SILM Workshop 2023

Work in Progress: Thwarting Timing Attacks in Microcontrollers using Fine-grained Hardware Protections

Jean-Loup Hatchikian-Houdot, Nicolas Gaudin,
Frédéric Besson, Pascal Cotret, Guy Gogniat,
Guillaume Hiet, Vianney Lapôtre, Pierre Wilke

June 30, 2023

Constant Time Programming Reminder

Sources of leakages

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

Constant Time Programming Reminder

Sources of leakages

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

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

Constant Time Programming Reminder

Sources of leakages

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

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

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

Constant Time Programming Reminder

Sources of leakages

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

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

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

Reminder on cache

Parsing of memory address:

Tag	Set	Offset
00000011	00001011	0010

Reminder on cache

Parsing of memory address:

	Tag	Set	Offset
	00000011	00001011	0010
In this presentation:	Digit	Letter	Irrelevant
addr	**3**	**B**	

addr**3B** $\rightarrow$ val**3B**

Set	Tag	Content
A	...	...
	...	...
B	**3**	*val3B*
	...	...
C	...	...
	...	...

Reminder on cache

Parsing of memory address:

	Tag	Set	Offset
	00000011	00001011	0010
In this presentation:	Digit	Letter	Irrelevant
addr	**3**	**B**	

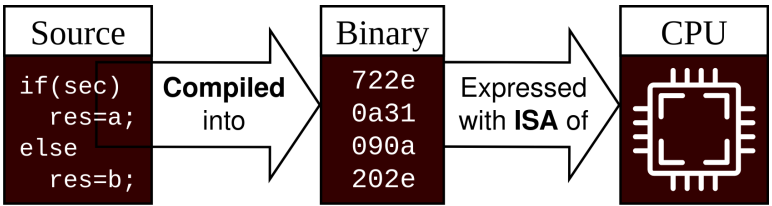
addr**3B** → val**3B**

Set	Tag	Content
A	...	...
	...	...
B	**3**	*val3B*
	...	...
C	...	...
	...	...

The cache is shared with the attacker process !

Existing solutions (few examples)

- Software: Constant time programming
- Hardware: Static partitioning
- **Hardware/Software cooperation**



Software solution example: Constant time programming

Generic solution:

- Access all array indexes, keep the good one with CMOVE

$\rightarrow$ Very inefficient

Software solution example: Constant time programming

Generic solution:

- Access all array indexes, keep the good one with CMOVE

→ Very inefficient

Specific solution (e.g. Bitslicing for Rijndael Sbox of AES)

- Re-computes the value instead of reading it from memory

→ Requires to have a computable array

→ Not always efficient either
(114 XOR/AND to replace 8 memory accesses).

Hardware solution example: Cache partitioning

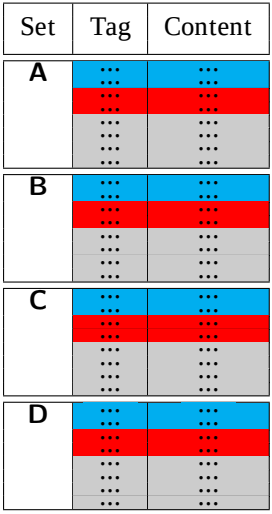
Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *Domnitser, Jaleel, Abu-Ghazaleh, Loew and Ponomarev* (ACM Trans. Archit. Code Optim)

Hardware solution example: Cache partitioning

Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *Domnitser, Jaleel, Abu-Ghazaleh, Loew and Ponomarev* (ACM Trans. Archit. Code Optim)

Example on 8-way cache with 4 sets

Allocation: *Process*$_1$ *Process*$_2$ *Shared*



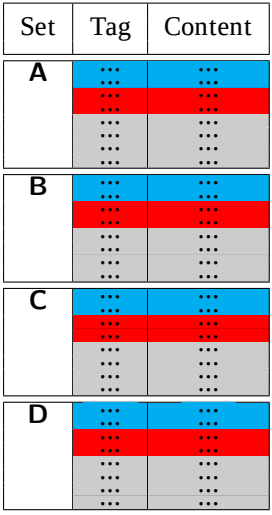
Hardware solution example: Cache partitioning

Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *Domnitser, Jaleel, Abu-Ghazaleh, Loew and Ponomarev* (ACM Trans. Archit. Code Optim)

- Requires a lot of ways in the cache
- Reduce cache availability for each process (increases cache miss rate, slows down execution)

Example on 8-way cache with 4 sets

Allocation: *Process*$_1$ *Process*$_2$ *Shared*



Hardware/Software cooperation with Partition-Locked Cache

New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks, *Wang & Lee* (ISCA '07).

Process P_1:
$Lock\_Cache(addr\mathbf{1A})$
$Lock\_Cache(addr\mathbf{1B})$
...
$res \leftarrow Load(addr\mathbf{1A})$
...
$Unlock\_Cache(addr\mathbf{1A})$
$Unlock\_Cache(addr\mathbf{1B})$

Hardware/Software cooperation with Partition-Locked Cache

New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks, *Wang & Lee* (ISCA '07).

Process P_1:
$Lock\_Cache(addr\mathbf{1A})$
$Lock\_Cache(addr\mathbf{1B})$
...
$res \leftarrow Load(addr\mathbf{1A})$
...
$Unlock\_Cache(addr\mathbf{1A})$
$Unlock\_Cache(addr\mathbf{1B})$

Table: PLcache

Set	Tag	Lock	Content
A	1	True	*val*1*A*
	...	...	...
B	1	True	*val*1*B*
	...	...	...
C	...	...	...
	...	...	...

Hardware/Software cooperation with Partition-Locked Cache

New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks, *Wang & Lee* (ISCA '07).

Process P_1:
$Lock\_Cache(addr\mathbf{1A})$
$Lock\_Cache(addr\mathbf{1B})$
...
$res \leftarrow Load(addr\mathbf{1A})$
...
$Unlock\_Cache(addr\mathbf{1A})$
$Unlock\_Cache(addr\mathbf{1B})$

Table: PLcache

Set	Tag	Lock	Content
A	1	True	*val*1*A*
	...	...	...
B	1	True	*val*1*B*
	...	...	...
C	...	...	...
	...	...	...

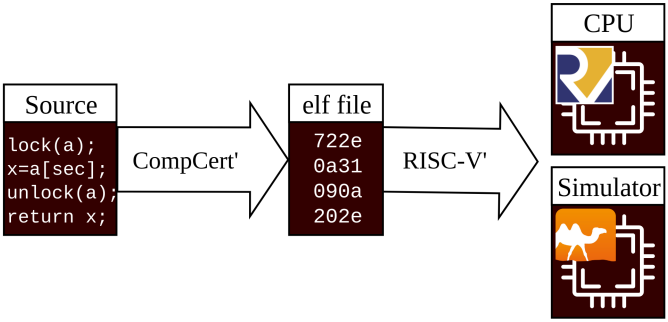
- Locked data stay in cache

→ Constant time access (cache hit)

→ Don't alter cache line (no eviction)

Contributions

- Attacks on PLcache
- RISC-V extension for efficient constant time security
- Hardware implementation with low overhead
- Hardware simulator to evaluate security



Attacks on PLcache

We found two attacks on PLcache

Lock can be removed by accident (or because of an attacker)

[1] Least Recently Used (LRU) for examples

Attacks on PLcache

We found two attacks on PLcache

Lock can be removed by accident (or because of an attacker)

Replacement policy[1] still updated on locked lines

[1] Least Recently Used (LRU) for examples

Attack on PLcache: Retrieve secret access with eviction pattern (Setup)

Victim:

Lock_Cache(*addr***1A**)

Lock_Cache(*addr***1B**)

Attacker:

Load(*addr***2A**)

Load(*addr***2B**)

Set	Tag	LRU	Lock	Content
A	1	Next	Vict.	*val1A*
	2	Last	none	*val2A*
B	1	Next	Vict.	*val1B*
	2	Last	none	*val2B*

Attack on PLcache: Retrieve secret access with eviction pattern (Setup)

Victim:

$Lock\_Cache(addr\mathbf{1A})$

$Lock\_Cache(addr\mathbf{1B})$

Attacker:

$Load(addr\mathbf{2A})$

$Load(addr\mathbf{2B})$

Set	Tag	LRU	Lock	Content
A	1	Next	Vict.	$val1A$
	2	Last	none	$val2A$
B	1	Next	Vict.	$val1B$
	2	Last	none	$val2B$

$res \leftarrow Load(addr\mathbf{1B})$

Set	Tag	LRU	Lock	Content
A	1	Next	Vict.	$val1A$
	2	Last	none	$val2A$
B	1	Last	Vict.	$val1B$
	2	Next	none	$val2B$

Attack on PLcache: Retrieve secret access with eviction pattern (Probing)

Set	Tag	LRU	Lock	Content
A	1	Next	Vict.	*val*1*A*
	2	Last	none	*val*2*A*
B	1	Last	Vict.	*val*1*B*
	2	**Next**	none	*val*2*B*

Attack on PLcache: Retrieve secret access with eviction pattern (Probing)

Set	Tag	LRU	Lock	Content
A	1	Next	Vict.	*val*1*A*
	2	Last	none	*val*2*A*
B	1	Last	Vict.	*val*1*B*
	2	**Next**	none	*val*2*B*

Attacker:	**Effect:**
Load(*addr***3A**)	Bypass cache
Load(*addr***3B**)	Evict addr**2B**

Attack on PLcache: Retrieve secret access with eviction pattern (Probing)

Set	Tag	LRU	Lock	Content
A	1	Next	Vict.	*val1A*
	2	Last	none	*val2A*
B	1	Last	Vict.	*val1B*
	2	**Next**	none	*val2B*

Set	Tag	LRU	Lock	Content
A	1	Last	Vict.	*val1A*
	2	Next	none	*val2A*
B	1	Next	Vict.	*val1B*
	3	Last	none	*val3B*

Attacker:	**Effect:**
Load(*addr***3A**)	Bypass cache
Load(*addr***3B**)	Evict addr**2B**

Attack on PLcache: Retrieve secret access with eviction pattern (Probing)

Set	Tag	LRU	Lock	Content
A	1	Next	Vict.	*val1A*
	2	Last	none	*val2A*
B	1	Last	Vict.	*val1B*
	2	**Next**	none	*val2B*

Attacker:	**Effect:**
Load(*addr***3A**)	Bypass cache
Load(*addr***3B**)	Evict addr**2B**

Set	Tag	LRU	Lock	Content
A	1	Last	Vict.	*val1A*
	2	Next	none	*val2A*
B	1	Next	Vict.	*val1B*
	3	Last	none	*val3B*

Load(*addr***2A**)	Cache hit
Load(*addr***2B**)	Cache miss

Attacker learn that Victim used set B !

RISC-V extension specifications

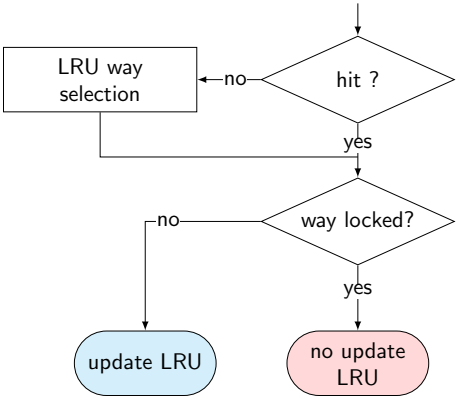
Security concerns

- No accidental unlock (Locks are removed only with Unlock instructions)
- Usage meta-data (LRU) is not updated by accesses on locked lines.

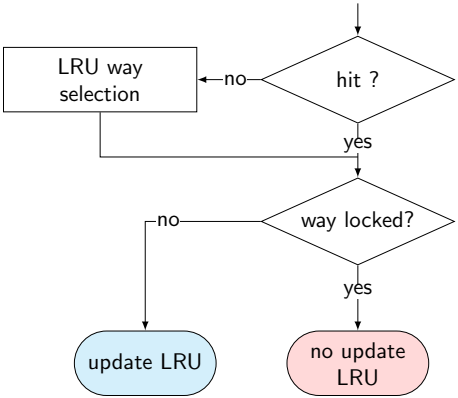
Performance

- At least one free way: Lock fail when only one unlocked way left in the cache set

Memory Access Handling



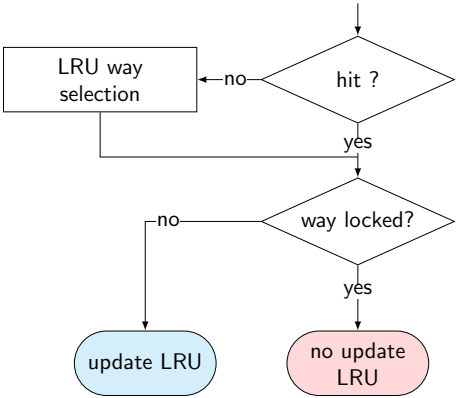
Memory Access Handling



- Access on locked ways never alter usage meta-data

← Prevents previous attack

Memory Access Handling



- Access on locked ways never alter usage meta-data
- Locked ways are never selected for eviction

← Prevents previous attack

Hardware implementation with low overhead

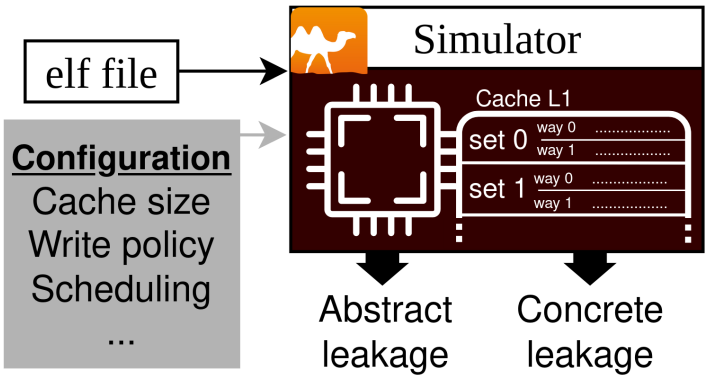
Core: CV32E40P (RISC-V based)
Cache: 8 KiB, 4-way set-associative, L1 data cache.

Table: Post-synthesis area results[2]

		BRAM	LUTs (%)	FFs (%)
New results [3]	Lock overhead	0	4.70	0.67

[2] Synthesis for Kyntex-7 chip using Vivado 2022 tool
[3] Published results are outdated

Simulator to evaluate security



Timing leakage of an execution

Classic leakage trace

int a = b + c;	[•]	"Nothing leaks (except program counter)"
int a = array[index];	index	"Index of memory access leaks"

Timing leakage of an execution

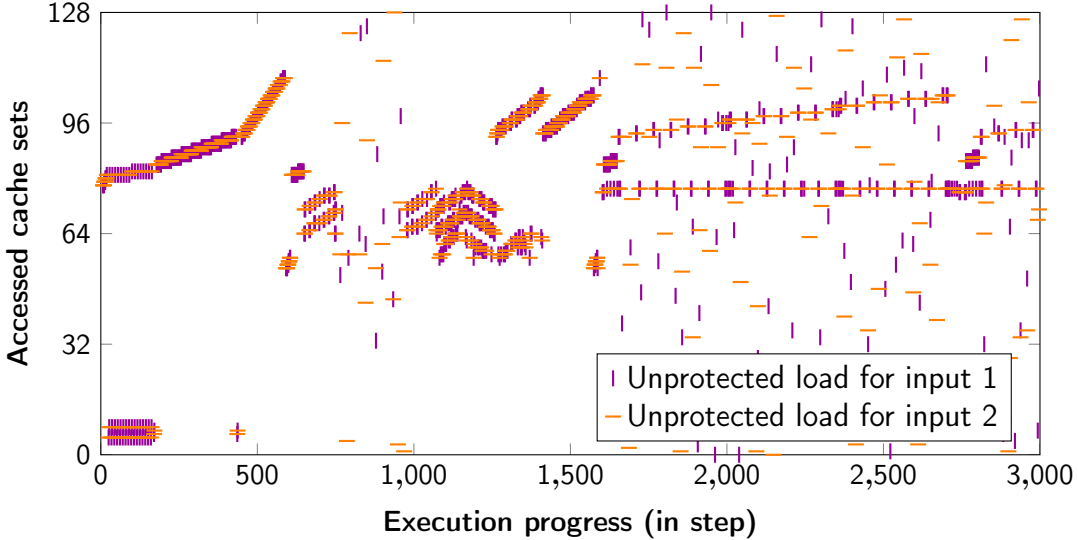
Classic leakage trace

int a = b + c;	[•]	"Nothing leaks (except program counter)"
int a = array[index];	index	"Index of memory access leaks"

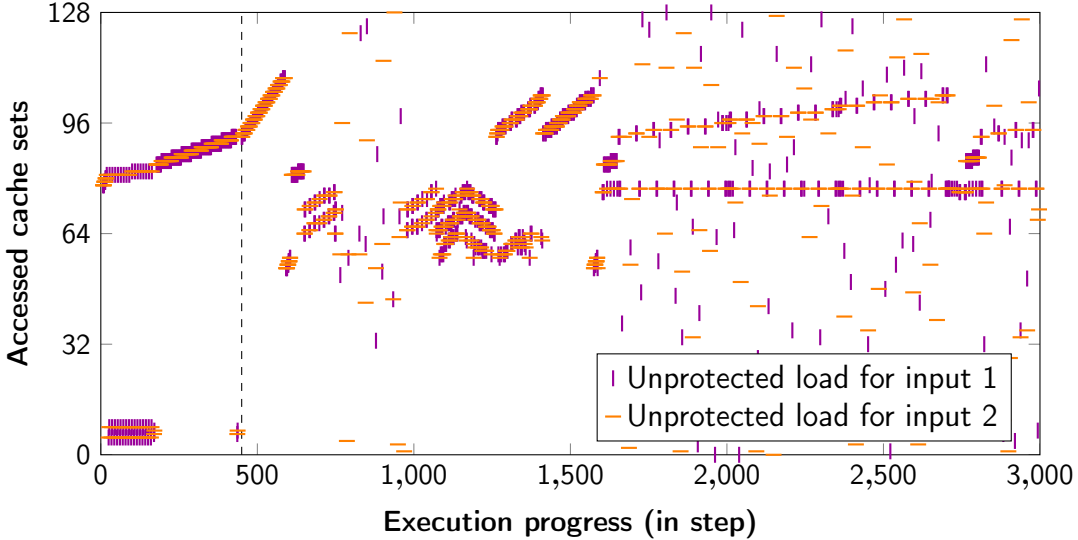
We introduce two derived leakages

	Abstract leakage (what could be seen)	**Concrete leakage** (what is currently seen)
int a = b + c;	[•]	[•]
int a = array[index];	cache_set(index)	cache miss
lock(array[index]):	cache_set(index)	cache hit
int a = array[index];	[•]	cache hit

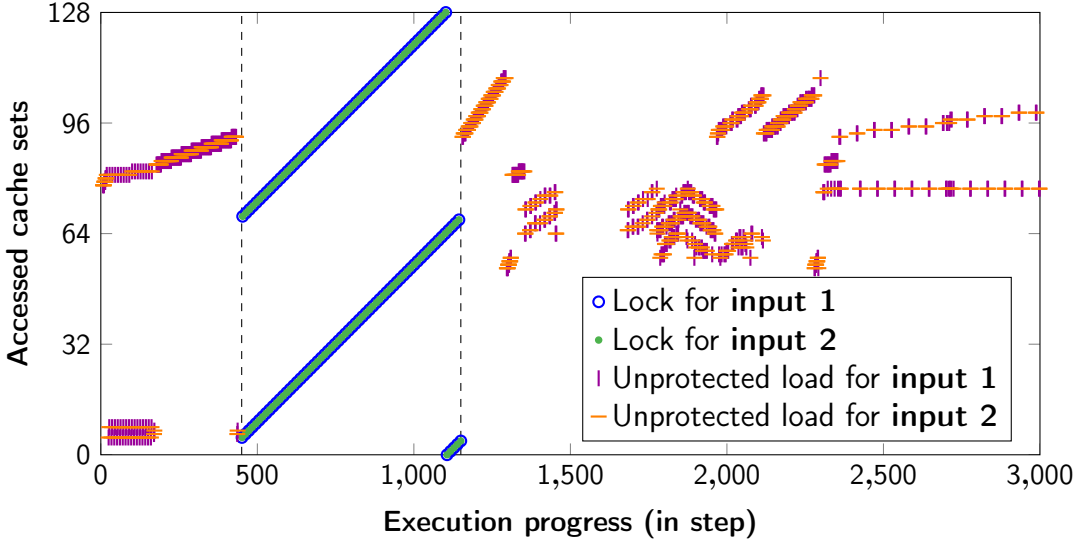
Abstract leakage of unprotected Camellia



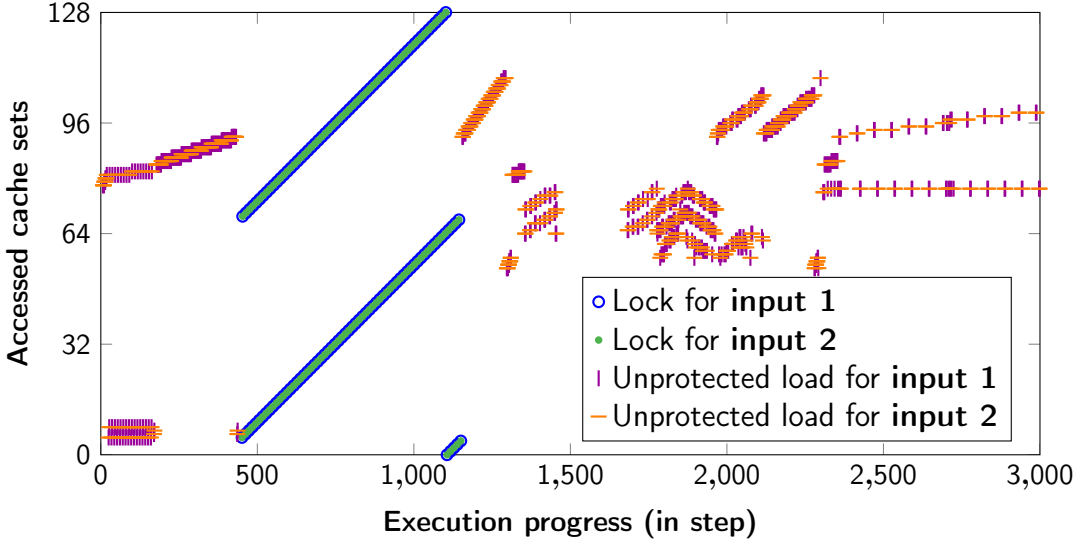
Abstract leakage of unprotected Camellia



Abstract leakage of protected Camellia



Abstract leakage of protected Camellia



Perspectives

Proof about Abstract and Concrete Leakages

Indistinguishability of abstract leakages is preserved on concrete leakages

$\forall exec_1, exec_2,$
$abstract\_leakage(exec_1) = abstract\_leakage(exec_2)$
(Abstract leakages are the same in both execution)
$\Longrightarrow$
(
$\forall context$ (Potential attacker running on the same hardware)
$concrete\_leakage(exec_1, context) = concrete\_leakage(exec_2, context)$
(Concrete leakages are also the same,
even if an attacker is tampering with the cache)
)

Other perspectives

Limitation of Lock	**Perspective**
L1-d only	Multi-level cache lock

Other perspectives

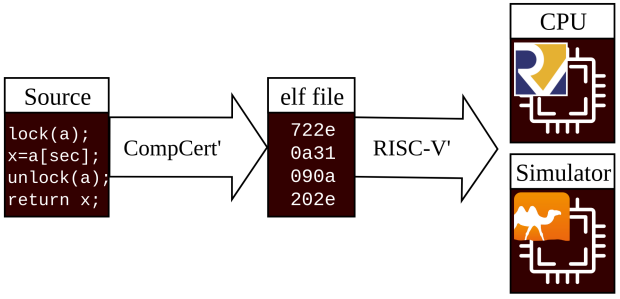
Limitation of Lock	**Perspective**
L1-d only	Multi-level cache lock
Reduce availability	Alternative mechanism (Restore-On-Context-Switch)

Other perspectives

Limitation of Lock	Perspective
L1-d only	Multi-level cache lock
Reduce availability	Alternative mechanism (Restore-On-Context-Switch)
Exception if set is full	OS support to catch the error and run a back-up solution

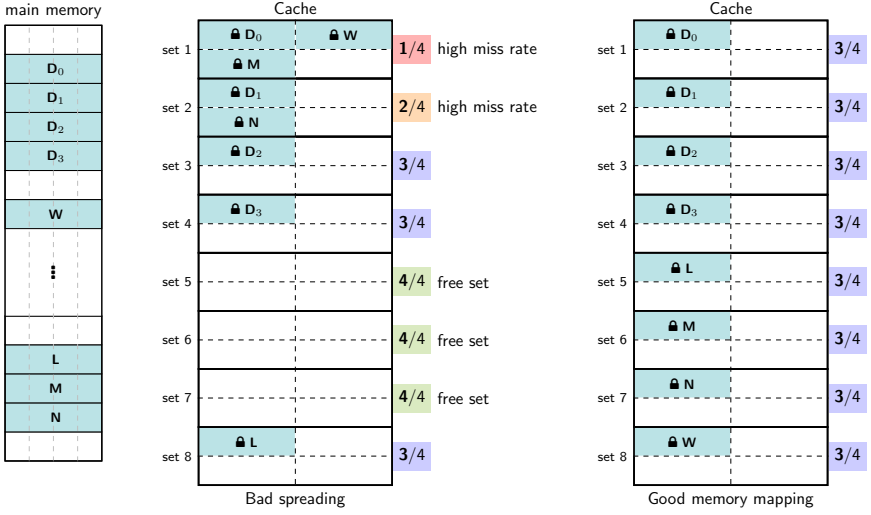
Current state of our work

- Attacks on PLcache
- RISC-V extension for efficient constant time
- Hardware implementation with low overhead
- Hardware simulator to evaluate security



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

Appendix: Lock spreading optimization



Appendix: Rijndael Substitution Box and Bitslicing

```
static const uint8_t sbox[256] = {
  //0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
  0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
  0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
  0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
  0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
  0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
  0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
  0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
  0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
  0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
  0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
  0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
  0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
  0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
  0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
  0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
  0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 };

#if (defined(CBC) && CBC == 1) || (defined(ECB) && ECB == 1)
static const uint8_t rsbox[256] = {  //inverse S-box
  0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb,
  0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb,
  0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e,
  0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25,
  0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92,
  0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84,
  0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06,
  0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b,
  0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73,
  0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e,
  0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b,
  0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4,
  0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f,
  0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef,
  0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,
  0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d };
#endif
```

Rijndael SBox is

- Constant
- Public
- Computable

Bitslicing uses 114 XOR and AND operations to replace 8 loads on Sbox

Appendix: Proofs

Set	Tag	Usage meta-data	Lock	Content
A	1	Black	True	$val1A$
	5	Box A	False	$val5A$
B	9	Black	False	$val9B$
	4	Box B	False	$val4B$

$$\frac{\textbf{protected}(\mathit{TargetAddress}(\mathit{Acc}))}{\mathit{State} \xrightarrow[\mathit{access}(\mathit{Acc})]{} \mathit{SetResult}_{\mathit{Acc}}(\mathit{State})}$$

$$\frac{\neg\textbf{protected}(\mathit{TargetAddress}(\mathit{Acc})) \wedge \textbf{cached}(\mathit{TargetAddress}(\mathit{Acc}))}{\mathit{State} \xrightarrow[\mathit{access}(\mathit{Acc})]{} (\mathit{UpdateUsage}_{\mathit{Acc}} \circ \mathit{SetResult}_{\mathit{Acc}})(\mathit{State})}$$

$$\frac{\neg\textbf{cached}(\mathit{TargetAddress}(\mathit{Acc}))}{\mathit{State} \xrightarrow[\mathit{access}(\mathit{Acc})]{} (\mathit{UpdateUsage}_{\mathit{Acc}} \circ \mathit{SetResult}_{\mathit{Acc}} \circ \mathit{Evict\&Replace}_{A}cc)(\mathit{State})}$$