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

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

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

side-channels. Noise is traditionally compensated for with statistical analysis over data acquired from multiple runs of the victim program. In a controlled-channel setting on the other hand, one prevailing research line is exploring the possibility of *amplifying* conventional side-channels so as to extract sensitive information in a single run, with limited noise. Recent work on Intel SGX platforms has practically demonstrated such side-channel amplification for the usual suspects: CPU caches [8, 25, 31, 51, 62] and branch prediction machinery [18, 42]. These results have prompted Intel to release an official statement, arguing that "in general, these research papers do not demonstrate anything new or unexpected" [38].

A second, more profound consequence of the PMA attacker model, however, is the emergence of an entirely new class of side-channels that were never considered relevant before. To date only page table-based attacks [64, 75, 77, 79] have been identified as one such innovative controlled-channel for high-end MMU-based architectures. By carefully revoking access rights on protected memory pages and observing the associated page accesses, an adversarial operating system is able to extract large amounts of sensitive data (cryptographic keys, full text, and images) from SGX enclaves. Several authors [13, 20, 43, 67, 70, 74] have since expressed their concerns on controlled-channel vulnerabilities in a PMA setting. An important research question therefore is to determine which novel controlled-channels exist, and to what extent they endanger the PMA protection model.

This paper contributes to answering this question. We present an innovative class of Nemesis[1] controlled-channel attacks that exploit subtle timing differences in the rudimentary fetch-decode-execute operation of programmable instruction set processors. We abuse the key microarchitectural property that hardware interrupts/faults are only served upon instruction retirement, *after* the currently executing instruction has completed, which can take a variable amount of CPU cycles depending on the instruction type and the microarchitectural state of the processor. Where Meltdown-type "fault latency" attacks [44, 71] exploit this time window in modern out-of-order processors to transiently leak unauthorized memory through a microarchitectural covert channel, Nemesis-type interrupt latency attacks abuse a more fundamental observation that equally affects non-pipelined processors. Namely, that delaying interrupt handling until instruction retirement introduces a subtle timing difference that *by itself* reveals side-channel information about the interrupted instruction and the microarchitectural state when the interrupt request arrived. Intuitively, an untrusted operating system can exploit this timing measurement when interrupting enclaved instructions to differentiate between secret-dependent program branches, or to extract information for different side-channel analyses (e.g., trace-driven cache [1], address translation [75], or false dependency [50] timing attacks).

We are the first to recognize the threat caused by instruction set architectures with variable interrupt latency. Previous PMA research has overlooked this subtle attack vector, claiming for instance that "timing of external interrupts does not depend on secrets within compartments, and does not leak confidential information" [20]. We show that Nemesis attacks affect a wide range of security architectures, covering the whole computing spectrum. In this, we are the first to identify a remotely exploitable microarchitectural side-channel vulnerability that is *both* applicable to embedded, as well as higher-end enclaved execution environments.

Summarized, the main contributions of this paper are:

- We leverage interrupt latency as a novel, non-conventional side-channel to extract information from enclaved applications, thereby advancing microarchitectural understanding.
- We present the first controlled-channel attack vector for embedded enclaved execution processors, and extract full application secrets in practical Sancus attack scenarios.
- We provide clear evidence that interrupt latency reveals microarchitectural instruction timings on modern Intel SGX processors, and illustrate Nemesis's increased instruction-granular potential in macrobenchmark evaluation scenarios.
- We explain how naive hardware-level defense strategies cannot defend against advanced Nemesis-style interrupt attack variants, demonstrating the consequential impact of our findings for provably side-channel resistant processors.

Our attack framework and evaluation scenarios are available as free software at https://github.com/jovanbulck/nemesis.

2 BACKGROUND AND BASIC ATTACK

We first refine the threat model and the class of security architectures affected by our side-channel. Next, we explain how interrupt latency can be leveraged in ideal conditions to extract sensitive data from secure enclaves.

2.1 Attacker Model and Assumptions

The adversary's goal is to derive information regarding the internal state of an enclaved application. In this respect, trusted computing solutions including Intel SGX have been explicitly put forward to protect sensitive computations on an untrusted attacker-owned platform, both in an untrustworthy cloud environment [6, 61], as well as to enforce enterprise right management on consumer hardware [32, 56]. Analogous to previous enclaved execution attacks [30, 42, 75, 79], we therefore consider an adversary with (*i*) access to the (compiled) source code of the victim application, and (*ii*) full control over the Operating System (OS) and unprotected application parts. This means she can modify BIOS options, load kernel drivers, configure hardware devices such as timers, and control scheduling decisions. Note that although PMAs can be leveraged [26, 61] to protect the confidentiality of sensitive code, this is not the default case in the security architectures analyzed in this work and for many of the PMA use cases [6, 32, 66].

At the architectural level, we assume the untrusted OS can securely interrupt and resume enclaves. Such interruptible isolated execution is supported by a wide range of mature embedded [7, 15, 41] as well as higher-end [14, 17, 33, 48, 65] PMAs that employ a trusted security monitor to preserve the confidentiality and integrity of a module's internal state in the presence of asynchronous interrupt events. In this paper we focus exclusively on hardware-level security monitors, but our timing channel may also be relevant for architectures where enclave interruption proceeds through a small trusted software layer [7, 14, 19, 33]. We assume that enclaves can be interrupted repeatedly within the same run, and for the

[1] From the ancient Greek goddess of retribution who inevitably intervenes to balance out good and evil; an inescapable agent much like a pending interrupt request.

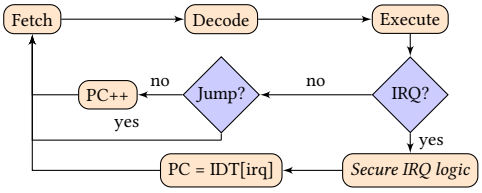


Figure 1: A processor fetches, decodes, and executes the instruction referred by the Program Counter (PC) register.

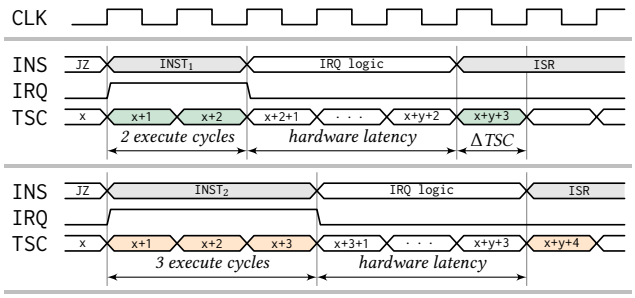


Figure 2: Interrupt latency leaks information about the instruction that was executing at the time of IRQ arrival.

Intel SGX application scenarios, can be made to process the same secret-dependent input repeatedly over multiple invocations.

Importantly, in contrast to previous controlled-channel attacks referenced above, our attack vector does *not* necessarily require advanced microarchitectural CPU features, such as paging, caching, branch prediction, or out-of-order execution. Instead, Nemesis-type interrupt timing attacks only assume a generic stored program computer with a multi-cycle instruction set, where each individual instruction is uninterruptible (i.e., executes to completion). This is the most widespread case for major embedded (e.g., TI MSP430, Atmel AVR) as well as higher-end (e.g., x86, openRISC, RISC-V) instruction set architectures.

2.2 Fetch-Decode-Execute Operation

Figure 1 summarizes the basic operational process of a CPU, traditionally referred to as the *fetch-decode-execute* operation. A dedicated Program Counter (PC) register holds the address of the next instruction to fetch from memory. PC is automatically incremented after every instruction in the program, and can be explicitly changed by means of jump instructions. Hardware devices furthermore have the ability to halt execution of the current program by means of Interrupt Requests (IRQs) that notify the processor of some asynchronous external event that requires immediate attention. Whenever the current instruction has completed, before fetching the next one, the processor checks if there are IRQs pending. If so, the PC is loaded from a predetermined location in the Interrupt Descriptor Table (IDT) that holds the address of the corresponding Interrupt Service Routine (ISR). Typical processor architectures only take care of storing the minimal execution context (e.g., PC and status register) before vectoring to the ISR. The trusted OS interrupt handling code then stores any remaining CPU registers as needed. However, when interrupting a protected module, the PMA hardware is responsible to securely store and clear *all* CPU registers, which is abstracted in the "secure IRQ logic" block of Fig. 1.

While the simplified fetch-decode-execute description above is representative for a class of low-end CPUs such as the TI MSP430 [69], optimizations found in modern higher-end processors considerably increase the complexity. A pipelined architecture improves throughput by parallelizing the fetch-decode-execute stages of subsequent instructions. In case of a complex instruction set such as Intel x86 [13, 36], individual instructions are first split into smaller *micro-ops* during the decode stage. Thereafter, an out-of-order engine schedules the micro-ops to available execution units, which may be duplicated to further increase parallelism. To minimize pipeline stalls from program branches, the processor will try to predict the outcome of conditional jumps. Simultaneous multithreading technology can interleave the execution of multiple independent instruction streams on the same physical CPU core to maximize the use of available execution units. Repeated memory accesses are furthermore sped up by means of an intricate cache hierarchy for among others micro-ops, instructions, data, and address translation. However, despite all these optimizations, Intel [36] confirms that the basic property remains that "all interrupts are guaranteed to be taken on an instruction boundary [...] located during the retirement phase of instruction execution".

2.3 Basic Nemesis Attack

We consider processors that serve interrupts after the execute stage has completed,[2] which can take multiple clock cycles depending on the microarchitectural behavior of the instruction. Our attacks are based on the key observation that *an IRQ during a multi-cycle instruction increases the interrupt latency with the number of cycles left to execute* – where interrupt latency is defined as the number of clock cycles between arrival of the hardware IRQ and execution of the first instruction in the software ISR. When interrupt arrival time is known (e.g., generated by a timer), untrusted system software can infer the duration of the interrupted instruction from a timestamp obtained on ISR entry.

Figure 2 illustrates our basic attack for an enclaved execution that branches on a secret. After the conditional jump *jz* in the victim enclave, either the two-cycle instruction $inst_1$ or the three-cycle instruction $inst_2$ is executed. In an ideal environment, a kernel-level attacker proceeds as follows to determine private control flow. First, before executing the enclave, a cycle-accurate timer is configured to schedule an IRQ at the beginning of the first clock cycle $x + 1$ after the conditional jump instruction. Next, the enclave is entered and the timer fires, interrupting either $inst_1$ or $inst_2$. After instruction completion, the secure hardware stores and clears protected execution state, and hands over control to the untrusted interrupt handler code. Here, the adversary compares the value of a timestamp counter with the known IRQ arrival time to yield a timing difference of one clock cycle, depending on whether the conditional jump in the enclaved execution was taken or not.

The above scenario is a clear example of how an untrusted OS can leverage interrupt latency to break the black box view on

[2] While not the focus of this paper, there are also issues with cancelling the currently executing instruction upon IRQ arrival, as outlined in Section 6.

protected modules. In line with previous enclaved execution attacks [8, 42, 75, 79], Nemesis-type interrupt timing attacks exploit secret-dependent control flow. Specifically, we require a different execution time for at least one instruction in the if/else branch. The adversary furthermore relies on (*i*) a timer device capable of generating cycle-accurate IRQs, and (*ii*) a Time Stamp Counter (TSC) peripheral that is incremented every CPU cycle. The main difficulty for a successful attack lies in determining a suitable timer value so as to interrupt the instruction of interest. This is non-trivial in that it requires one to predict the duration between the moment the timer is configured and the desired interruption point. For reasons pointed out above, it is challenging to precisely predict the execution time of an instruction stream on modern processors. We present our approach to configuring the timer and dealing with noise in Section 4.

Note that IRQ latency measurements capture an instruction-granular measurement of the CPU's microarchitectural state, such that the instruction opcode ($inst_1$ vs. $inst_2$) is only one of many properties that influence latency on modern processors. We will show in Section 5 that Nemesis adversaries can also distinguish instructions based on for instance CPU caching behavior, address translation, or data operand dependencies.

3 CASE STUDY PLATFORMS AND ATTACKS

We implemented and evaluated Nemesis-type interrupt timing attacks for both a representative embedded, as well as for an off-the-shelf higher-end enclaved execution processor. To illustrate the wide applicability of conditional control flow side-channel attacks, beyond common cryptographic key extraction [25, 51, 62, 64, 75], we follow a line of enclaved execution attacks [8, 31, 42, 74, 79] that target non-cryptographic case study applications. Such applications cannot be hardened straightforwardly using vetted crypto libraries, as secrets are generally non-trivial to identify and conditional control flow is more prevalent plus harder to eliminate.

3.1 Sancus and Embedded PMAs

Given the rise of tiny embedded devices in recent years, a new line of research [7, 16, 41, 53, 68] employs a lightweight program counter based memory isolation technique to secure small microcontrollers that lack hardware support for established security measures, such as virtual memory and processor privilege levels. The Sancus [53, 55] research prototype extends the memory access logic and instruction set of a low-end TI MSP430 microcontroller to allow the creation, authentication, and destruction of enclaved software modules with a hardware-only TCB. Furthermore, enclaves residing on the same device can securely link to each other using caller and callee authentication primitives. A dedicated LLVM-based C compiler hides low-level concerns such as secure linking, inter-module calling conventions, and private call stack switching by inserting short assembly code stubs to be executed whenever an enclave is entered or exited. Finally, recent research [54, 72] has shown that, in contrast to Intel SGX platforms, Sancus' memory isolation primitive can also be used to provide software enclaves with exclusive access to Memory-Mapped I/O (MMIO) hardware peripheral devices. However, since Sancus enclaves only feature a single contiguous private data section, secure I/O on Sancus requires the use of a small driver module entirely written in assembly code, using only registers for data storage.

The original Sancus architecture presumes uninterruptible isolated execution. Secure interruption of hardware-enforced embedded software modules was pioneered by the TrustLite [41] PMA. More specifically, TrustLite modifies the processor to push all CPU registers onto the private call stack of the interrupted module, before clearing them and vectoring to the untrusted ISR. Subsequent research [15] has since implemented a comparable hardware-level interrupt mechanism for a prototypic Sancus-like PMA with a single secure domain, and recent work-in-progress [73] reports on hardware and compiler support for fully interruptible and reentrant Sancus enclaves. For the work presented in this paper, we have implemented TrustLite's secure interrupt mechanism as an extension to the original Sancus architecture. Furthermore, we extended the compiler-generated entry stubs to restore private execution context on the next invocation of a previously interrupted enclave.

We selected Sancus as the case study architecture representative for the lowest end of the computing spectrum with strict security requirements for mutually distrusting stakeholders. A recent exhaustive PMA overview [45] indicates that Sancus is the only embedded architecture with a fully open-source[3] hardware design and tool chain, which allowed us to develop the secure interrupt extensions. In contrast to modern SGX processors, Sancus' openMSP430-based implementation embodies an elementary programmable microcontroller without advanced architectural features such as paging, caches, or out-of-order instruction pipelining. Given the simplistic design of the security extensions, as well as the underlying processor, the existence of remotely exploitable side-channels was considered rather unlikely by the original designers [52, §7.5.3]. To the best of our knowledge, we present the first controlled-channel attack vector for embedded enclaved execution processors.

3.1.1 Bootstrap Loader. We illustrate the applicability of our basic attack with a code snippet from an actual password comparison routine in Texas Instruments' MSP430 serial Bootstrap Loader (BSL) implementation. The BSL software is executed on platform reset, and enables remote, in-field firmware updates. To enforce that only legitimate device owners can reprogram the microcontroller, sensitive BSL commands are protected with a 32-byte password. Our first Sancus application scenario employs hardware-enforced isolation to shield critical BSL password-protected functionality from untrusted embedded firmware.

```
    cmp.b @r6+, r12          cmp.b @r6+, r12
    jz    1f                 jz    1f
    bis   #0x40, r11         bis   #0x40, r11
1:  ...                      jmp   2f
                           1: nop nop nop nop 2: ...
```

Listing 1: (Un)balanced BSL password comparison.

However, the password comparison routine in some BSL versions is known to be vulnerable to an execution timing attack [24]. The left hand side of Listing 1 provides the original, actually used

[3] https://distrinet.cs.kuleuven.be/software/sancus and https://github.com/sancus-pma

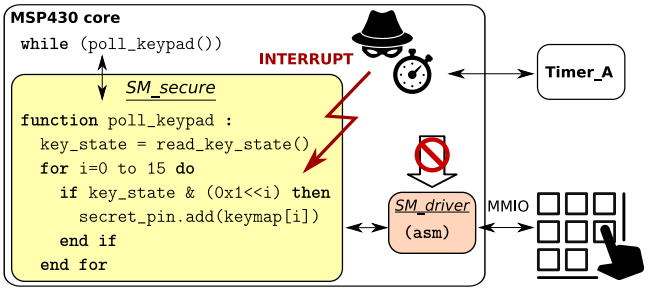


Figure 3: Secure keypad Sancus application scenario.

assembly code.[4] For clarity we only show the body of the password comparison loop, where the byte pointed to by `r6` is compared with the value in `r12`, and a bit in `r11` is set to invalidate access when the comparison fails. Observe that the code is unbalanced in that the two-cycle `bis` (bit-set) instruction is only executed for incorrect password bytes. Hence, an adversary can determine the correctness of individual bytes by observing the program's overall execution time. We close this vulnerability in the hardened version on the right by balancing the else branch with no-op compensation code, as previously suggested in literature [11, 60].

We show that, even when executing the balanced password comparison routine in a Sancus enclave, untrusted system software can still learn the correctness of individual password bytes by carefully timing interrupts. More specifically, an IRQ arriving in the first clock cycle after the conditional jump instruction, will either interrupt the two-cycle `bis` instruction or the single-cycle `nop` instruction. Hence, depending on the secret password byte, an IRQ latency difference of one clock cycle will be observed. The hardened routine thus properly closes the timing side-channel at the architectural assembly code level, but unknowingly introduces a new one at the microarchitectural level. As such, our elementary BSL case study serves as a clear demonstration of the additional attack surface induced by secure interrupts, where adversaries are no longer restricted to start-to-end timing measurements of the enclaved computation.

3.1.2 Secure Keypad. Various authors [16, 41, 53, 54, 72] have suggested the use of small PMAs to securely interface embedded platforms with peripheral I/O devices. Our second Sancus application scenario leverages secure I/O to guarantee the secrecy of a 4-digit PIN code towards an untrusted embedded operating system.

Figure 3 summarizes the core idea, where the security-sensitive application logic is implemented in a protected SM_{sec} enclave that securely links to a dedicated SM_{drv} assembly enclave to gain exclusive access to the MMIO region of the keypad peripheral, as explained above. The untrusted OS can only interact with the keypad indirectly, through the public interface offered by SM_{sec}. A single entry point `poll_keypad` fetches the current key state from the driver enclave, and processes each bit sequentially. The 16-bit key state indicates which keys are down, and a static lookup table is used to translate key numbers to the corresponding characters. This is similar to a reference implementation for an unprotected MSP430 keypad application by Texas Instruments [49]. To increase readability, the pseudo code in Fig. 3 omits practical concerns such as detecting key releases and limiting the length of the PIN code. We refer the interested reader to Appendix B for the full implementation, derived from a recently published open-source Sancus automotive application case study [72].

The keypad has to be polled regularly to detect key presses. For this, our application scenario relies on the untrusted operating system for availability of the CPU time resource. Since the OS is in control of scheduling decisions, it is allowed to interrupt SM_{sec} at all times.[5] Our attack exploits key state dependent control flow in the `poll_keypad` function. Appendix B provides the full compiler-generated assembly code, but it suffices to say that the conditional code path consists of two single-cycle instructions followed by either a single-cycle `tst` or a two-cycle `cmp` instruction. If we succeed in timing an IRQ two cycles after the conditional jump, we will thus observe a difference in interrupt latency of one clock cycle, depending on whether the private key state bit was set or not. Reconfiguring the timer to repeat the attack in each loop iteration allows an untrusted ISR to unambiguously determine which keys were pressed in a single run of SM_{sec}.

3.2 Intel Software Guard eXtensions

Recent Intel x86 processors include Software Guard eXtensions (SGX) [3, 48] that enable isolated execution of security-critical code in hardware-enforced enclaves, embedded in the virtual address space of a conventional OS process. SGX reduces the TCB to the point where a remote software provider solely has to trust the implementation of her own enclave, plus the underlying processor. Enclave code is restricted to user space (ring 3), and has access to all its protected pages, as well as to the unprotected part of the host application's address space. Dedicated CPU instructions switch the processor in and out of *enclave mode*, where hardware-level access control logic verifies the output of the untrusted address translation process to safeguard enclaved pages from outside accesses. The EENTER instruction transfers control from the unprotected application context to a predetermined location inside the enclave, and EEXIT leaves an enclave programmatically. Alternatively, in case of a fault or external interrupt, the processor executes an Asynchronous Enclave Exit (AEX) procedure that saves the execution context securely in a preallocated state save area inside the enclave, and replaces the CPU registers with a synthetic state to avoid direct information leakage to the untrusted ISR. The AEX procedure also takes care of pushing a predetermined Asynchronous Exit Pointer (AEP) on the unprotected call stack, so as to allow the OS interrupt handler to return transparently to unprotected trampoline code outside the enclave. From this point, a previously interrupted enclave can be continued by means of the ERESUME instruction.

Intel SGX serves as our case study architecture for higher-end enclaved execution platforms. A modern SGX-enabled CPU implements the complex x86 instruction set architecture, and includes all advanced microarchitectural features found in modern processors.

[4] Assembly code snippet from BSL v2.12, as published by [24].

[5] Note that Sancus' secure IRQ logic stores execution state in the protected data section of the interrupted enclave. For MMIO driver enclaves without general purpose private data region, our hardware mechanism clears registers without saving them.

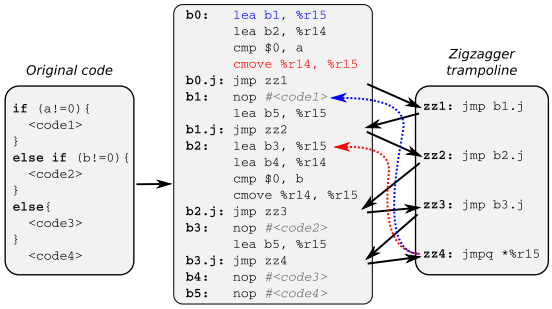


Figure 4: Zigzagger branch obfuscation example (from [42]).

3.2.1 Zigzagger Branch Obfuscation. Recent research on branch shadowing attacks [42] showed that enclaved control flow can be inferred by probing the CPU-internal Branch Target Buffer (BTB) after interrupting a victim enclave. Given the prevalence of conditional control flow in existing non-cryptographic applications, this work also includes a practical compile-time hardening scheme called Zigzagger. Figure 4 shows how secret-dependent program branches are translated into an oblivious CMOV instruction, followed by a tight trampoline sequence of unconditional jumps that ends with a single indirect branch instruction. The key idea behind Zigzagger is to prohibit probing the BTB for the current branch instruction by rapidly jumping back and forth between the instrumented code and the trampoline such that recognizing the current instruction pointer becomes difficult. It has since been shown, however, that Zigzagger-instrumented code can be reliably interrupted one instruction at a time [74], and concurrent research defeated Zigzagger in restricted circumstances through a segmentation-based side-channel [30].

We will show that even the contained conditional control flow in Zigzagger-hardened code exhibits definite instruction timing differences that can be recognized to extract application secrets from IRQ latency traces. Particularly, to emphasize Nemesis's increased precision over state-of-the-art SGX attacks, we aligned the assembly code of Fig. 4 to fit entirely within one cache line, such that execution paths cannot be distinguished by their corresponding code cache or page access profiles [64]. Our Zigzagger attack scenario thus illustrates that Nemesis-type interrupt latency attacks leak microarchitectural timing information at the granularity of *individual* instructions, whereas previous controlled-channels only expose enclaved memory accesses at a relatively coarse-grained 4 KiB page [75, 79] or 64-byte cache line [31, 62] granularity.

3.2.2 Binary Search. Intel SGX technology has been explicitly put forward for securely offloading privacy-sensitive data analytics to an untrusted cloud environment [61]. Our second SGX application scenario considers enclaves that look up secret values in a known dataset, as it occurs for instance in privacy-friendly contact discovery [57] or DNA sequence processing [8, 76]. In case of the former, the enclave is provided with a known large list of users, plus an encrypted smaller list of secret contacts, and is requested to return only those contacts that occur in the known user list. In case of the latter, the enclave may lookup values in a public reference human genome dataset, based on an encrypted secret input tied to an individual. In both scenarios, adversaries may track control flow decisions made for instance by the widely used binary search algorithm to learn (parts of) the secret input. In this respect, binary search serves as a particularly relevant example for the difficulty of eliminating conditional control flow in general-purpose enclave programs. The obvious alternative at the application level, an exhaustive scan of the public data, would increase the time complexity from a logarithmic to a linear effort.

```
for (lim = nmemb; lim != 0; lim >>= 1) {
    p = base + (lim >> 1) * size;
    cmp = (*compare)(key, p);
    if (cmp == 0) return p;
    if (cmp > 0) {   /* key > p: move right */
        base = p + size; lim--;
    } /* else move left */
}
```

Listing 2: Binary search routine in Intel SGX Linux SDK.

Listing 2 shows the relevant part of the actual binary search routine provided by the official Intel SGX Linux SDK. We refer to Appendix C for the complete unmodified source code, plus a disassembly of the compiled version. The implementation looks up a provided key in the sorted array between `base` and `lim` by repeatedly comparing it to the middle value. If the provided key was found, the function returns. Otherwise, the values of `base` and `limit` are adjusted according to whether the provided key was greater or smaller than the middle value. We will show that the assembly code paths corresponding to whether the algorithm took the left, right, or equal branch, manifest subtle yet distinct instruction latency patterns which are revealed in the extracted IRQ latency traces. As with the Zigzagger example above, the secret lookup key is learned even when the array fits entirely within a single cache line. For larger arrays, motivated adversaries can develop highly practical hybrid approaches that start with tracking array indices at a 4-KiB page-level granularity, over to a finer-grained 64-byte cache line granularity within a page, before finally leveraging Nemesis's instruction-granular interrupt timing differences to infer comparisons within a cache line.

4 IMPLEMENTATION ASPECTS

4.1 Implementation on Sancus Platforms

Our Sancus case study attacks exploit timing differences as subtle as a single CPU cycle. In order to do so, the timer interrupt has to arrive at exactly the right time, within the first clock cycle of the enclaved instruction of interest. There is no room for deviation here, as a shift of a single cycle may miss the instruction we are aiming for and corrupt the latency timing difference.

Conveniently, the standard TI MSP430 architecture [69] comes with a Timer_A peripheral capable of generating cycle-accurate interrupts. The timer features an internal Timer_A Register (TAR) that is incremented every clock cycle, and can be configured to generate an IRQ upon reaching a certain value. After generation of the interrupt request, Timer_A immediately restarts counting from zero. Hence, interrupt latency on MSP430 microcontrollers

can be measured trivially by reading TAR as the first instruction in the ISR. The key to a successful exploit thus comes down to determining the amount of clock cycles between configuring the timer, and execution of the instruction of interest in the protected module. Again, this is relatively straightforward on an MSP430 microcontroller where – in the absence of pipelining and caching – execution timing is *completely* deterministic. More specifically, instruction execution takes between one and six clock cycles, depending on the addressing modes of the source and destination operands. An MSP430 CPU [69] features seven different addressing modes, yielding a large variation in possible execution cycles. We refer to Appendix A for a full instruction timing table.

Careful analysis of the compiled source code thus suffices to establish appropriate timer configurations for the Sancus application scenarios. To make our exploits more robust against changes in the application's source code, however, we opted for a different approach where the attacker first deploys a near-exact copy of the victim module, adjusted to copy the value of TAR in a global variable directly after execution of the conditional jump of interest. Our practical attack combines the execution timings retrieved from this "spy" module with predetermined constant parameters to dynamically configure the timer at runtime.

4.2 IRQ Latency Traces on SGX Platforms

SGX enclave programs are explicitly left interrupt-unaware by design. While an x86 processor [36] in enclave mode ignores obvious hardware debug assistance features such as the single-step trap flag (RFLAGS.TF) or hardware breakpoints, recent research on interrupt-driven SGX attacks [31, 42, 51, 74] has shown that untrusted OSs can accurately emulate this behavior by leveraging first-rate control over timer devices. So far, these attacks have focussed on collecting side-channel information from frequent enclave preemptions via the page tables, CPU caches, or the branch prediction unit. We are the first to recognize, however, that the *act* of interrupting a victim enclave in itself leaks microarchitectural instruction timings.

We explain below how we extended and improved a state-of-the-art enclave single-stepping framework to collect precise interrupt latency measurements from SGX enclaves. The resulting IRQ latency traces describe the execution time for *each* subsequent instruction in the enclaved computation, and can thus be thought of as an "x-ray" of the microarchitectural processor state and the code executing in the enclave.

Single-Stepping Enclaved Execution. We based our implementation on the recently published open-source SGX-Step [74] framework that allows a privileged adversary to precisely "single-step" enclaves at most one instruction at a time. SGX-Step comes with a Linux kernel driver to establish convenient user space virtual memory mappings for enclave Page Table Entries (PTEs) and the local Advanced Programmable Interrupt Controller (APIC) device. A very precise single-stepping technique is achieved by writing to the APIC timer register directly from user space, eliminating any jitter from kernel context switches in the timer configuration path [31, 42, 51]. Carefully selecting a platform-specific timer interval ensures that interrupts reliably arrive with a very high probability (> 97%) within the first enclaved instruction after ERESUME [74].

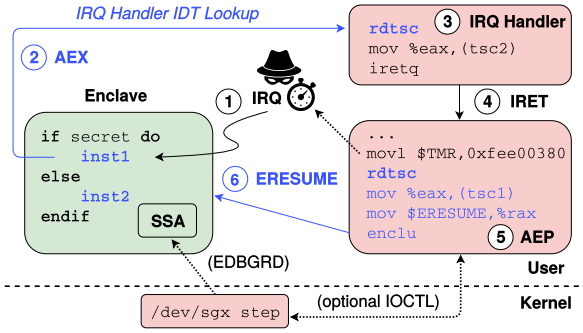


Figure 5: Enhanced SGX-Step framework for precise IRQ latency measurements (blue path) on Intel x86 platforms.

While SGX-Step allows APIC interrupts to be sent from a ring 3 user space process, the original framework still vectors to a conventional ring 0 kernel space interrupt handler. Execution will eventually return to the user space AEP stub where the single-stepping adversary collects side-channel information, and configures the local APIC timer for the next interrupt before resuming the enclave. This approach suffices to amplify conventional side-channels, but subtle microarchitectural timing differences can be affected by noise from kernel space interrupt handling code, privilege level switches, and cache pollution [31, 51]. As such, precisely measuring interrupt latency on Intel x86 platforms presents a substantial challenge over state-of-the-art enclave execution control approaches. As an important contribution, we therefore extended SGX-Step to handle interrupts *completely* within user space, without ever having to vector to the kernel.

Figure 5 summarizes our improved approach to interrupt and resume enclaves. In an initial preparatory phase, the privileged adversary queries the `/dev/sgx-step` Linux kernel driver to establish user space virtual memory mappings for the local APIC MMIO range plus the IA-64 Interrupt Descriptor Table (IDT) [36, 74]. Custom user space ISRs can now be registered directly by writing to the relevant IDT entry, taking care to specify the handler address relative to the user code segment and with descriptor privilege level 3 [36].[6] When the local APIC timer interrupt ① arrives within an enclaved instruction, SGX's secure AEX microcode procedure stores and clears CPU registers inside the enclave. Next, the conventional interrupt logic takes over and ② vectors to the user space interrupt handler. At this point, ③ we immediately grab a timestamp as the very first ISR instruction before ④ returning to the aforementioned AEP stub. ⑤ Here, we log the extracted latency timing measurements, optionally annotating them for benchmark debug enclaves with the stored in-enclave program counter that can be retrieved via the privileged EDBGRD instruction in the `/dev/sgx-step` driver. Thereafter, we configure the local APIC timer for the next interrupt by writing into the initial-count MMIO register, and grab another timestamp to mark the start of the interrupt latency measurement. We take care to ⑥ execute the ERESUME instruction immediately after storing the timestamp to memory. This ensures that the interrupt latency measurement path between the two timestamps

[6] We register our user space handlers as an x86 trap gate, since otherwise the interrupt-enable flag (RFLAGS.IF) does not get restored upon interrupt return.

(visualized in blue in Fig. 5) *only* includes (*i*) three unprotected instructions to store the first timestamp and resume the enclave, plus (*ii*) the enclaved instruction of interest, plus (*iii*) the AEX microcode procedure to vector to the untrusted interrupt handler.

Handling Noise. In contrast to an embedded Sancus-enabled MSP430 CPU, microarchitectural optimizations found in modern x86 processors are known to cause non-constant instruction execution times [11, 12]. Conformant to our attacker model, and closely following previous SGX attacks [8, 25, 31, 42, 51, 75] our experimental setup attempts to reduce measurement noise to a minimum by leveraging some of the unique untrusted operating system adversary capabilities to increase execution time predictability: disable HyperThreading and dynamic frequency scaling (C-states, SpeedStep, TurboBoost), and affinitize the enclave process to a dedicated CPU with Linux's `isolcpus` kernel parameter.

To compensate for the remaining measurement noise, we correlate IRQ latency observations from repeated enclaved executions over the same input, as is not uncommon practice in (SGX) side-channel research [8, 25, 42, 51, 62]. Specifically, we will show in Section 5 that the IRQ latency measurements extracted by our framework exhibit a normally distributed variance. As such, adversaries can rely on basic statistical analysis techniques (e.g., mean, median, standard deviation) to combine multiple IRQ latency observations into a representative overall trace of enclaved instruction timings. Our practical implementation uses a Python post-processing script to parse the raw measurements extracted by our framework for repeated enclaved executions. The resulting traces plot the median execution time (plus optionally a box plot describing the distribution) for each subsequent instruction in the enclaved execution.

Accurately aggregating IRQ latency measurements from repeated enclaved executions also presents another substantial challenge, however. That is, while SGX-Step guarantees that a victim enclave executes *at most* one instruction at a time, a relatively low fraction of the timer IRQs (< 3%) still arrives within ERESUME – before an enclaved instruction is ever executed [74]. Such "zero-step" events are harmless in themselves, but should be filtered out in order to correctly associate repeated measurements for the same step (i.e., instruction) in different enclave invocations. We therefore contribute a novel technique to deterministically recognize false zero-step interrupts by probing the "accessed" bit [36] in the unprotected page table entry mapping the enclaved code page. Specifically, we experimentally verified that the CPU *only* sets the code PTE accessed bit when the enclave did indeed execute an instruction (i.e., timer interrupt arrived *after* ERESUME). Merely clearing the PTE accessed bit for the relevant enclaved code page before sending the interrupt, and querying it afterwards thus suffices to filter out false zero-step observations and achieve noiseless single-stepping.

5 EVALUATION

Our embedded scenarios were evaluated on a development version of Sancus, extended with the hardware-level secure interrupt mechanism described in Section 3.1. We interfaced the Sancus core with a Diligent PmodKYPD peripheral for the secure I/O application. All SGX experiments were conducted on an off-the-shelf Dell Inspiron 13 7359 laptop with a generic Linux v4.13.0 kernel on a Skylake dual-core Intel i7-6500U CPU running at 2.5 GHz. Custom BIOS and kernel parameters were described in the previous section.

5.1 Effectiveness on Sancus

To evaluate our attack against the MSP430 bootstrap loader software, we encapsulated the relevant password comparison routine `BSL430_unlock_BSL` in a protected Sancus enclave. Texas Instruments eliminated secret-dependent control flow entirely from BSL v3 onwards (with a bitwise `or` of the `xor` of each pair of bytes). To the best of our knowledge, vulnerable BSL versions are no longer distributed. We therefore based our implementation on the latest BSL v9, where we replaced the invulnerable, `xor`-based password comparison with the hardened assembly code from Listing 1. The untrusted application context succeeds in recovering the full BSL password by iterating over all possible values for each input byte sequentially. A single interrupt per guess suffices to determine the correctness of the password byte under consideration. As such, our interrupt timing attack reduces an exhaustive search for the password from an exponential to a linear effort.

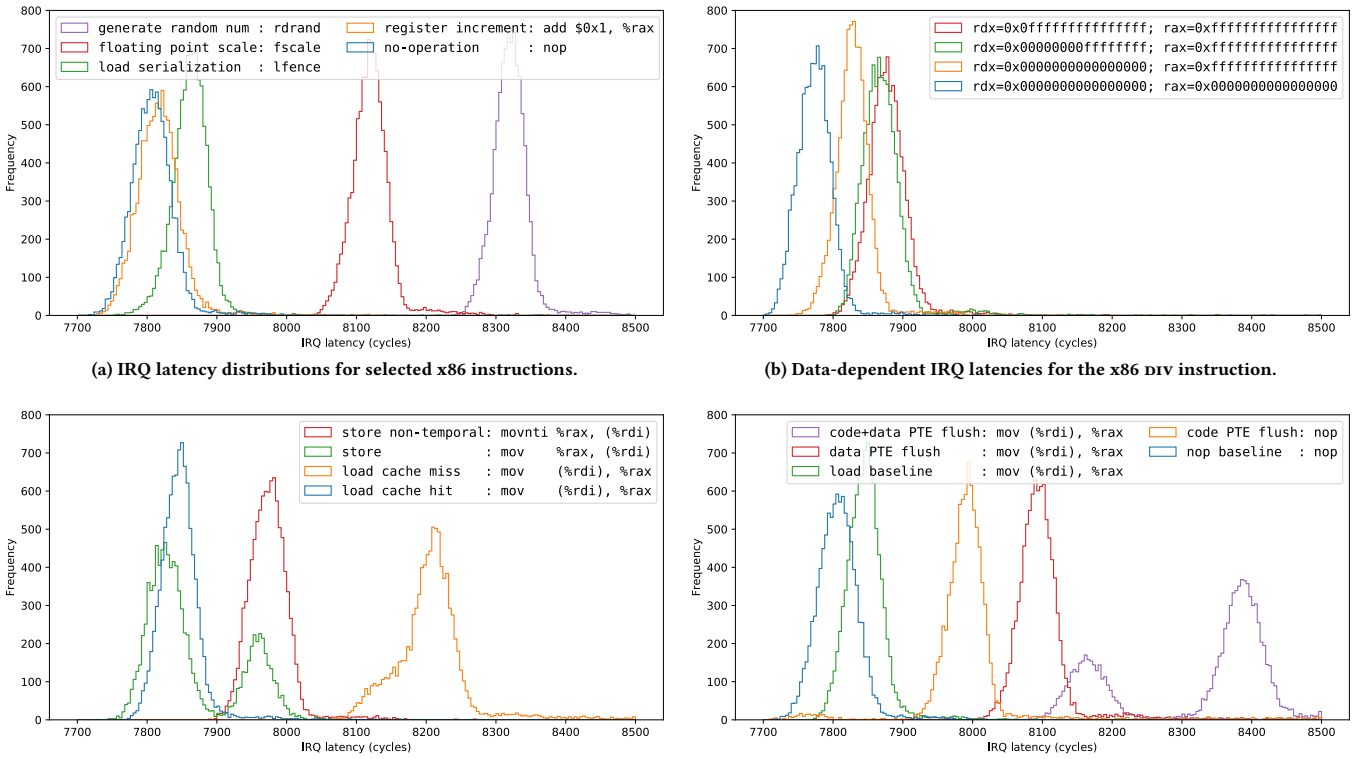
We provide the full source code of the `poll_keypad` function in Appendix B. The program was compiled with the Sancus C compiler based on LLVM/Clang v3.7.0. Our exploit recognizes all key presses without noise, in a single run of the victim enclave. This is an important property for I/O scenarios where, unlike cryptographic algorithms, a victim cannot be forced to execute the same code over the same secret data multiple times. Instead, key strokes should be recognized in real-time, while they are being entered by the human actor. Moreover, our secure keypad attack only requires a single IRQ per loop iteration, totaling no more than 16 interrupts to recover the full key mask from a single enclaved execution.

5.2 SGX Microbenchmarks

We first present microbenchmark experiments in order to quantify the effect of microarchitectural execution state and instruction type on the latency of individual x86 instructions. The microbenchmarks were obtained by single-stepping a benchmark SGX enclave that executes a slide of 10,000 identical assembly instructions. We refer to the original SGX-Step paper [74] for a thorough evaluation of its APIC timer-based single-stepping mechanism which guarantees that at most one enclaved instruction is executed per interrupt. Additionally, we used the code PTE "accessed" bit technique described in Section 4.2 to deterministically filter out false zero-step observations, resulting in perfect single-stepping capabilities.

Differentiating Instruction Types. Figure 6a provides the IRQ latency distributions for selected processor instructions. The horizontal axis lists the observed latency timings in CPU cycles, whereas the number of corresponding interrupts in this latency class is depicted on the vertical axis. Note that the horizontal axis does not start from zero, as our interrupt latency measurement path (Fig. 5) includes the execution times of the ERESUME and AEX microcode.

As a first important result, we can decisively distinguish certain low-latency enclaved operations such as NOP or ADD from higher-latency ones such as secure random number generation (RDRAND) or certain floating point operations (FSCALE), solely by observing the latency they induce on interrupt. This confirms our hypothesis that IRQ latency on x86 platforms depends on the execution time



(a) IRQ latency distributions for selected x86 instructions.

(b) Data-dependent IRQ latencies for the x86 DIV instruction.

(c) Increased IRQ latencies from enclaved data cache misses.

(d) Increased IRQ latencies from unprotected PTE data cache misses.

Figure 6: SGX microbenchmarks: IRQ latency distribution timing variability based on (a) enclaved instruction type, (b) secret input operands, (c) enclave private memory caching conditions, and (d) untrusted address translation data cache misses.

of the interrupted instruction. Hence, these benchmarks can be considered clear evidence for the existence of a timing-based side-channel in SGX's secure AEX procedure.

We can furthermore conclude that differentiating a NOP instruction from an ADD with immediate and register operands is much less obvious, however. These instructions are indeed very similar at the microarchitectural level, both requiring only a single micro-op [22]. As an interesting special case, we investigated the IRQ latency behavior of the LFENCE instruction, which serializes all prior load-from-memory operations. This instruction has recently become particularly relevant, for Intel officially recommends [37] to insert LFENCE instructions after sensitive conditional branches to protect SGX enclaves against Spectre v1 speculative bounds check bypasses [40]. While the microarchitectural timing differences are more subtle, Fig. 6a still shows that one can on average plainly separate LFENCE from ordinary NOP or ADD instructions.

Measuring Data Timing Channels. Variable latency arithmetic instructions are known to be an exploitable side-channel, even in code without secret-dependent control flow [4, 11, 12]. Previous research on microarchitectural data timing channels has established that the execution time of some commonly used x86 arithmetic instructions such as (floating point) multiplication or division depends on the operands they are being applied upon. Our second set of microbenchmark experiments therefore explore leakage of enclaved operand values through interrupt latency for the widely studied [4, 11, 12] unsigned integer division x86 DIV instruction.

Figure 6b shows the IRQ latency distributions for 10,000 enclaved executions of the DIV instruction applied on different 128-bit dividend operands and a fixed 64-bit divisor (`0xffffffffffffffff`). The average interrupt latency clearly increases as the dividend becomes larger, which confirms that "the throughput of DIV/IDIV varies with the number of significant digits in the input RDX:RAX" [35] As such, we conclude that IRQ latency leaks operand values for variable latency instructions. Importantly, in contrast to classical start-to-end timing measurements, Nemesis-style interrupt timing attacks leak this information at an *instruction-level* granularity, which allows to precisely isolate (successive) data-dependent instruction timing measurements in a larger enclaved computation.

Influence of Data Caching. Figure 6c investigates the IRQ latency distributions for selected MOV instructions to/from enclave memory. The store distribution is characterized by two prominent normally distributed peaks. Our hypothesis is that the right peak, representing measurements with a larger IRQ latency, is caused by a write miss in the data cache.[7] A write miss indeed forces the CPU to

[7]Intel SGX always uses a write-back caching policy for enclave memory [34]. This means that a write hit on enclave memory initially only updates the cache, unblocking the processor immediately, while writing to main memory is postponed until eviction of the dirty cache line. When the data was not yet in the cache (i.e., write miss),

wait for completion of the memory transaction before finishing the instruction. It appears that in this particular experimental setup, the processor's cache replacement policy rather frequently evicts the data accessed by the benchmark enclave. To support this hypothesis, we examined IRQ latency behavior for the x86 MOVNTI store operation with a non-temporal hint that forces the CPU to write the data directly into memory, without updating or fetching the corresponding cache line. MOVNTI clearly manifests an increased latency that overlaps with the right peak of the store distribution.

To investigate the impact of data cache misses on enclaved load operations, we instrumented the instruction slide in our benchmark enclave to explicitly invalidate the corresponding cache line by executing CLFLUSH before each MOV instruction. Our noiseless single-stepping techniques allows to afterwards filter out latency measurements for the interleaved CLFLUSH instructions, such that the resulting IRQ latency distributions are solely characterized by the execution times of the MOV instruction under consideration. Figure 6c shows a prominently increased latency for intra-enclave memory load operations that miss the data cache hierarchy. We suspect that the sparser distribution for load cache misses is caused by noise from the DRAM controller.

These experiments thus provide clear evidence for the fact that IRQ latency reveals cache misses. This finding may be particularly relevant for state-of-the-art fortified PMA designs like Sanctum [14] that include all known architectural countermeasures to prevent adversaries from gaining insight into enclave caching behavior.

Influence of Address Translation. SGX was explicitly designed to traverse untrusted page tables during enclaved execution, and verifies address translation metadata via an independent additional protection mechanism. Recent research on address translation side-channel attacks [27, 75], however, exploits the microarchitectural property that x86 page table entries are cached as with normal data. By spying on unprotected cache lines, adversaries can gain insight into enclaved memory page accesses.

Our last set of microbenchmark experiments explores the impact of untrusted address translation data cache misses on the latency of the interrupted instruction. We used CLFLUSH before resuming the benchmark enclave to invalidate the cache line for the *unprotected* PTE entry that stores the physical address of the code page containing the microbenchmark instruction slide. Figure 6d demonstrates that we can distinctly increase the latency of even ordinary NOP instructions in this way. Furthermore, for instructions with a memory operand, kernel-level adversaries can choose to flush the PTE entry for the data operand, and/or the enclaved code page to be executed. Figure 6d indeed shows a clear increase in IRQ latency for MOV instructions that need an additional memory access to retrieve the physical address of the private data operand. Likewise, latency even further increases when also flushing the PTE entry of the enclaved code page containing the load instruction.

We conclude that IRQ latency reveals data cache misses in the page table walk at instruction-level granularity. While SGX page tables reside in unprotected memory, this finding may once more impact fortified PMA designs [14, 17] that move page tables inside enclave memory, out of reach of the attacker, to protect against address translation side-channel attacks.

however, any dirty line about to be replaced has to be written back, and the new line has to be fetched from main memory.

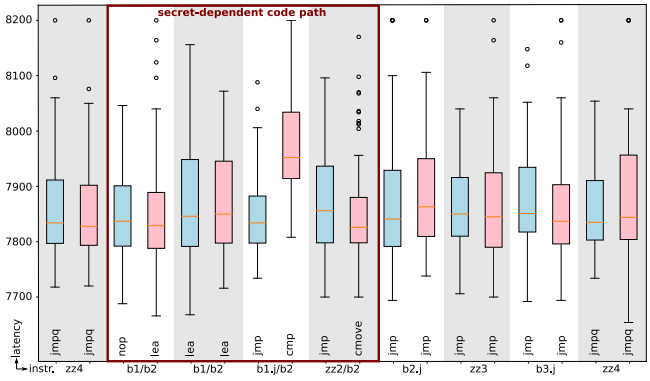


Figure 7: IRQ latency distributions for 100 runs of Zigzagger branch taken (blue) vs. not-taken (red) execution paths.

5.3 SGX Macrobenchmark Attack Scenarios

To demonstrate information leakage in larger enclave programs, we extracted full IRQ latency traces from the SGX case study applications introduced in Section 3.2. In contrast to the isolated microbenchmark experiments described above, our macrobenchmark results illustrate interrupt latency behavior in typical, compiler-generated mixed instruction streams.

Defeating Zigzagger. Since the Zigzagger compiler pass was not made publicly available, we copied the exemplary assembly code (Fig. 4) from the corresponding paper [42] in an SGX enclave. As explained in Section 3.2.1, we made sure to manually align secret-dependent code to fit entirely within one 64-byte cache line. Figure 7 shows the IRQ latency distributions extracted by our framework for 100 repeated runs of a victim enclave that either takes the first Zigzagger-obfuscated branch (a=1; blue) or not (a=0; red). The left-most box plots visualize IRQ latency measurements for the indirect branch instruction `zz4` at the end of the Zigzagger trampoline, whereas the following grouped box plots represent instruction latencies in the the conditional control flow path from either `b1` (blue) or `b2` (red) to the next secret-dependent jump at `zz4`. Note that both execution paths in the assembly code snippet of Fig. 4 already merge at `b2.j`, such that IRQ latency traces extracted from Zigzagger-hardened code only feature an extremely short secret-dependent sequence of 4 instructions, marked in Fig. 7.

A first important observation, in line with the microbenchmark results above, is that IRQ latency measurements are normally distributed such that we need to perform multiple observations before making decisive conclusions on the timing characteristics of the instruction under consideration. In this respect, the first two instructions in the secret-dependent execution paths exhibit similar and fairly indistinguishable IRQ latency distributions, which is indeed to be expected given that NOP and LEA instructions behave identical (micro-op count, latencies) at the microarchitectural level [22] The third secret-dependent instruction, either JMP (blue) or CMP (red), however, manifests a sharply visible (median) IRQ latency difference that can be exploited to unambiguously distinguish both

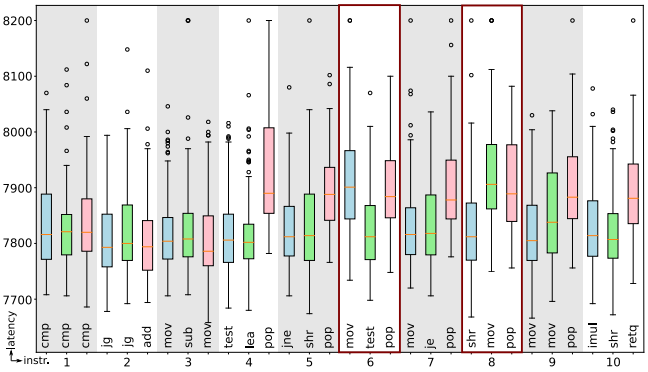


Figure 8: IRQ latency distributions for 100 runs of `bsearch` left (blue) vs. right (green) vs. equal (red) execution paths.

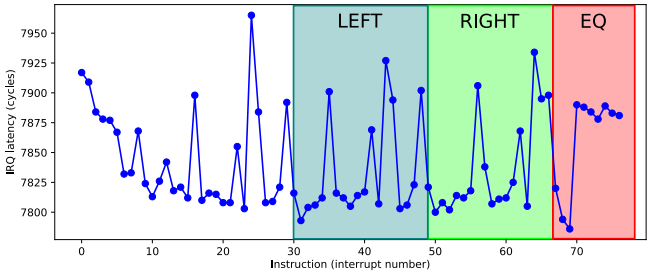


Figure 9: Median IRQ latencies over 100 `bsearch` invocations.

branches. Specifically, by relying on the noiseless single-stepping technique from Section 4.2, adversaries can collect IRQ latencies from repeated enclaved executions, and afterwards categorize the samples for the third secret-dependent instruction as either a JMP or CMP. To compensate for outliers, we use the median IRQ latency instead of the mean. Note that Fig. 7 was generated from 100 repeated enclave invocations to yield a representative overall plot, but we found that in practice secret-dependent Zigzagger branches can already be reliably identified after as little as 10 enclave invocations. Finally, also note that there exists a subtle yet potentially exploitable IRQ latency distribution difference for the last secret-dependent instruction JMP (blue) vs. register CMOV (red).

Inferring Binary Search Indices. To evaluate our binary search attack, we constructed an enclave that calls the Intel SGX SDK `bsearch` trusted library function to look up a value in a fixed integer array. We carefully selected the exemplary lookup value to ensure that `bsearch` first looks left, then right, and finally returns the requested address. Our practical exploit faults on the code page containing the `bsearch` function to enter single-stepping mode and then starts collecting IRQ latency measurements. Figure 9 plots the median IRQ latencies obtained from 100 enclaved `bsearch` executions, where each individual data point reveals the execution time of the corresponding assembly instruction in Appendix C. We annotated the trace to mark the three consecutive execution paths (left, right, equal) after comparing the value for that loop iteration. As with the Zigzagger benchmark, Fig. 8 furthermore compares relative IRQ latency distributions by means of box plots for each assembly instruction in the secret-dependent execution paths.

As a first important result, one can easily identify the relatively high-latency peaks from the 6 subsequent POP stack accessing instructions in the return path of the equal case (red; instructions 4-10 in Fig. 8). Furthermore, while distinguishing the left (blue) and right (green) cases is more subtle, the source code in Listing 2 indicates that the right case has to perform slightly more work before continuing to the next loop iteration. This is indeed reflected at the assembly code level by two more low-latency register instructions (SUB and LEA) before the right branch continues along the common execution path. Again, we found this extremely subtle difference to be sufficient to distinguish both branches via the relative position of a higher-latency MOV instruction at the start of the for loop. It is apparent from Fig. 9 that the IRQ latency patterns for the right branch are slightly shifted with respect to those of the left one. Particularly, the first high-latency peak in the left branch occurs 4 interrupts (instruction 6 in Fig. 8) after CMP, whereas for the right branch this peak only occurs after 6 interrupts (instruction 8 in Fig. 8). As with the Zigzagger benchmark, comparing median IRQ latency samples for specific instructions (identified by their single-stepping interrupt number) thus suffices to reliably infer control flow decisions in the binary search algorithm and establish the secret lookup key.

6 DISCUSSION AND MITIGATIONS

Interrupt Timing Leaks. While generally well-understood at the architectural level, asynchronous CPU events like faults and interrupts have not been studied extensively at the microarchitectural level. Recent developments on Meltdown-type "fault latency" attacks [44, 71] exposed fundamental flaws in the way modern out-of-order processors enforce software isolation, whereas Nemesis reveals a more intrinsic and subtle timing side-channel in the CPU's interrupt mechanism. We showed that the act of interrupting enclaved execution leaks microarchitectural timing information at an instruction-level granularity, even on the most rudimentary of microcontrollers. In this, we have presented the first remotely exploitable controlled-channel for embedded enclave processors, and we contribute to the understanding of SGX side-channel information leakage beyond the usual suspects.

IRQ latency traces (e.g., Fig. 9) can be regarded as an instruction-granular "x-ray" for enclaved execution. Our microbenchmark SGX experiments show that interrupt latency directly reveals certain high-latency enclaved operations, and can furthermore reliably quantify other microarchitectural properties that affect execution time on modern x86 processors [23], e.g., data-dependent instruction latencies, and data or page table cache misses. In this respect, we expect that Nemesis's ability to extract fine-grained microarchitectural instruction timings from SGX enclaves will enable new and improved side-channels such as MemJam-type [50] false dependency attacks. As a particularly relevant finding for fortified PMAs like Sanctum [14] that aim to eradicate known cache timing attacks, we identified what might well be one of the last remaining side-channels that provide insight into enclave caching behavior. Specifically, since we have shown that interrupt latency reveals cache misses, we can see IRQ latency traces being leveraged in a

trace-driven cache attack [1] for instance to reduce the key space of cryptographic algorithms.

We have demonstrated that interrupt latency timing attacks pose a direct and serious threat to the protection model pursued by embedded PMAs such as Sancus, though further research is needed to investigate the bandwidth of practical Nemesis side-channel attacks on SGX platforms. A particularly promising future work avenue in this respect would be to supersede reverse engineering and statistical analysis efforts by applying automated machine learning techniques on IRQ latency traces extracted from multiple invocations of the victim enclave.

Why Constant-Time IRQ Defenses are Insufficient. We have shown how interrupt-capable adversaries can dissolve black box-style start-to-end protected execution times into (a sequence of) execution timing measurements for individual enclaved instructions. This paper has focussed on exploring "interrupt latency timing" channels on multi-cycle instruction set processors, but we want to stress that attack surface from secure interrupts is *not* limited to timing side-channels only. Another potentially dangerous "interrupt counting" channel for instance would measure the total number of times the enclaved execution can be interrupted before it finally completes. For example, in the balanced BSL password comparison scenario of Listing 1, adversaries can interrupt the if branch twice (2 instructions), whereas the else branch featuring NOP compensation code can be interrupted four times (4 instructions). As such, while the total enclaved execution time remains constant, interrupt-capable single-stepping adversaries will still notice a decrease in the total IRQ count for each correct password byte.

The above interrupt counting channel seems particularly interesting, for it only assumes a multi-cycle instruction set architecture, and thus continues to persist on processors with constant-time IRQ latency. We for instance considered a hardware patch for Sancus that always enforces the worst-case interrupt response time by inserting dummy execution cycles depending on the enclaved instruction being interrupted. Alternatively, ARM Cortex M0 processors [5] abandon multi-cycle instructions to handle any pending interrupt immediately. While such processors are immune to the IRQ latency timing attacks described in this paper, they remain vulnerable to interrupt counting attacks and may additionally be exposed to advanced Nemesis-type interrupt timing attack variants.

We conclude that constant-time interrupt logic is a necessary but not sufficient condition to eradicate Nemesis-style interrupt attacks at the hardware level. In general processor-level solutions alone seem not to be able to completely prevent information leakage from secure interrupts in enclaved execution. This finding may have a consequential impact for fully abstract compilation schemes [59] and provably side-channel resistant processor designs [20, 21] that have so far not considered secure interrupt timing channels. We encourage further research and formal analysis to adequately address interrupt-based side-channels via hardware-software co-design.

Application Hardening. Considering that our attacks exploit secret-dependent control flow, an application-level solution should strive to eliminate conditional program branches and variable latency instructions completely. This can be realized by rewriting the enclave code manually (e.g., `xor`-based password comparison of Section 5.1), or by automated if-conversion in a compiler backend [12]. Such solutions remain compatible with existing PMA hardware, but also assume that sensitive information can be easily identified. Previous research [8, 79] in this area has shown that sensitive application data may be more ill-defined than the typical cryptographic keys of side-channel analysis. Moreover, if-conversion comes with a significant performance overhead [12] that somewhat invalidates the PMA promise of native code execution in a protected environment.

Alternatively, compilers could focus on detecting, rather than eliminating, IRQ timing attacks. Our interrupt extensions for Sancus indeed follow PMA designs [7, 14, 41] that explicitly call into an enclave to request resumption of internal execution. As such, Sancus enclaves are *interrupt-aware* and they could use excessive interrupt rates as an indicator to trigger some security policy that terminates the module and/or destroys secrets. Interrupts also occur in benign conditions, however, and a single interrupt already suffices to leak confidential information, as evident from our Sancus attack scenarios. Adversaries could thus adapt their attacks to the entry policy of a victim enclave.

Intel SGX on the other hand leaves enclave programs explicitly *interrupt-unaware* through the use of a dedicated ERESUME hardware instruction. However, a contemporary line of research [10, 28, 63] leverages hardware support for Transactional Synchronization eXtensions (TSX) in recent x86 processors to detect interrupts or page faults in enclave mode. More specifically, these proposals rely on the property that code executing in a TSX transaction is aborted and automatically rolled back when an external interrupt request arrives. TSX furthermore modifies the stored in-enclave instruction pointer upon AEX, such that a preregistered transaction abort handler is called on the next ERESUME invocation. Whereas TSX-based defenses would likely recognize suspicious interrupt rates when single-stepping enclaved execution, advanced Nemesis adversaries could construct stealthy Sancus-like IRQ timing attacks that only interrupt the victim enclave minimally and stay under under the radar of the transaction abort handler's probabilistic security policy. Moreover, TSX-based defenses also suffer from some important limitations [67, 74], ranging from the absence of TSX features in some processors to severe runtime performance impact and the false positive/negative rates inherent to heuristic defenses. In conclusion, we do *not* regard current ad-hoc TSX approaches as a solution, even apart from compatibility and performance issues, since they cannot prevent the root information leakage cause. Our attacks against Sancus show that a *single* interrupt can deterministically leak sensitive information, and we expect further development of the attacks against SGX to increase stealthiness, as has been shown for instance for page-table based attacks [75, 77].

7 RELATED WORK

We have discussed PMAs security architectures throughout the paper. In this section we focus on relating our work to existing side-channel analysis research. There exists a vast body of work on microarchitectural timing channels [23], but side-channel attacks in a PMA context are only being explored very recently. To the best of our knowledge, we have presented the first remotely exploitable controlled-channel for low-end embedded PMAs. Various authors [13, 20, 43] have explicitly expressed their concerns on software side-channel vulnerabilities in higher-end PMAs such as

Intel SGX. This paper argues, however, that current attack research efforts focus too narrowly on the "usual suspects" that are relatively well-known, and do not reveal anything really unexpected. Apart from our work, only page table-based attacks [30, 64, 75, 77, 79] have to date been identified as a novel controlled-channel. Compared to IRQ latency, the page fault channel has a coarser-grained granularity (instruction vs. page-level), but does not suffer from the noise inherent to microarchitectural channels.

Our attack vector is closely related to cache timing side-channels in that IRQ latency traces reveal cache misses. A powerful class of access-driven cache attacks based on the Prime+Probe technique [58] first primes the cache by loading congruent addresses, and thereafter measures the time to reload these addresses so as to establish memory access patterns by the victim. Such Prime+Probe cache timing attacks have been successfully applied against SGX enclaves [8, 25, 31, 51, 62]. When memory is shared between the attacker and the victim, Flush+Reload [80] and Flush+Flush [29] techniques improve the efficiency of cache timing attacks. In the context of Intel SGX, these techniques have recently been leveraged to spy on unprotected page table memory [75].

It has furthermore been shown that enclave-private control flow leaks via the CPU's branch prediction machinery [18, 42], which recently became particularly relevant for Spectre-type speculative execution attacks [9, 40]. Recent Intel microcode patches address Spectre attacks against SGX enclaves by clearing the BTB upon enclave entry/exit [9]. At the microarchitectural level, Nemesis-style interrupt latency timing attacks are more closely related to Meltdown [44] in that both abuse the property that asynchronous CPU events like faults and interrupts are only handled upon instruction retirement. While Intel SGX was initially considered to be resistant to Meltdown-type transient execution vulnerabilities, recent work presented Foreshadow [71, 78], which allows for arbitrary in-enclave reads and completely collapses isolation and attestation guarantees in the SGX ecosystem. To allow for TCB recovery, Intel has revoked the compromised attestation keys, and released microcode patches to address Foreshadow at the hardware level.

8 CONCLUSION

The security aspects of asynchronous CPU events like interrupts and faults have not been amply studied from a microarchitectural perspective. We contributed Nemesis, a subtle timing channel in the CPU's rudimentary interrupt logic. Our work represents the first controlled-channel attack against embedded enclaved execution processors, and we demonstrated Nemesis's applicability on modern Intel SGX x86 platforms.

ACKNOWLEDGMENTS

We thank Job Noorman for guidance on the Sancus secure interrupt extensions, and Pieter Maene for valuable feedback on early versions of this text. The research presented in this paper was partially supported by the Research Fund KU Leuven, and by a gift from Intel Corporation. Jo Van Bulck and Raoul Strackx are supported by a grant of the Research Foundation – Flanders (FWO).

REFERENCES

[1] Onur Aciiçmez and Çetin Kaya Koç. 2006. Trace-driven cache attacks on AES. Cryptology ePrint Archive, Report 2006/138. http://eprint.iacr.org/2006/138.

[2] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*. ACM, 312–320.

[3] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, Vol. 13.

[4] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On subnormal floating point and abnormal timing. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 623–639.

[5] ARM. 2009. *Cortex-M0 technical reference manual r0p0.* http://infocenter.arm.com/help/topic/com.arm.doc.ddi0432c/DDI0432C_cortex_m0_r0p0_trm.pdf.

[6] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding applications from an untrusted cloud with Haven. In *11th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 267–283.

[7] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. 2015. TyTAN: Tiny trust anchor for tiny devices. In *Design Automation Conference (DAC 2015)*. IEEE, 1–6.

[8] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT '17)*. USENIX Association.

[9] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2018. SGXPECTRE attacks: Leaking enclave secrets via speculative execution. *arXiv preprint arXiv:1802.09085* (2018).

[10] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. 2017. Detecting privileged side-channel attacks in shielded execution with Déjà Vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 7–18.

[11] Jeroen V Cleemput, Bart Coppens, and Bjorn De Sutter. 2012. Compiler mitigations for time attacks on modern x86 processors. *ACM Transactions on Architecture and Code Optimization (TACO)* 8, 4 (2012), 23.

[12] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. 2009. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *2009 IEEE Symposium on Security and Privacy*. IEEE, 45–60.

[13] Victor Costan and Srinivas Devadas. 2016. *Intel SGX explained.* Technical Report. Computer Science and Artificial Intelligence Laboratory MIT. https://eprint.iacr.org/2016/086.pdf.

[14] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium*. USENIX Association, 857–874.

[15] Ruan De Clercq, Frank Piessens, Dries Schellekens, and Ingrid Verbauwhede. 2014. Secure interrupts on low-end microcontrollers. In *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*. IEEE, 147–152.

[16] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. 2012. SMART: Secure and minimal architecture for (establishing a dynamic) root of trust.. In *NDSS*, Vol. 12. Internet Society, 1–15.

[17] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. 2014. Iso-x: A flexible architecture for hardware-managed isolated execution. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 190–202.

[18] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, Dmitry Ponomarev, et al. 2018. BranchScope: A new side-channel attack on directional branch predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 693–707.

[19] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 287–305.

[20] Andrew Ferraiuolo, Yao Wang, Rui Xu, Danfeng Zhang, Andrew Myers, and Edward Suh. 2015. *Full-processor timing channel protection with applications to secure hardware compartments.* Computing and Information Science Technical Report. Cornell University. http://hdl.handle.net/1813/41218.1.

[21] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C Myers, and G Edward Suh. 2017. Verification of a practical hardware security architecture through static information flow analysis. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 555–568.

[22] Agner Fog. 2018. *Instruction tables. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs.* Technical Report. Technical University of Denmark. http://www.agner.org/optimize/instruction_tables.pdf.

[23] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering* 8, 1 (2018), 1–27.

[24] Travis Goodspeed. 2008. Practical attacks against the MSP430 BSL. In *Twenty-Fifth Chaos Communications Congress*.

[25] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security (EuroSec'17)*.

[26] Johannes Götzfried, Tilo Müller, Ruan de Clercq, Pieter Maene, Felix Freiling, and Ingrid Verbauwhede. 2015. Soteria: Offline software protection within low-cost embedded devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 241–250.

[27] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Christiano Giuffrida. 2017. ASLR on the line: practical cache attacks on the MMU. *NDSS (Feb. 2017)* (2017).

[28] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security Symposium*.

[29] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A fast and stealthy cache attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer.

[30] Jago Gyselinck, Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Off-limits: Abusing legacy x86 memory segmentation to spy on enclaved execution. In *International Symposium on Engineering Secure Software and Systems (ESSoS '18)*. Springer, 44–60.

[31] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-resolution side channels for untrusted operating systems. In *2017 USENIX Annual Technical Conference (ATC '17)*. USENIX Association.

[32] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. 2013. Using innovative instructions to create trustworthy software solutions. In *HASP@ ISCA*. 11.

[33] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. 2013. InkTag: Secure applications on an untrusted operating system. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 265–278.

[34] Intel Corporation. 2014. *Intel software guard extensions programming reference*. Order no. 329298-002US.

[35] Intel Corporation. 2018. *Intel 64 and IA-32 architectures optimization reference manual*. Order no. 248966-040.

[36] Intel Corporation. 2018. *Intel 64 and IA-32 architectures software developer's manual*. Order no. 325384-067US.

[37] Intel Corporation. 2018. *Intel software guard extensions (SGX) SW development guidance for potential bounds check bypass (CVE-2017-5753) side channel exploits*. Rev. 1.1.

[38] Simon Johnson. 2017. Intel SGX and side-channels. https://software.intel.com/en-us/articles/intel-sgx-and-side-channels.

[39] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 207–220.

[40] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre attacks: exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[41] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. 2014. TrustLite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, Article 10, 14 pages.

[42] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proceedings of the 26th USENIX Security Symposium*. Vancouver, Canada.

[43] Andy Leiserson. 2018. Side channels and runtime encryption solutions with Intel SGX. https://www.fortanix.com/assets/Fortanix_Side_Channel_Whitepaper.pdf. (2018).

[44] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[45] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling, and I. Verbauwhede. 2017. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Trans. Comput.* 99 (2017).

[46] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB reduction and attestation. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 143–158.

[47] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. 2008. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008*. ACM, 315–328.

[48] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, Article 10, 1 pages.

[49] Mike Mitchell. 2002. *Implementing an ultralow-power keypad interface with the MSP430*. Technical Report. Texas Instruments. http://www.ti.com/lit/an/slaa139/slaa139.pdf.

[50] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2018. MemJam: A false dependency attack against constant-time crypto implementations in SGX. In *CryptographersâĂŹ Track at the RSA Conference*. Springer, 21–44.

[51] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX amplifies the power of cache attacks. In *Conference on Cryptographic Hardware and Embedded Systems (CHES '17)*.

[52] Job Noorman. 2017. *Sancus: A low-cost security architecture for distributed IoT applications on a shared infrastructure*. Ph.D. Dissertation. KU Leuven. https://lirias.kuleuven.be/bitstream/123456789/574995/1/thesis.pdf.

[53] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. 2013. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security Symposium*. USENIX Association, 479–494.

[54] Job Noorman, Jan Tobias Mühlberg, and Frank Piessens. 2017. Authentic execution of distributed event-driven applications with a small TCB. In *STM '17 (LNCS)*, Vol. 10547. Springer, Heidelberg, 55–71.

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

[56] Guevara Noubir and Amirali Sanatinia. 2016. Trusted code execution on untrusted platform using Intel SGX. *Virus Bulletin* (2016).

[57] Open Whisper Systems. 2017. Technology preview: Private contact discovery for Signal. https://signal.org/blog/private-contact-discovery/.

[58] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Cryptographers' Track at the RSA Conference*. Springer, 1–20.

[59] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. 2015. Secure compilation to protected module architectures. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 37, 2 (2015).

[60] Peter Puschner, Raimund Kirner, Benedikt Huber, and Daniel Prokesch. 2012. Compiling for time predictability. In *International Conference on Computer Safety, Reliability, and Security*. Springer, 382–391.

[61] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 38–54.

[62] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware guard extension: Using SGX to conceal cache attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA) (DIMVA)*.

[63] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[64] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16)*. ACM, 317–328.

[65] Shweta Shinde, Shruti Tople, Deepak Kathayat, and Prateek Saxena. 2015. *Podarch: Protecting legacy applications with a purely hardware TCB*. Technical Report. National University of Singapore.

[66] Raoul Strackx, Job Noorman, Ingrid Verbauwhede, Bart Preneel, and Frank Piessens. 2013. Protected software module architectures. In *ISSE 2013 Securing Electronic Business Processes*. Springer, 241–251.

[67] Raoul Strackx and Frank Piessens. 2017. The Heisenberg defense: Proactively defending SGX enclaves against page-table-based side-channel attacks. https://arxiv.org/abs/1712.08519. *arXiv preprint arXiv:1712.08519* (Dec. 2017).

[68] Raoul Strackx, Frank Piessens, and Bart Preneel. 2010. Efficient isolation of trusted subsystems in embedded systems. In *Security and Privacy in Communication Networks*. Springer, 344–361.

[69] Texas Instruments. 2006. *MSP430x1xx family: User's guide*. http://www.ti.com/lit/ug/slau049f/slau049f.pdf.

[70] Florian Tramer, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. 2017. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *2nd IEEE European Symposium on Security and Privacy (Euro S&P)*. IEEE.

[71] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association.

[72] Jo Van Bulck, Jan Tobias Mühlberg, and Frank Piessens. 2017. VulCAN: Efficient component authentication and software isolation for automotive control networks. In *Proceedings of the 33th Annual Computer Security Applications Conference (ACSAC'17)*. ACM.

[73] Jo Van Bulck, Job Noorman, Jan Tobias Mühlberg, and Frank Piessens. 2016. Towards availability and real-time guarantees for protected module architectures. In *MODULARITY Companion Proceedings '16*. ACM, New York, 146–151.

[74] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution (SysTEX'17)*. ACM, 4:1–4:6.

[75] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Security Symposium*. USENIX Association.

[76] Marcus Völp, Jérémie Decouchant, Christoph Lambert, Maria Fernandes, and Paulo Esteves-Verissimo. 2017. Enclave-based privacy-preserving alignment of raw genomic information: Information leakage and countermeasures. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution (SysTEX'17)*. ACM, Article 7, 6 pages.

[77] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2421–2434.

[78] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical Report* (2018).

[79] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 640–656.

[80] Yuval Yarom and Katrina Falkner. 2014. Flush+ reload: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium*. USENIX Association, 719–732.

A MSP430 INSTRUCTION CYCLES

This appendix provides the full instruction timings for the MSP430 architecture, as published by Texas Instruments [69]. All jump instructions require two clock cycles to execute, regardless of whether the jump is taken or not. The number of CPU cycles required for other instructions depends on the addressing modes of the source and destination operands, not the instruction type itself. Tables 1 and 2 list the number of cycles for respectively single and double operand instructions. Note that a number of MSP430 assembly operations (including `nop`, `incd`, `rla`, `ret`, and `tst`) are emulated by means of the listed machine instructions.

Table 1: MSP430 single operand instruction cycles.

Addressing Mode	No. of Cycles: RRA, RRC, SWPB, SXT	PUSH	CALL	Example
Rn	1	3	4	`SWPB R5`
@Rn	3	4	4	`RRC @R9`
@Rn+	3	5	5	`SWPB @R10+`
#N	–	4	5	`CALL #0F000H`
x(Rn)	4	5	5	`CALL 2(R7)`
EDE	4	5	5	`PUSH EDE`
&EDE	4	5	5	`SXT &EDE`

Table 2: MSP430 double operand instruction cycles.

Addressing Mode Src	Dst	No. of Cycles	Example
Rn	Rm	1	`MOV R5,R8`
	PC	2	`BR R9`
	x(Rm)	4	`ADD R5,4(R6)`
	EDE	4	`XOR R8,EDE`
	&EDE	4	`MOV R5,&EDE`
@Rn	Rm	2	`AND @R4,R5`
	PC	2	`BR @R8`
	x(Rm)	5	`XOR @R5,8(R6)`
	EDE	5	`MOV @R5,EDE`
	&EDE	5	`XOR @R5,&EDE`
@Rn+	Rm	2	`ADD @R5+,R6`
	PC	3	`BR @R9+`
	x(Rm)	5	`XOR @R5+,8(R6)`
	EDE	5	`MOV @R9+,EDE`
	&EDE	5	`MOV @R9+,&EDE`
#N	Rm	2	`MOV #20, R9`
	PC	3	`BR #2AEH`
	x(Rm)	5	`MOV #0300H,0(SP)`
	EDE	5	`ADD #33,EDE`
	&EDE	5	`ADD #33,&EDE`
x(Rn)	Rm	3	`MOV 2(R5),R7`
	PC	3	`BR 2(R6)`
	x(Rm)	6	`ADD 2(R4),6(R9)`
	EDE	6	`MOV 4(R7),EDE`
	&EDE	6	`MOV 2(R4),&EDE`
EDE	Rm	3	`AND EDE,R6`
	PC	3	`BR EDE,R6`
	x(Rm)	6	`MOV EDE,0(SP)`
	EDE	6	`CMP EDE,EDE`
	&EDE	6	`MOV EDE,&EDE`
&EDE	Rm	3	`MOV &EDE,R8`
	PC	3	`BR &EDE`
	x(Rm)	6	`MOV &EDE,0(SP)`
	EDE	6	`MOV &EDE,EDE`
	&EDE	6	`MOV &EDE,&EDE`

B SECURE KEYPAD IMPLEMENTATION

The enclaved keypad program below was derived from a recently published open-source[8] automotive Sancus application scenario [72], which we had to minimally modify in order to run without function callbacks in a stand-alone enclave.

The start-to-end timing of the `poll_keypad` function only reveals the number of times the if statement was executed, i.e., the number of keys that were down (cf. return value). By carefully interrupting the function each loop iteration, an untrusted ISR can learn the value of the secret PIN code.

[8] https://github.com/sancus-pma/vulcan/blob/master/demo/ecu-tcs/sm_tcs_kypd.c

```
int         SM_DATA(secure) init      = 0x0;
int         SM_DATA(secure) pin_idx   = 0x0;
uint16_t    SM_DATA(secure) key_state = 0x0;
char        SM_DATA(secure) pin[PIN_LEN];
const char SM_DATA(secure) keymap[NB_KEYS] =
{
  '1', '4', '7', '0', '2', '5', '8', 'F',
  '3', '6', '9', 'E', 'A', 'B', 'C', 'D'
};

int SM_ENTRY(secure) poll_keypad( void )
{
  int is_pressed, was_pressed, mask = 0x1;

  /* Securely initialize SM on first call. */
  if (!init) return do_init();

  /* Fetch key state from MMIO driver SM. */
  uint16_t new_key_state = read_key_state();

  /* Store down keys in private PIN array. */
  for (int key = 0; key < NB_KEYS; key++)
  {
    is_pressed  = (new_key_state & mask);
    was_pressed = (key_state & mask);
    if (is_pressed
        /* INTERRUPT SHOULD ARRIVE HERE */
        && !was_pressed && (pin_idx < PIN_LEN))
    {
      pin[pin_idx++] = keymap[key];
    }
    /* .. OR HERE. When configuring the timer
    for the key comparison in the next loop
    iteration, ISR should take into account
    key presses from previous runs to be able
    to detect key releases. */
    mask = mask << 1;
  }
  key_state = new_key_state;

  /* Return the number of characters still
  to be entered by the user. */
  return (PIN_LEN - pin_idx);
}
```

For completeness, we also provide a disassembled version of this function, as compiled with LLVM/Clang v3.7.0.

```
poll_keypad:
   push    r4
   mov     r1, r4
   push    r11
   push    r10
   push    r9
   tst     &init
   jz      3f
   call    #read_key_state
   mov     #1, r12
   clr     r13
   mov     &key_state, r14
1: mov     &pin_idx, r11
   cmp     #4, r11
   jge     2f
   mov     r12, r10
   and     r15, r10
   tst     r10               # test key state
   jz      2f                # V no. of cycles
   mov     r14, r10          # 1
   and     r12, r10          # 1
   tst     r10               # 1
   jnz     2f
   mov.b   518(r13), r10
   mov     r11, r9
   inc     r9
   mov     r9, &pin_idx
   mov.b   r10, 550(r11)     # V no. of cycles
2: rla     r12               # 1
   incd    r13               # 1
   cmp     #32, r13          # 2
   jnz     1b
   mov     r15, &key_state
   mov     #4, r15
   sub     &pin_idx, r15
   jmp     4f
3: call    #do_init
4: pop     r9
   pop     r10
   pop     r11
   pop     r4
   ret
```

C INTEL SGX SDK BINARY SEARCH IMPLEMENTATION

In this appendix, we provide the full C source code of the `bsearch` function from the trusted in-enclave `libc` in the official Intel SGX Linux SDK v2.1.2 (`linux-sgx/sdk/tlibc/stdlib/bsearch.c`).

```
/*
 * Copyright (c) 1990 Regents of the University
 * of California. All rights reserved.
 */

#include <stdlib.h>

/*
 * Perform a binary search.
 *
 * The code below is a bit sneaky.  After a
 * comparison fails, we divide the work in half
```

```
 * by moving either left or right. If lim is
 * odd, moving left simply involves halving
 * lim: e.g., when lim is 5 we look at item 2,
 * so we change lim to 2 so that we will look
 * at items 0 & 1.  If lim is even, the same
 * applies.  If lim is odd, moving right again
 * involves halving lim, this time moving the
 * base up one item past p: e.g., when lim is 5
 * we change base to item 3 and make lim 2 so
 * that we will look at items 3 and 4.  If lim
 * is even, however, we have to shrink it by
 * one before halving: e.g., when lim is 4, we
 * still looked at item 2, so we have to make
 * lim 3, then halve, obtaining 1, so that we
 * will only look at item 3.
 */

void *
bsearch(const void *key, const void *base0,
    size_t nmemb, size_t size,
    int (*compar)(const void *, const void *))
{
   const char *base = (const char *)base0;
   size_t lim; int cmp; const void *p;

   for (lim = nmemb; lim != 0; lim >>= 1) {
      p = base + (lim >> 1) * size;
      cmp = (*compar)(key, p);
      if (cmp == 0)
         return ((void *)p);
      if (cmp > 0) {  /* key > p: move right */
         base = (char *)p + size;
         lim--;
      } /* else move left */
   }
   return (NULL);
}
```

Since Nemesis-type IRQ latency attacks exploit information leakage at an instruction-level granularity, we also provide a disassembled version of this function, as compiled with gcc v5.4.0.

```
bsearch:
   push   %r15
   push   %r14
   push   %r13
   push   %r12
   push   %rbp
   push   %rbx
   sub    $0x18,%rsp
   test   %rdx,%rdx
   mov    %rdi,0x8(%rsp)
   je     3f
   mov    %rsi,%r12
   mov    %rdx,%rbx
   mov    %rcx,%rbp
   mov    %r8,%r13
   jmp    2f

/* base = (char *)p + size; lim--; */
1: sub    $0x1,%rbx
   lea    (%r14,%rbp,1),%r12
   shr    %rbx
   test   %rbx,%rbx
   je     3f

/* for (lim = nmemb; lim != 0; lim >>= 1) */
2: mov    %rbx,%r15
   mov    0x8(%rsp),%rdi
   shr    %r15
   mov    %r15,%rdx
   imul   %rbp,%rdx
   lea    (%r12,%rdx,1),%r14
   mov    %r14,%rsi
   callq  *%r13
   cmp    $0x0,%eax
   je     4f                  # cmp == 0
   jg     1b                  # cmp > 0
   mov    %r15,%rbx           # else move left
   test   %rbx,%rbx
   jne    2b

/* return (NULL); */
3: add    $0x18,%rsp
   xor    %eax,%eax
   pop    %rbx
   pop    %rbp
   pop    %r12
   pop    %r13
   pop    %r14
   pop    %r15
   retq

/* return ((void *)p); */
4: add    $0x18,%rsp
   mov    %r14,%rax
   pop    %rbx
   pop    %rbp
   pop    %r12
   pop    %r13
   pop    %r14
   pop    %r15
   retq
```

For completeness, we finally list the source code and dissassemly of the integer comparison function we used in the macrobenchmark evaluation of Section 5.3.

```
int int_comp(const void *p1, const void *p2)
{
    int a = *((int*) p1), b = *((int*) p2);
```

```
    if (a == b)
        return 0;
    else if (a > b)
        return 1;
    else
        return -1;
}
```

```
int_comp:
   xor    %eax, %eax
   mov    (%rsi), %edx
   cmp    %edx, (%rdi)
   je 1f
   setg   %al
   movzbl %al,%eax
   lea    -0x1(%rax,%rax,1),%eax
1: retq
```