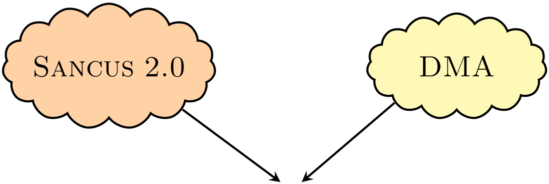
DMA Support for the Sancus Architecture

Lightweight and Open-Source Trusted Computing for the IoT

Sergio Seminara
✉ seminara@kth.se

KTH - Kungliga Tekniska Högskola

Examiner: prof. Mads Dam
Supervisor: prof. Roberto Guanciale



Goal of the Thesis

To extend Sancus 2.0 architecture with DMA capabilities, without affecting its security guarantees

Contributions of the Thesis:

1. Provide a background on Sancus and PMA in general
2. Show that a direct implementation of DMA breaks security guarantees
3. Propose secure DMA implementations on Sancus that preserve security properties

What is DMA?

Direct Memory Access (DMA)

It is a feature of CPUs that allows hardware subsystems to directly access the memory, without the participation of the Control Unit (CU).

☞Without DMA, the CPU would be fully occupied during I/O operations. In this sense, DMA speeds up the system, by unburdening the CPU from I/O loads.

What is Sancus 2.0?

Sancus

Target architecture of the thesis is Sancus[a][5], an open-source, lightweight PMA with a specific focus for networked embedded devices.

[a] Sancus version with secure DMA support is currently maintained on GitHub at `https://github.com/S3rg7o/sancus-core`

Protected Modules Architectures

Security architectures running independently from an operating system, that can execute code in an isolated area of the memory.

Secure Code Execution on Embedded Devices

Embedded device are required to be cheap in terms of:

- Chip area
- Chip complexity
- Power consumption

→ unsuitable to implement established solutions from high-end devices world

A promising solution is found in **Protected (software) Module Architectures**, security architectures that offers:

- Isolated execution of protected software module
- Secure remote attestation
- Divide-and-conquer approach, as complex software is splitted into smaller protected modules, easier to verify [4]

Program-Counter Based Memory Access Control

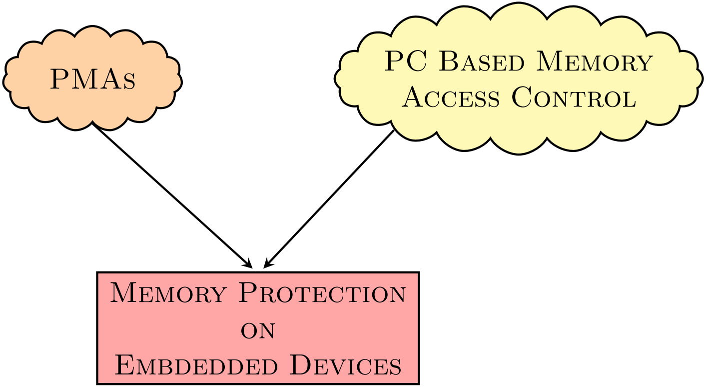
PC Based MAC

Memory protection technique which sets different memory permissions depending on the current value of the PC.

- Hardware-only solution, with minimal TCB[1]
- Strong modules isolation and confidentiality guarantees
- Low cost → compatible with lightweight embedded devices

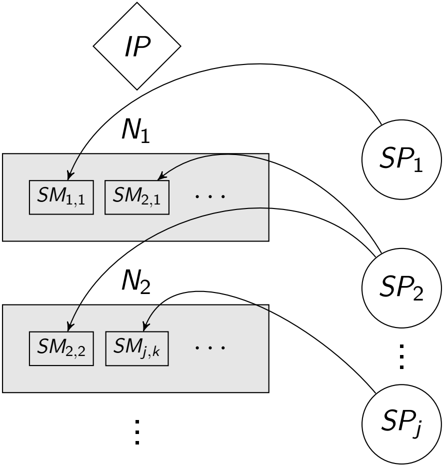
	Memory access rights				
	from \ to	Protected			Unprotected
		Entry point	Code	Data	
PC	Entry point	r - x	r - x	r w -	r w x
	Text section	r - x	r - x	r w -	r w x
	Unprotected \ Other SMs	- - x	- - -	- - -	r w x

[1] Trusted Computing Base


PMAs
PC Based Memory Access Control
Memory Protection on Embedded Devices

System Model

A single infrastructure provider *IP* owns and administers a set of networked microprocessor-based systems, referred as *nodes* N_i.



Security Properties:

- **SMs Isolation**
- **Remote Attestation**
- **Secure Communication & Secure Linking**
- **Secure Key Management**

Attacker Model

Attackers can:

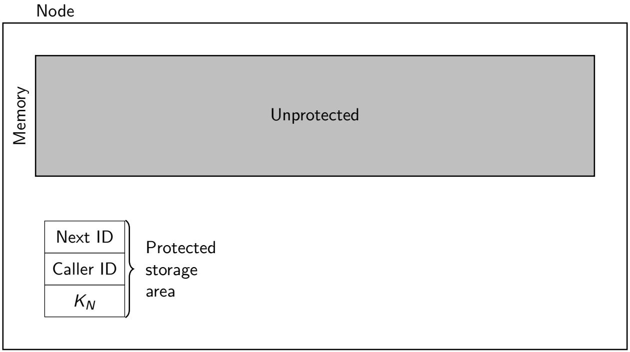
- Manipulate all the software on the node and act as software providers
- Control the communication network, independently of its security protocol
- Perform protocol-level attacks on cryptographic functions
- Plug-in their own peripherals before the system is started. Any further alteration at runtime is not considered in this model

Attackers cannot:

- Have physical access to the hardware of the system. At anytime they cannot:
  - Access CPU internal registers
  - Place probes on memory buses
  - Disconnect components at runtime
- Break cryptographic primitives

Software Module on a Sancus Node

Processor `protect(layout,SP)` instruction supervises SMs deployment

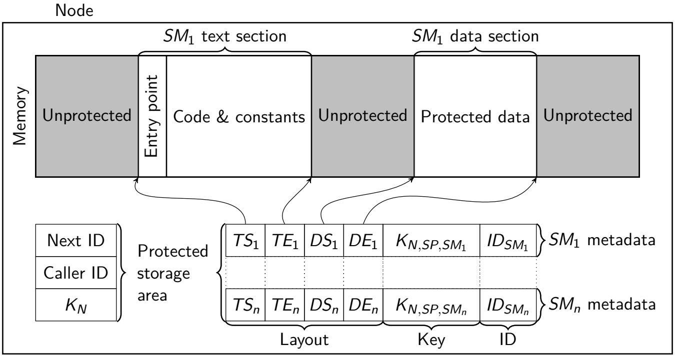


A software module is composed by:

- Code section, containing protected code and constants, that can be entered only via few predefined entry points
- Data section, containing the module private data

Software Module on a Sancus Node

Processor `protect(layout,SP)` instruction supervises SMs deployment

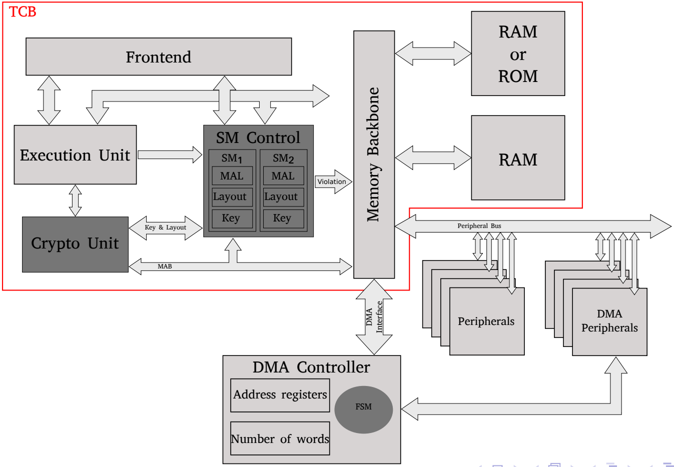


☞The node key K_N, together with all SMs keys K_{N,SP,SM_i}, are stored in the Protected Storage Area (PSA).

☞PSA is not mapped into the system memory ➞ keys never leak on Sancus!

Trusted Computing Base (TCB)

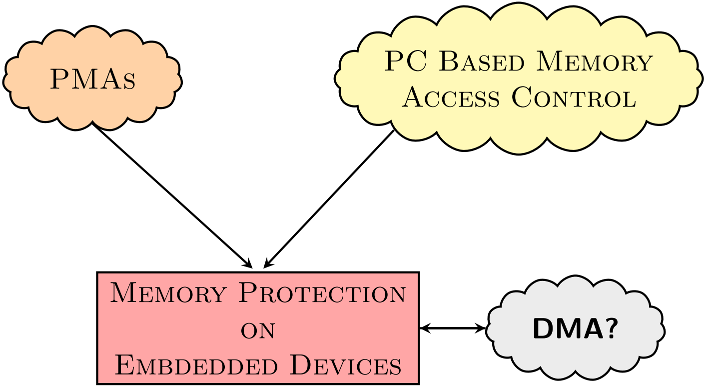
☞The set of hardware or software components critical for the security of the system. Sancus TCB is hardware-only



DMA Controller

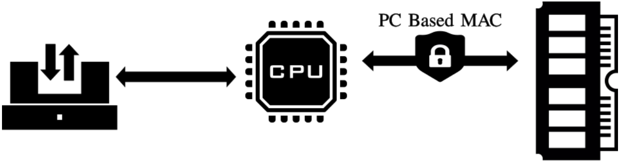
Sancus is provided with a default DMA controller. Main benefits from its inclusion in the system are:

- Multiplex different devices, with a positional priority arbitration
- Incorporate all the complexity of the DMA protocol in use on Sancus


PMAs
PC Based Memory Access Control
Memory Protection on Embedded Devices
DMA?

PMAs generally do not support DMA

PC based MAC is enforced over the CPU memory access bus (MAB).

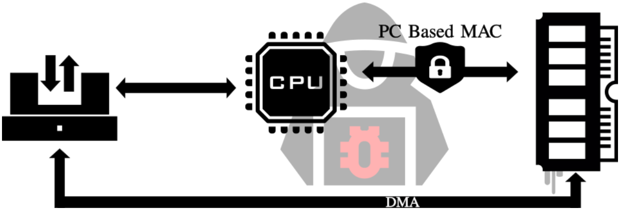


- What if the untrusted element resides outside CPU domain?
- What if there was a way to directly access the memory, bypassing any CPU control, so that no violation is raised on illegal accesses?

DMA Exploitation on PMAs

⚠ Watch out ⚠

An attacker with DMA capabilities can tamper with any location of the system memory at runtime, as DMA bypasses any MMU-like control.



An example of DMA exploitation, for the Sancus architecture, is provided at

`https://github.com/S3rg7o/sancus-examples/blob/master/hello-DMA/Readme.md`.

Sancus 2.0

What about Sancus?

It does not support DMA natively

Direct DMA Implementation

Breaks Sancus security guarantees!

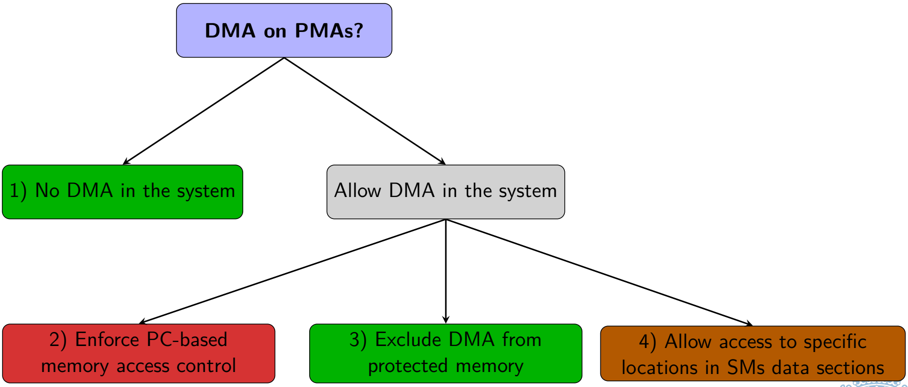
- Every memory location can be accessed, including the SMs protected sections. Modules isolation reneges
- The $K_{N,SP,SM}$ key is computed only once, on module deployment. If isolation reneges, it can't be no longer be considered a sufficient assurance of modules integrity
- Nodes and modules keys are inaccessible from DMA

⚠ SMs isolation is a crucial security property ⚠

An attacker can entirely rewrite the text section of a module, making it de facto a Trojan horse

☞Although alarming, this scenario differs from keys disclosure since attacker's computational capabilities are still confined to compromised node

Need for solutions!



1) No DMA in the System

☞ Sacrifice DMA capabilities in order to preserve security guarantees, without adding complexity to the architecture.

$$\textbf{(1)} \begin{cases} \text{PC} \mathrel{!}= \text{TS} \\ \text{TS} \leq \text{PC} < \text{TE} \\ !(\text{TS} \leq \text{PREV\_PC} < \text{TE}) \end{cases}$$

$$\textbf{(2)} \begin{cases} !(\text{TS} \leq \text{PC} < \text{TE}) \\ \text{DS} \leq \text{MAB} < \text{DE} \end{cases}$$

$$\textbf{(3)} \begin{cases} !(\text{TS} \leq \text{PC} < \text{TE}) \\ \text{TS} \leq \text{MAB} < \text{TE} \end{cases}$$

$$\textbf{(4)} \begin{cases} \text{WRITE\_MEM} == 1 \\ \text{TS} \leq \text{MAB} < \text{TE} \end{cases}$$

violation

TS: Text Section Start Add
TE: Text Section End Add
DS: Data Section Start Add
DE: Data Section End Add

Figure: Example of Memory Access Logic for a single SM

1) No DMA in the System

☞ Sacrifice DMA capabilities in order to preserve security guarantees, without adding complexity to the architecture.

Pros	**Cons**
• Doesn't add complexity to the system • Reasonable trade-off for lightweight, resource constrained, systems	• Does not provide any DMA capability

2) Enforce PC based MAC over DMA Accesses

(1) $\begin{cases} \text{PC} \mathrel{!=} \text{TS} \\ \text{TS} \leq \text{PC} < \text{TE} \\ !(\text{TS} \leq \text{PREV\_PC} < \text{TE}) \end{cases}$

(2) $\begin{cases} !(\text{TS} \leq \text{PC} < \text{TE}) \\ \text{DS} \leq \boxed{\text{MAB or DMA\_ADDR}} < \text{DE} \end{cases}$

(3) $\begin{cases} !(\text{TS} \leq \text{PC} < \text{TE}) \\ \text{TS} \leq \boxed{\text{MAB or DMA\_ADDR}} < \text{TE} \end{cases}$

(4) $\begin{cases} \text{WRITE\_MEM} == 1 \\ \text{TS} \leq \boxed{\text{MAB or DMA\_ADDR}} < \text{TE} \end{cases}$

violation

TS: Text Section Start Add
TE: Text Section End Add
DS: Data Section Start Add
DE: Data Section End Add

2) Enforce PC based MAC over DMA Accesses

Pros	Cons
• Allows DMA in the system • Expands the already present Sancus MAL circuitry with minimal hardware additions	• Flawed idea of relating two independent entities as the PC and the DMA bus • Opens to privilege escalation attacks, as PC is free to vary during a DMA operation

3) Exclude DMA from Protected Memory

(1) $\begin{cases} \text{PC} \mathrel{!=} \text{TS} \\ \text{TS} \leq \text{PC} < \text{TE} \\ !(\text{TS} \leq \text{PREV\_PC} < \text{TE}) \end{cases}$

(2) $\begin{cases} !(\text{TS} \leq \text{PC} < \text{TE}) \\ \text{DS} \leq \text{MAB} < \text{DE} \end{cases}$

(3) $\begin{cases} !(\text{TS} \leq \text{PC} < \text{TE}) \\ \text{TS} \leq \text{MAB} < \text{TE} \end{cases}$

(4) $\begin{cases} \text{WRITE\_MEM} == 1 \\ \text{TS} \leq \text{MAB} < \text{TE} \end{cases}$

(5) $\begin{cases} \text{DMA\_EN} == 1 \\ \text{TS} \leq \text{DMA\_ADDR} < \text{TE} \end{cases}$

(6) $\begin{cases} \text{DMA\_EN} == 1 \\ \text{DS} \leq \text{DMA\_ADDR} < \text{DE} \end{cases}$

→ violation

TS: Text Section Start Add
TE: Text Section End Add
DS: Data Section Start Add
DE: Data Section End Add

3) Exclude DMA from Protected Memory

Pros

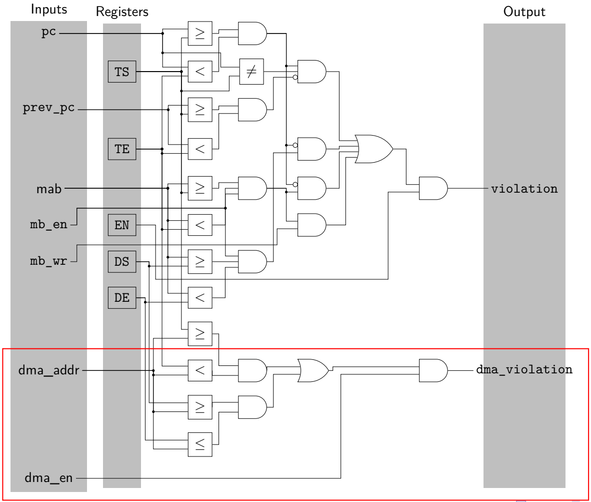
- Allows DMA in the system, preventing accesses to protected memory (SMs integrity and confidentiality preserved)
- Reuse the already instantiated MAL registers TS, TE, DS and DE
- No software overhead or SMs direct intervention required

Cons

- Solely allows DMA operations on unprotected memory. Does not really extend SMs functionalities

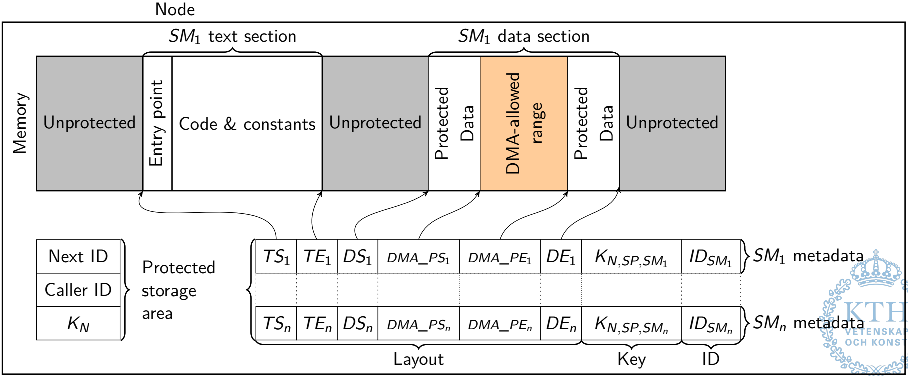
3) Exclude DMA from Protected Memory

MAL Circuit for a Single Software Module



4) Allow Access to Specific Locations inside SMs Data Sections

☞ Enhances SMs functionalities with DMA capabilities, by relaxing integrity and confidentiality guarantees for a specific subset of the data section



4) Allow Access to Specific Locations inside SMs Data Sections

$$\textbf{(1)} \begin{cases} \text{PC} \mathrel{!}= \text{TS} \\ \text{TS} \leq \text{PC} < \text{TE} \\ !(\text{TS} \leq \text{PREV\_PC} < \text{TE}) \end{cases}$$

$$\textbf{(2)} \begin{cases} !(\text{TS} \leq \text{PC} < \text{TE}) \\ \text{DS} \leq \text{MAB} < \text{DE} \end{cases}$$

$$\textbf{(3)} \begin{cases} !(\text{TS} \leq \text{PC} < \text{TE}) \\ \text{TS} \leq \text{MAB} < \text{TE} \end{cases}$$

$$\textbf{(4)} \begin{cases} \text{WRITE\_MEM} == 1 \\ \text{TS} \leq \text{MAB} < \text{TE} \end{cases}$$

$$\textbf{(5)} \begin{cases} \text{DMA\_EN} == 1 \\ \text{DS} \leq \text{DMA\_ADDR} < \text{DE} \\ !(\text{DMA\_PS} \leq \text{DMA\_ADDR} < \\ \text{DMA\_PE}) \end{cases}$$

$$\textbf{(6)} \begin{cases} \text{DMA\_EN} == 1 \\ \text{TS} \leq \text{DMA\_ADDR} < \text{TE} \end{cases}$$

violation

TS: Text Section Start Add
TE: Text Section End Add
DS: Data Section Start Add
DE: Data Section End Add
DMA_PS: DMA Protected Start
DMA_PE: DMA Protected End

4) Allow Access to Specific Locations inside SMs Data Sections

Pros	**Cons**
• Allows DMA in the system, preserviung SMs integrity and confidentiality • Reuse of the already instantiated MAL registers TS, TE, DS and DE • Full configurability of the system	• Extension of the ISA with a new instruction to set the boundaries of the DMA-allowed subset • Register overhead: two extra registers for each SM • Implicit trustworthiness of all the DMA peripherals: currently impossible to selectively provide peripherals with access to protected memory

Reduce Register Overhead

4.1) Fix the start or the end addresses of the DMA-allowed subset

Pros	Cons
• Register overhead is halved, since only the loose boundary has to be stored	• Reduced system flexibility in positioning the subset inside the data section

Reduce Register Overhead

4.2) Allow only one DMA-allowed subset per time

Pros	Cons
• Register overhead is dramatically reduced	• No direct data transfer between two SMs with DMA • Software overhead, as each SM must load the boundaries of its DMA allowed memory subset before any DMA operation

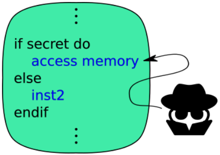
```
// ============================
//    ENABLE SMs PROTECTION   |
// ============================
          ....
SM with ID 2 enabled:
    0x7588 0x78c2 0x02aa 0x03b4
          ....
// ============================
//     START THE ATTACK       |
// ============================
[attacker] Reading into SM2 text section...
[attacker] Num. of Words: 6

Data0 at addr. 0x7588: 0xc232
Data1 at addr. 0x758a: 0x4182
Data2 at addr. 0x758c: 0x03b0
Data3 at addr. 0x758e: 0x40b2
Data4 at addr. 0x7590: 0x03ac
Data5 at addr. 0x7592: 0x03b2

[attacker] Writing into SM2 text section...
[attacker] Num. of Words: 6
...
[attacker] Reading into SM2 text section after
     having written...
[attacker] Num. of Words: 6

Data0 at addr. 0x7588: 0x0000
Data1 at addr. 0x758a: 0x0001
Data2 at addr. 0x758c: 0x0002
Data3 at addr. 0x758e: 0x0003
Data4 at addr. 0x7590: 0x0004
Data5 at addr. 0x7592: 0x0005
// ============================
//      SIMULATION PASSED     |
// ============================
```

```
// ============================
//    ENABLE SMs PROTECTION   |
// ============================
          ....
SM with ID 2 enabled:
    0x7588 0x78c2 0x02aa 0x03b4
          ....
// ============================
//     START THE ATTACK       |
// ============================
[attacker] Reading into SM2 text section...
[attacker] Num. of Words: 6
--> Illegal mem. access detected!
Data0 at addr. 0x7588: 0x0000
Data1 at addr. 0x758a: 0x0000
Data2 at addr. 0x758c: 0x0000
Data3 at addr. 0x758e: 0x0000
Data4 at addr. 0x7590: 0x0000
Data5 at addr. 0x7592: 0x0000

[attacker] Writing into SM2 text section...
[attacker] Num. of Words: 6
--> Illegal mem. access detected!
[attacker] Reading into SM2 text section after
     having written...
[attacker] Num. of Words: 6
--> Illegal mem. access detected!
Data0 at addr. 0x7588: 0x0000
Data1 at addr. 0x758a: 0x0000
Data2 at addr. 0x758c: 0x0000
Data3 at addr. 0x758e: 0x0000
Data4 at addr. 0x7590: 0x0000
Data5 at addr. 0x7592: 0x0000
// ============================
//      SIMULATION PASSED     |
// ============================
```

Open Problems

– **Rowhammer:** DMA widens the threats of Rowhammer attacks by allowing to repeatedly access specific memory locations without any CPU involvement

– **Side Channel Attacks:** DMA support inclusion extends the side channel attack base.



Conclusions and Future Work

Conclusions

- The disruptive outcomes of direct DMA implementation prompt the need of providing secure DMA support.
- Suitable solutions are to totally exclude DMA from protected memory, or to relax securities guarantees and allow DMA in specific protected memory locations

Future Work

- Implement and investigate the theoretical solution that allows DMA in confined memory subsets of SMs data sections
- Include the DMA controller in the TCB, allowing it to:
  - Introduce the concept of trusted peripherals IDs
  - Selectively grant access to DMA interface
  - Store private informations, like the identity or the ID, of the party that started a DMA operation

References I

Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens.
Secure compilation to modern processors, 2012.

Victor Costan and Srinivas Devadas.
Intel sgx explained.
Cryptology ePrint Archive, Report 2016/086, 2016.

Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley.
Iso-x: A flexible architecture for hardware-managed isolated execution.
In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 190–202, Washington, DC, USA, 2014. IEEE Computer Society.

Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki.
Flicker: An execution infrastructure for tcb minimization.
In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 315–328, New York, NY, USA, 2008. ACM.

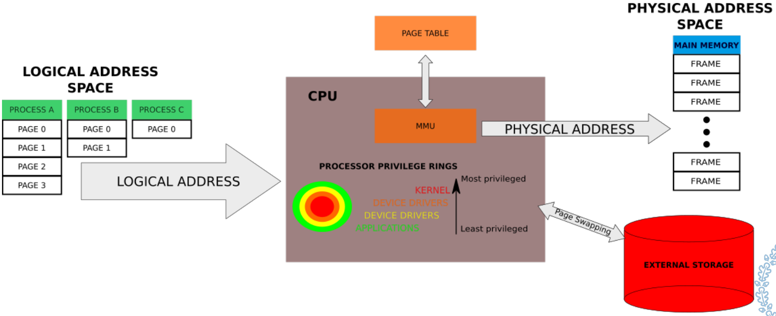
References II

Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling.
Sancus 2.0: A low-cost security architecture for iot devices.
ACM Transactions on Privacy and Security (TOPS), 20(3):7:1–7:33, September 2017.

Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens.
Secure compilation to protected module architectures.
ACM Trans. Program. Lang. Syst., 37(2):6:1–6:50, April 2015.

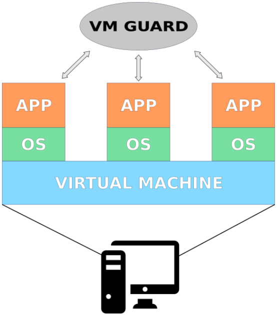
Solutions on High-End Devices

1. Use of hardware support for virtual memory in combination with processor privilege levels.
2. Use of a memory-safe virtual machine equipped with a VM guard



Solutions on High-End Devices

1. Use of hardware support for virtual memory in combination with processor privilege levels.
2. Use of a memory-safe virtual machine equipped with a VM guard



Disadvantages of Classical Solutions

1. Non-trivial support for remote attestation[2]
2. Expensive to implement → non compatible with lightweight embedded devices
3. Rely on a software layer – OS or hypervisor → Do not protect from system-level attacks.

⚠ Watch out ⚠

A tampered with OS allows attackers to fully manipulate the software, breaking the root of trust ➞ need for hardware-based solutions.

[2] Remote attestation is a property of a system which a remote stakeholder relies on to verify that a specific software module is running untampered on a remote device.

Program-Counter Based Memory Access Control

PC Based MAC

Memory protection technique which sets different memory permissions depending on the current value of the PC.

	Memory access rights				
	from \ to	Protected			Unprotected
		Entry point	Code	Data	
PC	Entry point	r - x	r - x	r w -	r w x
	Text section	r - x	r - x	r w -	r w x
	Unprotected \ Other SMs	- - x	- - -	- - -	r w x

Program-Counter Based Memory Access Control

- Hardware-only solution, with minimal TCB[3]
- Strong modules isolation and confidentiality guarantees
- Low cost → compatible with lightweight embedded devices
- Preserves isolation of compiled code from modern programming languages (C++, Java, etc...) [1, 6]

	Memory access rights				
	from \ to	Protected			Unprotected
		Entry point	Code	Data	
PC	Entry point	r - x	r - x	r w -	r w x
	Text section	r - x	r - x	r w -	r w x
	Unprotected \ Other SMs	- - x	- - -	- - -	r w x

[3] Trusted Computing Base

Program-Counter Based Memory Access Control

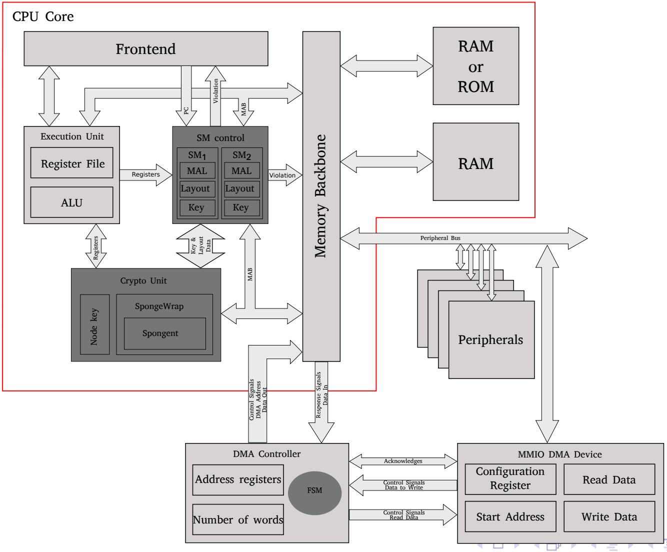
- Hardware-only solution, with minimal TCB
- Strong modules isolation and confidentiality guarantees
- Low cost → compatible with lightweight embedded devices
- Preserves isolation of compiled code from modern programming languages (C++, Java, etc...) [1, 6]

```
    ***Java code***

public class Foo{
private int secret = 0;


public void add() {
this.secret += 1;
}
}
```

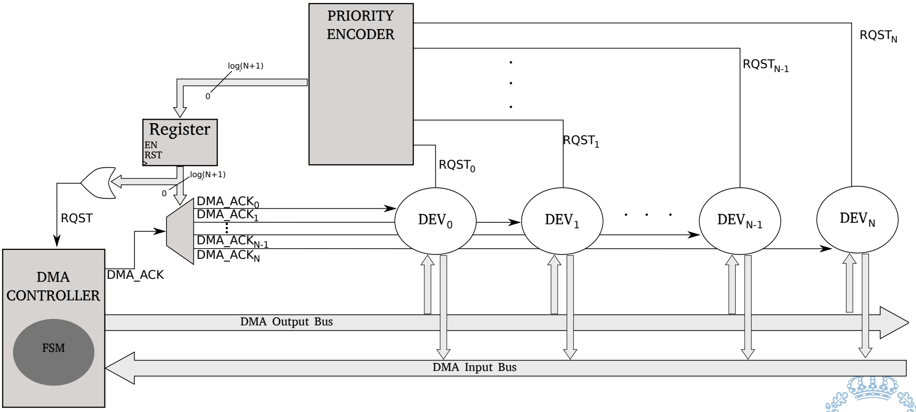
```
     ***C code***

typedef struct foo_t {
int secret = 0;

void (*add)(struct Foo*, int)
= add_f; } Foo;

void add_f(Foo* a, int amount)
    {
a → secret += 1;
return; }
```

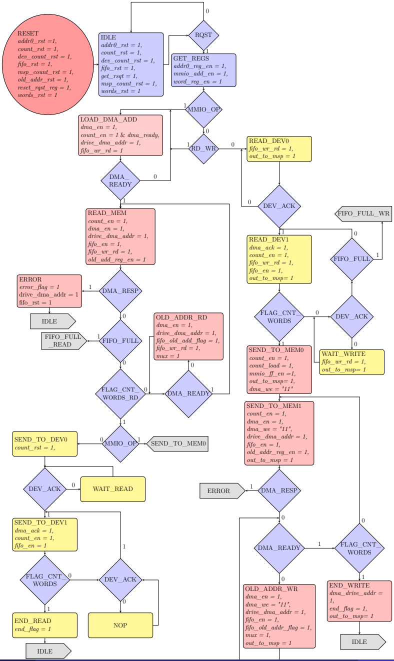
leaked_secret =
  *(Foo_ptr+sizeof(int))

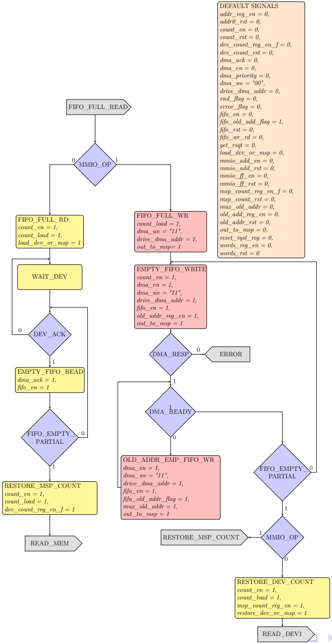
Sancus Extended System View



Arbitration Circuit

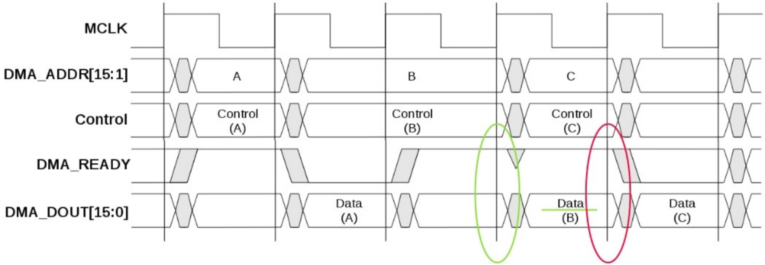



RESET
addr0_rst =1,
count_rst = 1,
dev_count_rst = 1,
fifo_rst = 1,
msp_count_rst = 1,
old_addr_rst = 1,
reset_rqst_reg = 1,
words_rst = 1
IDLE
addr0_rst = 1,
count_rst = 1,
dev_count_rst = 1,
fifo_rst = 1,
get_rsqt = 1,
msp_count_rst = 1,
words_rst = 1
RQST
0
1
GET_REGS
addr0_reg_en = 1,
mmio_add_en = 1,
word_reg_en = 1
MMIO_OP
LOAD_DMA_ADD
dma_en = 1,
count_en = 1 & dma_ready,
drive_dma_addr = 1,
fifo_wr_rd = 1
RD_WR
READ_DEV0
fifo_wr_rd = 1,
out_to_msp = 1
DMA_ READY
DEV_ACK
FIFO_FULL_WR
READ_MEM
count_en = 1,
dma_en = 1,
drive_dma_addr = 1,
fifo_en = 1,
fifo_wr_rd = 1,
old_add_reg_en = 1
READ_DEV1
dma_ack = 1,
count_en = 1,
fifo_wr_rd = 1,
fifo_en = 1,
out_to_msp= 1
FIFO_FULL
ERROR
error_flag = 1
drive_dma_addr = 1
fifo_rst = 1
DMA_RESP
IDLE
FLAG_CNT_ WORDS
DEV_ACK
OLD_ADDR_RD
dma_en = 1,
drive_dma_addr = 1,
fifo_old_add_flag = 1,
fifo_wr_rd = 1,
mux = 1
FIFO_FULL _READ
FIFO_FULL
SEND_TO_MEM0
count_en = 1,
count_load = 1,
mmio_ff_en =1,
out_to_msp= 1,
dma_we = "11"
WAIT_WRITE
fifo_wr_rd = 1,
out_to_msp= 1
FLAG_CNT_ WORDS_RD
DMA_READY
SEND_TO_MEM1
count_en = 1,
dma_en = 1,
dma_we = "11",
drive_dma_addr = 1,
fifo_en = 1,
old_addr_reg_en = 1,
out_to_msp = 1
SEND_TO_DEV0
count_rst = 1,
MMIO_OP
SEND_TO_MEM0
DEV_ACK
WAIT_READ
ERROR
DMA_RESP
SEND_TO_DEV1
dma_ack = 1,
count_en = 1,
fifo_en = 1
DMA_READY
FLAG_CNT_ WORDS
FLAG_CNT_ WORDS
DEV_ACK
END_READ
end_flag = 1
NOP
OLD_ADDR_WR
dma_en = 1,
dma_we = "11",
drive_dma_addr = 1,
fifo_en = 1,
fifo_old_addr_flag = 1,
mux = 1,
out_to_msp = 1
END_WRITE
dma_drive_addr = 1,
end_flag = 1,
out_to_msp= 1
IDLE
IDLE

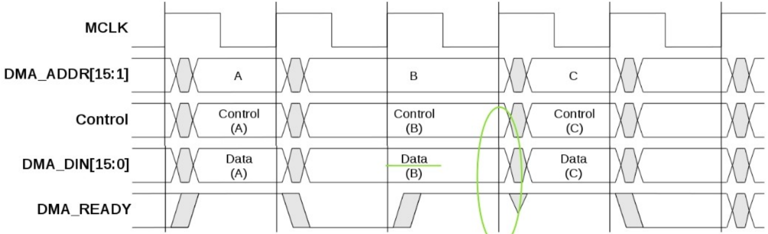

DEFAULT SIGNALS
addr_reg_en = 0,
addr0_rst = 0,
count_en = 0,
count_rst = 0,
dev_count_reg_en_f = 0,
dev_count_rst = 0,
dma_ack = 0,
dma_en = 0,
dma_priority = 0,
dma_we = "00",
drive_dma_addr = 0,
end_flag = 0,
error_flag = 0,
fifo_en = 0,
fifo_old_add_flag = 1,
fifo_rst = 0,
fifo_wr_rd = 0,
get_rsqt = 0,
load_dev_or_msp = 0,
mmio_add_en = 0,
mmio_add_rst = 0,
mmio_ff_en = 0,
mmio_ff_rst = 0,
msp_count_reg_en_f = 0,
msp_count_rst = 0,
mux_old_addr = 0,
old_add_reg_en = 0,
old_addr_rst = 0,
out_to_msp = 0,
reset_rqst_reg = 0,
words_reg_en = 0,
words_rst = 0
FIFO_FULL_READ
MMIO_OP
0
1
FIFO_FULL_RD
count_en = 1,
count_load = 1,
load_dev_or_msp = 1
FIFO_FULL_WR
count_load = 1,
dma_we = "11",
drive_dma_addr = 1,
out_to_msp= 1
WAIT_DEV
EMPTY_FIFO_WRITE
count_en = 1,
dma_en = 1,
dma_we = "11",
drive_dma_addr = 1,
fifo_en = 1,
old_addr_reg_en = 1,
out_to_msp = 1
DEV_ACK
0
1
DMA_RESP
0
ERROR
1
EMPTY_FIFO_READ
dma_ack = 1,
fifo_en = 1
DMA_READY
1
0
FIFO_EMPTY_
PARTIAL
0
1
OLD_ADDR_EMP_FIFO_WR
dma_en = 1,
dma_we = "11",
drive_dma_addr = 1,
fifo_en = 1,
fifo_old_addr_flag = 1,
mux_old_addr = 1,
out_to_msp = 1
FIFO_EMPTY_
PARTIAL
0
1
RESTORE_MSP_COUNT
count_en = 1,
count_load = 1,
dev_count_reg_en_f = 1
READ_MEM
RESTORE_MSP_COUNT
1
MMIO_OP
0
RESTORE_DEV_COUNT
count_en = 1,
count_load = 1,
msp_count_reg_en = 1,
restore_dev_or_msp = 1
READ_DEV1

DMA Protocol for Read/Write Operations

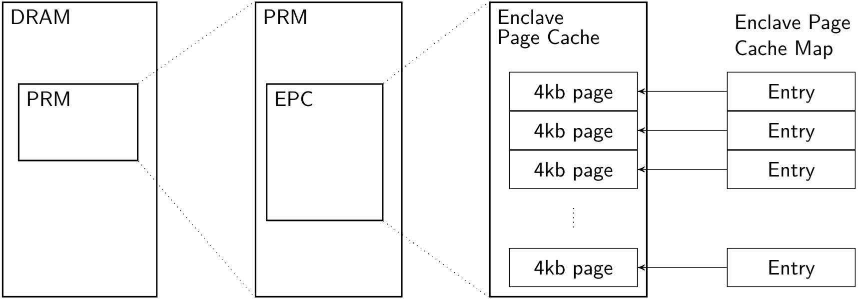
Timing diagram for a read operation



Timing diagram for a write operation



Detail of Intel SGX Processor Reserved Memory (PRM)



In Intel SGX [2] (or Iso-X [3]) the equivalent of modules protected sections are stored in a specific range of the memory. Hence, the protection mechanism consists in denying every DMA accesses to those regions.