Constant Time Security at a Low Cost for Embedded Systems Through Hardware/Software Cooperation

Jean-Loup Hatchikian-Houdot

Epicure Team, Inria
Supervised by Frédéric Besson, Guillaume Hiet, Pierre Wilke
In collaboration with Nicolas Gaudin, Pascal Cotret, Guy Gogniat, Vianney Lapôtre

March 27, 2023

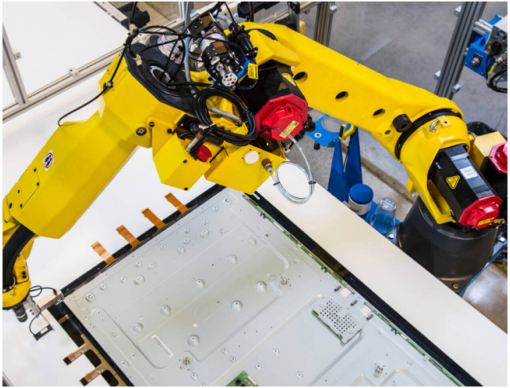
Context

Constant time operations

Cache protection

Improvement to Lock

Embedded systems and Internet-of-Things (IoT)



- Used in a lot of devices (industrial, medical, etc.)

→ Must be tiny, cheap, and have low power consumption

→ Can handle sensitive data

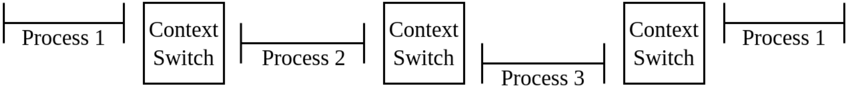
- Often have internet access (Software updates, cloud access, remote control, etc.)

→ Attackers can force their own code to execute on those device to steal data to victim process

Use case : IoT

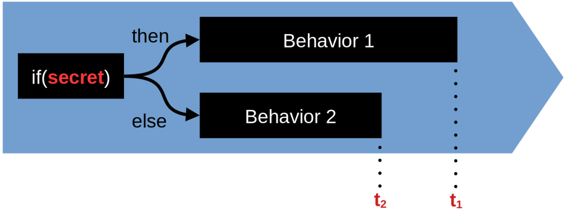
Cheap embedded system with low power consumption :

- No speculation
- In-Order
- Mono-threading:

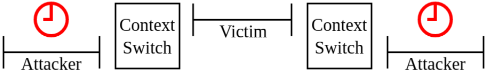


Definition : Timing leakage & Constant Time Security

Program leaking the value of a secret:



The attacker can observe leakages if its code run on the same hardware and can measure execution time:



Constant Time Security (CTS):

→ No secrets exposed through timing leakage

Causes of leakages

- Computations time depending on operands

e.g.: $res \leftarrow div(x, y)$ $[log_2(y)]$

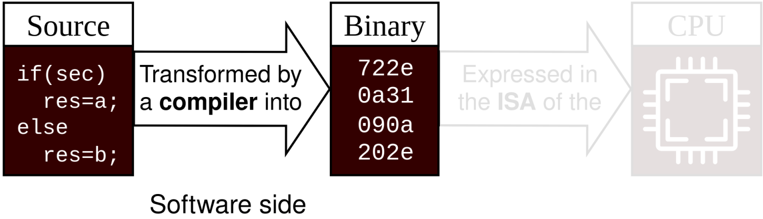
- Memory accesses

e.g.: $res \leftarrow load(address)$ $[cache\_line(address)]$

- Conditional jumps (future work)

e.g.: $if(condition)$ $[condition]$

Software only Countermeasure



Constant time programming (Timing does no depends on secrets)

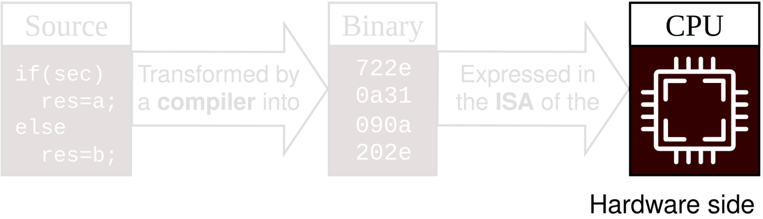
- Restrict the programmer

→ E.g., **no memory access on a secret address**.

- Could rely on undefined micro-architectural behaviors

→ E.g., **multiplication is not CTS on every processor**.

Hardware only Countermeasure

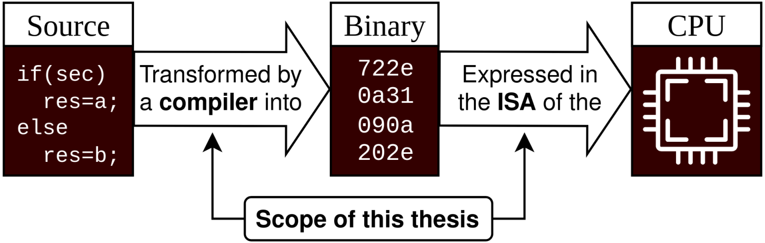


E.g., Cache partitioning

- Cannot tell apart secret from public data

→ Unnecessary high cost when handling public data

Proposal: Cooperation between Hardware and Software



Source
Transformed by a **compiler** into
Binary
Expressed in the **ISA** of the
CPU
Scope of this thesis

```
if(sec)
   res=a;
else
   res=b;
```

```
722e
0a31
090a
202e
```

New instructions in the ISA

- Software and Hardware can communicate about security

→ The software can use costly security only when needed

- Timing behavior specification

→ Security guaranties against timing attacks

Requirement and Hypothesis

Requirements:

- The software developer does not need to know the hardware implementation
- Secrets defined by the source code
- No timing leakages on secrets
- Security cost must be kept low regarding execution time, memory usage and hardware requirements

Hypothesis:

- The source code, compiler and hardware will comply to the ISA specification
- The attacker does not have physical access to the hardware

Context

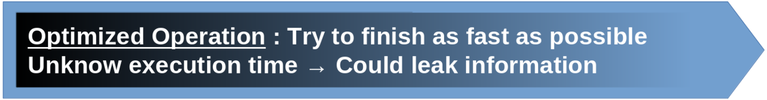
Constant time operations

Cache protection

Improvement to Lock

Safe Operations

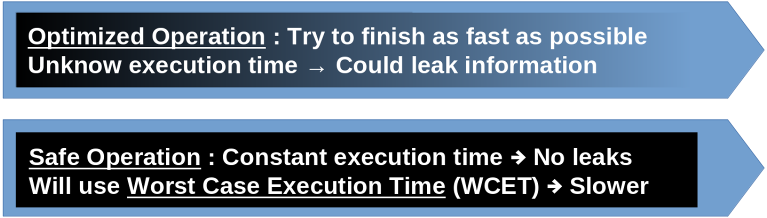
Some operations have huge timing variations caused by optimizations



Optimized Operation : Try to finish as fast as possible
Unknow execution time → Could leak information

Timing behavior of operations

Some operations have huge timing variations caused by optimizations



Optimized Operation : Try to finish as fast as possible
Unknow execution time → Could leak information

Safe Operation : Constant execution time ➔ No leaks
Will use Worst Case Execution Time (WCET) ➔ Slower

We can define a safe version of them for constant time mode.

Constant time mode

Code in pseudo-assembly	Leakage
$x_1 \leftarrow add(x_2, x_3)$	$[\bullet]$
$x_1 \leftarrow div(x_2, x_3)$	$[log_2(x_3)]$
$begin\ constant\ time\ mode$	$[\bullet]$
$x_1 \leftarrow add(x_2, x_3)$	$[\bullet]$
$x_1 \leftarrow div(x_2, x_3)$	$[\bullet]$ (performance loss)
$end\ constant\ time\ mode$	$[\bullet]$

Context

Constant time operations

Cache protection

Improvement to Lock

Cache: direct mapping

RAM

Address	Value
00000	a
00001	b
00010	c
00011	d
00100	e
00101	f
00110	g
00111	h
...	...

Direct mapped cache

Line id	Tag	Word 0	Word 1
Line 0	111	x	x
Line 1	111	x	x

$b_{1\sim3}$	b_4	b_5
Tag	Line	Word

Cache: direct mapping

RAM

Address	Value
00000	a
00001	b
00010	c
00011	d
00100	e
00101	f
00110	g
00111	h
…	…

Direct mapped cache

Line id	Tag	Word 0	Word 1
Line 0	111	x	x
Line 1	111	x	x

load(00110):	001	1	0
	$b_{1\sim3}$	b_4	b_5
	Tag	Line	Word

Cache: direct mapping

RAM

Address	Value
00000	a
00001	b
00010	c
00011	d
00100	e
00101	f
00110	g
00111	h
…	…

Direct mapped cache

Line id	Tag	Word 0	Word 1
Line 0	111	x	x
Line 1	111	x	x

load(00110):	001	1	0
	$b_{1\sim3}$	b_4	b_5
	Tag	Line	Word

Cache: direct mapping

RAM

Address	Value
00000	a
00001	b
00010	c
00011	d
00100	e
00101	f
00110	g
00111	h
...	...

Direct mapped cache

Line id	Tag	Word 0	Word 1
Line 0	111	x	x
Line 1	111	x	x

load(00110):	001	1	0
	$b_{1\sim3}$	b_4	b_5
	Tag	Line	Word

Cache: direct mapping

RAM

Address	Value
00000	a
00001	b
00010	c
00011	d
00100	e
00101	f
00110	g
00111	h
...	...

Direct mapped cache

Line id	Tag	Word 0	Word 1
Line 0	111	x	x
Line 1	111	x	x

Cache miss!

load(00110): 001 1 0

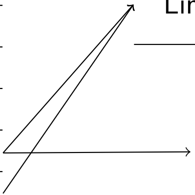
$b_{1\sim3}$ b_4 b_5

Tag Line Word

Cache: direct mapping

RAM

Address	Value
00000	a
00001	b
00010	c
00011	d
00100	e
00101	f
00110	g
00111	h
…	…



Direct mapped cache

Line id	Tag	Word 0	Word 1
Line 0	111	x	x
Line 1	**001**	**g**	**h**

Load from RAM (slow)

load(00110):	001	1	0
	$b_{1\sim 3}$	b_4	b_5
	Tag	Line	Word

Cache: direct mapping

RAM

Address	Value
00000	a
00001	b
00010	c
00011	d
00100	e
00101	f
00110	g
00111	h
...	...

Direct mapped cache

Line id	Tag	Word 0	Word 1
Line 0	111	x	x
Line 1	001	g	h

load(00111):	001	1	1
	$b_{1\sim3}$	b_4	b_5
	Tag	Line	Word

Cache: direct mapping

RAM

Address	Value
00000	a
00001	b
00010	c
00011	d
00100	e
00101	f
00110	g
00111	h
…	…

Direct mapped cache

Line id	Tag	Word 0	Word 1
Line 0	111	x	x
Line 1	001	g	h

load(00111):	001	1	1
	$b_{1\sim3}$	b_4	b_5
	Tag	Line	Word

Cache: direct mapping

RAM

Address	Value
00000	a
00001	b
00010	c
00011	d
00100	e
00101	f
00110	g
00111	h
...	...

Direct mapped cache

Line id	Tag	Word 0	Word 1
Line 0	111	x	x
Line 1	001	g	h

Cache hit!

load(00111):	001	1	1
	$b_{1\sim3}$	b_4	b_5
	Tag	Line	Word

Cache: direct mapping

RAM

Address	Value
00000	a
00001	b
00010	c
00011	d
00100	e
00101	f
00110	g
00111	h
…	…

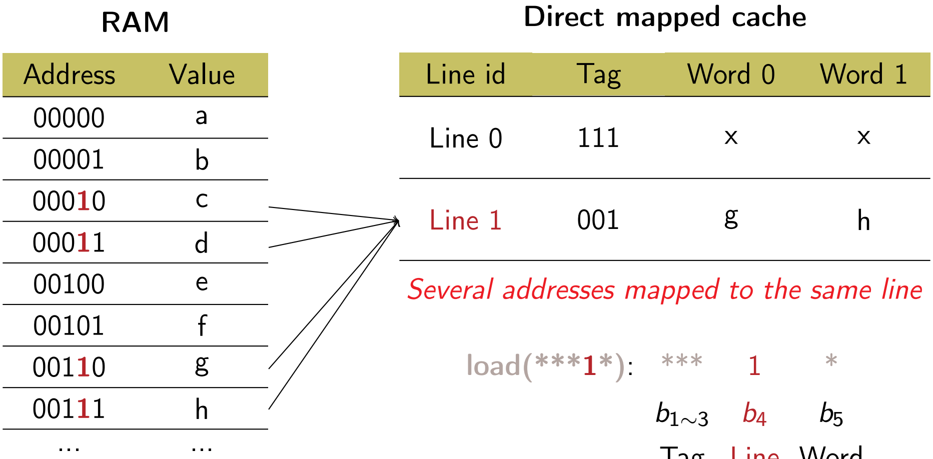
Direct mapped cache

Line id	Tag	Word 0	Word 1
Line 0	111	x	x
Line 1	001	g	h

Load from cache (fast)

load(00111):	001	1	1
	$b_{1\sim3}$	b_4	b_5
	Tag	Line	Word

Cache: direct mapping - eviction



RAM

Address	Value
00000	a
00001	b
00010	c
00011	d
00100	e
00101	f
00110	g
00111	h
…	…

Direct mapped cache

Line id	Tag	Word 0	Word 1
Line 0	111	x	x
Line 1	001	g	h

Several addresses mapped to the same line

load(***1*):	***	1	*
	$b_{1\sim3}$	b_4	b_5
	Tag	Line	Word

Cache attack

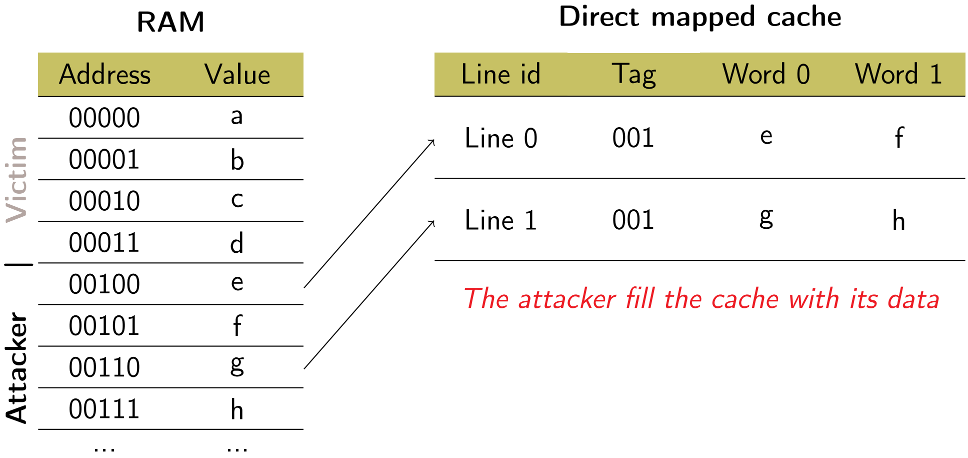
RAM

	Address	Value
Victim	00000	a
	00001	b
	00010	c
	00011	d
Attacker	00100	e
	00101	f
	00110	g
	00111	h
	...	...

Direct mapped cache

Line id	Tag	Word 0	Word 1
Line 0	111	x	x
Line 1	001	g	h

Cache attack



RAM

	Address	Value
Victim	00000	a
	00001	b
	00010	c
	00011	d
Attacker	00100	e
	00101	f
	00110	g
	00111	h
	...	...

Direct mapped cache

Line id	Tag	Word 0	Word 1
Line 0	001	e	f
Line 1	001	g	h

The attacker fill the cache with its data

Cache attack

RAM

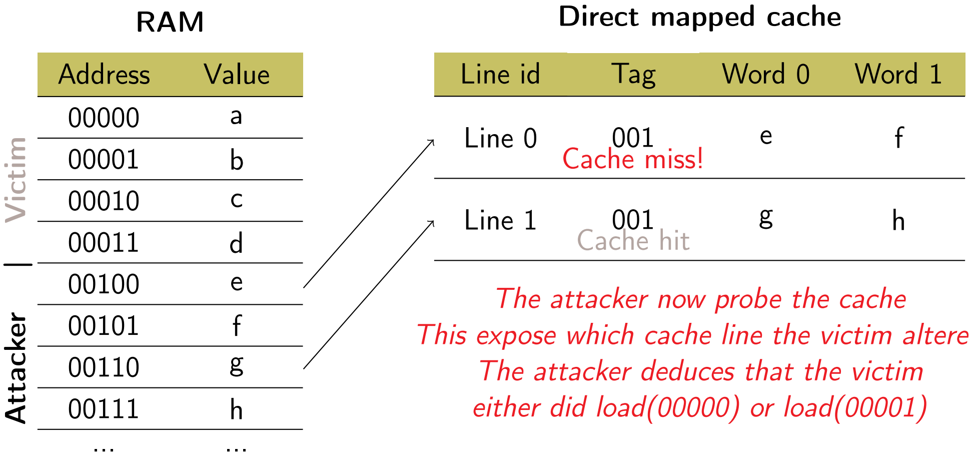
	Address	Value
Victim	00000	a
	00001	b
	00010	c
	00011	d
Attacker	00100	e
	00101	f
	00110	g
	00111	h
	...	...

Direct mapped cache

Line id	Tag	Word 0	Word 1
Line 0	000	a	b
Line 1	001	g	h

The victim try to secretly load the word at ***00001***

Cache attack



RAM

Victim | Attacker

Address	Value
00000	a
00001	b
00010	c
00011	d
00100	e
00101	f
00110	g
00111	h
...	...

Direct mapped cache

Line id	Tag	Word 0	Word 1
Line 0	001	e	f
Line 1	001	g	h

Cache miss!

Cache hit

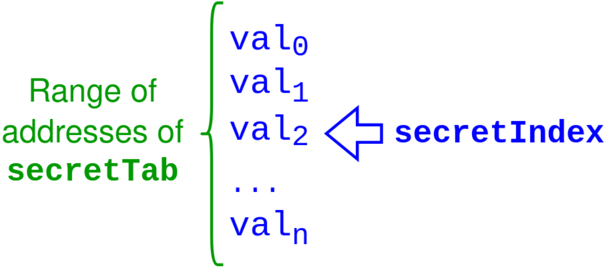
The attacker now probe the cache
This expose which cache line the victim altered
The attacker deduces that the victim
either did load(00000) or load(00001)

What we want to protect

We want to be able to do secret memory accesses (i.e. to not leak at which index we access an array)

Public source code

```
...
int x = secretTab[secretIndex];
...
```



Range of addresses of **secretTab**

$\mathtt{val}_0$

$\mathtt{val}_1$

$\mathtt{val}_2$ **secretIndex**

...

$\mathtt{val}_n$

Solution : Lock line in cache

Process P_1:

```
$Lock\_Cache(00001)$
$Lock\_Cache(00010)$
$res \leftarrow load(00001)$
$Unlock\_Cache(00001)$
$Unlock\_Cache(00010)$
```

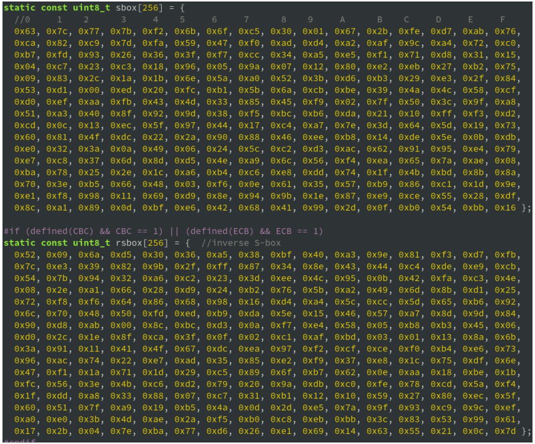
Direct mapped cache

Line id	Lock	Tag	Word 0	Word 1
Line 0	P_1	000	a	b
Line 1	P_1	000	c	d

Attacker can no longer tamper with lines 0 and 1

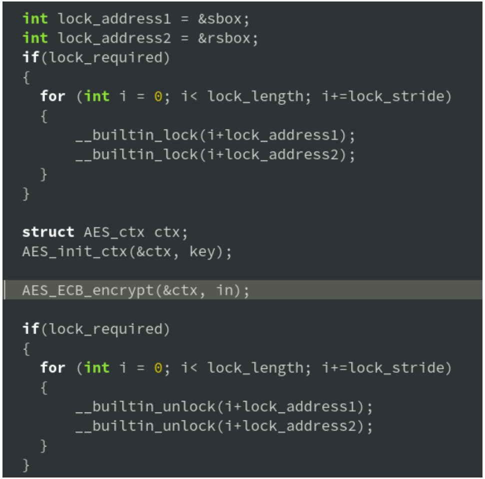
Partionned Lock cache (PLcache) proposed by Zhenghong Wang and Ruby B. Lee in 2007

Example on the S-box of AES



```c
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
```



```c
int lock_address1 = &sbox;
int lock_address2 = &rsbox;
if(lock_required)
{
  for (int i = 0; i< lock_length; i+=lock_stride)
  {
      __builtin_lock(i+lock_address1);
      __builtin_lock(i+lock_address2);
  }
}

struct AES_ctx ctx;
AES_init_ctx(&ctx, key);

AES_ECB_encrypt(&ctx, in);

if(lock_required)
{
  for (int i = 0; i< lock_length; i+=lock_stride)
  {
      __builtin_unlock(i+lock_address1);
      __builtin_unlock(i+lock_address2);
  }
}
```

Constant time AES with lookup tables (S-box) !

Context

Constant time operations

Cache protection

Improvement to Lock

Issues of PLcache and proposition

- Memory access on locked lines still alter cache state (LRU policy)
- The victim can accidentally unlock it's own locked lines in some cases

We want a stronger version of lock that guarantees no timing leakage could occurs.

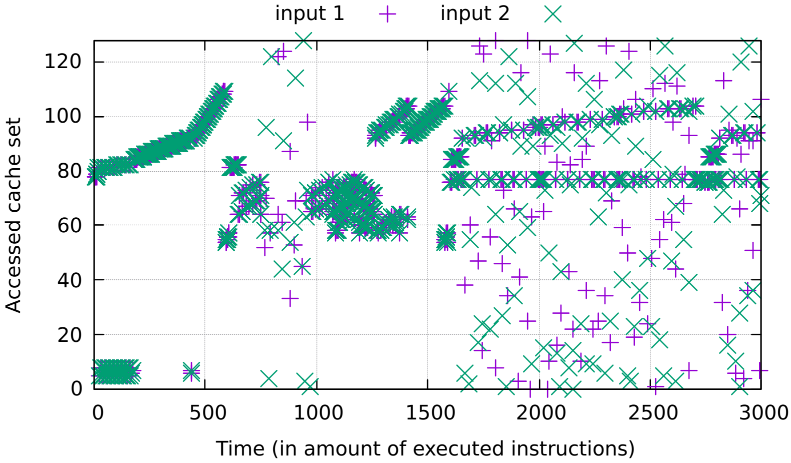
Issues of PLcache and proposition

- Memory access on locked lines still alter cache state (LRU policy)
- The victim can accidentally unlock it's own locked lines in some cases

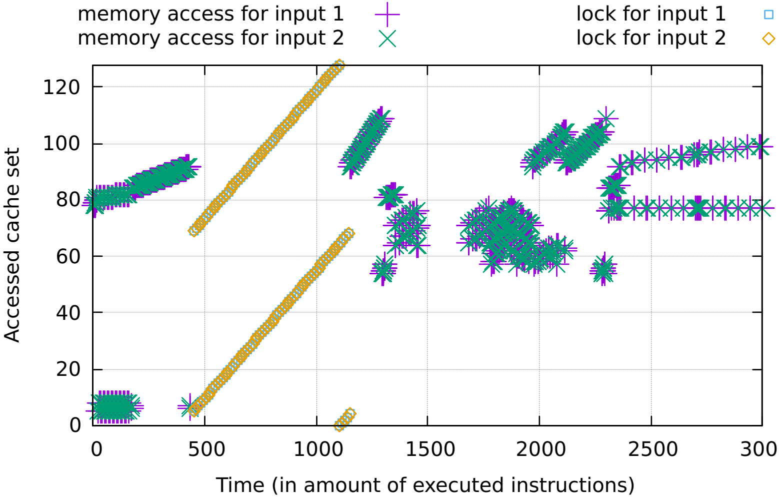
We want a stronger version of lock that guarantees no timing leakage could occurs.
We propose the following properties as requirement of any lock implementation :

- Memory access on a locked line cannot alter the cache in an observable manner
- Locked line can only be unlocked explicitly (with the unlock instruction)

Simulation on Camellia encryption : Vulnerable S-box



Simulation on Camellia encryption : Protected S-box



Perspectives

Priorities:

- Formal proof of the security guarantees
- Performance evaluations

Next perspectives:

- Generalize the lock on multi-level caches
- Protection on branching (branch balancing + instruction cache protection)
- Additional protections for a alternative trade-off between spend cache space and execution time