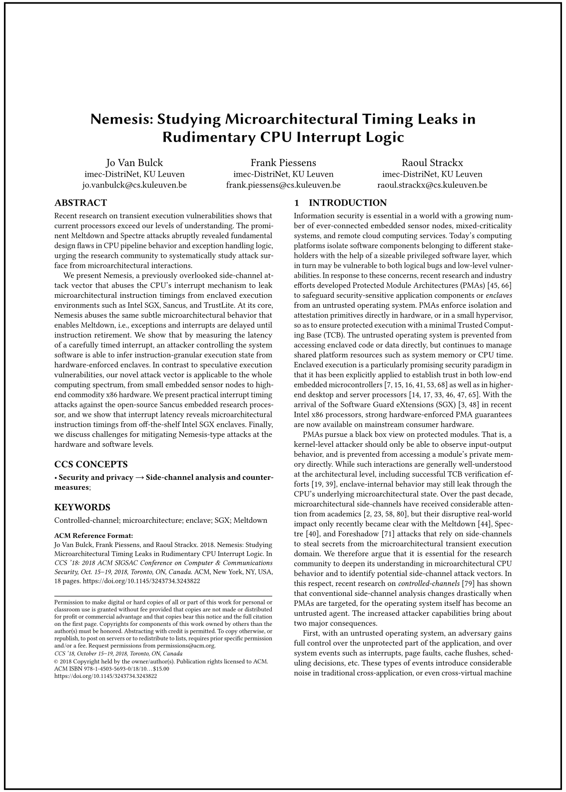
KU LEUVEN

Compiler-Assisted Hardening of Embedded Software Against Interrupt Latency Side-Channel Attacks

Hans Winderix, Jan Tobias Mühlberg, Frank Piessens

DistriNet

Nemesis, an Interrupt Latency Side-Channel Attack



Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic

Jo Van Bulck
imec-DistriNet, KU Leuven
jo.vanbulck@cs.kuleuven.be

Frank Piessens
imec-DistriNet, KU Leuven
frank.piessens@cs.kuleuven.be

Raoul Strackx
imec-DistriNet, KU Leuven
raoul.strackx@cs.kuleuven.be

ABSTRACT

Recent research on transient execution vulnerabilities shows that current processors exceed our levels of understanding. The prominent Meltdown and Spectre attacks abruptly revealed fundamental design flaws in CPU pipeline behavior and exception handling logic, urging the research community to systematically study attack surface from microarchitectural interactions.

We present Nemesis, a previously overlooked side-channel attack vector that abuses the CPU's interrupt mechanism to leak microarchitectural instruction timings from enclaved execution environments such as Intel SGX, Sancus, and TrustLite. At its core, Nemesis abuses the same subtle microarchitectural behavior that enables Meltdown, i.e., exceptions and interrupts are delayed until instruction retirement. We show that by measuring the latency of a carefully timed interrupt, an attacker controlling the system software is able to infer instruction-granular execution state from hardware-enforced enclaves. In contrast to speculative execution vulnerabilities, our novel attack vector is applicable to the whole computing spectrum, from small embedded sensor nodes to high-end commodity x86 hardware. We present practical interrupt timing attacks against the open-source Sancus embedded research processor, and we show that interrupt latency reveals microarchitectural instruction timings from off-the-shelf Intel SGX enclaves. Finally, we discuss challenges for mitigating Nemesis-type attacks at the hardware and software levels.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**;

KEYWORDS

Controlled-channel; microarchitecture; enclave; SGX; Meltdown

ACM Reference Format:
Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *CCS '18: 2018 ACM SIGSAC Conference on Computer & Communications Security, Oct. 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3243734.3243822

1 INTRODUCTION

Information security is essential in a world with a growing number of ever-connected embedded sensor nodes, mixed-criticality systems, and remote cloud computing services. Today's computing platforms isolate software components belonging to different stakeholders with the help of a sizeable privileged software layer, which in turn may be vulnerable to both logical bugs and low-level vulnerabilities. In response to these concerns, recent research and industry efforts developed Protected Module Architectures (PMAs) [45, 66] to safeguard security-sensitive application components or *enclaves* from an untrusted operating system. PMAs enforce isolation and attestation primitives directly in hardware, or in a small hypervisor, so as to ensure protected execution with a minimal Trusted Computing Base (TCB). The untrusted operating system is prevented from accessing enclaved code or data directly, but continues to manage shared platform resources such as system memory or CPU time. Enclaved execution is a particularly promising security paradigm in that it has been explicitly applied to establish trust in both low-end embedded microcontrollers [7, 15, 16, 41, 53, 68] as well as in higher-end desktop and server processors [14, 17, 33, 46, 47, 65]. With the arrival of the Software Guard eXtensions (SGX) [3, 48] in recent Intel x86 processors, strong hardware-enforced PMA guarantees are now available on mainstream consumer hardware.

PMAs pursue a black box view on protected modules. That is, a kernel-level attacker should only be able to observe input-output behavior, and is prevented from accessing a module's private memory directly. While such interactions are generally well-understood at the architectural level, including successful TCB verification efforts [19, 39], enclave-internal behavior may still leak through the CPU's underlying microarchitectural state. Over the past decade, microarchitectural side-channels have received considerable attention from academics [2, 23, 58, 80], but their disruptive real-world impact only recently became clear with the Meltdown [44], Spectre [40], and Foreshadow [71] attacks that rely on side-channels to steal secrets from the microarchitectural transient execution domain. We therefore argue that it is essential for the research community to deepen its understanding in microarchitectural CPU behavior and to identify potential side-channel attack vectors. In this respect, recent research on *controlled-channels* [79] has shown that conventional side-channel analysis changes drastically when PMAs are targeted, for the operating system itself has become an untrusted agent. The increased attacker capabilities bring about two major consequences.

First, with an untrusted operating system, an adversary gains full control over the unprotected part of the application, and over system events such as interrupts, page faults, cache flushes, scheduling decisions, etc. These types of events introduce considerable noise in traditional cross-application, or even cross-virtual machine

---

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243822

- Microarchitectural side-channel attack
  - Exploits fetch-decode-execute logic
  - Attacker measures interrupt latency

⇓

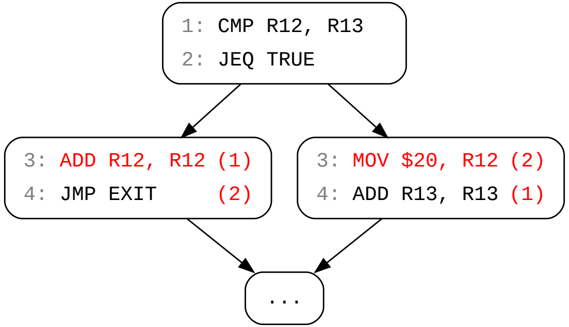
- **Reveals the latency trace of an execution**, i.e.
  - Machine instruction count
  - Timing of individual machine instructions

⇓

- Reveals which side of branch has been executed
  - Leaks info about branch predicate values

Nemesis - Illustrative Example

```
  CMP R12, R13
  JEQ TRUE
FALSE:
  ADD R12, R12   /* 1 cycle  */
  JMP EXIT       /* 2 cycles */
TRUE:
  MOV $20, R12   /* 2 cycles */
  ADD R13, R13   /* 1 cycle  */
EXIT:
  ...
```



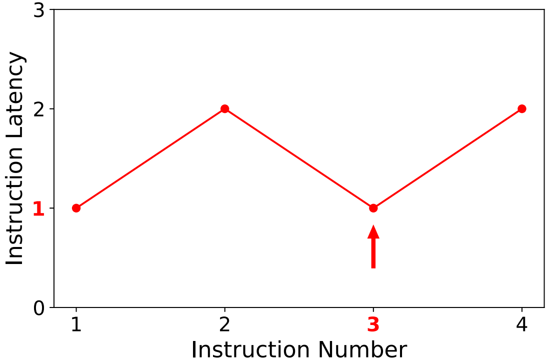
```
1: CMP R12, R13
2: JEQ TRUE
```

```
3: ADD R12, R12 (1)
4: JMP EXIT     (2)
```

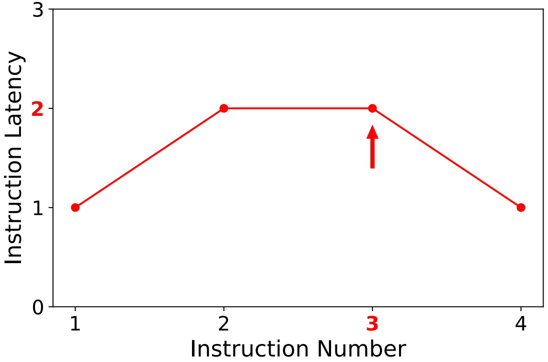
```
3: MOV $20, R12 (2)
4: ADD R13, R13 (1)
```

...

Latency Trace of False Path



Latency Trace of True Path



Constant-Time Programming Policy

- An established security policy
  - To protect against timing side-channel attacks
  - **No secret-dependent control-flow**
  - *No secret-dependent memory accesses*
  - *No secret-dependent instruction latencies*

No secret-dependent control-flow

⇓

No secret-dependent branch instructions

⇓

Effective protection against Nemesis

Constant-Time Programming Policy (Concerns)

- Strict rules with **a status of absoluteness**
- Typically **manually implemented** at the highest abstraction level (**source code**)
    - Harms **readability** and **maintainability**
        - Prevents using familiar programming constructs
        - Developer must use obscure tricks to deceive compiler (*brittle*)
    - No **separation of concerns** (harms **portability**)
        - Tight coupling between security policy and source code
    - **Policy must be honored by the compiler early on** (*brittle*)
        - Optimiser cannot introduce secret-dependent constructs
        - Compiler cannot introduce secret-dependent constructs when lowering abstractions
- **Performance** impact

```
  CMP R12, R13
  JEQ TRUE
FALSE:
  ADD R12, R12    /* 1 cycle  */
  JMP EXIT        /* 2 cycles */
TRUE:
  MOV $20, R12    /* 2 cycles */
  ADD R13, R13    /* 1 cycle  */
```

```
  /* Hardened program (balanced branch) */
  CMP R12, R13    /* 1 cycle , 1 byte  */
  JEQ TRUE        /* 2 cycles, 1 byte  */
FALSE:
  ADD R12, R12    /* 1 cycle , 1 byte  */
  JMP EXIT        /* 2 cycles, 1 byte  */
TRUE:
  ADD R13, R13    /* 1 cycle , 1 byte  */
  MOV $20, R12    /* 2 cycles, 2 bytes */
```

	Time (cycles)	Size (bytes)
Balanced	6	7
Eliminated	22	22

```
/* Hardened program (eliminated branch) */
CMP R12, R13  /* 1 cycle, 1 byte
MOV R2 , R10  /* 1 cycle, 1 byte  R2 is status register
RRA R10       /* 1 cycle, 1 byte
AND $1 , R10  /* 1 cycle, 1 byte  extract Z bit
ADD $-1, R10  /* 1 cycle, 1 byte  store TRUE mask
MOV R10, R11  /* 1 cycle, 1 byte
XOR $-1, R11  /* 1 cycle, 1 byte  store FALSE mask
MOV R12, R9   /* 1 cycle, 1 byte  store original value
AND R10, R9   /* 1 cycle, 1 byte  apply TRUE mask
ADD R12, R12  /* 1 cycle, 1 byte  actual computation
AND R11, R12  /* 1 cycle, 1 byte  apply FALSE mask
BIS R13, R12  /* 1 cycle, 1 byte  conditional select
MOV R12, R9   /* 1 cycle, 1 byte  store original value
AND R11, R9   /* 1 cycle, 1 byte  apply FALSE mask
MOV $20, R12  /* 1 cycle, 1 byte  actual computation
AND R11, R13  /* 1 cycle, 1 byte  apply TRUE mask
BIS R9 , R12  /* 1 cycle, 1 byte  conditional select
MOV R13, R9   /* 1 cycle, 1 byte  store original value
AND R11, R9   /* 1 cycle, 1 byte  apply FALSE mask
ADD R13, R13  /* 1 cycle, 1 byte  actual computation
AND R11, R13  /* 1 cycle, 1 byte  apply TRUE mask
```

Research Hypothesis

- The constant-time programming policy is not absolute
- Relaxing the constant-time rules can be secure (*depends on leakage model*)
- Relaxing the constant-time rules can produce more performant programs
- Balancing branches is an effective countermeasure against timing attacks on some low-end processors

Objectives

- Decouple security policy from source code
- Automate program hardening
- Make latency trace secret-independent
- Balance secret-dependent branches (*instead of eliminating them*)
- Less overhead (*compared to eliminating branches*)

Assumptions

- **Attacker model**
  - Access to cycle-accurate clock
  - Ability to precisely schedule and handle interrupts
  - Attacker can interrupt victim code running in another protection domain
- **System model**
  - Interrupts are handled upon instruction retirement
  - Execution environment leaks latency trace of execution
  - A *dummy instruction* can be constructed for every latency class

Dummy instruction

An instruction without observable effects besides its time to execute

The Defense

A Recursive Control-Flow Graph Algorithm

- **Phase 1 - Static analysis**
  - Taint analysis, loop analysis

- **Phase 2 - Program hardening**
  - Balance secret-dependent branches according to their latency trace

    ⇒ Insert dummy instructions if latencies don't match
  - Three operations
    1. Equalise path lengths
    2. Compute level structure
    3. Equalise execution times (level-wise)

Implementation

- LLVM compiler infrastructure
    - *MachineFunction* pass
    - MSP430 backend

Evaluation

- Platform = openMSP430 + Sancus TEE extensions
- Benchmark suite, consisting of
    - Synthetic programs
    - Third-party programs

https://github.com/hanswinderix/sllvm

Experimental Results (openMSP430)

Benchmark	**Vulnerable Baseline**		**Balancing Overhead**		**Elimination Overhead**	
	Code size (bytes)	Exec time (cycles)	Code size	Execution time	Code size	Execution time
call	302	112	1.09x	1.05x	-	-
diamond	284	103	1.16x	1.12x	-	-
fork	264	91	1.06x	1.05x	-	-
ifcompound	384	372	1.06x	1.02x	-	-
ifthenloop	284	143	1.27x	1.19x	-	-
ifthenloopif	342	179	1.38x	1.60x	-	-
ifthenlooploop	308	378	1.54x	1.36x	-	-
ifthenlooplooptail	350	387	1.63x	1.25x	-	-
indirect	274	97	1.18x	1.16x	-	-
loop	400	2841	1.06x	1.02x	-	-
multifork	290	100	1.19x	1.09x	-	-
triangle	266	94	1.09x	1.06x	-	-
bsl	394	984	1.12x	1.20x	1.27x	1.47x
keypad	672	1119	1.28x	1.56x	1.24x	1.81x
kruskal	634	2460	1.14x	1.08x	1.16x	1.24x
modexp2	702	23537	1.05x	1.31x	1.05x	1.32x
mulhi3	416	904	1.37x	1.59x	1.34x	2.01x
mulmod8	482	425	1.49x	1.07x	1.40x	1.36x
sharevalue	480	3398	1.06x	1.04x	1.05x	1.07x
switch16	402	115	1.41x	1.09x	2.29x	4.65x
switch8	402	115	1.41x	1.09x	2.29x	4.65x
twofish	8872	92745	1.06x	1.00x	1.02x	1.02x
Geometric Mean			**1.23x**	**1.19x**	**1.35x**	**1.76x**