Proving the security of interrupt handling against interrupt-based side-channel attacks: a case study

Frank Piessens

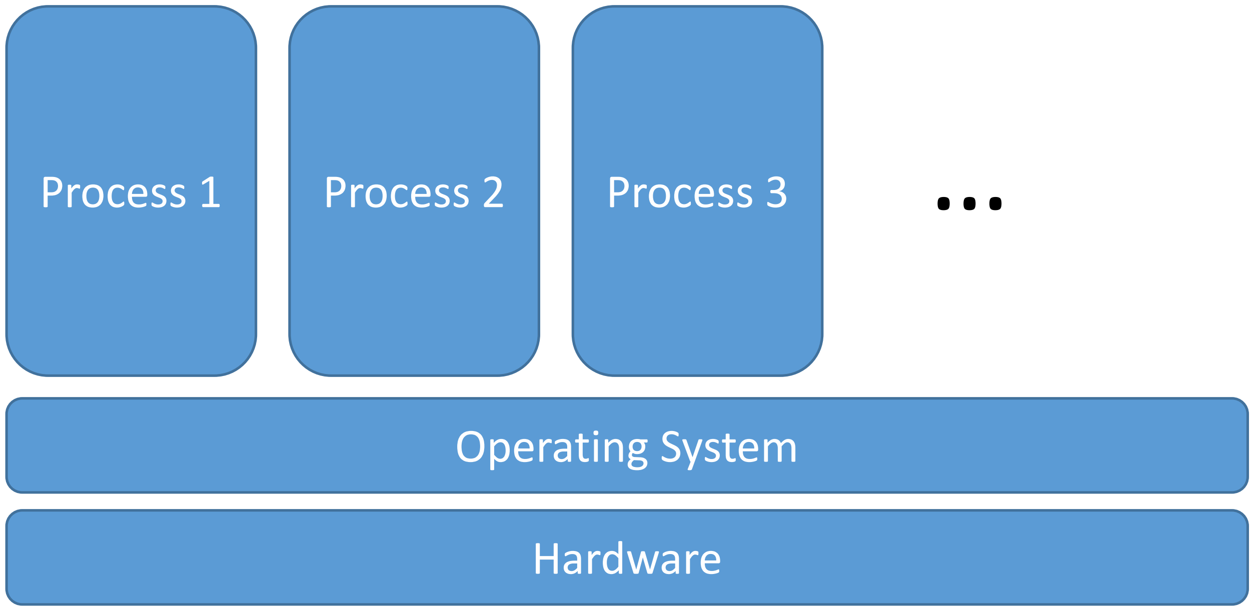
Entropy 2019 Workshop

(Joint work with: Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg)

Overview

- Introduction: hardware isolation mechanisms and micro-architectural attacks
- Enclaved execution: Sancus
- Extending Sancus with interrupts
- Formalization and security proof
- Implementation
- Conclusions

Hardware isolation mechanisms



Process 1

Process 2

Process 3

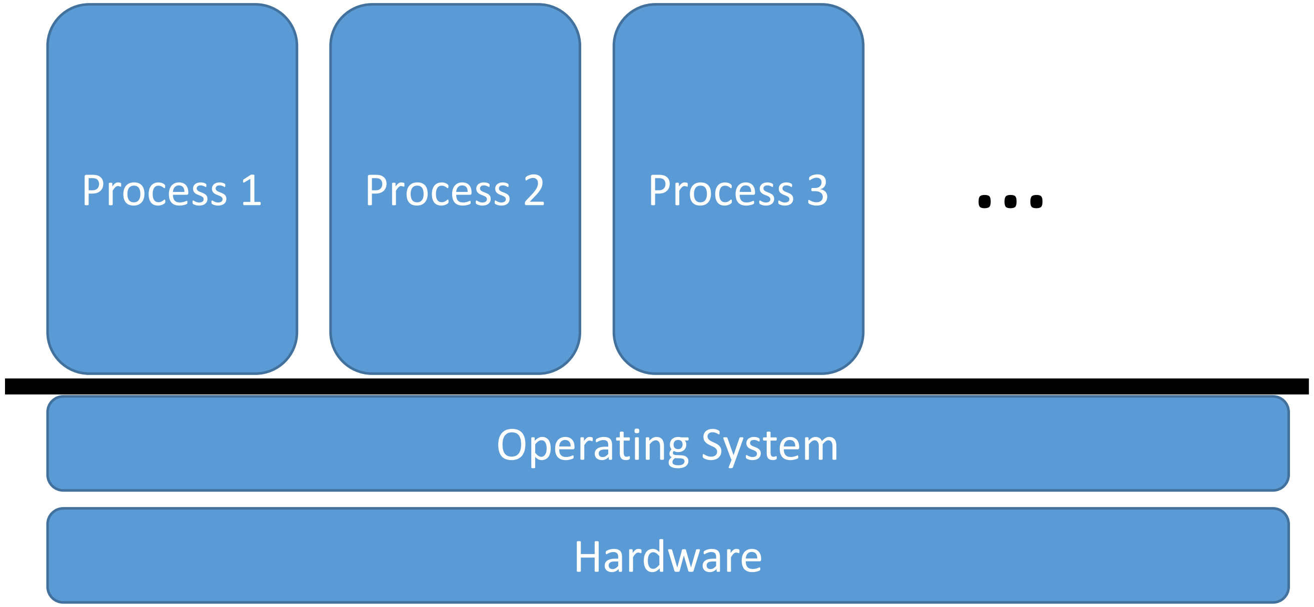
…

Operating System

Hardware

Protecting the kernel: privilege levels

OS is protected
from applications
by privilege level



Process 1

Process 2

Process 3

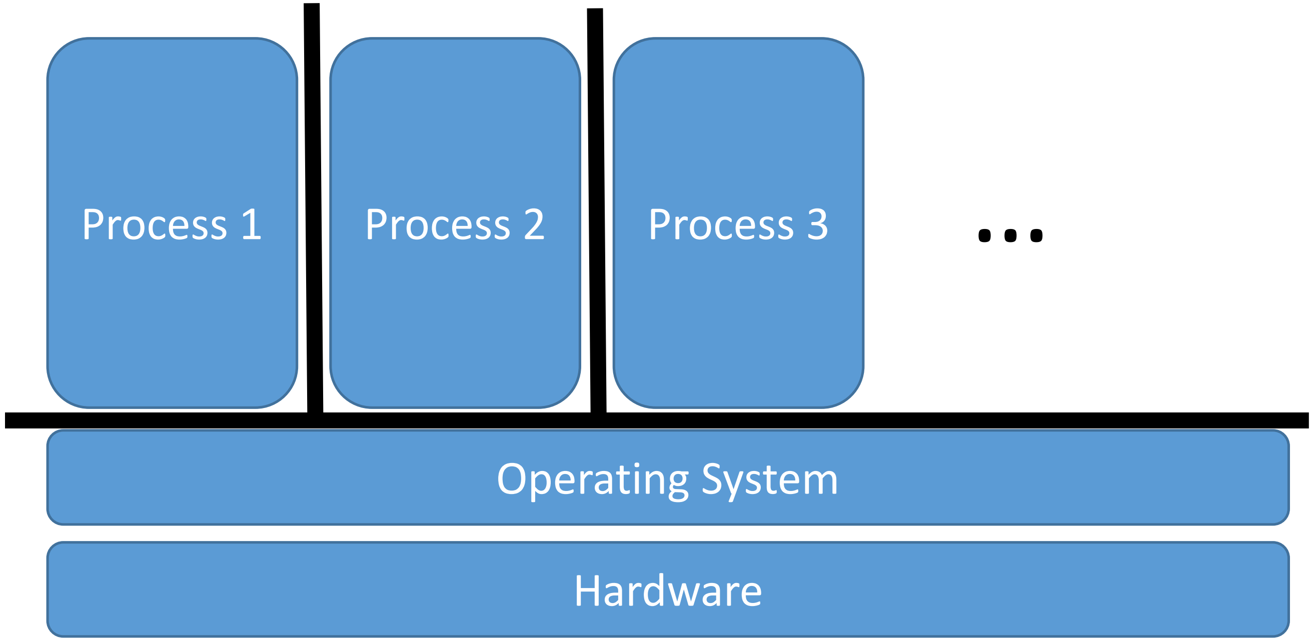
…

Operating System

Hardware

Protecting processes: virtual memory

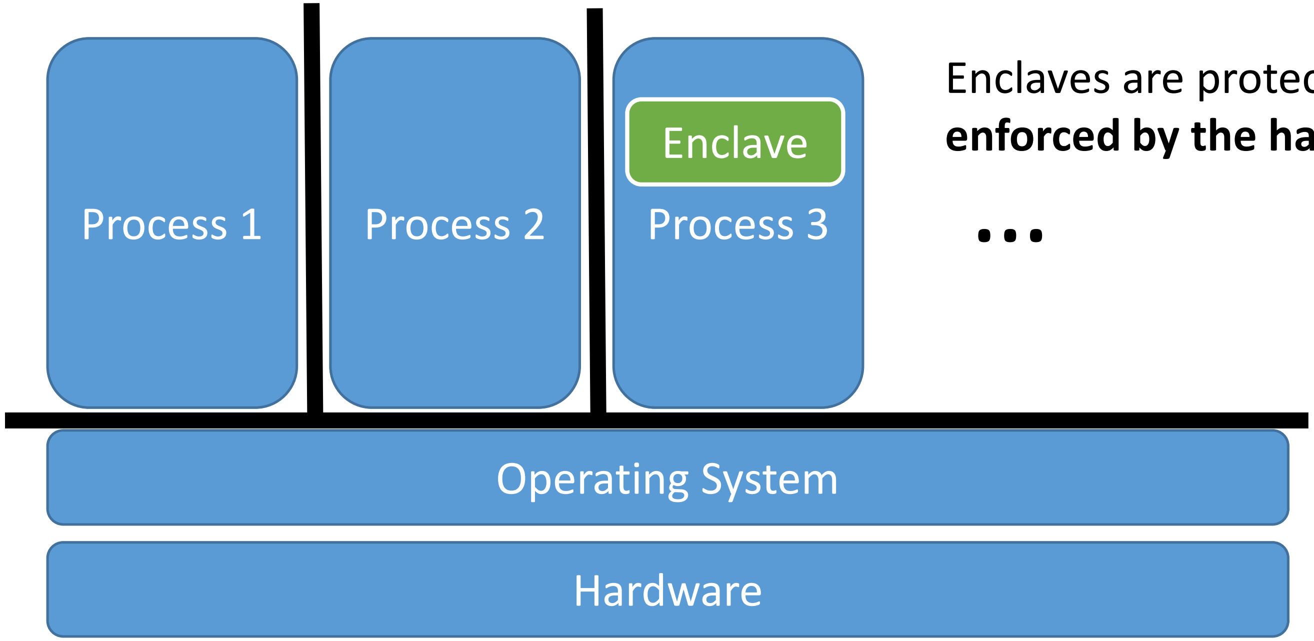
Processes are protected from each other through memory isolation



OS is protected
from applications
by privilege level

Protecting critical software: enclaves

Processes are protected from each other through memory isolation



Enclaves are protected by memory isolation **enforced by the hardware only**

…

OS is protected
from applications
by privilege level

Micro-architectural attacks

- Over the past two years, all these isolation mechanisms have been broken dramatically:
  - Meltdown breaks user/kernel isolation
  - Spectre breaks several isolation including process boundaries and software defined boundaries
  - Foreshadow breaks SGX enclave isolation
- And older but less impactful micro-architectural attacks have been known for over a decade

References:

Paul Kocher et al. *Spectre Attacks: Exploiting Speculative Execution*, IEEE S&P 2019

Moritz Lipp et al. *Meltdown: Reading Kernel Memory from User Space*, USENIX Security Symposium 2018

Jo Van Bulck et al. *Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution*, USENIX Security Symposium 2018

Objective of our work

Study **one specific attack mechanism**

- More specifically, interrupt-based attacks

against **one specific isolation mechanism**

- More specifically, enclaves on small microprocessors

very **rigorously**

- More specifically, fully formal security objectives and proofs

Overview

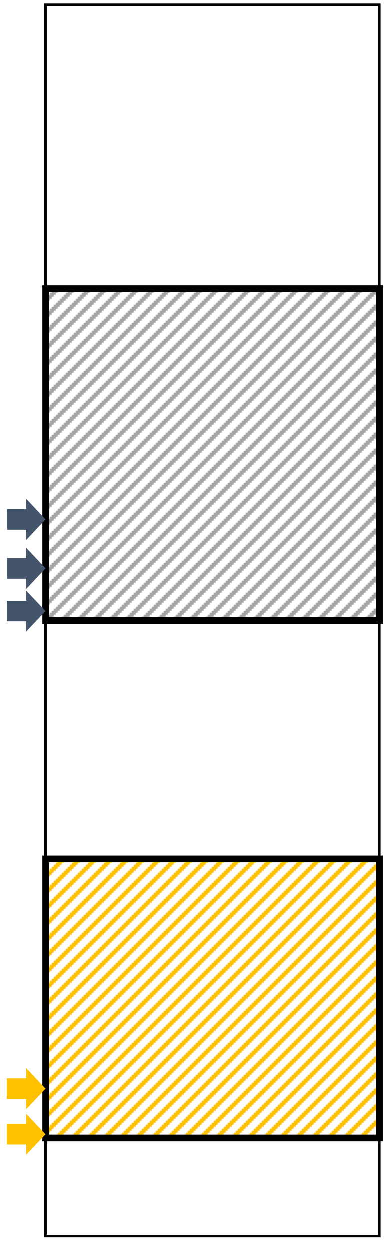
- Introduction: hardware isolation mechanisms and micro-architectural attacks
- ➡ Enclaved execution: Sancus
- Extending Sancus with interrupts
- Formalization and security proof
- Implementation
- Conclusions

Enclaved execution

- Security mechanism that enables **secure remote computation**
    - An isolation mechanism that relies only on the hardware
    - Remote attestation to provide assurance on proper initialization
    - Support for secure communication
- Implementations exist for small and large processors
    - Cloud-level processors: Intel Software Guard Extensions (Intel SGX)
    - IoT level processors: Sancus, Trustlite, Soteria, …
- For this talk we focus on just the isolation mechanism

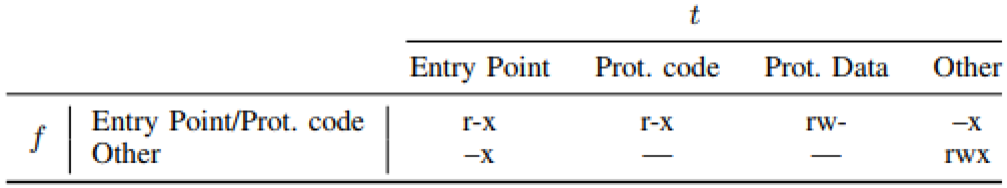
Sancus isolation

- Instructions to create protected modules or *enclaves*
  - Contiguous memory range with list of entry points
- PC-based memory access control
  - PC within enclave: full access to enclave memory
  - PC outside enclave: only jumping to entry point is allowed
- Key property: **encapsulation**
  - By keeping code and data of a module within one enclave, the code of the module has exclusive access to the data of the module

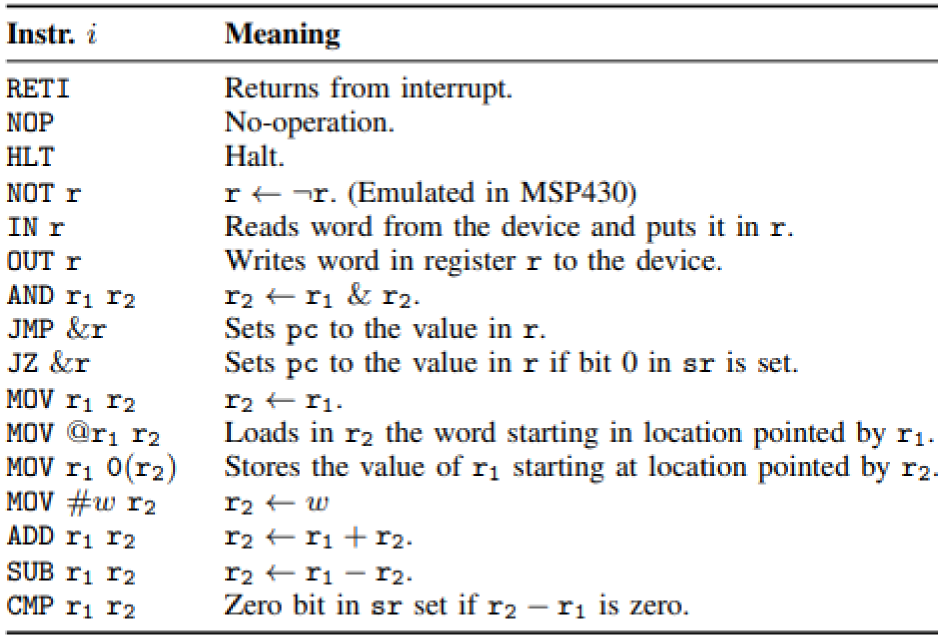


Our model of Sancus

- A simplified TI MSP430 processor
  - Standard instruction set + HLT/IN/OUT
  - 64KB of byte addressable memory
  - Supporting a single enclave
- A single I/O device
  - Can model a cycle-accurate timer
  - Can be an arbitrary deterministic I/O automaton
- PC-based memory access control



		t			
		Entry Point	Prot. code	Prot. Data	Other
f	Entry Point/Prot. code	r-x	r-x	rw-	–x
	Other	–x	—	—	rwx



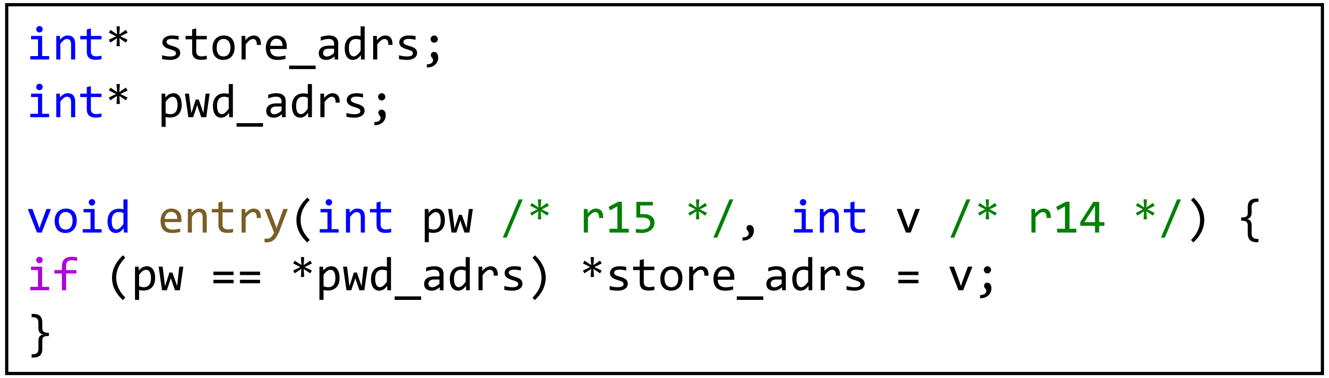
Instr. i	Meaning
RETI	Returns from interrupt.
NOP	No-operation.
HLT	Halt.
NOT r	$r \leftarrow \neg r$. (Emulated in MSP430)
IN r	Reads word from the device and puts it in r.
OUT r	Writes word in register r to the device.
AND r_1 r_2	$r_2 \leftarrow r_1 \,\&\, r_2$.
JMP &r	Sets pc to the value in r.
JZ &r	Sets pc to the value in r if bit 0 in sr is set.
MOV r_1 r_2	$r_2 \leftarrow r_1$.
MOV @r_1 r_2	Loads in r_2 the word starting in location pointed by r_1.
MOV r_1 0(r_2)	Stores the value of r_1 starting at location pointed by r_2.
MOV #w r_2	$r_2 \leftarrow w$
ADD r_1 r_2	$r_2 \leftarrow r_1 + r_2$.
SUB r_1 r_2	$r_2 \leftarrow r_1 - r_2$.
CMP r_1 r_2	Zero bit in sr set if $r_2 - r_1$ is zero.

Security definitions

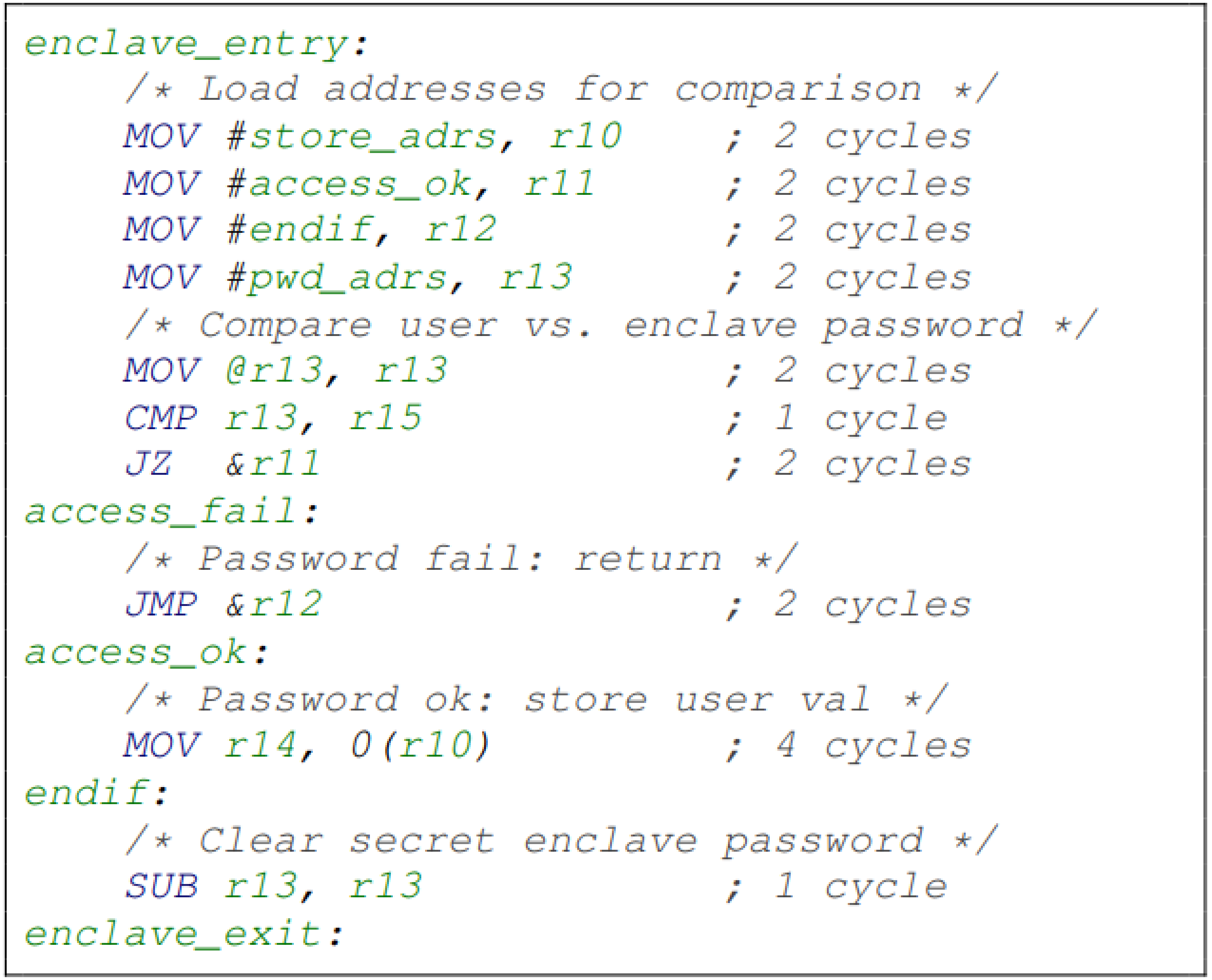
- Attacker model: attacker controls the entire **context** of an enclave
  - All of the unprotected memory
  - The connected device
- Isolation properties are formalized by means of contextual equivalence
  - Our security objective is to “not weaken isolation on extension of the processor”
  - We formalize this as “preservation of contextual equivalence”

Example

- Two instances of this enclave differing in the value at pwd_adrs:
  - Are contextually equivalent if the attacker does not have a timer device
  - Are not contextually equivalent otherwise
- Sancus is vulnerable to end-to-end timing attacks



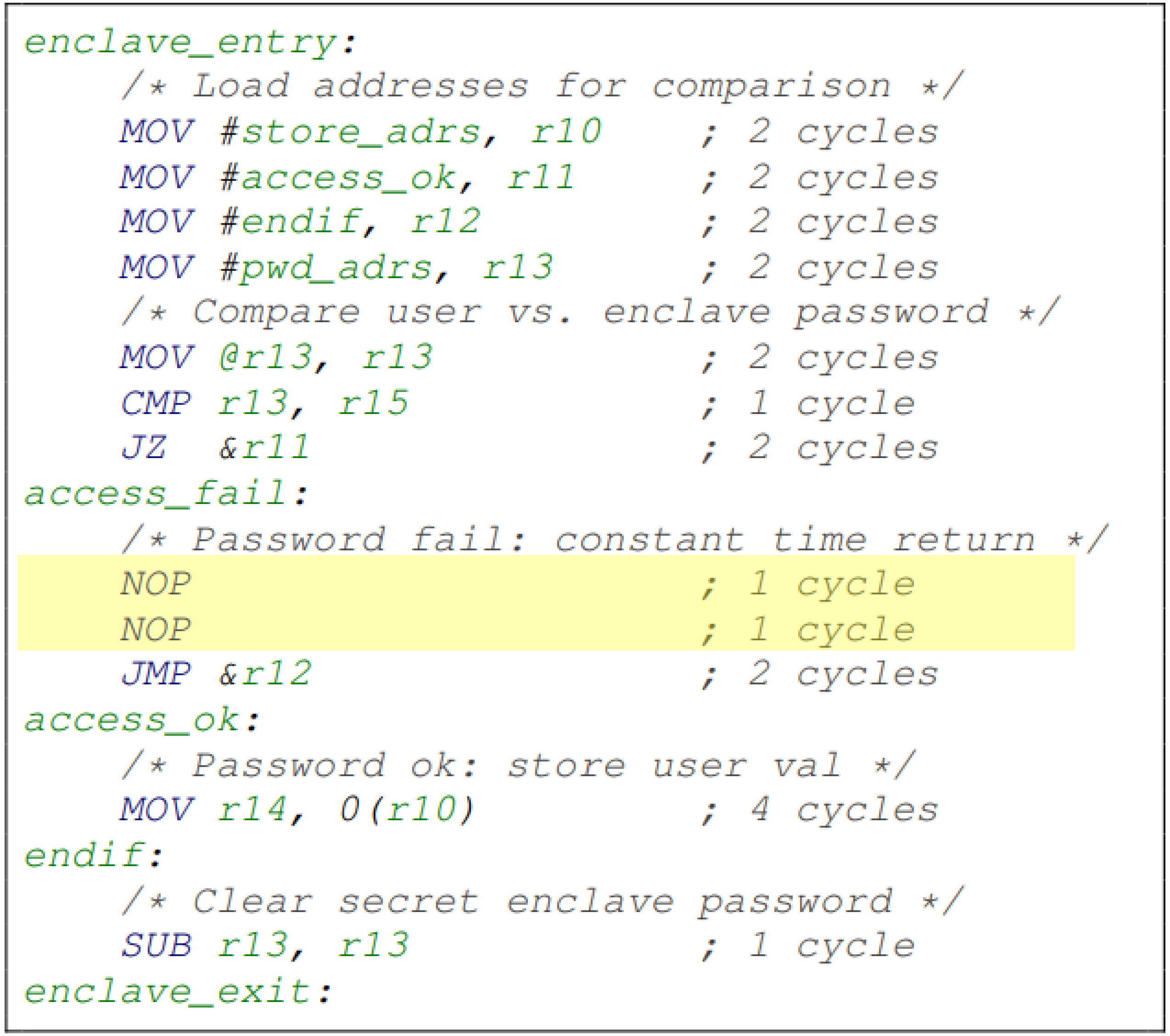
```
int* store_adrs;
int* pwd_adrs;

void entry(int pw /* r15 */, int v /* r14 */) {
if (pw == *pwd_adrs) *store_adrs = v;
}
```



```
enclave_entry:
    /* Load addresses for comparison */
    MOV #store_adrs, r10      ; 2 cycles
    MOV #access_ok, r11       ; 2 cycles
    MOV #endif, r12           ; 2 cycles
    MOV #pwd_adrs, r13        ; 2 cycles
    /* Compare user vs. enclave password */
    MOV @r13, r13             ; 2 cycles
    CMP r13, r15              ; 1 cycle
    JZ  &r11                  ; 2 cycles
access_fail:
    /* Password fail: return */
    JMP &r12                  ; 2 cycles
access_ok:
    /* Password ok: store user val */
    MOV r14, 0(r10)           ; 4 cycles
endif:
    /* Clear secret enclave password */
    SUB r13, r13              ; 1 cycle
enclave_exit:
```

Closing the timing leak

- Balancing out execution time of the two if-branches closes the timing leak
- Now, two instances of the enclave with different values at address pwd_adrs are contextually equivalent



```
enclave_entry:
    /* Load addresses for comparison */
    MOV #store_adrs, r10     ; 2 cycles
    MOV #access_ok, r11      ; 2 cycles
    MOV #endif, r12          ; 2 cycles
    MOV #pwd_adrs, r13       ; 2 cycles
    /* Compare user vs. enclave password */
    MOV @r13, r13            ; 2 cycles
    CMP r13, r15             ; 1 cycle
    JZ  &r11                 ; 2 cycles
access_fail:
    /* Password fail: constant time return */
    NOP                      ; 1 cycle
    NOP                      ; 1 cycle
    JMP &r12                 ; 2 cycles
access_ok:
    /* Password ok: store user val */
    MOV r14, 0(r10)          ; 4 cycles
endif:
    /* Clear secret enclave password */
    SUB r13, r13             ; 1 cycle
enclave_exit:
```

Overview

- Introduction: hardware isolation mechanisms and micro-architectural attacks
- Enclaved execution: Sancus
- ➡ Extending Sancus with interrupts
- Formalization and security proof
- Implementation
- Conclusions

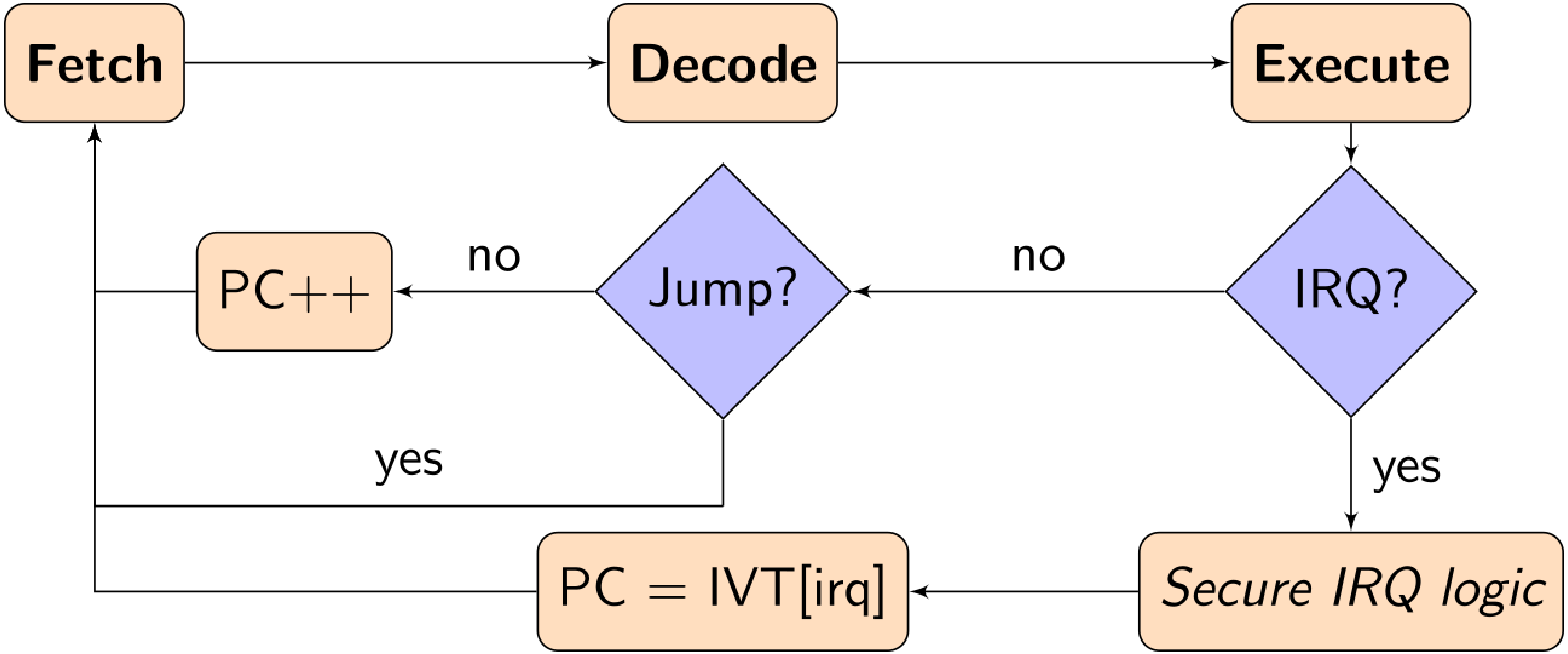
The extension: interruptible enclaves

- In Sancus, interrupts are disabled during the execution of an enclave
- This makes it impossible to protect against denial-of-service by a module
- Several authors have proposed secure ways to interrupt enclaves
  - Ruan De Clercq, Dries Schellekens, Frank Piessens, Ingrid Verbauwhede, **Secure Interrupts on Low-End Microcontrollers**, ASAP 2014
  - Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan, **TrustLite: a security architecture for tiny embedded devices**, EuroSys 2014

But all these proposals are vulnerable to side-channel attacks

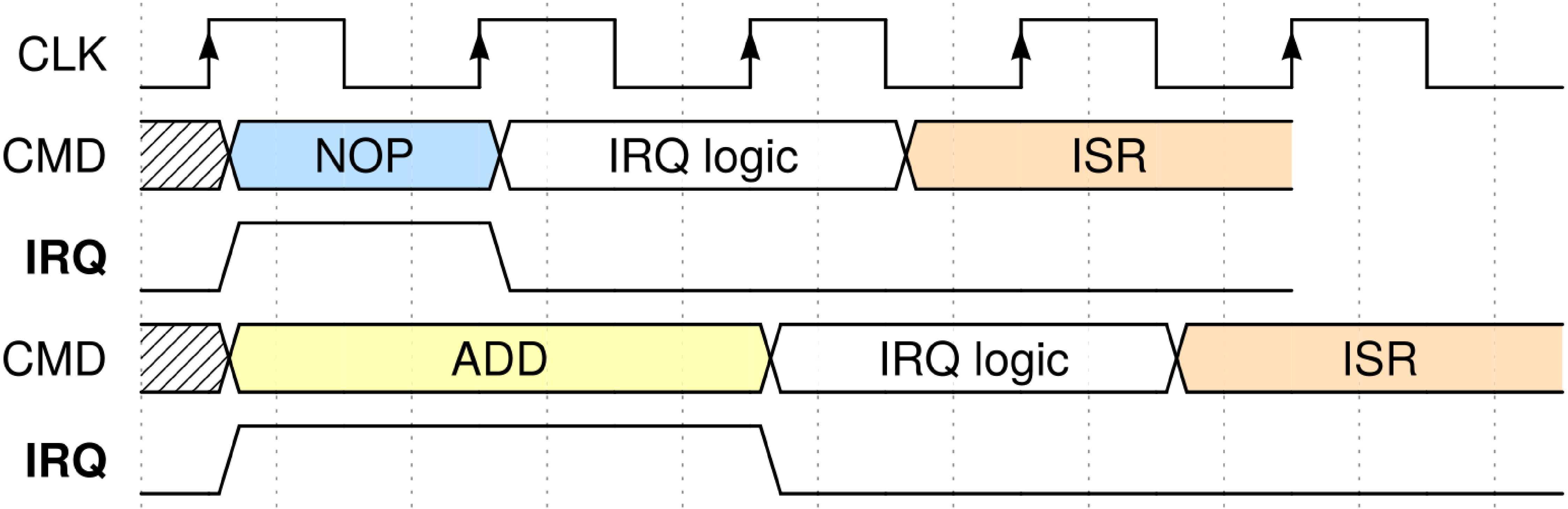
- Full discussion of the main attack:
  - Jo Van Bulck, Frank Piessens, Raoul Strackx, **Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic**, ACM CCS 2018
- Nemesis performs measurements on the micro-architectural state by measuring *interrupt latency*
  - On small embedded platforms, this can leak information on the instruction that was interrupted, and hence on control flow
    - Sancus, Trustlite, …
  - On large processors, this is an instruction-granular measurement of the CPU's micro-architectural state, where the instruction opcode is only one of many aspects that influence the latency
    - See the paper for details, including an attack against Intel SGX

The rudimentary CPU Interrupt logic …



… and how it leaks information

```
…
if secret {
  ADD @R5+,R6 // 2 cycles
}
else {
  NOP; NOP // 2 x 1 cycle
}
…
```

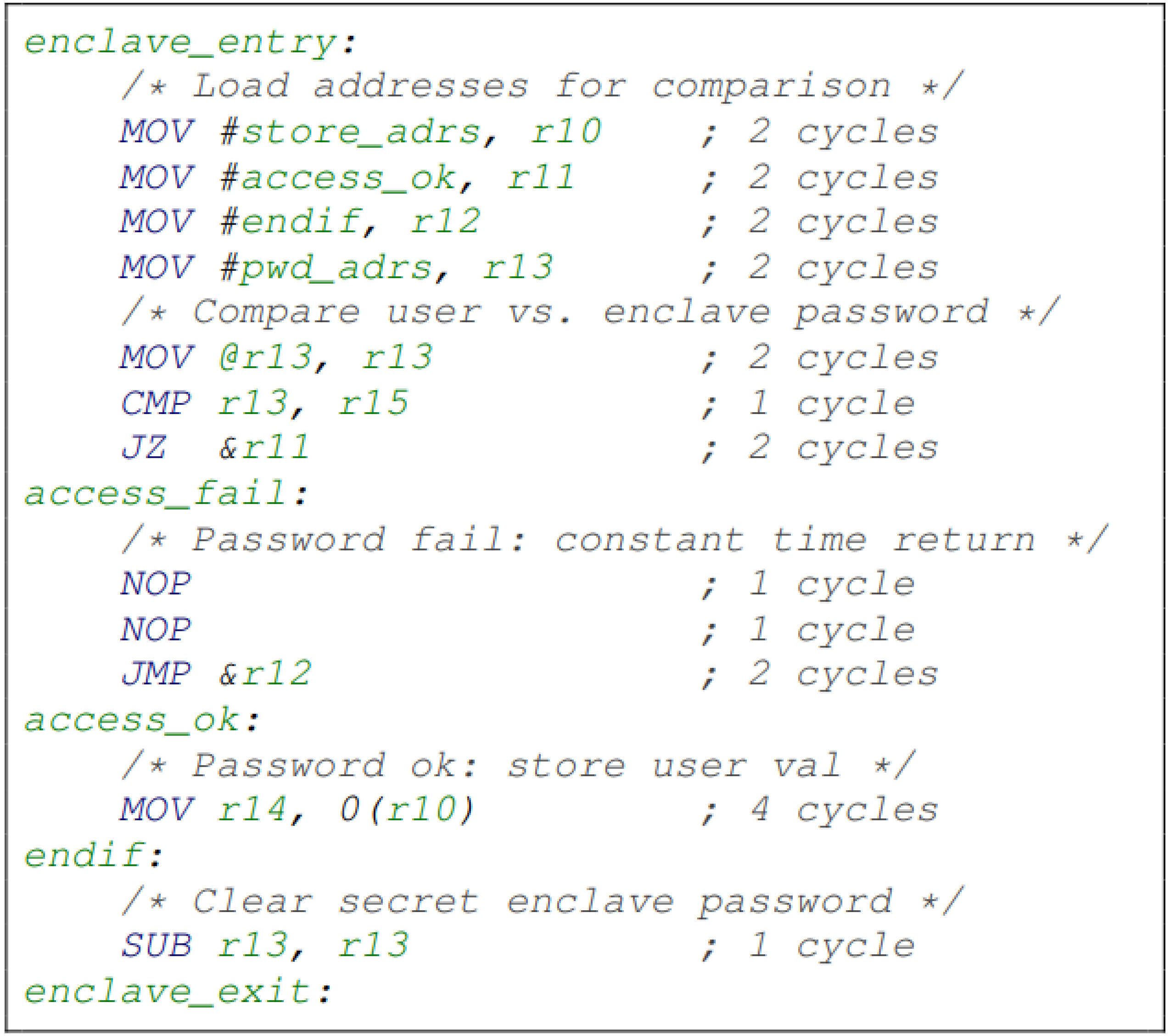


See the Nemesis paper for more information

- Case studies showing how to use this attack on Sancus to
  - Extract a password from a bootstrap loader
  - Extract a PIN from a secure keypad
- An extension of the attack to larger processors:
  - Where each interrupt latency measurement is an instruction-granular measurement of the micro-architectural state
  - A case study attacking privacy-sensitive data analytics in SGX

Example

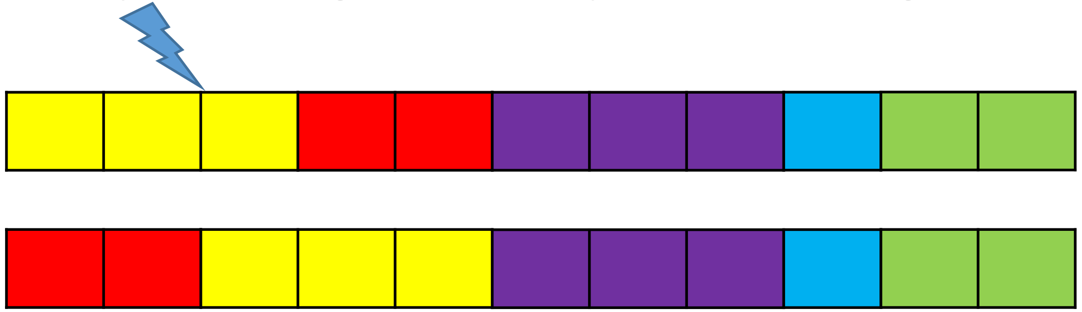
- Balanced enclave implementation becomes vulnerable again
    - Fail-branch: 1,1,2
    - Ok-branch: 4
- Hence: adding interrupts **weakens** isolation properties



```
enclave_entry:
    /* Load addresses for comparison */
    MOV #store_adrs, r10      ; 2 cycles
    MOV #access_ok, r11       ; 2 cycles
    MOV #endif, r12           ; 2 cycles
    MOV #pwd_adrs, r13        ; 2 cycles
    /* Compare user vs. enclave password */
    MOV @r13, r13             ; 2 cycles
    CMP r13, r15              ; 1 cycle
    JZ  &r11                  ; 2 cycles
access_fail:
    /* Password fail: constant time return */
    NOP                       ; 1 cycle
    NOP                       ; 1 cycle
    JMP &r12                  ; 2 cycles
access_ok:
    /* Password ok: store user val */
    MOV r14, 0(r10)           ; 4 cycles
endif:
    /* Clear secret enclave password */
    SUB r13, r13              ; 1 cycle
enclave_exit:
```

The attack is trickier than it seems

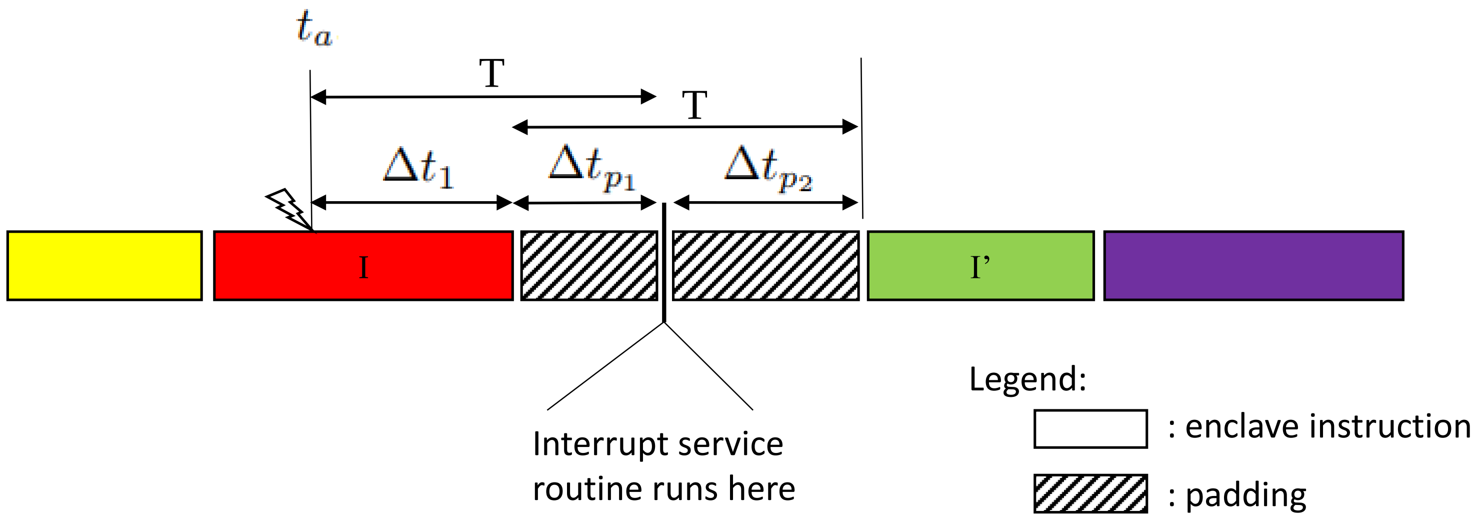
- Just padding interrupt handling time is not enough



- Three “measurements” to keep in mind:
  - Interrupt latency
  - Resume-to-end time
  - Interrupt counting

Nemesis-resistant Sancus

- Designing the “Secure IRQ logic” such that it is secure against Nemesis attacks:
  - Cycle accurate interrupt delivery
  - Pre- and post-padding such that:
    - Interrupt latency is constant (T)
    - Resume-to-end-time does not change on interrupt



Interrupt service routine runs here

Legend:

□ : enclave instruction

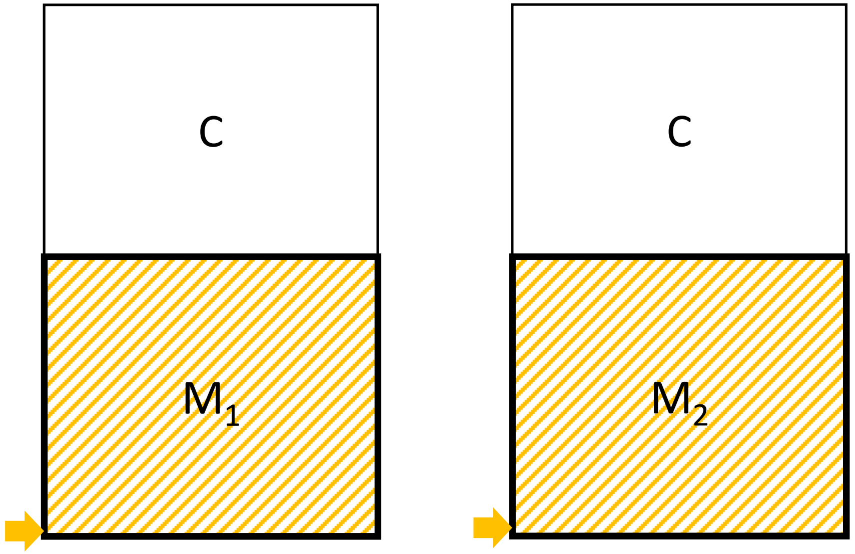
▨ : padding

Overview

- Introduction: hardware isolation mechanisms and micro-architectural attacks
- Enclaved execution: Sancus
- Extending Sancus with interrupts
- Formalization and security proof
- Implementation
- Conclusions

Formalizing the security objective

- Informally:
  - the interrupts extension does not introduce new information leaks
- We formalize this as a full abstraction property
  - What “leaks from a module” is defined by means of *contextual equivalence*
    - Modules M_1 and M_2 are contextually equivalent ($M_1 \approx M_2$) iff:
      - $\forall\, C{:}\, C[M_1]\downarrow \;\Leftrightarrow\; C[M_2]\downarrow$
    - $M_1 \approx M_2$ means:
      - Difference between them does not leak
  - Full abstraction is defined as the preservation (and reflection) of contextual equivalence before and after the extension



High-level overview of the proof

- Provide operational semantics for both versions of Sancus
- Reflection of contextual equivalence is trivial
- Preservation is proved by using a trace-semantics
  - Traces: $\beta ::= \bullet \mid \texttt{jmpIn?}(\mathcal{R}) \mid \texttt{jmpOut!}(\Delta t; \mathcal{R}).$
  - Structure of the proof:

$$\begin{tikzcd}
\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'} \arrow[d, Rightarrow, "(iii)"'] \arrow[dr, Rightarrow, "(ii)"] & \\
\mathcal{M}_M \simeq^{\mathbf{L}} \mathcal{M}_{M'} & \mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'} \arrow[l, Rightarrow, "(i)"]
\end{tikzcd}$$

Step (i) $\textit{If } \mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'} \textit{ then } \mathcal{M}_M \simeq^{\mathbf{L}} \mathcal{M}_{M'}.$

- Sufficient to prove: $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'} \implies (\forall C. C[\mathcal{M}_M]{\Downarrow^{\mathbf{L}}} \Rightarrow C[\mathcal{M}_{M'}]{\Downarrow^{\mathbf{L}}}).$
- Intuition behind the proof:
  - Consider the executions of $C[\mathcal{M}_M]$ and $C[\mathcal{M}_{M'}]$
  - They proceed in lockstep while in unprotected mode
  - On entry of protected mode:
    - By trace-equivalence they will either return the same result after the same time, or will both halt
    - The interrupts that will go off during protected execution are exactly the same

Step (ii): $\textit{If } \mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'} \textit{ then } \mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}.$

- Intuition behind the proof:
  - Find a trace β of M that M’ does not have
  - Find a trace β_{min} of M’ with a maximal common prefix
    - The first difference must be in a halt or jump-out action
  - Construct a context that generates β_{min} and turns the first difference into a difference in termination
    - This construction relies on the fact that we can use an arbitrarily complex device to help us construct calls to the protected module

Some surprising observations from doing the proof

- Several other “attacks” break contextual equivalence:
  - “Concurrency-like” issues:
    - If an enclave can read unprotected memory, interrupts break contextual equivalence
    - If an enclave can be “re-entered” on interrupt, this breaks contextual equivalence
  - Saving execution state:
    - Storing saved execution state of the module on an in-enclave stack breaks contextual equivalence
  - Manipulating interrupt enable bits within the enclave breaks contextual equivalence
- Handling corner cases is tricky:
  - What if a new interrupt arrives while still padding for the previous one?

Overview

- Introduction: hardware isolation mechanisms and micro-architectural attacks
- Enclaved execution: Sancus
- Extending Sancus with interrupts
- Formalization and security proof
- Implementation
- Conclusions

Implementation

- We have implemented our secure design as an extension of the current Sancus processor
  - Performance overhead is predictable and small
  - Area overhead is significant, mainly because of the need to back up registers on interrupt
    - Needed anyway to support other secure interrupt designs
    - Can be reduced by saving registers in memory

Overview

- Introduction: hardware isolation mechanisms and micro-architectural attacks
- Enclaved execution: Sancus
- Extending Sancus with interrupts
- Formalization and security proof
- Implementation
- Conclusions

Conclusions

- We propose an approach to give high-assurance arguments that an (architectural or micro-architectural) extension of a base system does not introduce new software exploitable side-channel leaks
  - For small deterministic systems, this appears to be a very strong guarantee
  - Scaling it to bigger or non-deterministic systems is a challenge for future work
- We have applied it to a significant case-study:
  - extending an embedded processor supporting enclaves with interrupts
- For bigger systems, we need to find ways to “factor” the problem in smaller sub problems