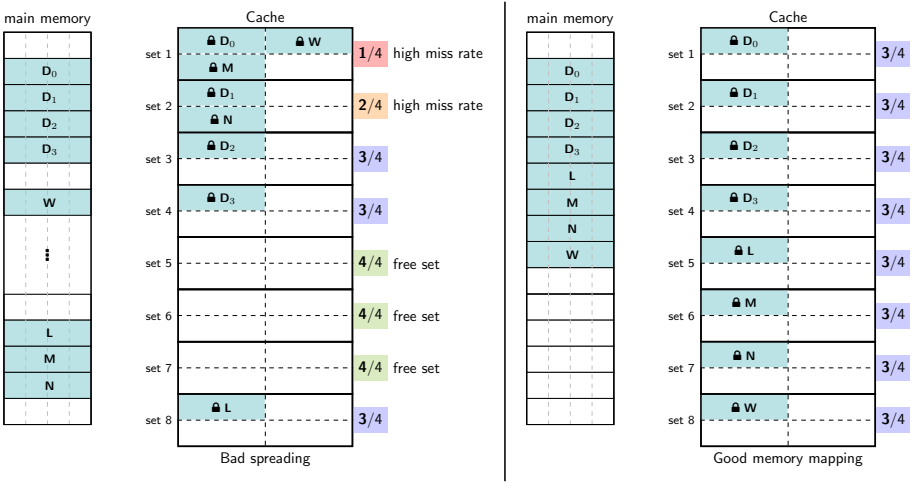
Considering AMD/Xilinx Kintex-7 FPGA family with Vivado 2022.2

Implementation Synthesis

At a glance :

- Provide **security guarantee** for critical applications
  - Skewed random cache for overall applications
  - Locking mechanism reserved for critical applications
- C**ritical applications remain immune** against known timing based CSCAs
- This implementation allows to defend against **new technique** to build eviction set
- Locking mechanism implementation implies a **low area** overhead (<2.5% on cache)

Impact of locked addresses in memory space



Evict+Time attack

Objectives :

- Infer which cache set(s) is accessed by the victim
- Observe the victim execution time

Requirements :

- Eviction Set matching with victim addresses
- Shared cache resource
- (High) Precision Timer

How ? :

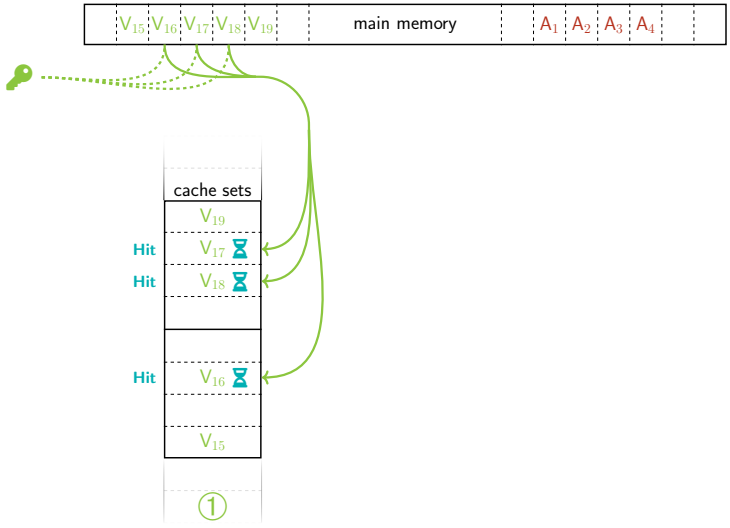
- with a 3-step attack

① measure victim execution

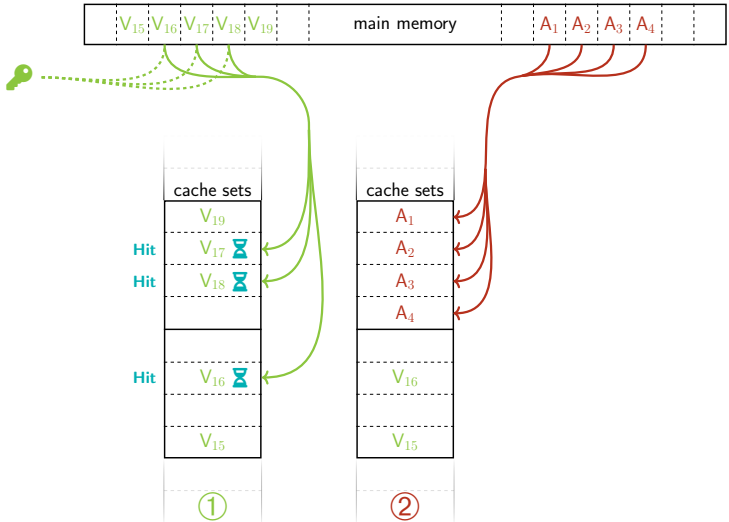
② fill a cache set using the eviction set

③ measure the affected execution to compare differences

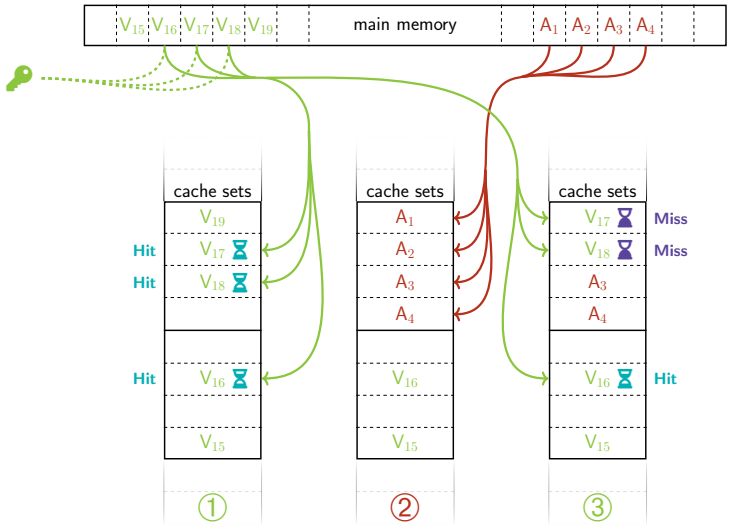
Evict+Time attack



Evict+Time attack



Evict+Time attack



Constant time operations in the div hardware module

Involved instructions : `div`, `divu`, `rem`, `remu`

	Execution Mode				
	Classic		Constant time		
Divider value	Cycles	Leak[1]	Idle cycles	Total	Leak
0x0000 0000	35	0x0000 0000	0	35	ø
0x0000 0001	34	0x0000 0001	1	35	ø
0x0000 0002	33	0x0000 000**2**	2	35	ø
0x0003 F5A2	17	0x000**2** xxxx	18	35	ø
0xBE63 20C1	3	0x**8**xxx xxxx	32	35	ø

[1] **2** = 0b001x, **8** = 0b1xxx

How to implement and use it

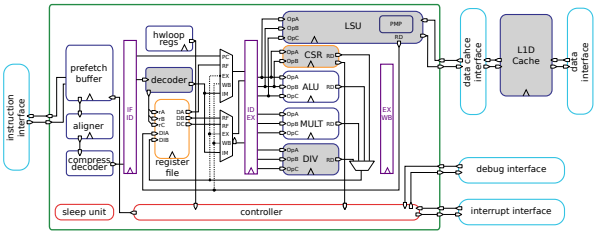


Figure 1: CV32E40P Block diagram

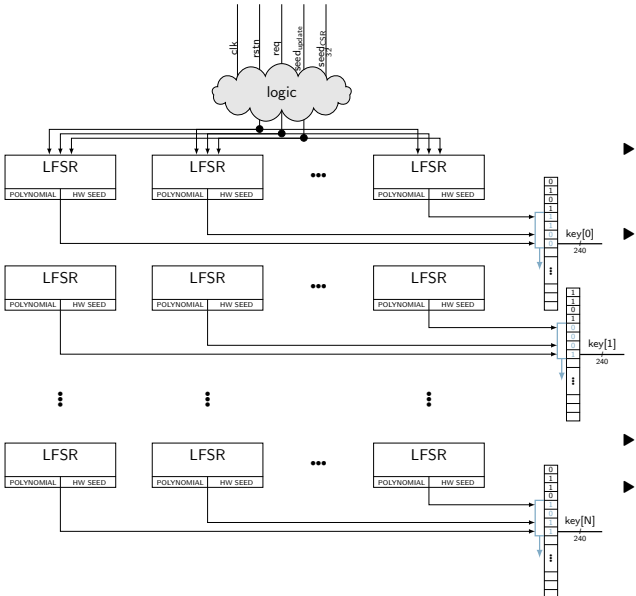
With the `swctm` pseudo instruction compiled as :

```
csrr{s|c} rd, CONSTANT_TIME, rs1
```

```
1  # block of sensitive code
2  swctm #set CT mode
3  add a2, t0, a5
4  c.addi a3, 82
5  div a0, a3, a2
6  rem t1, a5, a2
7  lw a2, 4(sp)
8  div a0, a3, a2
9  swctm #reset CT mode
```

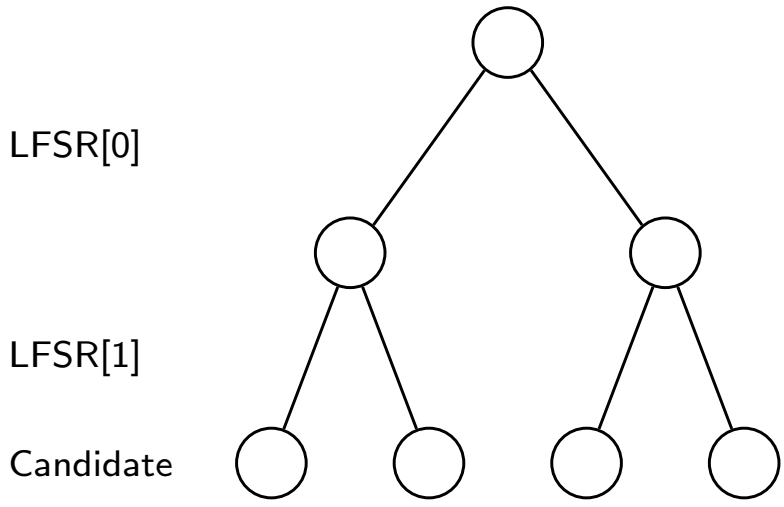
Listing 2: Application example

Key Renewing

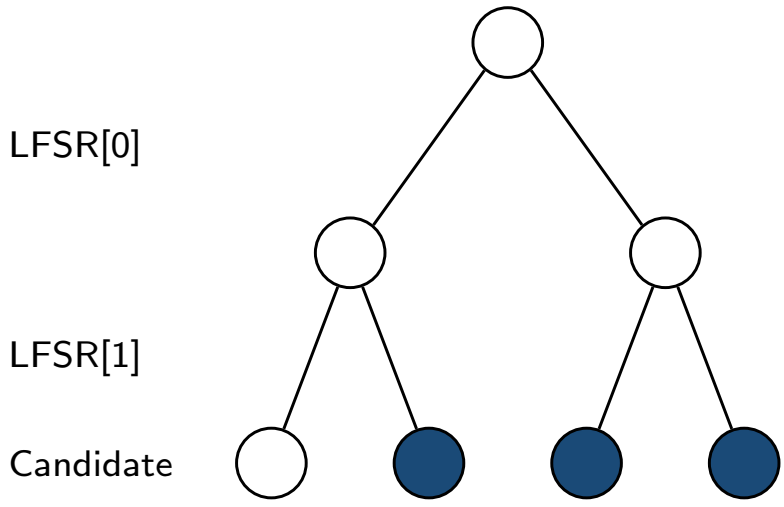


- provide **N** 240-bits keys (depending on the number of ways)
- generate each key using **M** LFSRs
  - 32-bit LFSR
  - unique polynomial for each LFSR
  - unique hardware seed for each LFSR
- seed can be updated from a CSR
- $\text{seed}_{\text{CSR}} \oplus \text{seed}_{\text{HW}}$

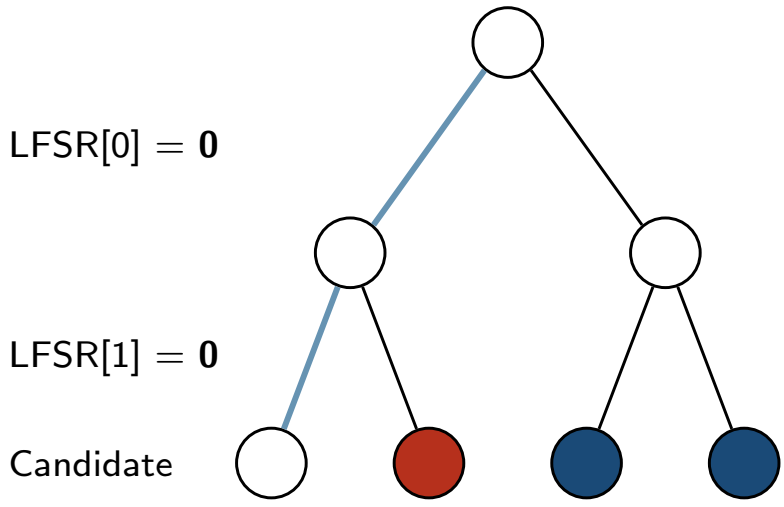
Binary Decision Tree through random number to select way to evict



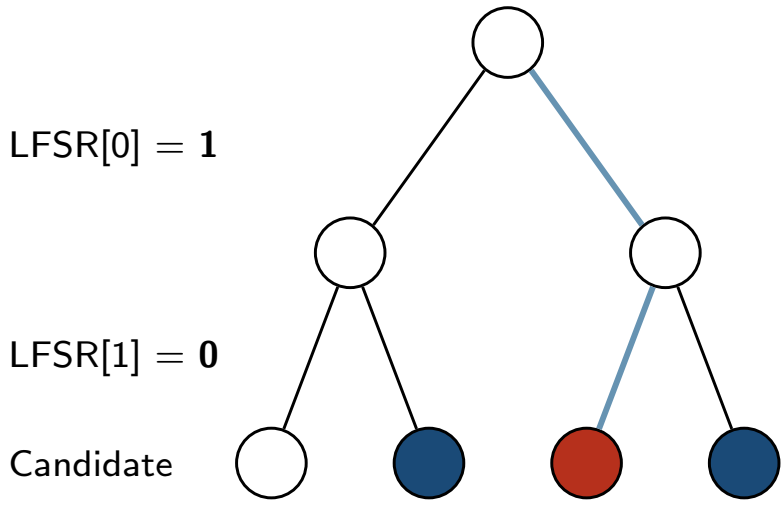
Binary Decision Tree through random number to select way to evict



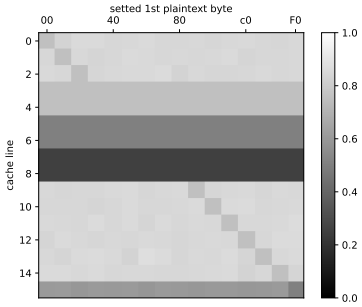
Binary Decision Tree through random number to select way to evict



Binary Decision Tree through random number to select way to evict



Covert channel



Takeaway from this figure

- Prime+Probe on AES-128
- 1 line locked on sets 3-4
- 2 lines locked on sets 5-6
- 3 lines locked on sets 7-8
- application stack collides with SBOX

Threat Model

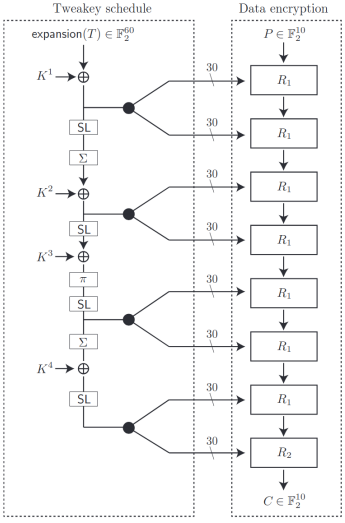
- a victim application manipulates secrets
- an attacker application tries to retrieve the secret
- multiple applications sharing resource with (and)

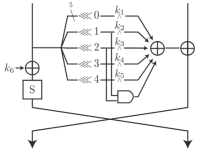
and processes are concurrently executing on the processor
only considers timing side channel

The attacker :

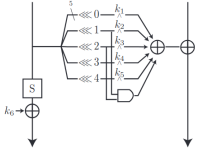
- knows the victim program.
- measures time to cycle.
  - victim execution **E**+**T**
  - its memory accesses **P**+**P**
- can interrupt the victim.
- shares cache with victim.
- has different memory spaces.

Choice to extend up to 1024 sets - SCARF limits



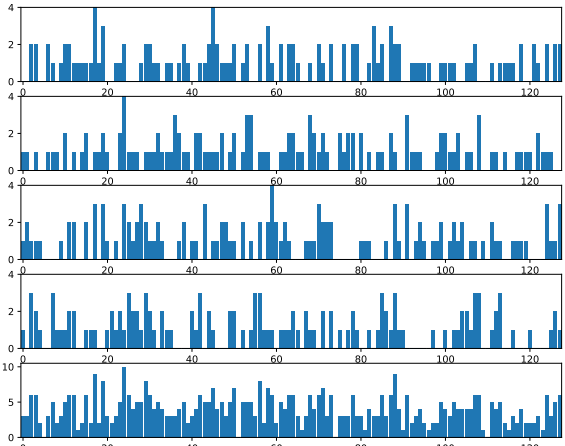


Function R_1



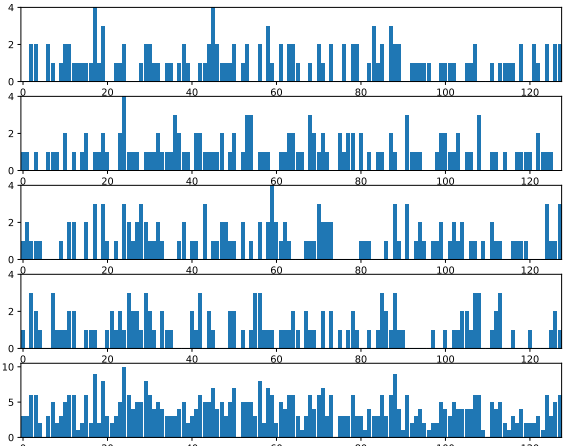
Function R_2

Choice to extend up to 1024 sets - SCARF limits

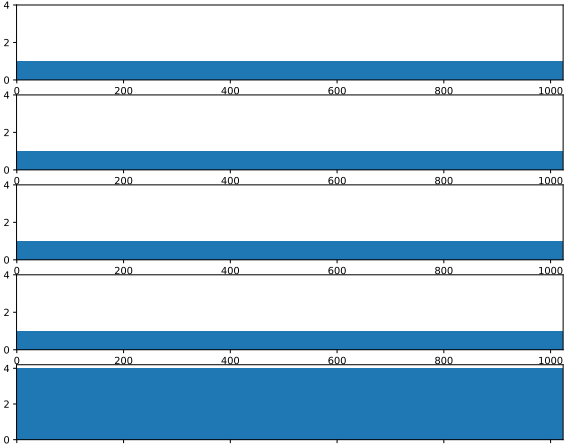


Considering 7 bits

Choice to extend up to 1024 sets - SCARF limits



Considering 7 bits



Considering 10 bits

Lock with skewed cache - Impact on Performances