Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

On Performing Accurate Time Measurements of SGX Enclave Instructions

Bachelor Thesis

M. Haller

June 6, 2019

Advisors: Prof. Dr. S. Capkun, I. Puddu, M. Schneider
Department of Computer Science, ETH Zürich

Abstract

SGX enclaves allow to securely execute a program on hardware owned by an untrusted entity. Since precise timestamps are not available inside enclaves and not even the OS can manipulate their computations, it is difficult to measure the timings of instructions executing inside enclaves. This thesis serves as a reference on how to obtain accurate timings in general and especially in connection with SGX enclaves. We explain a variety of improvements and new features that we contribute to an existing measurement framework in detail such that the resulting tool can be used for further research. By improving the instruction granular time measurements of SGX enclaves, we show that even more detailed information about the enclave's execution state can be leaked than previously assumed. We demonstrate the capabilities of our measurement framework by exploiting a timing side channel that is only caused by code alignment, which was not detectable with previous measurement methods.

Contents

Chapter 1

Introduction

SGX enclaves should enable someone to securely execute a program on hardware owned by an untrusted entity. Securely means that the third party neither is able to tamper with the computation nor to learn anything new about the executed program or its result. The SGX instruction set is widely available as it is shipped in Intel's standard CPUs, from the generation Skylake onwards. There are many commercial products that use SGX enclaves, for example IBM Cloud [17], Microsoft Azure Confidential Computing [22] and 1Password [13]. Moreover, the research in this area is very active, for instance Tesseract (a real-time cryptocurrency exchange service that uses trusted hardware) [4] or SCONE (secure linux containers using SGX) [3] to only name a few. Since a lot of these works rely on the security of SGX enclaves by building services on top of them, it is important to test and challenge the safety of SGX enclaves.

We will focus on time side channels against enclaves. In this area it was already shown that information can be leaked by exploiting page tables ([27]) or CPU caches ([5], [19]). Nemesis ([7]) contributed a side-channel exploiting interrupts (using SGX-Step, see chapter 2.2). We will build on the latter to leak more information about the microarchitectural state of SGX enclaves by improving and extending SGX-Step to obtain more precise timings.

In this thesis we will first refresh the relevant technical background information on SGX enclaves and explain the SGX-Step framework in sections 2.1 and 2.2. Then, in chapter 3, we show in detail what features we added – examples are other methods to cross-check measurement results (section 3.3) and different plot types to inspect the data from multiple perspectives (section 3.4) – and how we measure instructions. After this we discuss in chapter 4 eleven challenges to make precise measurements inside enclaves and present for each one how we extended SGX-Step to cope with it. Many challenges also provide instructive insights into the microarchitecture of Intel processors. Furthermore, we apply our improved tool in chapter 5 to investigate an interesting behaviour in connection with memory writes. Then we use those insights to exploit a side channel based on code alignment (which in turn changes instruction timings) in section 5.2. To conclude, we summarize our improvements and extensions of the SGX-Step measurement tool (section 6.1) and discuss possible further research directions (section 6.2).

Chapter 2

Background

In this chapter we first explain the threat model of SGX enclaves. Then we briefly introduce the technical details that are relevant for this thesis before we discuss how the SGX-Step framework – the essential building block for measuring instructions inside enclaves – works.

2.1 SGX Enclaves

Intel's Software Guard Extensions, often abbreviated as SGX, are a 'set of extensions to the Intel architecture that aims to provide integrity and confidentiality guarantees to security-sensitive computation performed on a computer where all the privileged software (kernel, hypervisor, etc) is potentially malicious' [10]. The goal of SGX is therefore to enable someone to securely execute a program on hardware owned by an untrusted entity.

2.1.1 Threat Model

SGX Enclaves should isolate their content and computations from the rest of the system, including the operating system itself. The goal is to run trusted code on a remote, untrusted system. This threat model is stronger than usual and allows an attacker to configure or even modify the OS. We will see that the SGX-Step framework takes advantage of this by using special BIOS settings, raising interrupts, modifying the Intel SGX SDK (just for convenience) and even introducing a kernel module.

Iago Attacker. This term was coined in [8] and is used for a malicious kernel that attacks a trusted application that it cannot modify directly. SGX enclaves provide such an execution environment that is protected (through hardware design) from direct access – even from high privileged OS instructions.

2.1.2 Technical Overview

Much of this subsection is based on information from the paper 'Intel SGX Explained' by V. Costan and S. Devadas [10] which does a great job in collecting and summarizing public documentation about SGX.

SGX Enclave. SGX relies on 'trusted hardware'[1] to provide a secure container for the user's computation, a so called *enclave*. Although the untrusted hardware owner loads the initial code and data of the enclave, a malicious configuration would be detected during the *software attestation*. After authenticating the enclave, the user communicates over an encrypted channel which is unreadable for attackers (even for the hardware owner himself). SGX has some reserved memory, that is separated from the normal one and can only be accessed from inside an enclave (i.e. even a malicious OS cannot read this memory).

Software Attestation. Software attestation is the process where a remote entity proves to a verifier that some particular code is running in a genuine execution environment. Without attestation the owner could create a malicious enclave that does not hide its content.

Paging. Part of the enclave-exclusive memory is the *Enclave Page Cache* (EPC). The page content is cryptographically protected to avoid leaks and tampering. However, the untrusted OS manages the EPC and can maliciously evict pages or set the flags of entries to its advantage, which proves to be useful for SGX-Step as we will see in chapter 2.2.

Entry and Exit. The three instructions relevant for this thesis are: *EENTER*, *EEXIT* and *ERESUME*. The first, *EENTER*, prepares the enclave execution and enters it by setting the processor to 'enclave mode'. In the normal case, the enclave is exited explicitly with *EEXIT*. However, if an interrupt arrives while the enclave is executing, an *Asynchronous Enclave Exit* (AEX) is performed[2]. This does not directly exit the enclave, instead it first stores the execution state of the enclave and clears all registers. The AEX also pushes an Asynchronous Exit Pointer (AEP) on the call stack. This allows the ISR[3], which is called to handle the interrupt after the AEX, to resume the execution of the enclave. This service routine uses the AEP to jump to trampoline code (outside the enclave), which then usually calls *ERESUME*. (This section is aggregated from [10], [6] and [23]).

[1]Terminology from [10]

[2]We look at interrupts in more detail in section 2.2.4

[3]Interrupt Service Routine

Debug, Pre-Release and Release Enclaves. An SGX enclave can be built in different modes, which is useful for debugging during code development. Intel describes the differences between those modes in a blog post ([24]), from which we summarize the three that are interesting for us[4]:

1. *Debug enclaves* include debug symbols and can be inspected by an enclave aware debugger. Compiler optimizations are disabled.
2. *Pre-Release enclaves* do not support debugging (no debug symbols are included) and have compiler optimizations enabled. However, the enclave still does not have to be signed and is launched in enclave-debug mode. We explain below, why this is useful.
3. *Release enclaves* are used for final releases. They are the same as pre-release enclaves except that they have to be signed and they run in enclave-production mode.

Pre-release enclaves are useful because release enclaves have to be signed by a valid launch token. Such a token can only be issued by Intel after completing their production licensing process. Therefore, pre-release enclaves are the same as release enclaves, except that they cannot be used in production for remote connections (because users cannot verify their integrity). In particular, the local behaviour (e.g. the performance) of both types of enclaves should be the same.

We also use pre-release enclaves for our tests. We adapted the code of SGX-Step as demonstrated by the code samples from the official linux-sgx repository[5]. That means we added optimization flags (-O2) and do-not-debug flags (-DNDEBUG, -DEDEBUG, -UDEBUG). We also added the option to build production enclaves.

2.2 SGX-Step: Framework for Precise Enclave Execution Control

SGX-Step [6] is a kernel framework that allows a (malicious) host of SGX enclaves to step through the execution of an enclave one instruction at a time. This enables an attacker to observe the execution state after each instruction and therefore facilitates side-channel attacks.

2.2.1 Brief Overview of How SGX-Step Works

The core idea of SGX-Step is to frequently preempt enclave execution, such that it always can execute at most one instruction inside the enclave before

[4]There is also a *Simulation* mode, which does not create a real enclave and therefore is not a good reference for measuring instruction execution times.

[5]https://github.com/intel/linux-sgx/tree/master/SampleCode

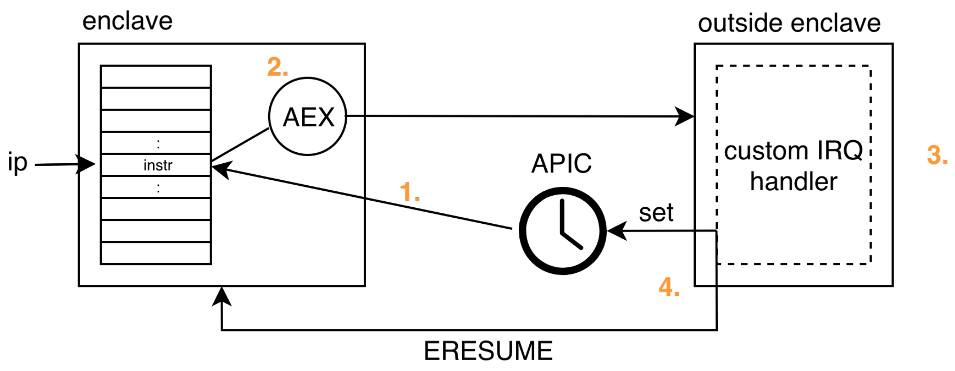


Figure 2.1: Summary of how the SGX-Step framework interrupts SGX enclaves instruction by instruction

it is interrupted again.

SGX-Step uses the APIC[6] to schedule those fine-grained interrupts in one-shot mode. This mode delivers a single interrupt after some configurable time interval. The time resolution is bounded by the CPU's bus frequency, which is slower than the CPU's frequency. More accurate APIC modes exist, but they are harder to control from user space (which is important in order to get precise timings as we will see in section 3.2.1).

SGX-Step comes with a library and a patched kernel driver, which simplifies the configuration of the APIC. Moreover, the driver enables the registration of a custom AEP, which can be used to add custom code that is executed every time the enclave is interrupted.

Figure 2.1 summarizes and simplifies the procedure that SGX-Step follows when executing a single instruction[7]. We assume that the enclave is already running and currently executes some instruction (*instr* in figure 2.1 as indicated by the instruction pointer *ip*).

1. APIC timer interrupt arrives while the enclave is executing
2. An AEX is performed to exit the enclave
3. The modified kernel module returns the control flow to custom attack code
4. A new APIC timer is set and the enclave is resumed

[6]Advanced Programmable Interrupt Controller; A device, which is local to each core and can be configured to schedule and deliver interrupts.

[7]I aggregated and simplified the six steps explained in the original paper [6]

2.2.2 APIC Timer Interval

The remaining challenge is to set the APIC timer such that we single-step through the code running inside an enclave. When we set it too long, we could execute more than one instruction before the interrupt arrives (an event referred to as *multi-step* in [6]). But if we set the timer too short we have *zero-steps*, which means we make no progress at all, because we never start executing an enclave instruction and therefore make no progress. The perfect timer interval is platform-specific, since it depends on the architecture (e.g. the CPU frequency and how long ERESUME and restoring the enclave takes).

The right APIC timer interval can be found empirically. In the SGX-Step paper [6] they generate a program only consisting of *nop*s[8] because those are among the fastest instructions. Therefore, we choose the largest timer interval such that all *nop*s are still reliably detected. This way no multi-steps should happen (since other instructions take at least as long as a *nop*), while the number of zero-steps is small. However, some of the latter will still occur (in our tests it was usually less than 0.1%) bit this does not matter since, as we explain next, we can detect and filter them.

2.2.3 Filtering Zero-Steps

We have seen in section 2.1.2 that the OS takes care of caching the enclave's page tables. Because of the strong adversary model of SGX, the OS can be malicious and manipulate the *access bits* of unprotected Page Table Entries (PTEs). As its name suggests, that bit signals whether a page was accessed or not. This is used for page replacement policies to approximate if a page was used recently.

SGX-Step sets the access bit of the page that contains the enclave's code to 0 (not accessed) on every interrupt. After at least one instruction has been executed, the access bit is set to 1. However, in case we interrupted before the enclave started (or continued) to execute code, the bit will still be 0 and thus we know that this was a zero-step.

Note that we cannot detect multi-steps just by looking at access bits, because it is indistinguishable if one or more than one instruction (of the same page) were executed. However, in the instruction measurements we present in section 3.3, we previously know how many instructions will be executed; Thus we can make sure that SGX-Step does not multi-step by checking whether the number of steps performed matches the expected number of instructions.

[8]No Operation; an instruction that uses no execution unit and does not perform any action. This is useful for synchronization, to align instructions or as a place-holder.

2.2.4 Interrupt Handling and Exceptions

Since SGX-Step interrupts the enclave for each instruction, it is important to understand how and when exactly interrupt handling is done.

Interrupts. In Intel's developer's manual [14] interrupts are generally defined as 'an asynchronous event that is typically triggered by an I/O device'. When an interrupt arrives, the processor pauses execution and switches to a fault handler that is registered for this specific interrupt. When this handler is done, execution resumes to the code that was running when the interrupt arrived. ([14]).

As we mentioned in section 2.1.2, when an interrupt arrives during the execution of an SGX enclave, an AEX is performed before calling the interrupt handler. What is not clarified in the SGX-Step paper is how an APIC interrupt, whose precision is only multiple CPU cycles, manages to reliably single-step instructions that just take one CPU cycle. Our speculative attempt to explain this is the following: Interrupts are usually only handled after an instruction finished and not immediately when they arrive. Additionally, it would make sense that the enclave starts with an empty pipeline. Together, if we assume that the interrupt is not handled until the instruction exits the pipeline, this would give the interrupt a window of the size of the pipeline's depth in which it has to arrive. This means it does not have hit at exactly the right cycle to interrupt an instruction that is processed in one cycle.

Exceptions. Exceptions are defined as 'a synchronous event that is generated when the processor detects one or more predefined conditions while executing an instruction' [14]. They are handled very similar as interrupts.

Because single-stepping has a large overhead, SGX-Step can use exceptions to only interrupt the enclave when specific code pages are executed. To do this, it registers a fault handler for segmentation faults and marks the first code page that should be observed as not executable. When the enclave tries to execute the first instruction of that page, a segmentation fault (because of the access violation) is raised. This traps into the registered fault handler which marks the page executable again. After that the custom AEP is executed and sets the APIC timer interval for the next interrupt and thereby initiates single-stepping.

Chapter 3

Measurement Setup and Methods

In this chapter we start by explaining our test environment in detail, then we discuss how we obtain precise instruction timings inside and outside enclaves. Furthermore, we introduce three different measurement methods; one for instructions outside the enclave, the interrupt method of SGX-Step and lastly one that does the whole time measurement from inside the enclave. Finally, we explain the different plots that we produce to analyse instructions.

3.1 General Setting

Hardware. We tested on two devices. The first one is a Dell Latitude E5470 laptop with a dual-core Intel(R) Core(TM) i7-6600U CPU @ 2.60 GHz. The second device is an Intel Skull Canyon NUC6i7KYK mini PC with a quad-core Intel(R) Core(TM) i7-6770HQ CPU running at 2.60 GHz. Although both processor models support SGX, they are not in the list of supported processors of SGX-Step. However, this just means that we had to find the right APIC timer intervals for our processors ourselves (as described in section 2.2.2). We discuss the special BIOS settings and kernel parameters we used in section 3.2.2.

Software. We tested on two different versions of Ubuntu. The laptop runs Ubuntu 17.10 with the Linux kernel version v4.13.0. Note that this Ubuntu version is also not compliant with the SGX-Step recommendations. To be able to run the framework, we had to use an older compiler version (gcc-4.8), because the newer version reports problems with position dependent code in some handwritten assembly files. Those are essential for SGX-Step to resume the enclave and rewriting them only leads to other problems. The Intel NUC runs Ubuntu 16.04 (the same version the authors of SGX-Step used) with a Linux kernel version v4.15.0.

3.2 Precise Time Measurement

We want a precise measurement routine that produces reliable results in the presence of superscalar and out-of-order execution with as little noise as possible. To achieve this, SGX-Step already reduces the code between starting and stopping the timer as much as possible and uses specific kernel and BIOS options. We add a better instruction serialization to prevent non-deterministic overlapping of the measured code with instructions before and after.

3.2.1 Time-Stamp Counter

In order to have reliable time measurements, we want to minimize other code that runs between the time sampling. For this reason, SGX-Step includes a kernel module that enables us to set the APIC from user space. This way, we can avoid a context switch before the enclave is resumed. We cannot avoid measuring the code that resumes the enclave and restores its execution context (since we cannot modify it without invalidating the enclave). Therefore we sample the time right before ERESUME and right after AEX.

Due to superscalar and out-of-order execution it is important to serialize the instruction stream before and after starting the time measurement. Otherwise instructions before the start timer could overlap (i.e. be in the pipeline at the same time as) instructions after it. This could non-deterministically slow down the measured code, because depending on the execution state before starting the measurement, more or less additional instructions are measured. In other words, measuring the same code twice could have inconsistent results due to the noise from other code (that can be different in the different measurements) that is being executed in parallel.

mfence. In SGX-Step they use the instruction *mfence* (memory fence) before starting the timer. However, according to the Intel manual [15] *mfence* 'does not serialize the instruction stream'. It only guarantees that the effect of every load and store operation before *mfence* becomes visible prior to the execution of memory operations after it. This means *mfence* has no effect on instructions that do not involve memory (except for other fences and serializing instructions), so those could still overlap with our measurements.

cpuid and rdtscp. According to Intel's white paper on 'How to Benchmark Code Execution Times' [20], the best way to serialize instructions is to use *cpuid* and *rdtscp* in the way we summarized in code snippet 1. To understand why this is a smart way to measure time, we have to first know what those instructions do. Although *cpuid* is actually an instruction for CPU identification (it identifies the processor and returns information about features), we only use it here because it 'can be executed at any privilege level to serialize instruction execution' [15]. There are no other unprivileged instructions that really serialize the whole instruction stream. The instruction *rdtsc* 'reads the current value of the processor's time-stamp counter' [15]. While *rdtscp* basically does the same, it additionally waits until all previous instructions are finished (because it reads the processor's ID). Now we can explain the code snippet: First, on line 1, we wait until all previous instructions have terminated, then we read the current time stamp and store it (lines 2-3). After the code we want to measure, we use *rdtscp* (line 5) instead of *rdtsc*, because this waits until all previous instructions are finished. This is desirable because then all operations that we want to measure are actually done and the time stamp is not read in parallel to the last of them finishing. It is also better than to use an additional *cpuid* before line 5, because this instruction has quite high variance itself, which would propagate into the measurement. On the last line, we use cpuid again to prevent later instructions to already start executing while the time stamp is read and thereby slowing this operation down. Note that storing the time stamp on line 6 cannot interfere with *rdtscp* because it has a read-after-write dependency[1].

Code Snippet 1 Code benchmarking with *cpuid* and *rdtscp*

```
1: cpuid
2: rdtsc
3: Store timestamp
4: ⟨Measured code⟩
5: rdtscp
6: Store timestamp
7: cpuid
```

lfence and sfence There are execution contexts in which *cpuid* and *rdtscp* are not available, e.g. in SGX enclaves. In such cases, we can use an *lfence* before *rdtsc*, which is also the recommendation of the Intel manual [15]. In our measurements we obtained the best results by using both *lfence* and *sfence* together before *rdtsc* and also instead of *cpuid* on line 7 to protect against later instructions. A more detailed explanation with plots that show why we use this combination compared to a single *mfence* can be found in appendix A. We use this measurement methodology for SGX enclaves in the counter method presented in chapter 3.3.4.

[1]Storing the timestamp requires reading the two registers *edx* and *eax* that *rdtscp* writes to, so that has to wait until *rdtscp* is finished anyways.

3.2.2 Reducing Noise

The most obvious solution to deal with noise is to do multiple measurements and then analyse the statistical properties of the results. We found that repeating each measurement 100'000 times is a good reference point, since doing more tests only takes longer (after all, we perform an enclave entry and exit for every single instruction) but does not exhibit a different behaviour regarding the observed empirical distribution of the measurements. We will explain in chapter 3.4 the different methods that we used to visualize the results.

To make execution times more predictable, we set the following kernel parameters to disable different performance optimizations and other updates, following the suggestions of the SGX-Step paper [6]:

1. *isolcpus=1*: This parameter isolates the CPU core number 1 from the general scheduler. This means that no process will be scheduled to this core, except if you manually affinitize it. We do this in the measurement preparation for the process that creates and runs the enclave. That means the observed code should run alone on a core without any interference from the scheduler.

2. *dis_ucode_ldr*: This disables the microcode loader. We do this because microcode updates to protect against foreshadow [25] affect the duration of ERESUME significantly. Setting this parameter reduced the mean number of cycles required for a single step of SGX-Step by approximately 25% on our machines.

There are three important parameters in the boot menu configurations that SGX-Step [6] and Nemesis [7] advise to set: We disabled TurboBoost, dynamic frequency-scaling (C-States, SpeedStep) as well as HyperThreading. Additionally, SGX-Step sets the P-States[2] maximum and minimum to the same fixed value before executing the enclave.

Note that all modifications from above are compliant with the threat model described in section 2.1.1, because the OS is considered to be malicious. For consistency, we take the same actions for all measurement methods (described next in 3.3).

3.3 Measurement Methods

In this subsection we present how we used the insights about precise time measurements that we have seen so far in this chapter to build multiple instruction measurement methodologies for different applications. Before

[2]P-States optimise the voltage and CPU frequency while the core running, which could introduce jitter.

that, we first provide some background information on how we generate test cases and introduce terminology that will make it easier to explain the measurement methods afterwards. Then we start in subsection 3.3.2 with the first measurement method: Measuring instructions outside the enclave, which is simple to do but nonetheless essential for detecting different implementations of instructions inside and outside enclaves. The other two approaches both measure instructions inside the enclave. The first one, described in 3.3.3, uses SGX-Step to single-step through the enclave's code and take the time from before ERESUME to after AEX. In subsection 3.3.4 we explain the last method, which uses a counter in shared memory that is incremented by a different process. The measurement is performed completely inside the enclave by sampling this counter before and after executing an instruction.

3.3.1 Background Information and Terminology

In this subsection we introduce the general structure of our measurements. We call a *test case* the observation of a single instruction, this can involve instructions to prepare the measurement or multiple repetitions of the measured instruction (the interrupt method in 3.3.3 does multiple measurements by completely unrolling the loop). However, different instructions are never measured in the same test case. It is necessary to directly define the assembly code for our test cases because otherwise the code could be translated to more instructions than intended[3]. Since we interrupt for every instruction, it is critical that we know how exactly the assembly code of our test case looks like (see also challenges 4.6 and 4.7). We will always use the AT&T syntax when we mention assembly instructions (i.e. with two operands the syntax is *instruction source, destination*).

For convenience, we introduce the following terminology for the rest of this thesis. We distinguish three different types of instructions in a test case:

1. The *test instruction* is what we actually want to measure. Usually, this is the only instruction type of which we log the timings[4].
2. *Initial instructions* are only executed once per test case, for example to move a value to a register that is never overwritten.
3. *Prepare instructions* are repeated every time before the test instruction. This can be necessary to restore the execution environment for obtaining consistent results over multiple measurements of the same test instruction.

[3]This also applies to inline assembly since the compiler might add code, for example to save and restore registers

[4]This is possible, since we know how many instructions there are of each type and we single-step through them, we can keep track of the current position in the code

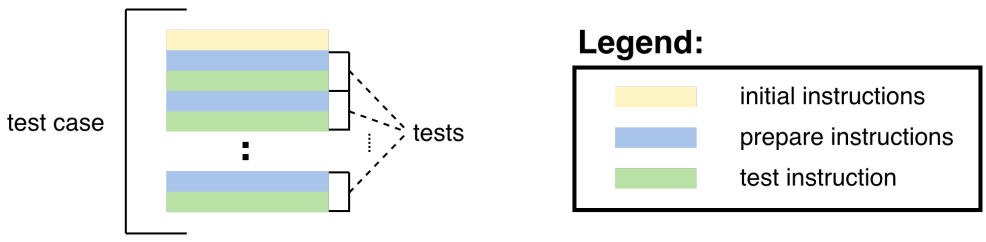


Figure 3.1: Visualization of a test case consisting of multiple tests and what type of instructions they contain.

We further divide a test case into single *tests*, which consist of zero or more prepare instructions and one test instruction. A test case can contain many repetitions of the same test, which is summarized in figure 3.1. Finally, an enclave can contain multiple test cases to perform and compare many different test instructions in the same enclave.

All three measurement methods that are presented in the rest of this section can use the same specification of instructions – grouped by the types presented above – to measure a test case. However, some methods (have to) organise the test cases differently, as we will see next.

3.3.2 Outside Enclave

Outside the enclave, we can use the optimal time measurement with the *cpuid* instruction as explained in paragraph 3.2.1. Getting the timing is straight forward as the right side of figure 3.2 shows: We first execute the initial and normal prepare instructions, then we start the timer and execute the test instruction before we sample the timer again. We repeat this in a loop until we have enough measurements and continue with the next test case. We have to include initial instructions in the loop because we cannot guarantee that the architectural state is preserved from one loop iteration to the next. Initial instructions might set some flags or use registers that are overwritten by the stop timer, the loop counter or the loop condition check. Therefore, initial instructions effectively become prepare instructions for this measurement method.

3.3.3 Interrupt Method

Measuring instructions inside enclaves is more complex, because neither *rdtsc* nor *cpuid* are available. The interrupt method approaches this problem by measuring the time outside, entering the enclave and allowing it to only execute a single instruction before exiting again. The best way to construct test cases is to explicitly repeat each test instead of using loops, which introduce more instructions (especially jumps) that have to be tracked carefully during single-stepping. Completely unrolling the loop makes it easier to measure the correct instructions (instruction tracking is discussed in more detail in chapters 4.5 and 4.6). Figure 3.2 visualizes this approach on the left side and shows the difference to other methods.

Example Time Measurement. In detail, a time measurement with the interrupt method involves the following steps:

1. Mark first page of the observed code not executable
2. Start the enclave and let it run
3. The enclave traps to our custom fault handler when it tries to execute the not executable code section
4. Mark this page as executable again and set the APIC timer before resuming the enclave
5. Right before ERESUME, sample the processor's time stamp and save it
6. The enclave resumes executing instructions
7. Scheduled APIC interrupt arrives
8. The enclave performs an AEX, we sample time stamp as early as possible
9. Log the time difference between the two time stamps and the access bit to memory, filter the data later
10. Set the APIC timer again and continue from 5. until all instructions from the enclave were executed
11. Parse the data, filter out zero-steps, initial and prepare instructions before saving the remaining data to a file

3.3.4 Counter Method

To cross-validate the results of interrupt based measurements, we created the counter method. We basically implemented our own time stamp that can be read inside the enclave (unlike the processor's time stamp). This enables us to perform the measurements inside the enclave without having to interrupt and exit/resume it. Note that this takes advantage of our control over the enclave's code. SGX-Step on the other hand can also be used to single-step code that we cannot change. For this reason, the counter method is unlikely to be useful in an attack scenario, however it can still be used for to validate the interrupt method.

We implement the counter in a different process which just keeps increasing a variable in a shared memory location by one. It is affinitized to another

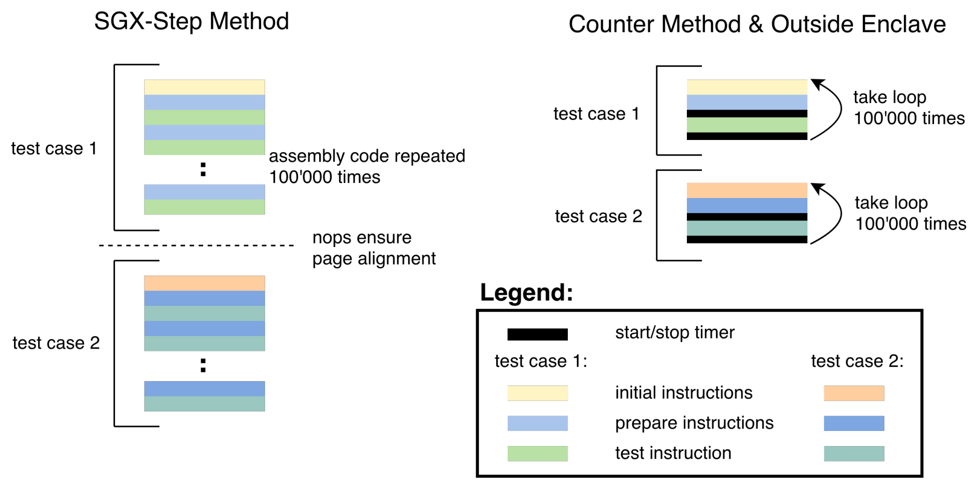


Figure 3.2: Visualization of the different structure of test cases for the Interrupt method compared to outside the enclave and the counter method.

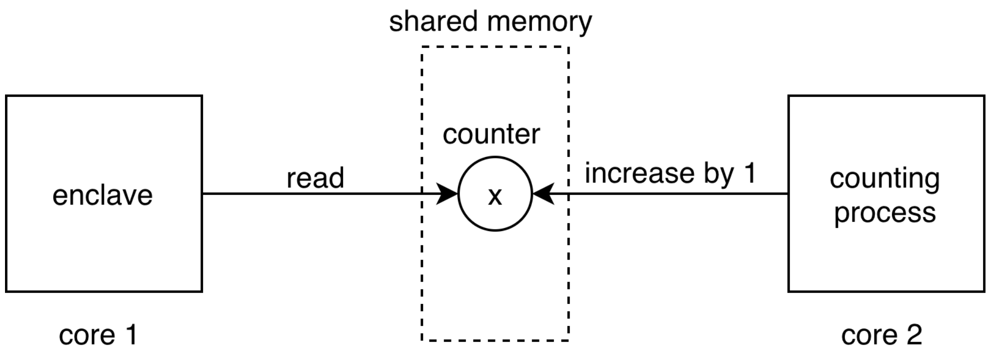


Figure 3.3: Implementation of the counter method with two processes: One running the enclave, reading the time stamp before and after executing the test instruction and the other increasing the counter in an infinite loop.

isolated (physical) core to make sure that it is not affected by scheduling and does not interfere with the enclave. The counter is placed in shared memory so that the enclave can access it as well, which is visualized in figure 3.3. Identical to the outside method (cf. 3.3.2), we use a loop for measuring (see figure 3.2). But instead of using *rdtsc* (which is not available inside enclaves) we sample our counter before and after we execute the test instruction. Additionally, to replace *cpuid*, we have to use the combination of fences that was explained in paragraph 3.2.1 to serialize instructions.

To optimize the accuracy of the counter, we implemented it directly in assembly because even inline assembly generates some unnecessary *mov* instructions. Additionally, we use loop unrolling to reduce the impact of the jump;

we execute one hundred *add*s in the loop body before we do one jump back to the start. The counter has to be stopped by killing the process, since the loop has no condition (which further reduces the number of instructions).

Counter Accuracy. We measured the accuracy of this counter by sampling it and the processor's time stamp before and after calling the enclave. This allows us to compare the difference of our two counter values with the number of cycles that were actually executed. We observed that our counter catches approximately 18.5% of all cycles on the NUC. This precision means that we can only measure instructions on a five cycle granularity. We argue that this not because the counter is in shared memory but rather seems to be a fundamental limitation. To show this, we consider two different ways to implement unshared counters: The first one, shown on the left side of figure 3.4, increases a counter in unshared memory. The second one, visualized on the right side of the same figure, increments a register. As before, we let those two counters run for a certain amount of time and control how many cycles they caught, i.e. how many times they were incremented compared to the number of processor cycles that were actually executed (which we obtain with *rdtsc*, see 3.2.1). The register counter caught 99.5% of all cycles in our test, which shows that the loop unrolling and implementing the counter in assembly generally enables us to build a counter with high accuracy. In other words, we can perform enough *add* operations to increase the counter almost at CPU frequency. The counter in unshared memory caught only 18.5% cycles, which is exactly the same as the shared counter. The reason is probably that the enclave only reads the shared counter and never writes it, thus the cache coherency protocol does not cause much additional effort. It is clear that we cannot share registers between cores, i.e. we cannot use the register counter for our counter method. Therefore we conclude that the counter accuracy cannot be improved since writing to memory (including all protocols that need to be followed to ensure consistency) is the underlying bottleneck.

3.3.5 Overview over all Measurement Methods

Table 31 summarizes the three measurement methods that were presented in this section: The one outside enclaves, the interrupt and the counter method.

3.4 Plot types

In this section we will explain the four different plots that proved to be useful: A simple histogram of the timings of test instructions, one that additionally shows the prepare instructions, a plot that shows the mean of different sequences of instructions, and one that shows single measurements over

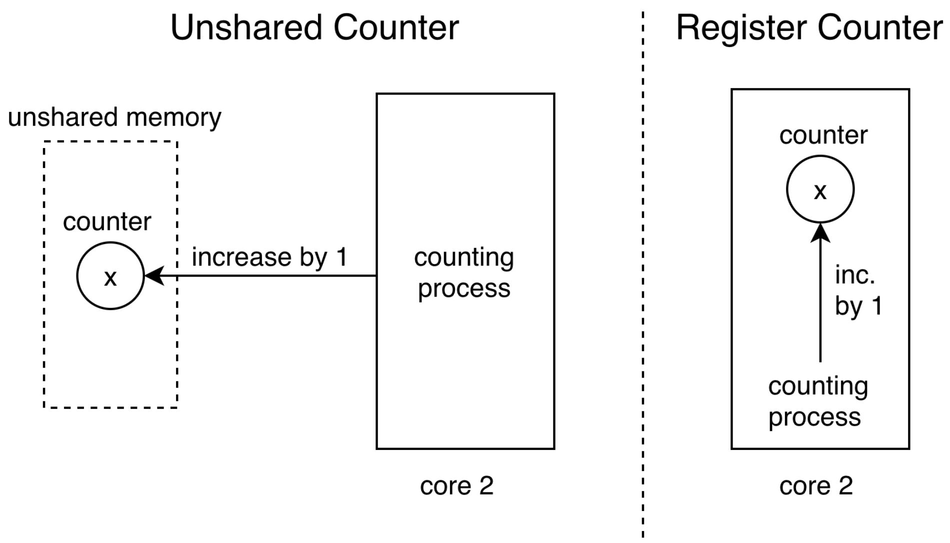


Figure 3.4: Visualization of the comparisons for the counter method: The unshared counter value resides in unshared memory, while the register counter only increments a register and does not access memory.

Table 31: Overview over all measurement methods

	Outside Enclave	Interrupt Method	Counter Method
Serialising Instruction	*cpuid*	*cpuid*	*sfence*, *lfence*
Additionally captured in the Measurement	*movs* to restore registers after first *cpuid*	ERESUME, AEX and some *movs* to save registers before *cpuid*	overlapping non-memory operations
Timestamp	processor	processor	shared variable incremented by a counter thread
Special	instruction timings outside enclaves	can be used even if we cannot control the enclave's code	measurements directly inside enclaves

time. This section is intended to serve as a reference of what plot types are available, applications of them will follow in chapter 4.

But before we start, a few words on how the data is processed. The C program that runs the test (e.g. in the interrupt method it runs the enclave and performs the single-stepping) logs the measured instructions to log files. We use a Python script to post process this data with the Matplotlib module. Except for the cycles-over-time plot, we always filter points that are further away from the mean than three times the standard deviation[5] to get more compact plots. If filtering was used, we mention the percentage of filtered points on the bottom right of each plot. In the legend of the plot, we indicate the instruction and other relevant information (operands, prepare instructions, flags). We also state the mean (μ) and the standard deviation (σ) for each measurement.

3.4.1 Histogram of Test Instructions

The histogram is the plot type that we will use most of the times to represent our measurements. Figure 3.5 shows it for the three very different instructions *fscale*, *lfence* and *rdrand*[6]. On the x-axis we have the number of cycles (grouped into bins) that were measured for the instructions and we have the amount of measurements that each bin contains on the y-axis. The number of bins that were used are always mentioned in the label of the x-axis. In our case there is no disadvantage in choosing small bins[7] since the noise is normally distributed and only affects the cycle count. That means it does not produce large single outliers on the y-axis that would have to be filtered out and therefore it is advisable to choose small bins[8] so that we retain most information of the measurements. The dotted line around each histogram shows the normal distribution with the measured mean and standard deviation to give an idea of how closely the measurements follow a normal distribution.

Since we also measure enclave entry and exit with the interrupt method used in figure 3.5, the mean of instructions that only take one cycle (like the *lfence*) is multiple thousand cycles. It proved to be hard to read off how many cycles an instruction needs (because of the challenges described later in 4.1 and 4.10). Therefore we have to look at the relative distance of instructions; in figure 3.5 we can see that the instructions are clearly separated; *rdrand* is the slowest, while an *lfence* that has nothing to serialize (since there are no load instructions around it inside the enclave) is more than 350 cycles faster.

[5]We will see that our data follows a normal distribution, therefore this only cuts away

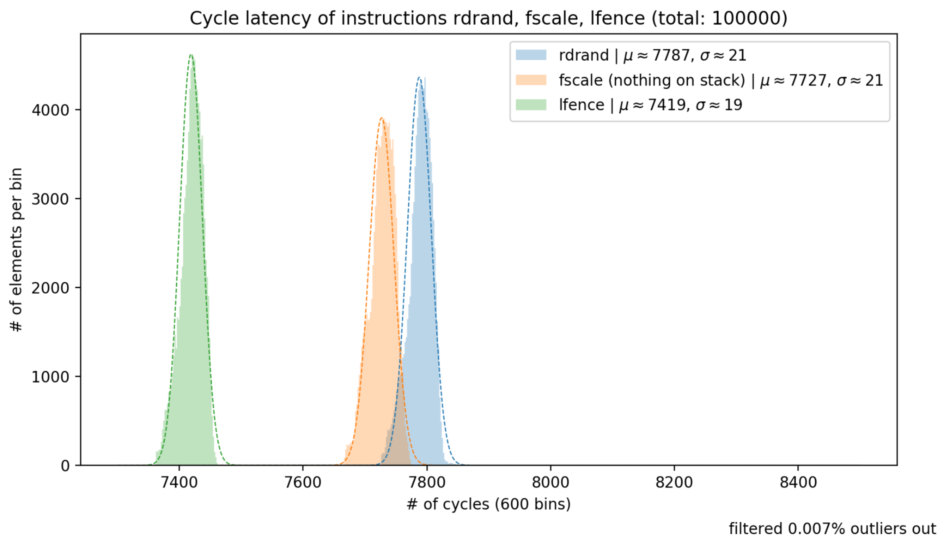


Figure 3.5: Histogram plot of the three instructions *fscale*, *lfence* and *rdrand* measured on the NUC.

3.4.2 Histogram of Prepare and Test Instructions

While we usually only need to see the histogram of test instructions, sometimes essential information can be hidden in the prepare instructions. Figure 3.6 gives a sneak preview of such a case, but we have to defer the explanation to chapter 5.1. The test instructions are displayed in the lower plot. The legend explains that both test cases essentially measure the same sequence of instructions: The first case tests *movq %rcx, -8(%rsp)* with *test %rax, %rax* as prepare instruction. The second one tests the same *movq* instruction, but it measures only every second *movq* since one is in the prepare instructions (which are a single *movq*, surrounded by two *test* instructions). Therefore, since the instruction streams follow the exact same pattern, it is not trivial to understand why the first test case has two clearly separated peaks while the second one has only a single peak. Plotting the prepare instructions provides some previously hidden insights: The violet prepare instruction (prep 1 of the second test case) is the *movq* that is always not measured in the second test and we can clearly see that this is the second peak that was previously hidden. It surely is still very surprising that every second *movq* is slow and we will discuss this in depth in chapter 5.1.

around 0.3% of all data points

[6]Nemesis [7] did the same test and has a similar representation

[7]The height of too small bins depends on noise in some scenarios, while to large bins loose information ([1])

[8]We usually chose them such that they are only two cycles wide

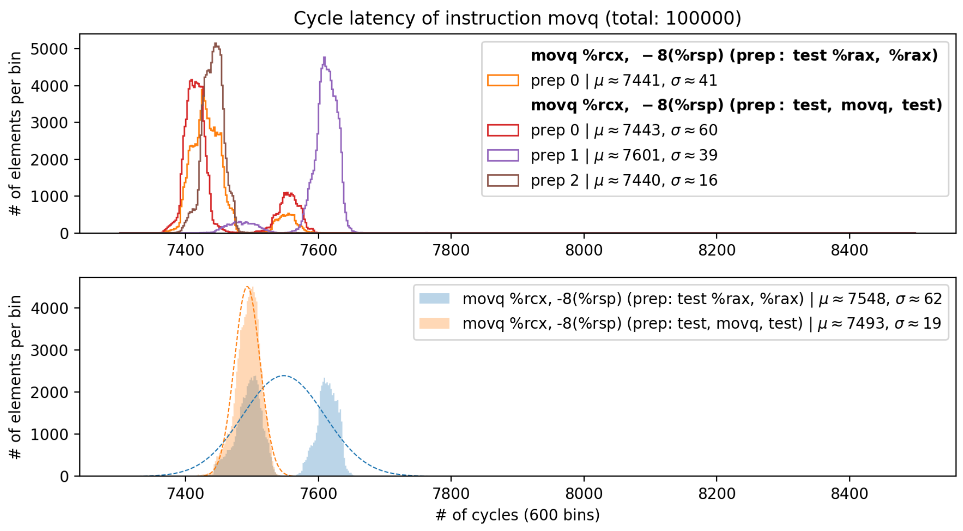


Figure 3.6: Histogram plot (showing also prepare instructions) of *movq* to stack with an independent *test* prepare instruction, measured on the NUC.

3.4.3 Measurements-over-Time Plot

The measurement-over-time plot type provides a more detailed view on unprocessed measurements: It shows how many cycles single test instructions have taken, i.e. here the x-axis shows the tests and the y-axis the corresponding number of cycles that were measured. The goal is to show patterns in the measurements (we discuss some in the challenge 4.2 and section 5.1). Figure 3.7 shows the same test as figure 3.6 in the previous chapter 3.4.2 and we see again that every second measurement is slower. The first test measures all instructions and thus has a double peak in the histogram plots. There are also in the aforementioned figure 3.7 two clearly separated groups of points for the first test case: One slightly above 7600 cycles and one around 7500. The second test case only measures every second *movq*, which means it only measures the fast ones.

We have to plot less instructions in this plot type and disable filtering because otherwise it would be hard to see patterns: The graphs would show too many data points or the filtering would shift patterns (e.g. instead of every even measurement being fast, after filtering one out, every odd one would be fast). A disadvantage of this detailed plot type is that it can be confusing because it also shows all outliers, for example on the last 50 instructions of this plot (we will discuss further what happens there in chapter 4.2).

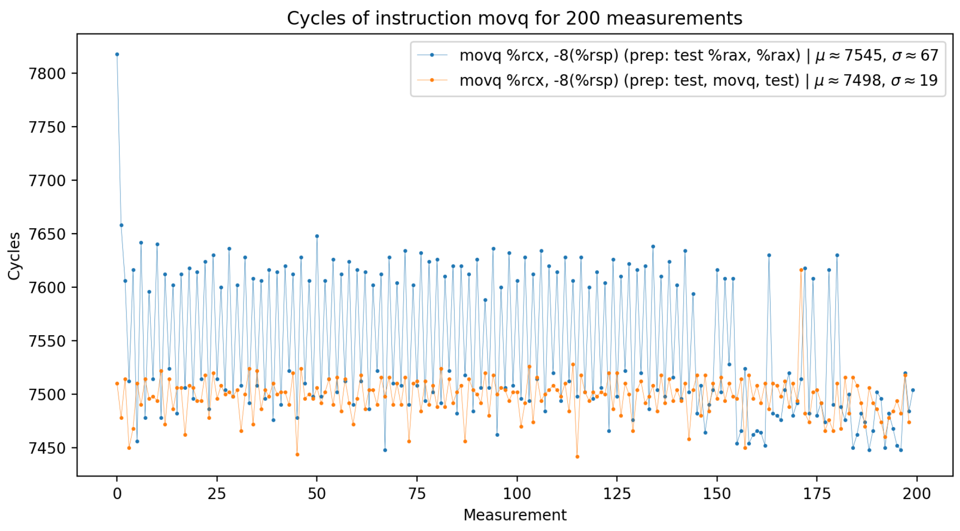


Figure 3.7: Plotting single measurements over time of *movq* to stack with an independent *test* prepare instruction, measured on the NUC.

3.4.4 Bar Plot

The bar plot is useful to visualize patterns that occur with slower and faster executions of the same instruction. We can, unlike the measurements-over-time plot from subsection 3.4.3, use this plot type for arbitrarily many instructions. Figure 3.8 demonstrates the bar plot on our running example of the double peaks when we measure *movq* with an independent *test* as prepare instruction that we already saw in the previous sections 3.4.2 and 3.4.3. The x-axis is grouped by periodicity. For each group, we calculate the mean of all possible instruction sequences: For example, for the period three, we measure three disjoint sequences of instructions: $0 + k * 3$, $1 + k * 3$ and $2 + k * 3$ for $k \in \mathbb{N}$. This means that every period p has exactly p bars, one for each possible offset. In figure 3.4.3 we clearly see at period two that every even test instruction is slow and every odd one is fast. In our tests we saw more complex patterns (e.g. only two of 16 instructions are slow) that were hard to notice in the measurements-over-time plot but are clearly visible with the bar visualization.

The maximal period is configurable, however in most cases the it is not larger than 16. As a guideline, the correct period usually shows the highest bars and all multiples of it repeat the same pattern. In figure 3.8, the period is two and it is repeated at every even period, while for the odd ones the difference between the instructions average out since the same number of slow and fast *movq* are measured.

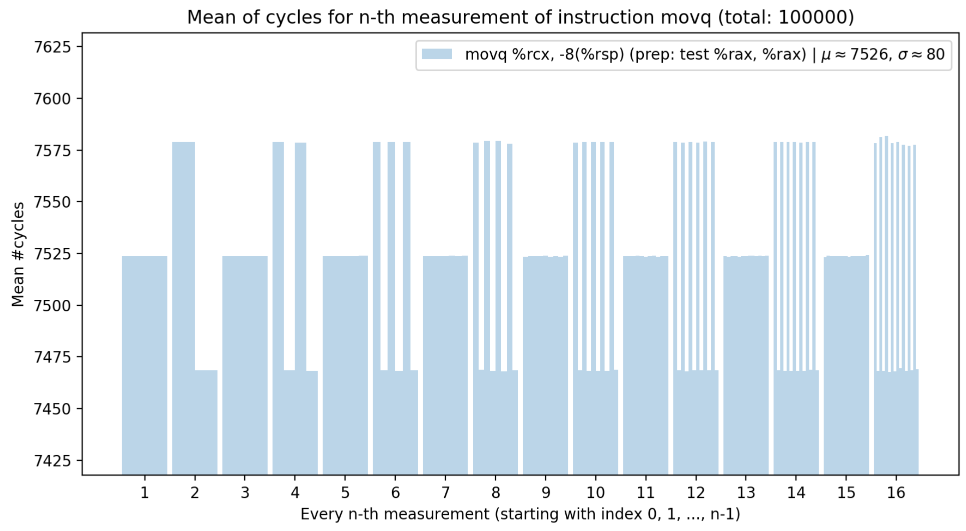


Figure 3.8: Bar plot of *mov* from register to memory with *test %rax, %rax* as prepare instruction.

Chapter 4

11 Challenges Towards Precise Measurement

This section discusses eleven challenges that we encountered while measuring instructions inside SGX enclaves. These challenges have to be taken into account to obtain precise enough timings, but beside this technical aspect, we think a thorough understanding of them helps to shed light upon the complexity of modern Intel processors. Some insights of this chapter might give rise to complex side channels that leak specific details of the microarchitectural state. We discuss such an attack, including a proof of concept, in chapter 6.

4.1 Challenge 1: Incomparability of Different Enclaves

This challenge discusses our observation that latencies for ERESUME and AEX[1] are different from enclave to enclave, even when we run the same code on the same machine. In the following we show that this can be very misleading when comparing measurements which were run in different enclaves. We will see that this problem is solved by simply running all test cases in the same enclave.

We look at the comparison of division with different operands. The first plot in figure 4.1 is the original one from the Nemesis paper [7]. It shows the *div* instruction for a fixed divisor (0xffffffffffffffff) and different 128 bit dividends (the upper 64 bits are stored in *%rdx*, the lower ones in *%rax*). The authors concluded that 'the average interrupt latency clearly increases as the dividend becomes larger'. The source code of Nemesis[2] shows that they measured the four dividends in different enclaves. This is problematic as the middle plot of figure 4.1 shows: This is a measurement that we did on

[1]ERESUME and AEX were described in chapter 2.1.2

[2]Which is published on https://github.com/jovanbulck/nemesis

our laptop[3] but with the source code of Nemesis. It shows a completely different ordering; the test with the largest dividend is now the second fastest instruction, while the second smallest dividend is the slowest one. The reason for this is that the time needed for enclave entry and exit seems to differ for enclaves and this shifts the plots in an unpredictable manner. The last plot in the same figure shows yet another order of the same instructions measured with Nemesis on our laptop. Figure 4.2 supports the point that different enclaves are incomparable by showing four *nop* instructions that were measured with the code from Nemesis in four different enclaves. Although those operations are completely identical, their measurements are shifted on the x-axis.

The simple solution to this problem is to run all tests inside the same enclave. Section 3.3 already explained in detail how we implemented this. Figure 4.3 shows our plot of the almost identical[4] *div* test, which consistently shows the same relative distance between different enclaves (of course, the whole pattern can still be shifted by the differences in enclave entry and exit). There are two clearly separated peaks: One for the instructions that have zero in *%rdx* and one for the others. In fact, we did other tests which showed that an instruction is already in the second peak if it has 0x1 in *%rdx*. We also changed the code of Nemesis to measure all instructions in the same enclave and the results were consistent with ours.

> *Conclusions from challenge 1:* Comparing instruction measurements is only consistent if they were all performed in the same enclave.

4.2 Challenge 2: Measuring Across Page Borders

The enclave code size can get considerably large because we have to add all test cases to the same enclave (cf. 4.1) and for each case we repeat its test instruction many times (sections 3.2.2 and 3.3). Therefore, it usually has to span across many pages, which introduces two problems: Firstly, SGX-Step only handles code contained in one page, secondly, the measurements at page borders are significant outliers. We will discuss how to handle those issues in this section.

Since SGX-Step always checks the accessed bit of the code's page to determine if an instruction was executed or not, it is limited to one page of code.

[3]Note that since this is a different device than they used for Nemesis, the whole measurements are shifted on the x-axis (our device is slower)

[4]We replaced the largest dividend of Nemesis (0x0fffffffffffffff in *%rdx*) with 0xefffffffffffffff, because this is actually the largest possible value, since for a larger value, the result when dividing by 0xffffffffffffffff overflows and throws an exception

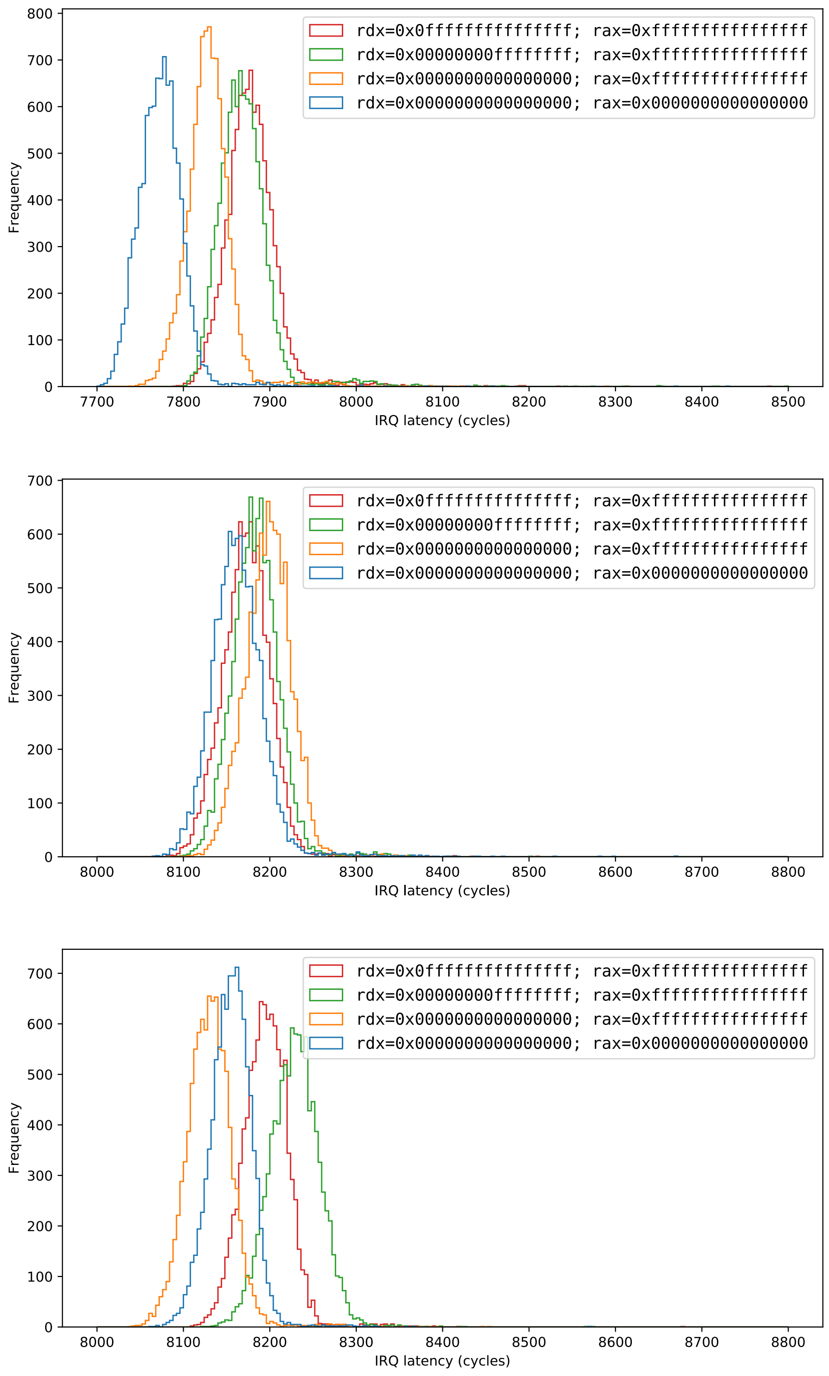


Figure 4.1: All plots show the x86 *div* instruction, with different dividends (stored in *%rdx:%rax*) and the fixed divisor 0xffffffffffffffff. The first plot is from the Nemesis paper [7], the second and third plots were measured on our laptop using the source code of Nemesis. They both show a completely different relative order of instructions than the first plot.

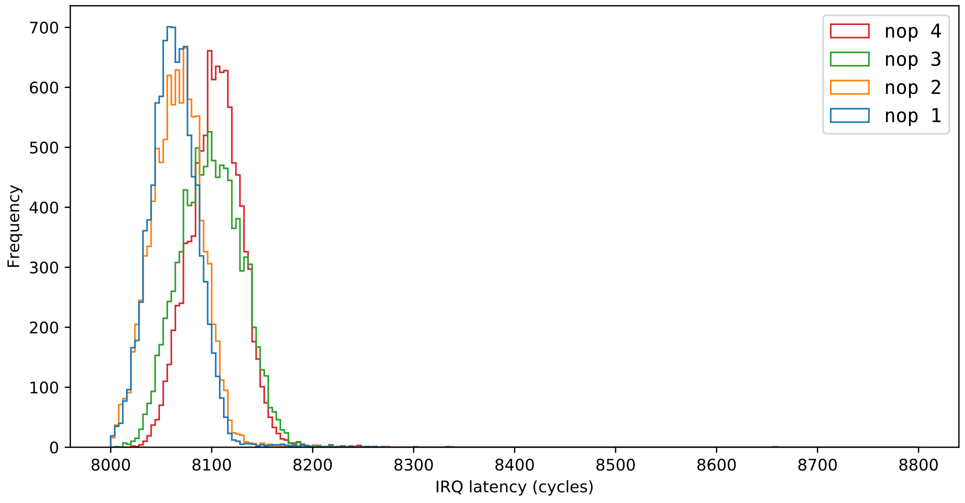


Figure 4.2: Measurement of four *nop* instructions with the code of Nemesis [7] on our laptop.

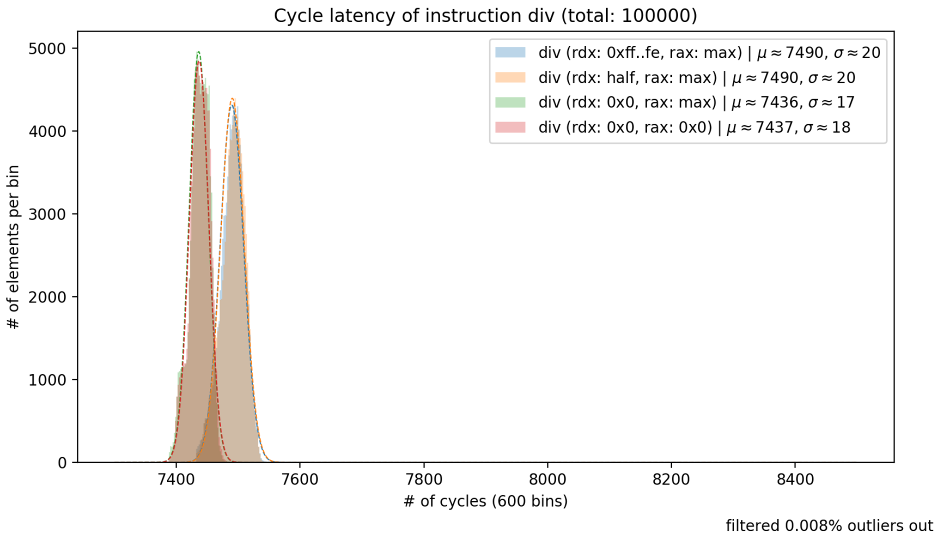


Figure 4.3: Our plots of the x86 *div* instruction with different dividends (stored in *%rdx:%rax*) and the fixed divisor 0xffffffffffffffff, measured on the NUC.

This implies for instance that we could only test 2^{12} *nop* instructions at once (which use 4 kB space, the page size on our test devices). First, we tried to remove this limitation by keeping an array of all pages that contain enclave code that we want to observe. Then, at every interrupt, we checked for each page in the array if it was accessed and, if so, marked it as 'not accessed' again to detect zero steps. However, it became apparent that this drastically slows down the tests (cf. 4.3 will show that already accessing two pages is problematic). To increase the performance without making the tool unnecessarily complex, we exploit the fact that the code under measurement is always linear (i.e. there are no branches). Therefore, we can just keep track of the current and next page and each time the latter is accessed for the first time, we move one page further[5]. Nemesis [7] solved this problem in a different way: Instead of keeping track of each page, they track the Page Middle Directory (PMD) entry (i.e. a higher entry in the hierarchical page table that is the same for a large number of pages). While this is simpler and can even handle non-linear code, there are two shortcomings compared to our approach: First, it can have false positives since the observed access bit is set to one for any accessed page with this PMD entry, not only the ones we intend to observe. For example, there is no guarantee that the enclave code before and after the observed code does not use pages with the same PMD entry. Second, in case the code happens to be aligned such that it spans over multiple PMD entries, the Nemesis variant will only catch instructions in the first entry. Our solution cannot have false positives, since we can just add a small padding to the end of the whole measurement to make sure that no other code starts in the same or the next page.

When we measure across page borders we have the additional problem of high outliers for the first instruction of each page. Figure 4.4 shows this for some test case (which instruction does not matter in this case) that spanned over 20 pages: We can count 19 outliers that are exactly 4KB apart (one test had 8B size in this case, so every 512 measurements we see an outlier). This is because new pages have to be fetched before their first instruction can be executed, which causes a large overhead for the first measurement. The less obvious observation that we made in figure 4.4 is that the last 50 measurements at the end of each page are faster. While we have not noticed this effect in the histogram, the outliers have increased the variance significantly, which is why we started to filter them out in all plots[6].

Conclusions from challenge 2:

[5] i.e. the old next page becomes the new current page

[6] One subtleness is that for the bar plot, we have to be careful to not filter them out too early, because then we would again shift the counting as we already have seen in section 3.4

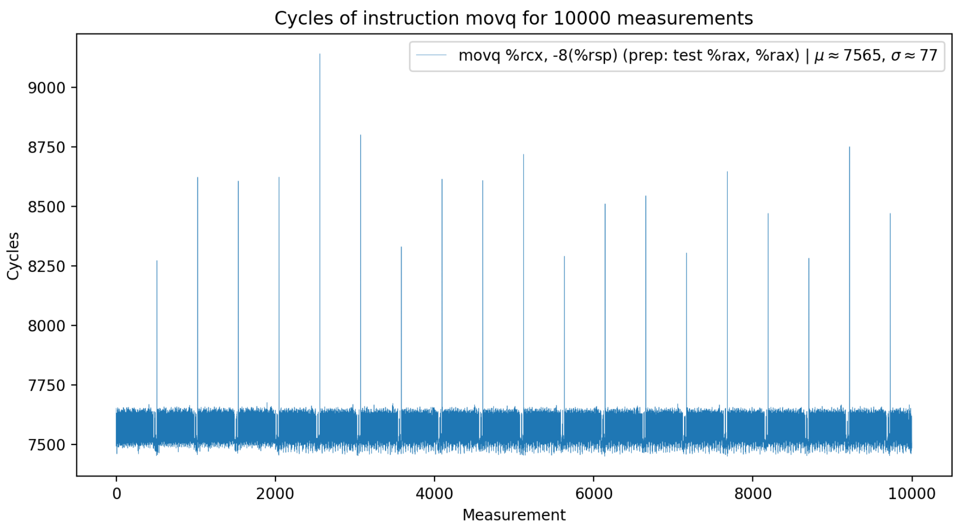


Figure 4.4: Measurements over time plot showing outliers at page boundaries (measured on the NUC).

1. Multiple pages of linear code can be observed without false positives by tracking the current and next code pages.
2. The first measurement of each page is a slow outlier that should be filtered in order to get a meaningful variance.

4.3 Challenge 3: Cache Conflicts

This section shows that the measurements are really sensible to how much code is prefetched into the cache between enclave entry and resume. This contributes to the noise in measurements and should thus be minimized as much as possible when the goal is to capture only the execution time of instructions.

In the previous section (4.2) we explained how we track the current and next page to handle multiple pages of code. When we use the *prefetch* instruction to make sure that both pages are in the cache right before we resume the enclave, some measurements contain characteristic noise: Figure 4.5a shows double peaks that we sporadically observed. In this figure we see the measurement of an *add* instruction, but the double peaks actually occur independent of the operation (but never consistently). They disappear when we only prefetch the current page: Now every measurement consistently has no double peaks and looks like figure 4.5b. This is clearly desirable, since it makes the results more stable and reproducible. We suspect that the second peak happens when prefetching the next page eliminates another page from cache that was needed by the enclave or ERESUME/AEX and therefore slows down the measurement because this page has to be fetched again.

> *Conclusions from challenge 3:* Cache pollution should be minimized between AEX and ERESUME to reduce noise (caused by re-fetching pages) in the measurements.

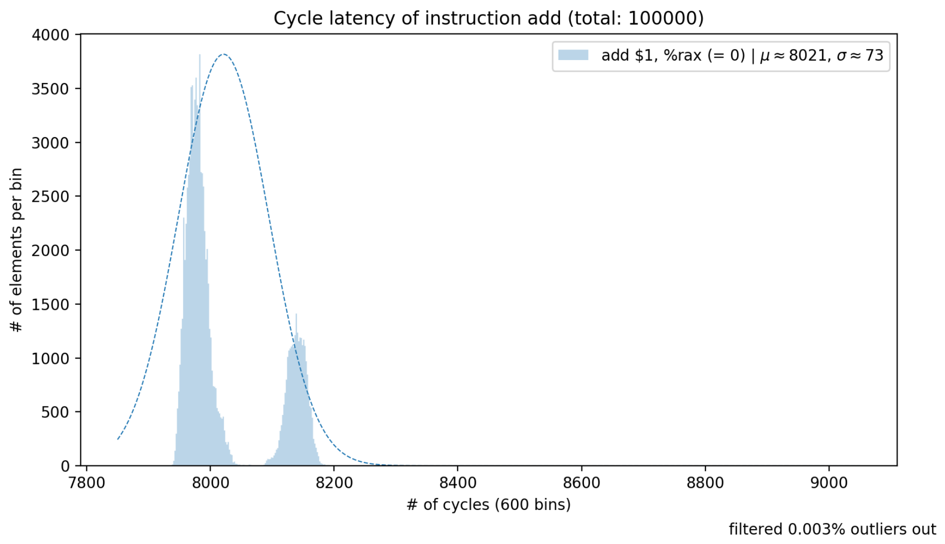
4.4 Challenge 4: Constant Time Measurement Code

In this section we argue that we do not only have to consider cache pollution (cf. 4.3) between enclave exit and resume, but the code between measurements should in general avoid asymmetries: We argue that non-constant time code[7], e.g. slow and fast execution paths in the code that logs instruction measurements, should be avoided.

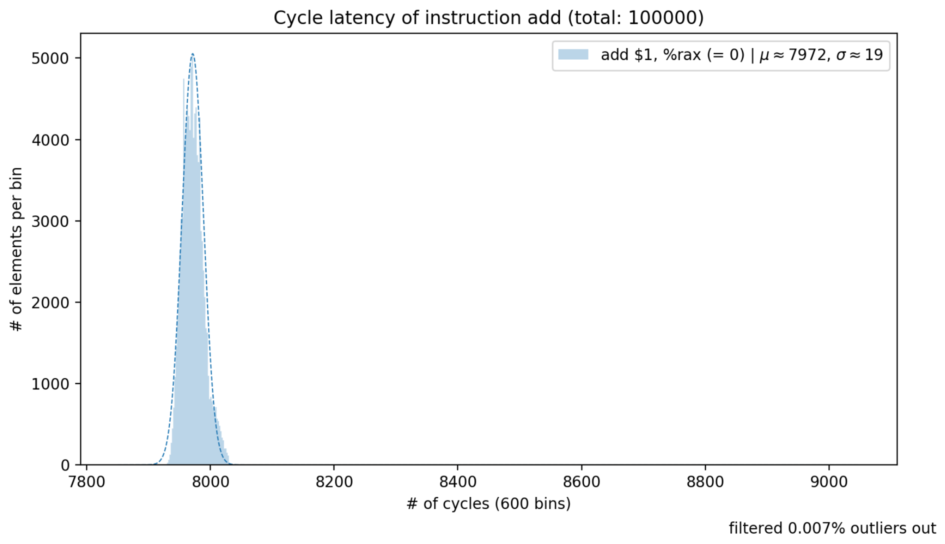
We first show the problem of non-constant time code in figure 4.6a: It shows the comparison of *add* $0, *%rax* and *add* $1, *%rax* where *%rax* is always 0 in both test cases. This plot seems to suggest that adding zero is slower than adding a one. However, there is a key difference between those two test cases: While for the second test we have to use a prepare instruction to reset the value of *%rax* to 0 after we have added one to it, we use no prepare instruction for the first test case. This itself cannot directly explain the measured difference, since we still single-step through the code and therefore measure each instruction separately. However, without constant-time code outside the enclave, a prepare instruction can be treated faster than a test instruction, since it does not have to be logged. This means that for the second test case that fewer instructions are executed before we measure *add* $1, *%rax* because we just measured a prepare instruction. This seems to slightly affect the timings, since we measured the same time for both test cases after we rewrote the time measuring code to be (almost) constant time, as figure 4.6b shows.

We applied three changes to fix this problem. First, we log all zero-steps, prepare and test instructions together with the accessed bits and filter after the measurements are done instead of between the enclave exit and resume. Second, we log to an array in memory and not to a file, because writes to the latter can be buffered which introduces unpredictable timings. Third, we replace most branches with the constant time instruction *cmov*, which linearises the code and avoids speculative execution. The only branches that we kept are special cases that do not influence the path that a normal test

[7]Constant time code takes the same time to execute independent of its inputs. This means that we cannot have any branches that do operations of different computational complexity.



(a) Spurious double peak



(b) Consistent single peak after reducing cache pollution

Figure 4.5: Both plots show measurements of an *add* instruction on the laptop. 4.5a shows a double peak which sometimes appears when we *prefetch* the next page. 4.5b shows the result that consistently appears when we only prefetch the current page.

takes[8]. However, we provide the compiler with branch prediction information to make sure that it optimizes for the path that measurements take[9].

> *Conclusions from challenge 4:* Code between AEX and ERESUME affects the microarchitectural state and hence should always be the same independent of the measured instruction to reduce noise.

4.5 Challenge 5: Imprecise APIC Timer

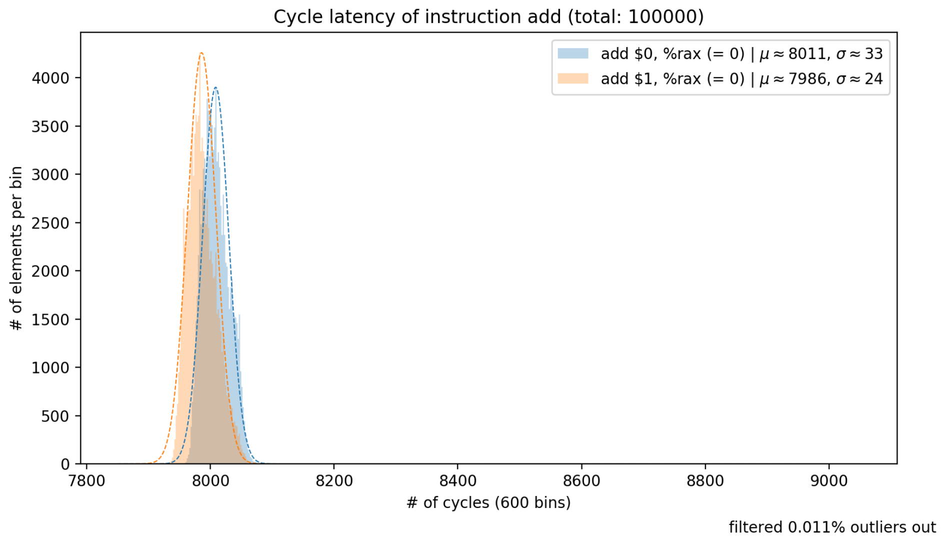
In this section we briefly discuss the problem caused by measuring instructions with interrupts generated by the APIC timer. In particular, it can happen that the synchronization with the instruction stream is lost. In section 2.2.1 we already mentioned that this timer runs at bus frequency, i.e. it has a lower precision than the processor's cycle counter (since it has a lower frequency than the CPU). Furthermore, we described in section 2.2.2 how to empirically choose the APIC timer interval so that no multi-steps and as few as possible zero-steps happen. In this challenge, we discuss the problem that occasionally, we still miss more than half of all instructions, which means that multiple instructions execute between two interrupts (i.e. that we have multi-steps). We can only speculate about the root cause of this behaviour, but a fairly convincing explanation would be that this is due to the variance of enclave entry and exit times (we showed their existence in section 4.1), because if those procedures take much shorter, then multiple instructions have time to execute before the next APIC interrupt arrives. This is consistent with the fact that we either catch all instructions or miss many of them: We saw that AEX and ERESUME are consistent for the same enclave, so if they happen to be very fast, then every single-step is likely to have actually executed multiple instructions.

Fortunately, this is easy to solve for instruction benchmarking: We know how many tests we have and can just count the instructions that we caught inside the enclave and throw an error if this does not match our expectation. This way the rare cases of such offsets can be detected and we can repeat the measurement. However, in an attack setting where we do not know the code, this case is undetected and would increase the prediction inaccuracy.

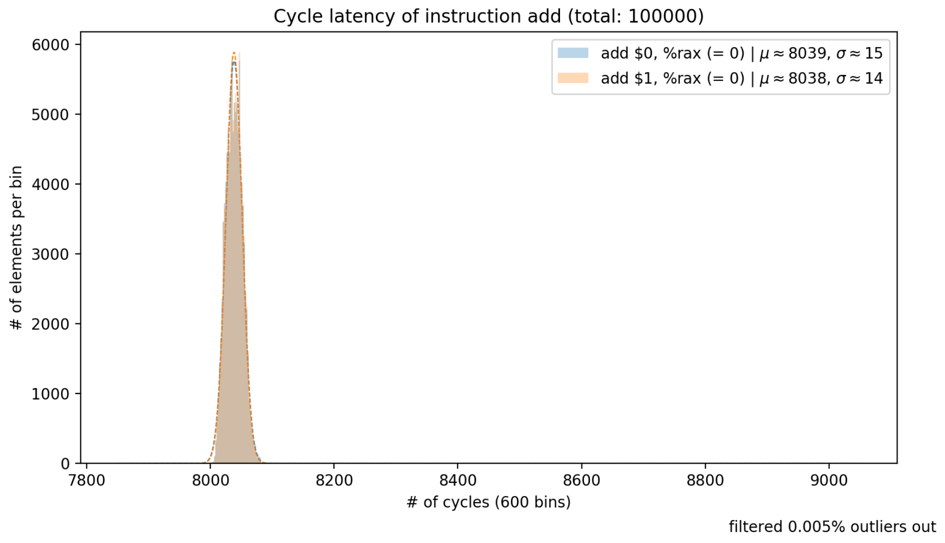
> *Conclusions from challenge 5:* Since our APIC timer does not generate interrupts on cycle accuracy, it can happen that variances in ERESUME

[8] For example, a branch that is only taken when an error happens does not affect the measurement, since if it is taken once, the whole run is aborted.

[9] The compiler built-in function *__builtin_expect* can be used to annotate the expected value of a branch



(a) Non-constant measurement code



(b) Constant measurement code

Figure 4.6: Both plots compare *add*ing 0 or 1 to a register that has the value 0, measured on the laptop. In 4.6a we used non-constant time measurement code while in 4.6b we show the result for constant time code.

desynchronise the instruction tracking and cause multi-steps. We have to detect and handle those rare cases.

4.6 Challenge 6: Keeping Track of Instructions

In this section we will see that counting instructions as described in challenge 4.5 is not enough, but we also need to make sure that the right instructions are measured. We will introduce further mechanisms to detect such cases.

Precisely tracking instructions is essential as we will see on the example of floating point multiplication with *fmul*. The first results that we obtained for those measurements were hard to explain. We show in this section the issue with our original measurements before we will discuss the fixed plots in the next section 4.7. The problem is that there are instructions that consist of multiple parts, which can be interrupted in the middle. This is a problem, because it shifts our instruction tracking and we start measuring the wrong operation. In our case the problematic instruction is *finit* (opcode *9B DB E3*), which initializes the 'FPU[10] after checking for pending unmasked floating-point exceptions' [15]. There is another instruction, *fninit*, which does the same but without checking for exceptions. Interestingly, this has the last two bytes of *finit* as opcode: *DB E3*. Printing the instruction pointer of a debug enclave (cf. 2.1.2) confirmed that *9B* is executed on its own, i.e. that *finit* is split into one instruction that checks for exceptions and one that initializes the FPU. By single-stepping, we interrupt the enclave twice for *finit* and see two executed instructions, which shifts our instruction count. Since we only use *finit* once as an initial instruction, the total count of measured test instructions is correct, but we are measuring prepare instead of test instructions. If *finit* is used as a prepare instruction, then there would be much more interrupts than we expect and this would be easier to catch.

We now discuss our approach to deal with this problem: Figure 4.7 shows the modified measurement code, where the coloured part are the additional flags that we added to accurately keep track of measured instructions. Before we start measuring, we set a variable *observed* in shared memory to true from inside the enclave. We only count instructions if the page was accessed and *observed* is set to true. We set this flag to false before we jump to the next test case or before we exit the enclave. This is an additional measure[11] to protect against false positives, since we can be sure that no code outside this *observe* flags is counted. In other word, the *observe* flag inserts verifiable checkpoints in the code. Counting the number of instructions between

[10] Floating Point Unit

[11] The other measure is to pad the test cases such that no other code is in the same page, as we explained in section 4.2

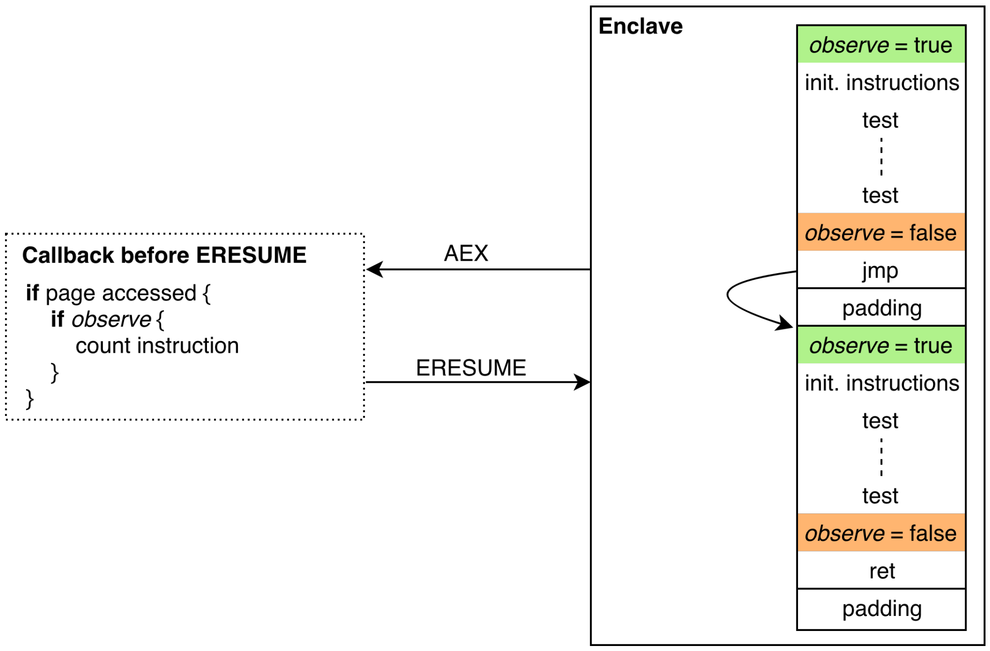


Figure 4.7: Instructions are only counted if a code page was accessed and the *observe* variable is set to true. The callback code is a simplification, because we actually have to use constant time code as we saw in section 4.4.

these checkpoints and comparing them with the expected number allows us to detect anomalies at a fine granularity level. Figure 4.8 shows the concrete example of *finit*: On the left side we see how the measurement was planned; there is one initial instruction (*finit*) and two tests, both consisting of a prepare and a test instruction. There are five interrupts and we only log the timings of the two test instructions. On the right side we see what happens when *finit* is interrupted twice: Now we measure prepare instead of test instructions. There is an additional interrupt at the end, however it is counted as a prepare instruction, so the number of measurements is still correct. We check with the modification of this section for unexpected trailing instructions, such as the last test instruction in this example. We can detect them because they still have the *observe* flag set to true and are a valid time measurement although we do not expect any additional measurements.

While strictly speaking it is enough to only do this detection once for every test case and then adapt to the correct number of instructions, keeping the *observe* flag as a fixed sanity check in all measurements has additional benefits. First, we do not have to worry about any other statements that are in our observed pages. For example the jumps from one test case to the next one can be tricky to catch because, depending on how large the relative immediate of the label is, this instruction can be faster than a *nop* and thus can

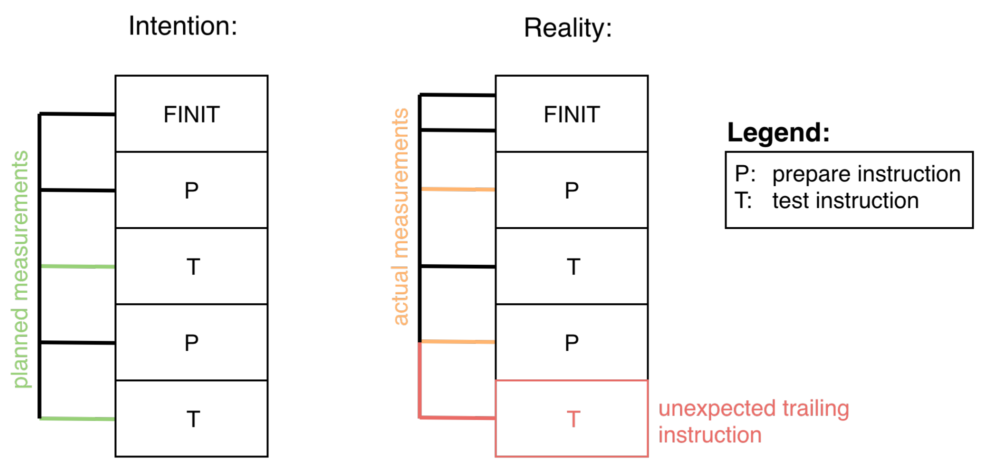


Figure 4.8: Visualisation of how an initial instruction (*finit*) that has more interrupts than expected, can slightly shift the instruction measurements. On the left side is we see how the measurement is intended (it measures all test instructions), on the right side we see the shifted version that measures prepare instructions by mistake.

be missed by SGX-Step. Second, the usability is better since test cases often change frequently and an additional step to verify them each time would cost time and can be forgotten easily.

> *Conclusions from challenge 6:* To catch special instruction that take multiple steps we have to track the enclave execution precisely and detect trailing instructions.

4.7 Challenge 7: Verifying Tests

After we saw that we need to precisely track instructions in sections 2.2.2 and 4.6, we will argument here that it is indispensable to check prepare instructions as well. They interact with test instructions by directly modifying the (micro)architectural state and we need to verify that their effect is what we expect and not some corner case that behaves differently.

We will discuss this more in depth on the example of measuring floating point multiplication, since those results are also interesting apart from illustrating the point of this section. To check the accuracy of our tool, we measure instructions that we know should have data dependent execution times. According to Agner Fog the 'handling of subnormal numbers is very costly in some cases because the subnormal results are handled by microcode exceptions' [12]. We construct the following test case, using specific floating point values for which D. Kohlbrenner and H. Shacham have measured different times in [18]: First, the initial instruction *finit* initializes the FPU[12], then we have two *fld* as prepare instructions to load the operands of the multiplication on the FPU register stack. As test instruction we use *fmul*, which multiplies the first two elements of the FPU stack and then pushes their product back on this stack. However, this test case is not showing any difference between subnormal and normal floats. There is a subtle mistake in this experiment: According to the Intel manual volume 1 [14], the FPU stack has space for eight values. When you push more values, it is not only wrapping around, but depending on the instruction it might also overwrite the first value on the stack with a NaN (Not a Number). This means in the test case from before that independent of the operands that we want to measure, we always overflow the stack and actually only observe the multiplication with NaNs. We will now show a test case with modified prepare instructions which confirms that overflowing the stack and multiplying with a NaN have the same execution time while non-exceptional multiplication is significantly faster. Figure 4.9 shows this test case with three tests, all of which measure *fmul*, but they use different prepare instructions. The first, *fmul 1.0, 1.0, wrap around*, pushes nine times 1.0 on the stack, i.e. it causes an overflow. The second test, *fmul 1.0, 1.0, ST(0)-ST(7)*, initializes all eight FPU stack registers with 1.0 and therefore does not overflow. The last test *fmul 1.0, NaN* pushes 1.0 and a NaN on the stack. We see that the last two tests overlap, because they are actually both performing the multiplication with a NaN value. The first test is faster, which makes sense because multiplying with a NaN is probably an expensive special case.

To get back to the original motivation to test floating point values, also after we fixed the tests (by always reinitializing the floating point stack with *finit* as preparation), we still did not see a difference when one operand is a subnormal value versus when both are normal values as the first two *fmul* tests of figure 4.10 show. However, we can also see in the same figure that using the newer SSE[13] instruction *mulsd* shows the expected difference. This leads us to believe that *fmul* instructions handle subnormal values differently. In [18] they measured the multiplication in JavaScript, so we do not know which assembly instructions were actually run, but it seems to be likely that this was also *mulsd*, because e.g. also *gcc* compiles floating point multiplication to SSE instructions on our machines.

> *Conclusions from challenge 7:* Prepare instructions influence test instruction by modifying the (micro)architectural state. We have to carefully check that they execute as expected and not exhibit exceptional behaviour (e.g.

[12]Floating Point Unit
[13]Streaming SIMD Extensions

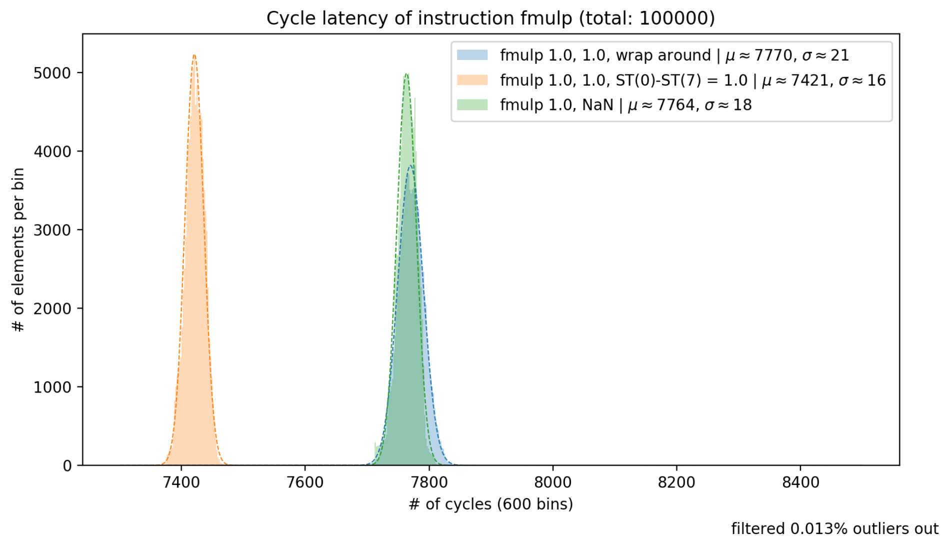


Figure 4.9: Measurement of floating point multiplication with *fmul* on the NUC. We compare three test cases: The orange histogram shows *fmul 1.0, 1.0, ST(0)-ST(7)*, which multiplies 1.0 with 1.0 by taking the first two elements of the FPU stack. While this test case only pushes eight values on the stack, the blue test case (*fmul 1.0, 1.0, wrap around*) pushes nine values and thus overflows the stack. And green test case, *fmul 1.0, NaN*, multiplies 1.0 with a Not-a-Number (NaN) value.

by using resources such as the Intel manuals [14] and [15] or Agner Fog's optimization guide [12]).

4.8 Challenge 8: Setting Flags

While previous challenges showed mistakes and how we can avoid them, this and the next section 4.9 will be different in the sense that they explain useful operations to construct advanced test cases. In this section we will show how we set status flags for testing instructions like *cmov*.

The instruction *cmov* performs a conditional move depending on the value of the status flags set by previous instructions. Those flags are either set as side effects of instructions like *add* (e.g. when an overflow happens) or explicitly, for example with the *test* instruction. The latter performs a logical *and* of its operands and then sets the registers according to the result. Figure 4.11 serves as motivation why we want to perform tests that use flags: It shows the comparison of *cmov* from stack[14] to a register where it once ac-

[14]Section 4.9 discusses how memory operands can be used in detail

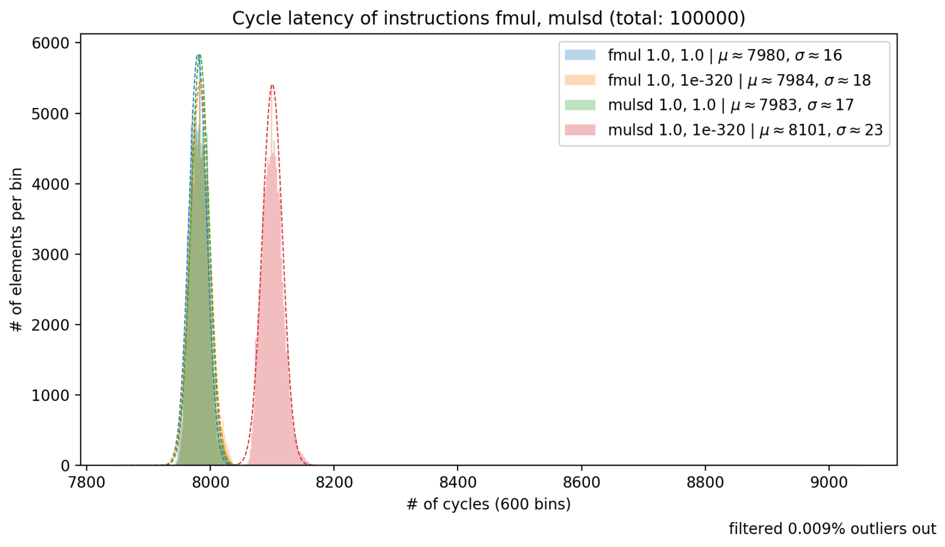


Figure 4.10: Measuring the time difference between multiplying normal and subnormal values with *fmul* (blue and orange) and *mulsd* (green and red), measured on the laptop. Except for the multiplication of a normal with a subnormal value using *mulsd*, all test cases show the same timings.

tually performs the move (ZF=1) and once not (ZF=0). It is important that those two cases cannot be distinguished because they are the fundamental assumptions of many data oblivious frameworks like Racoon [21].

We use *test %rax, %rax* to set the zero flag in the following way: We set the register *%rax* to either zero or one; if it is zero, the logical *and* will also be zero and the zero flag (ZF) is set to one. In the other case, we have ZF=0 because the result of the *and* is one. Then we can use the version of the conditional command that performs its operation based on ZF, e.g. for the mentioned conditional move of figure 4.11 we used *cmovz* (move if ZF=1). Since the enclave has to save and restore the execution environment, it is not surprising that status flags are preserved across AEX and ERESUME. Therefore, if prepare and test instructions do not set flags themselves (one can check this in the 'Flags Affected' section of the Intel manual [15]), it is enough to set them once in the initial instructions. We actually verified that the flags are restored by setting them in the first test case and then performing a conditional jump[15] to the end of the enclave code in the initial instructions of the second test case. If the test aborts after exactly half the instructions when we set one flag, but it executes all instructions if we set the opposite flag, then we know that the flag value is preserved for the first test case.

[15]*jz lab*: Jump to the label *lab* if the zero flag is set to one

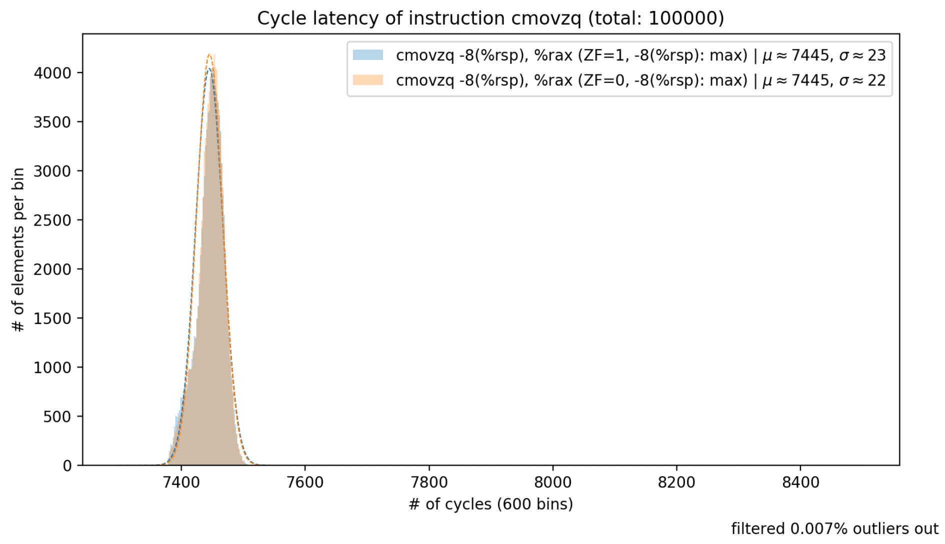


Figure 4.11: Comparing the conditional move *cmov* from stack to register where once the value is actually moved and once it is not.

Conclusions from challenge 8: Flags can be set with *test %rax, %rax* to ZF=1 for *%rax=1* and ZF=0 for *%rax=0* and are preserved over enclave exit and resume. If prepare and test instructions do not affect those flags, then it is sufficient to set them in the initial instructions.

4.9 Challenge 9: Writing to Memory

In this section we discuss three convenient options to perform tests that read or write to memory. They cover different use cases and are simple to implement. In particular, we look at storing data in the red zone, the read-only data and the data section of the program:

1. The *red zone* is an area below the stack pointer that is commonly used by compilers as an optimization to store temporary data without manipulating the stack pointer. According to the documentation for GNU compilers [9], a 128-byte area below the stack pointer is guaranteed to exist. For our use case, this is convenient to use, since it does not require any extra steps and we can read and write to that location. However, it is also a special case to write to the red zone and might not be a realistic option for all instructions to test. For example, global variables or large arrays will not be stored in the red zone in normal programs.

2. We can also place values in the *read-only data section* of the program. To do so, we use the *.rodata* in ELF[16] binaries. This is demonstrated in code snippet 2, which shows how one can place the 8B aligned and 8B long floating point value 1.0 in the read-only data section. The obvious downside is that we can only read from this memory.

3. The last and often best option is to use the *data section* of the ELF binary, which saves space for uninitialized data. This is done in the same way as we see in snippet 2 for the read-only section, but we use the *.comm* assembler directive instead of *.rodata*. Besides that it allows read and write operations, it also makes more sense to flush those locations than the red zone (e.g. to test the difference of instructions when operands are cached or not).

Code Snippet 2 Assembly code to place the floating point value 1.0 in read-only memory

```
.section .rodata
    .align 8
.ONE:
                ▷ the upper and lower 32 bits of 1.0 are interpreted as integers
    .long 0                                                      ▷ lower bits
    .long 1072693248                                             ▷ upper bits
```

Table 41: Comparison of different options to write to memory

Name	Usage	Write	Assembler Directive
red zone	temporary data	✓	-
read-only section	static data	✗	.rodata
data section	many applications	✓	.comm

Conclusions from challenge 9: Table 41 summarizes three different options (red zone, read-only and data section) that can be used to store values in memory. For each option, we mention for what applications it makes sense to use it, if this memory is writeable and the assembler directive that has to be used to put values in this section. It makes sense to choose the option that is closest to what a compiler would produce for the imagined test case because this produces realistic measurements.

[16]Executable and Linkable Format

4.10 Challenge 10: Two Noise Sources

In this section we discuss the fundamental problem that we have two sources of noise when we measure inside enclaves: On one hand the enclave entry and exit and on the other hand the instruction measurement (including the test instruction itself). We will discuss our attempts to filter the noise of AEX and ERESUME and their problems.

We will first look at this noise filtering problem from a mathematical point of view. Supported by the histogram plots of previous sections (for instance figure 4.10), it is reasonable to assume that we have Gaussian noise. Let $X \sim \mathcal{N}(\mu_X, \sigma_X^2)$ be the random variable for the execution time of an instruction and $Y \sim \mathcal{N}(\mu_Y, \sigma_Y^2)$ be the one for enclave entry and exit. In that case our overall measurement results follow a third random variable $Z = X + Y$. Now if f_X is the probability density function of X and f_Y is the one of Y, then the distribution of Z is the convolution of f_X and f_Y. In the case of normal distributions, this is

$$f_Z = conv(f_X, f_Y) = \mathcal{N}(\mu_X + \mu_Y, \sigma_X^2 + \sigma_Y^2) \tag{4.1}$$

which is also consistent with our measurements of Z, that closely followed a normal distribution. The basic idea to obtain the measurements of X if to filter the effect of Y and then perform a deconvolution, i.e. we want to measure Y so that together with Z we can recover X by fitting distributions f_Z and f_Y to their measurements and deconvolve their approximated density functions. Figure 4.12 demonstrates our implementation of this on artificial data: We generated instruction and noise timings according to normal distributions and added them to get the test data. The idea is that we can measure noise and overall timings as shown in the upper graph and then recover the instruction timings shown in the lower plot. This would have two main benefits: First, it would factor out the different effects of enclave entry and exit and thus make graphs from different enclaves comparable (see the problem explained in section 4.1). Second, it would reduce the variance significantly[17]. Most of the variance that we see should be caused by AEX and ERESUME (because they take much more cycles than the measured instruction).

However, in practice there are problems to measure only the noise from enclave entry and exit. Measuring the time of zero steps that naturally happen during a test case in isolation has too high variance to be used for the deconvolution, as figure 4.13 shows. According to equation 4.1, the zero steps must have a smaller variance (σ_Y) than the instruction measurements ($\sigma_Z^2 = \sigma_X^2 + \sigma_Y^2$), but measuring e.g. *add* has a standard deviation of

[17]Figure 4.12 can be misleading, because the x-axis is different. But also as an absolute value, the variance of the test data was chosen quite high (compared to tests outside the enclave which had a variance below three).

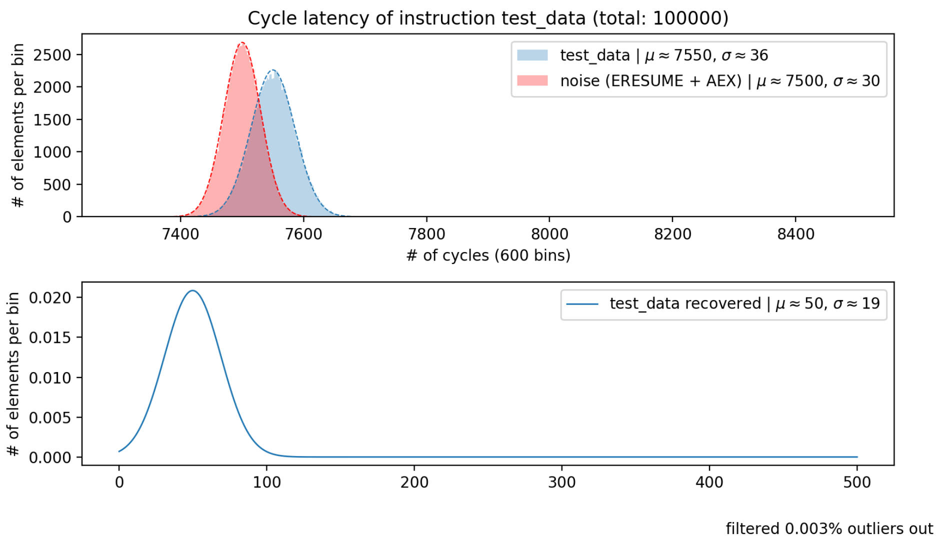


Figure 4.12: Show case of the deconvolution on sample data. The y-axis for the recovered data is not scaled to the number of measurements, since the probability distribution is independent of the number of measured instructions.

$\sigma_Z \approx 15$ (as we saw in figure 4.6b) while the zero steps from figure 4.13 have $\sigma_Y > 1000$. Additionally, we do not know when exactly the interrupt arrived because it is difficult to time (cf. 2.2.2, 2.2.4 and 4.5). That means the interrupt could have arrived just before the test instruction started to execute, but also much earlier.

Another approach to measure zero steps is to leave the first code page of the enclave marked as not executable. This way we are sure that an AEX is performed right when it tries to execute the first instruction, because this raises a page fault exception. However, figure 4.14 shows that this exit path, which is different from when an interrupt arrives, is slower by multiple thousand cycles (normal *nop* instructions take around 7500 cycles on the NUC). The problem is that while SGX-Step has registered their own fault handler for interrupts, page faults go through the kernel and we can not stop our timer before we return to user space. Therefore, we measure all kernel code that was executed before we could end our time measurement. This increases the mean as well as the variance and thus cannot be used for deconvolution for the same reason as the first approach. When we modify the Ubuntu kernel such that it takes the second timer right in the beginning of the page fault handler and writes it to the kernel log, we obtain measurements that are fast enough but still have too high variance, as figure 4.15 shows.

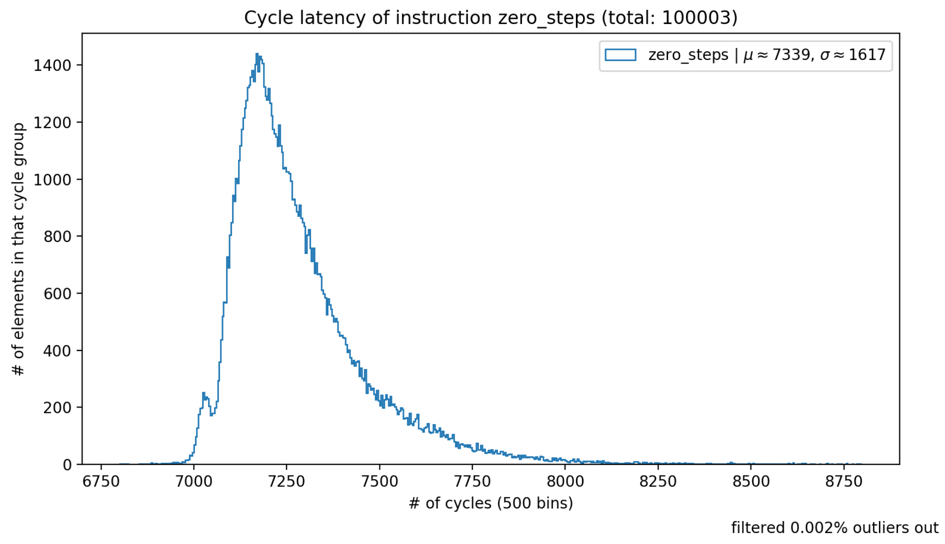


Figure 4.13: Repeat normal test cases on the NUC and collect zero steps that naturally occur until you have measured a significant amount of them.

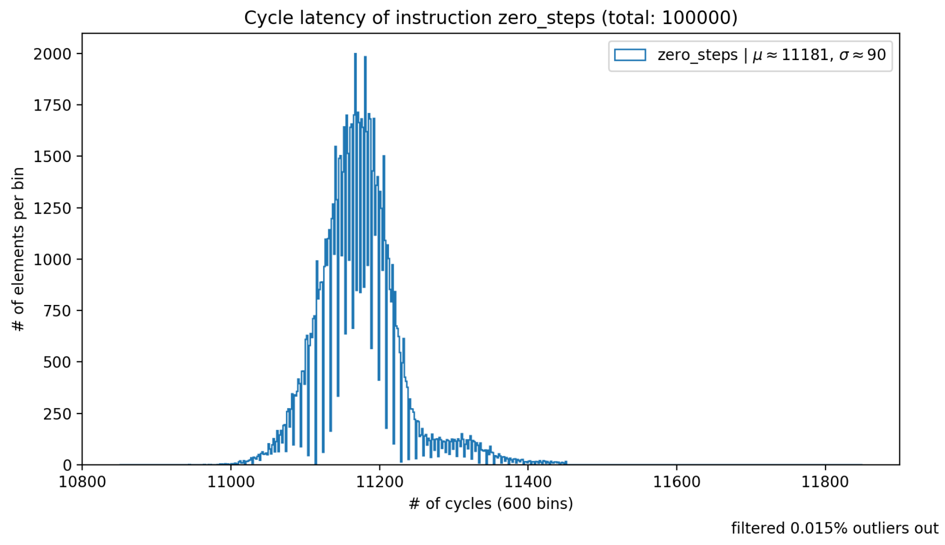


Figure 4.14: We mark the first code page as not executable and then measure the time it took to enter the enclave and exit with an error before executing a single instruction, measured on the NUC.

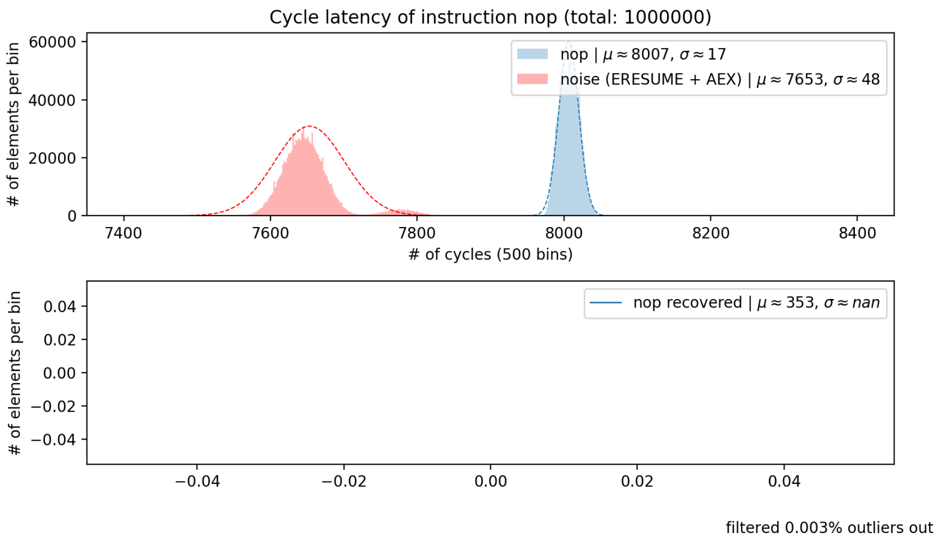


Figure 4.15: Same as figure 4.14, but we use a modified kernel to take the second timer directly in the page fault routine. The recovered data is not plotted, because the variance is invalid.

Conclusions from challenge 10: We have two convoluted distributions: One normal distribution for AEX and ERESUME and one for the instruction itself. It proved to be difficult to reliably measure zero steps to filter out the first noise source. An alternative is to use the measurement of *nop* instructions for deconvolution, since we know from tests outside the enclave, that *nop*s only take one cycle.

4.11 Challenge 11: Synthetic State on AEX

In this section we discuss the technical challenge of keeping the synthetic state consistent when inserting custom code between ERESUME and AEX.

The synthetic state replaces the processor's state on (asynchronous) enclave exit to prevent leakage. Apart from using placeholder values, this also sets the AEP[18] and prepares arguments for ERESUME. A full specification of the values of each register can be found in the third volume of the Intel developer's manual ([16]). The problem is that by adding instructions between AEX and ERESUME we may overwrite the synthetic state. For instance, this is a problem with *cpuid* (which we use for serialization, cf. 3.2.1): This instruction writes to four registers, among those *%rcx* which is used by the synthetic state to store the AEP. Therefore, we need to preserve this value

[18] Asynchronous Exit Pointer

because otherwise we would jump to an invalid address. Another instance is C code in the fault handler (e.g. for zero step measurement in 4.10). The compiler is free to use any volatile registers (to store local variables) without preserving their value. The mentioned *%rcx* register is one of them and thus needs to be preserved manually.

> *Conclusions from challenge 11:* Code between AEX and ERESUME must preserve the synthetic state.

Chapter 5

Applications

In this chapter we will look at an application of our improved tool to measure memory writes. We will first discuss the occurrence of reproducible double peaks and then use this knowledge to exploit a side-channel based on instruction alignment.

5.1 Double Peaks on Memory Write

We will discuss in this section reproducible double peaks in the distribution of time measurements of memory writes with *mov*. We will see that this problem is different from the spurious cache conflicts of section 4.3, because it happens consistently and with unique patterns. Moreover, we only observed the double peaks in this section in connection with *mov*. After looking at some examples of double peaks we will explore possible reasons that could justify why they might happen. In the end, we will get back to the first examples and reconsider them from the perspective of our hypothesis.

Double Peak Variants. We first introduce four cases that show different variants of double peaks. All characteristics of the plots that we discuss in this section are stable in occurrence as well as dimension, unlike the double peaks that we have seen in section 4.3. All four test cases that we present next measure the same test instruction, *movq %rcx, -8(%rsp)*, which writes a value from register *%rcx* to the red zone (cf. 4.9). Figure 5.1 shows two clearly separated peaks for a test that solely contains the aforementioned test instruction. However, we only observe a single peak when we add a *nop* as a prepare instruction. While the height of the peaks is stable across executions, it changes if other prepare instructions are used: Figure 5.2 shows that the peaks are of equal height when using the single prepare instruction *test %rcx, %rcx* (the blue histogram). However, there is only one peak when we add a *nop* to the prepare instructions, as we see with the orange measurements.

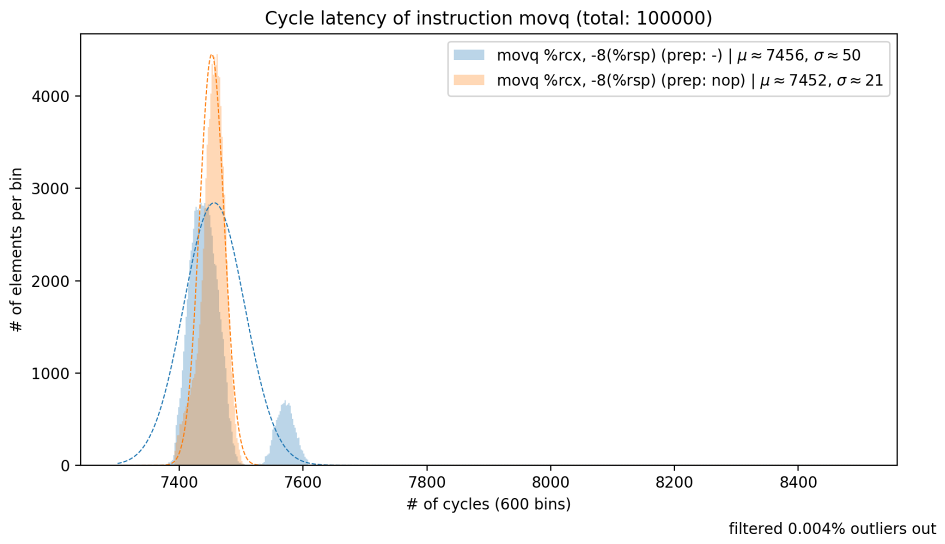


Figure 5.1: While a single *mov* from register to stack without any prepare instructions shows a double peak, the same test with a *nop* as prepare instruction does not. Measured on the NUC.

Additionally, the shape of the peaks also depends on the operands: Figure 5.3a shows that we have two peaks with the prepare instruction *mov $1, %rcx*. On the other hand, with the prepare instruction *mov $1, %rax*, i.e. a *mov* to another register than the one used by the test instruction, we only measure a single peak, as figure 5.3b shows. In both cases, there are no double peaks with an additional prepare *nop*. Table 51 summarises the four double peak variants that we discussed. For each figure, it shows the prepare instructions of the test cases and whether they have a single fast peak or two peaks (and their sizes).

We now shift the focus from the variants of double peaks to the temporal behaviour of the test instruction timings to understand the temporal relation between slow and fast *mov*s. We consider this relation on the example of a *mov* from register to memory with *test %rax, %rax* as prepare instruction. We have already seen in histogram 5.2 that this test case shows two double peaks of equal size. Figures 5.4 and 5.5[1] show that every second instruction is fast. In the first, we see that when we put one *mov* into the prepare instructions[2] then the test instruction measurements suddenly show only the fast peak. However, we can see that the prepare *mov* (the violet *prep 1* in figure 5.4) now only measures slow timings. Figure 5.5 makes it clear that *mov*s at an odd position are on average approximately 100 cycles faster

[1]We have already seen both figures in the plot types introduction in section 3.4
[2]I.e. use *test %rax, %rax; movq %rcx, -8%rsp; test %rax, %rax* as prepare instructions

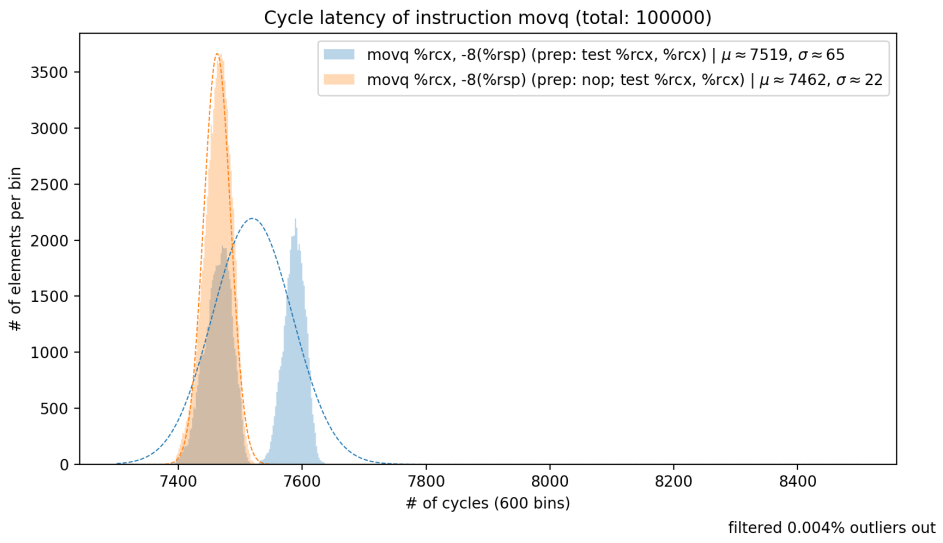
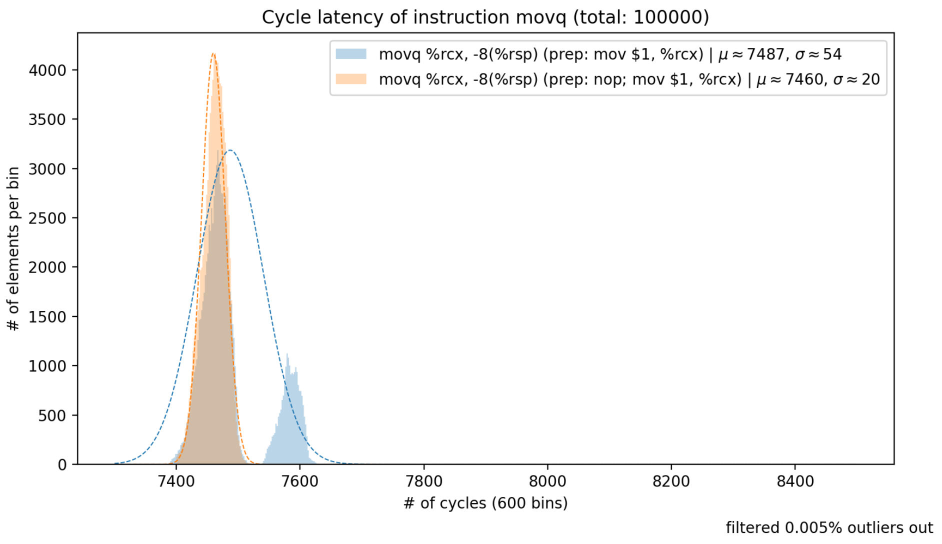


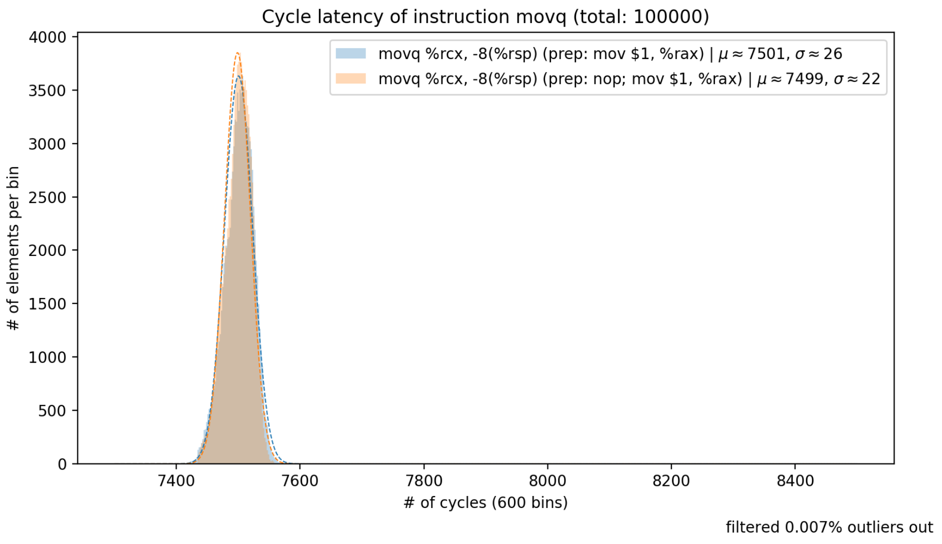
Figure 5.2: Double peak of *mov* from *%rcx* to stack with the dependent prepare instructions *test %rcx, %rcx*. Adding a prepare *nop* makes the slower peak disappear. Measured on the NUC.

Table 51: Overview over the four different variants of double peaks presented in the paragraph 'Double Peak Variants' in 5.1. For each figure, we specify the prepare instructions and whether there is a fast and/or a slow peak.

Figure	Prepare Instructions	Double Peaks	Heights
5.1	-	yes	larger fast peak
	nop	no	-
5.2	*test %rcx, %rcx*	yes	equal peaks
	nop; test %rcx, %rcx	no	-
5.3a	*mov $1, %rcx*	yes	larger fast peak
	nop; mov $1, %rcx	no	-
5.3b	*mov $1, %rax*	no	-
	nop; mov $1, %rax	no	-



(a) Dependent prepare instruction



(b) Independent prepare instruction

Figure 5.3: Double peak of *mov* from *%rcx* to stack with the prepare instructions *mov $1, register*. The test case in plot 5.3a uses the same register *%rcx* in the prepare and test instructions, while in 5.3b independent registers are used. In both cases, adding a *nop* to the prepare instructions makes the double peaks disappear. Measured on the NUC.

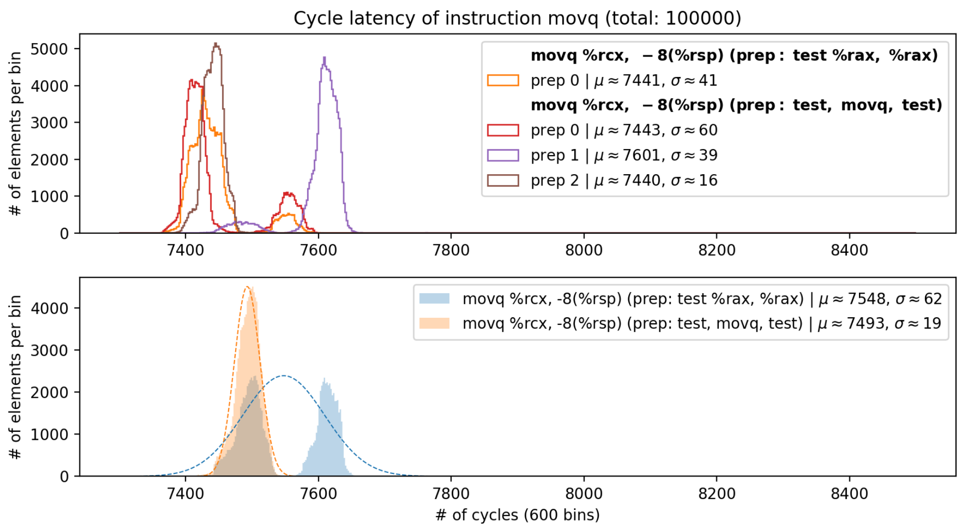


Figure 5.4: Histogram plot of *mov* from register to memory with *test %rax, %rax* as prepare instruction. When we measure every *mov*, we see the double peak, while if we only measure every second *mov* we do not.

than those in an even one. We can even take this one step further and see in figure 5.6 that every second *mov* to memory – except on page boundaries – is always going to be either fast or slow.

Cache Bypass Hypothesis. We now explore a theory that could possibly explain the source of those double peaks. Then we will briefly reconsider the four variants of double peaks that we saw in the beginning of this section. Essentially, we hypothesize that due to optimizations, the CPU sometimes decides to bypass the cache and directly write to memory. In that case, we measure a longer execution time than for a write to cache, because we measure the time until the instruction successfully finished its write. In other words, there is a difference in the point where an instruction retires: While a cache bypassing write waits until the value is committed to memory, a write to cache is faster and already finishes when the value is in cache but the memory transaction is still pending. Of course, a cached value also has to be written back to memory eventually, however, this is done by the cache coherency protocol and not by the instruction that we measure. This can thus happen after, or even in parallel, to our measurement and, therefore, does not prolong the timings of the *mov*. Given the sheer complexity of Intel CPUs and the unavailability of internal optimization details, we can only speculate what is happening behind the curtain. The architectural details in this paragraph are based on the information from WikiChip [2]. We assume that memory write operations look at the allocation queue (which holds all

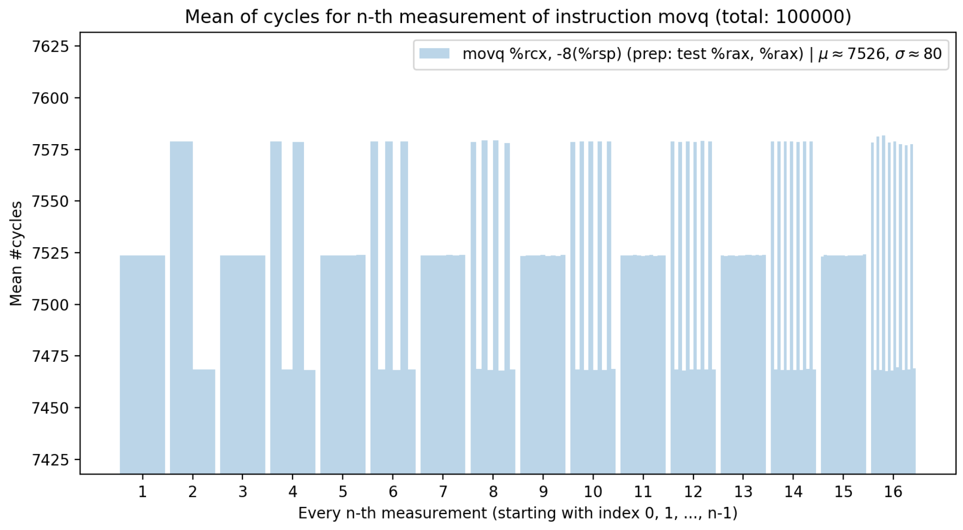


Figure 5.5: Bar plot of *mov* from register to memory with *test %rax, %rax* as prepare instruction. On the x-axis is the period for each of which we calculate the mean of the instruction sequences with all possible offsets (e.g. for period 3 we measure three disjoint sequences of instructions: $0 + k * 3$, $1 + k * 3$ and $2 + k * 3$ for $k \in \mathbb{N}$)

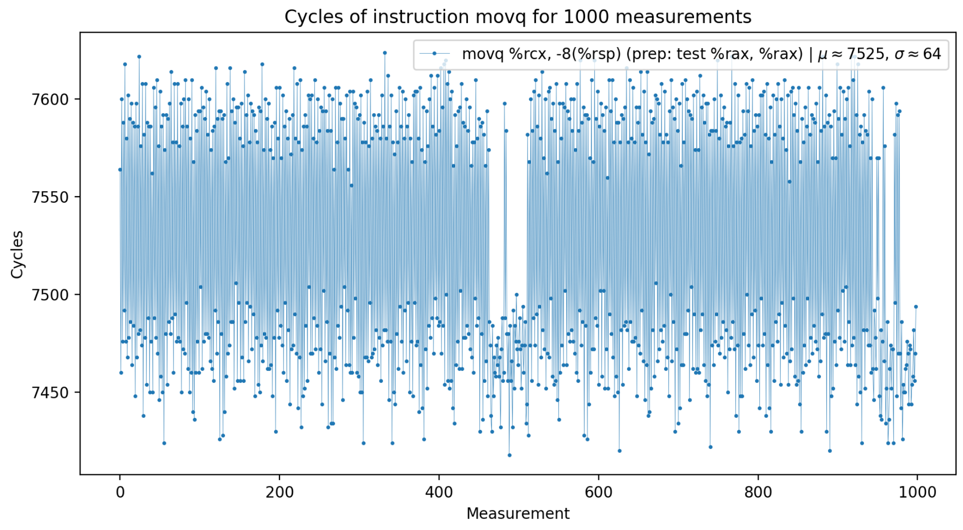


Figure 5.6: Pattern plot of *mov* from register to memory with *test %rax, %rax* as prepare instruction.

currently decoded instructions) and then decide if the written location is likely to be used again in the future (and thus should be cached) or if it is better to directly write it to memory and not pollute the cache. The allocation queue's content depends on the fetch and decode window, which has multiple implications: First, code alignment is important, since this causes the 16B fetch windows to include different instructions. Second, how the instructions are decoded to micro-ops[3] also changes the allocation queue. According to [2] there is only one complex decoder, which means only one unit can decode instructions that produce multiple micro-ops. Since *mov* produces, according to Agner Fog [11], two (unfused[4]) micro-ops, it will have a lower throughput when other complex instructions are in the same fetch window.

Supporting Arguments. We now present three figures that support this theory. First, figure 5.7 compares *mov* and *movnti*. The second instruction uses a non-temporal hint, informing the processor that this data will not be used in the near future. Thus this memory write usually bypasses the cache. It nicely fits our hypothesis that the measurements of *movnti* overlap with the slower peak of *mov*, since our proposition claims that those are the instructions that bypass the cache as well. In general, also the gap between the two peaks can be explained, because writing to memory has a cycle penalty that is in the same range (around 100 cycles) as the difference between the peaks. Second, figure 5.8 shows the measurement of *mov* from register to memory. Apart from the first test case, all others have the same number of instructions, the only difference between them is that the *nop* that they have as prepare instruction is encoded using a different amount of bytes. We only see double peaks for zero[5] as well as two and three byte *nop*s. This is compatible with our hypothesis in the sense that the number of instructions is exactly the same, but the fetch windows do look different for those test cases. To be able to understand why some particular sizes of *nop* instructions show double peaks, we would need to know the exact criteria that trigger the cache bypass optimization as well as the precise content of the allocation queue. The third test case, shown with two plot types in figure 5.9, consists of the prepare instruction *movq -8(%rsp), %rcx* followed by the test instruction *movq %rcx, -8(%rsp)*, i.e. we first read a value from the stack and then we write it back to the same location. Figure 5.9a shows that the read operation (the orange *prep 0*) also shows a double peak when it is combined with a write operation (which is the blue histogram in the lower plot of 5.9a). This is consistent with our theory, since if the second peak of

[3]Micro-operations are the elementary units that assembly instructions are broken into. They cause execution units to perform basic operations that realize the effect of a more complex operation.

[4]Certain micro-ops can be merged to a single one (after decoding)

[5]I.e. no prepare instruction

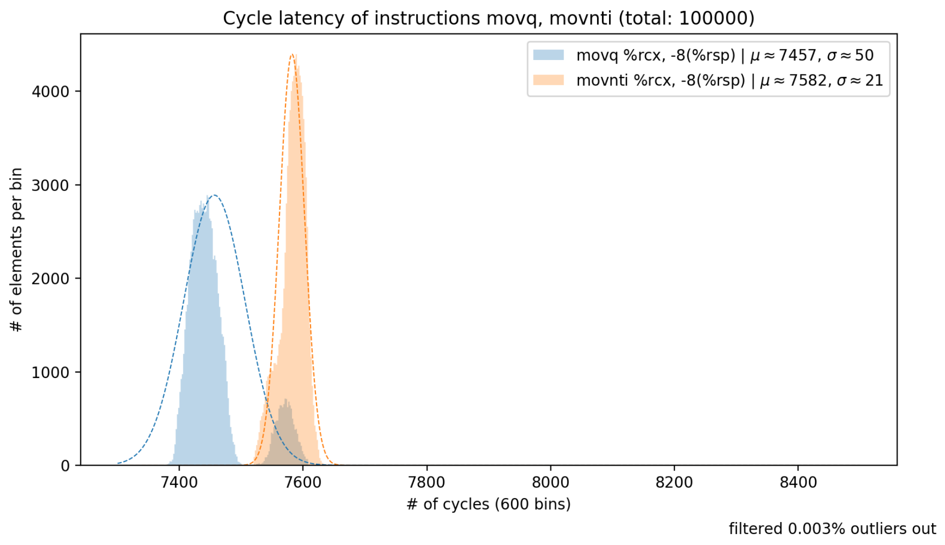


Figure 5.7: Comparison of *mov* and *movnti* which both move a value from a register to memory, but the second one uses a non-temporal hint to tell the processor that this data will not be used again in the near future. Measured on the NUC.

writes are bypassing the cache, then the next read operation has to fetch its value from memory, and will therefore be slow. Indeed, figure 5.9b shows at period 8[6] that most slow reads are preceded by slow writes, i.e. for the slow reads at offset 5 and 7, there are preceding slow writes at offsets 4 and 6.

Reconsidered Double Peak Variants We now reconsider the double peak plots from the beginning and explain how they fit to our hypothesis. Before we start with that, we like to point out that it makes sense that we never saw double peaks before figure 5.1, because only write operations can bypass the cache: We could not observe this behaviour e.g. for *cmov*, because it can only write to registers – not directly to memory. We speculate that other write operations like *add %rcx, -8(%rsp)* do not show double peaks, because it is likely that the result of an addition is used in the near future, so optimizations probably work differently for *add* than for *mov*. We will now explore how our theory explains why some prepare instructions have two equal double peaks while others have not and how this depends on the operands.

First towards the sizes of double peaks: The regular pattern of figures 5.2,

[6]The period seems to be 8 here, because the pattern starts repeating at 16 (and also higher periods that are not shown in figure 5.9b)

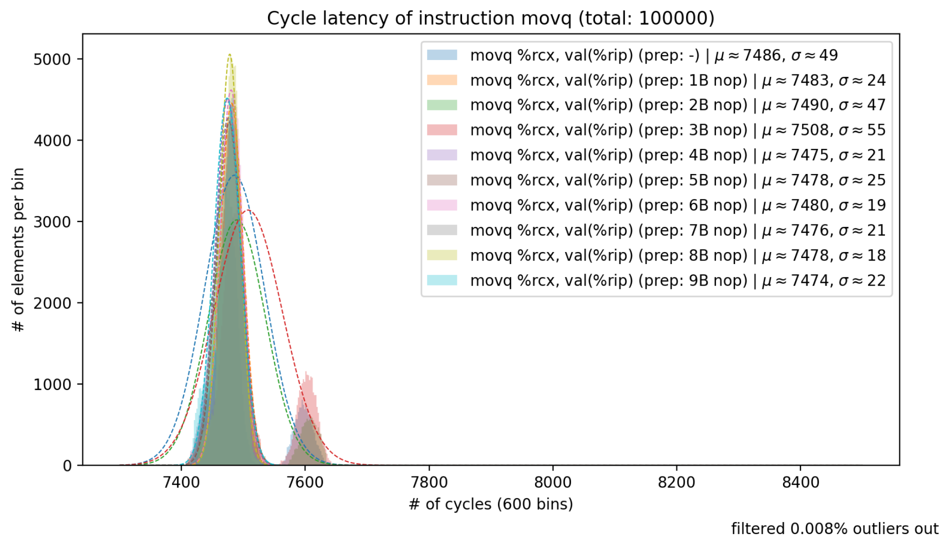
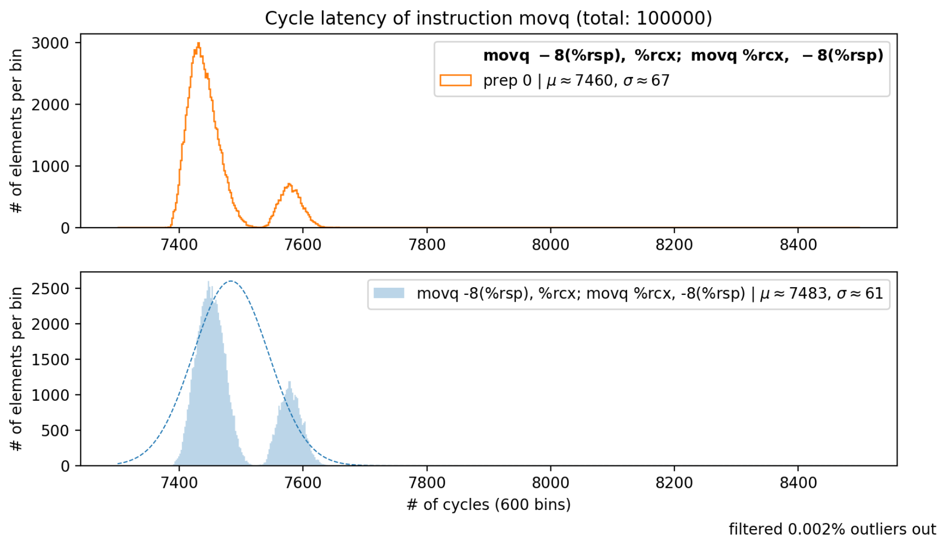


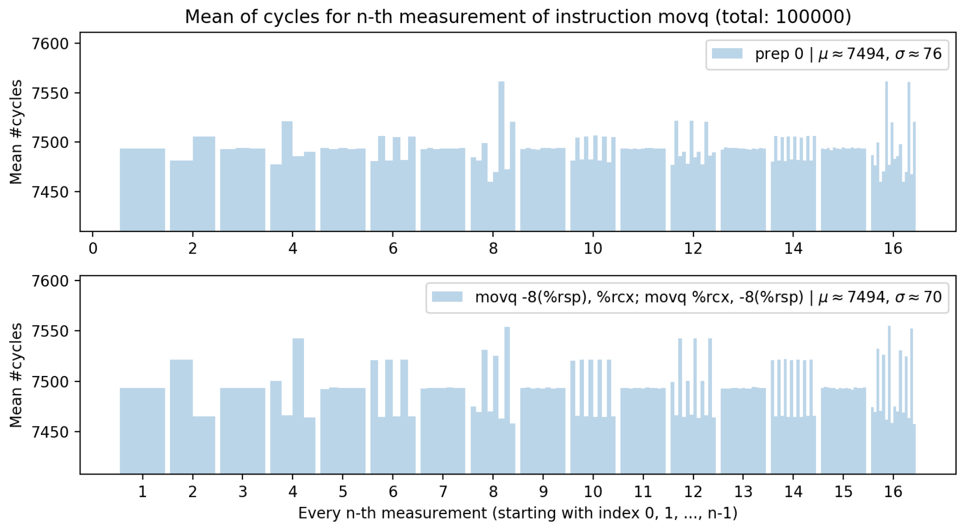
Figure 5.8: Plot of *movq* with *nop*s of different lengths (up to 9B) as prepare instruction. MEasured on the NUC.

5.5 and 5.6 is justified by the fact that the size of an encoded *mov* to memory is 5B and that of *test %rax, %rax* is 3B, so together a test is exactly 8B long. This fits to our theory, because the 16B fetch windows will always look at the same instructions and therefore lead to the same allocation queue for this test case. Every second instruction probably has the same microarchitectural environment because two tests fit in one fetch window. If one state triggers the cache bypass optimization, then it will happen for every second instruction, because they have the same state. Other prepare instructions, like the *mov* in figure 5.3a, do not repeat the same preconditions (structure of the fetch window and the allocation queue) as frequently for their test instruction. Consequently less instructions have a state that triggers the slow behaviour and thus there is a smaller second peak.

Secondly, concerning the operand dependency: In figure 5.3 we saw that the appearance of double peaks also depends on whether the prepare instruction is independent from the test instruction or not. On one hand, independent instructions can be reordered and executed in parallel to the *mov*, on the other hand, the dependent instruction has a read-after-write dependency and has to be stalled. Consistent with our proposition, this changes the allocation queue and thus also the optimization decision to bypass the cache. Apart from influencing the microarchitectural state for an instruction, out-of-order execution has little influence on our test cases because we single-step through the enclave code.



(a) Histogram with prepare instruction



(b) Bar plot

Figure 5.9: Both figures are different plot types for the same test case. They show a *movq* from *%rcx* to stack where we first read this stack location to *%rcx* as a prepare instruction (i.e. both instructions read and write the same location and are thus dependent).

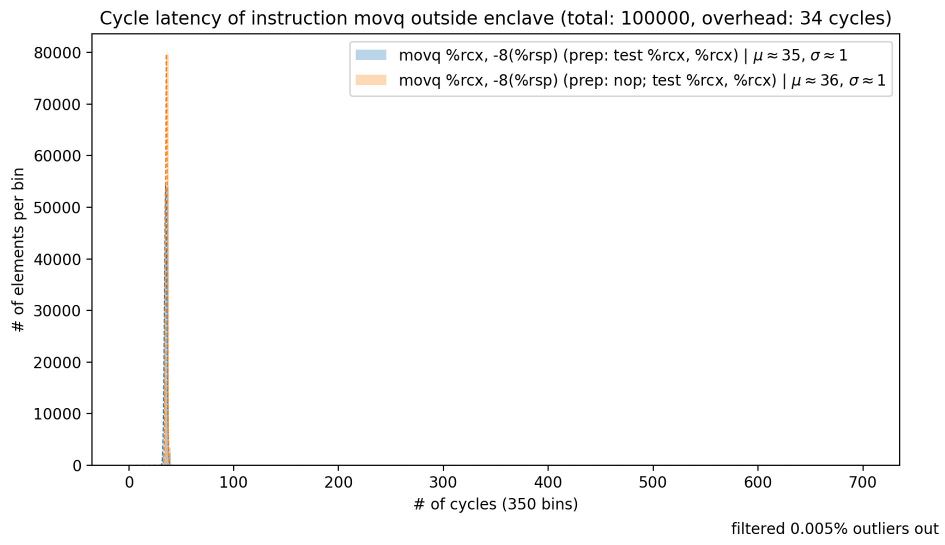


Figure 5.10: Double peak of *mov* from *%rcx* to stack with the dependent prepare instructions *test %rcx, %rcx*. Measured on the NUC but, unlike figure 5.2, outside the SGX enclave.

No Double Peaks Outside Enclaves We briefly discuss why no double peaks appear in measurements outside enclaves. Figure 5.10 shows the same test case that has two peaks of equal height inside the enclave (which we saw earlier in figure 5.2). It clearly has only a single peak, despite having much less noise (because measuring outside is easier as we saw in 3.3). However, this does not necessarily mean that there are no double peaks outside, because our measurement method is rather different outside enclaves: We do not single-step instructions, we execute them in a loop. As we discussed in sections 3.3.2 and 3.3.3, we do this because there is no AEX[7] that preserves our execution environment between measurements. Without this, it is non-trivial to single-step through an execution and measure instruction timings unless we explicitly insert the time measurements in the code – which we do in the loop. This impacts the (micro)architectural state for each test instruction significantly and can very well be the reason why we do not measure double peaks outside enclaves. The same argument holds for the counter method (section 3.3.4), which also does not show double peaks as we can see in figure 5.11. The counter method has the additional problem of being less precise.

Further Implications. The insight that code alignment changes the measurements implies that if we want to measure under the exact same conditions for all test cases, they need to have the same alignment. If we neglect

[7]Asynchronous Enclave Exit, see 2.1.2

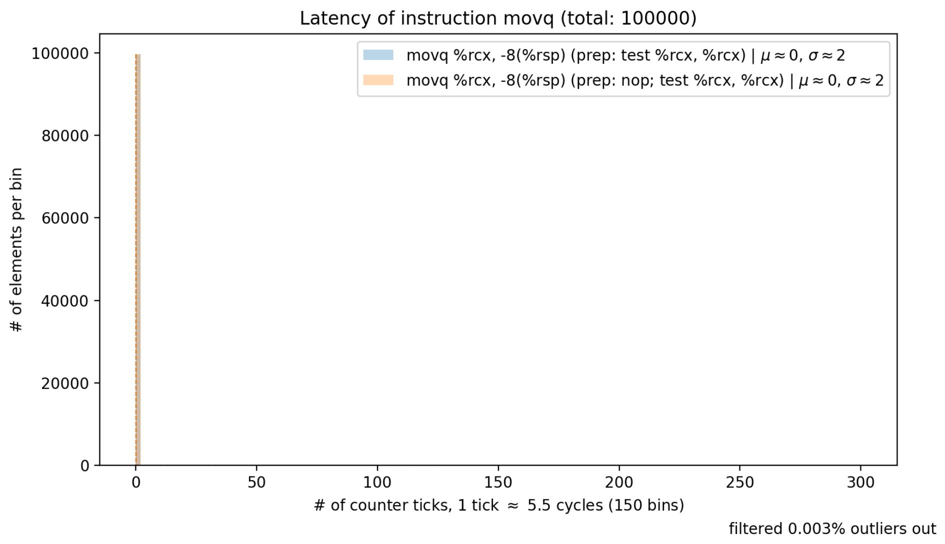


Figure 5.11: Same as figure 5.10 but measured inside enclaves with the counter method.

to do this, then different test cases with the same instruction could show varying measurements because they have different alignments. This is the reason why we padded them in section 3.3, so that each new test case starts aligned to pages. We conclude from this section that the processor architecture, especially optimizations, can influence our measurements in complex ways that might even be difficult to see outside enclaves (because there, the execution environment is not preserved between measurements).

5.2 Poor Man's cmov

In this section, we present a timing side channel attack that is based on the double peaks from the previous section 5.1. We will leak which branch of a (specially crafted) program running in an SGX enclave has been taken. Both branches use the same instructions (*mov* and *test*) with different registers as operands. Despite being almost identical, we are still able to distinguish the branches, and thereby show our improved precision in comparison with Nemesis [7].

The victim code (from which we will leak which branch it took) is shown in code snippet 3. The basic idea behind this example is to build something similar to a *cmov*: Based on some secret, we will either move a value from *%rdx* or *%rcx* to a memory location. However, instead of using *cmov*, we do this manually with two branches. We will now discuss code snippet 3 in more detail:

1. *Lines 1-2*: Align the measured instructions such that one is slow and the other is fast. Line 1 generally aligns this function to a page boundary (here 4KB) and *.space* does a more fine-grained alignment on byte granularity which could also be achieved with instructions that have an encoding of the required size.
2. *Lines 4, 19, and 28*: Set the *observe* flag (cf. 4.6) to start measuring instructions. The register *%rdi* contains the pointer to this shared variable.
3. *Lines 5-9, 15-18, and 24-27*: Necessary such that the double peaks occur. We assume, consistent with our hypothesis from 5.1, that this creates the discussed microarchitectural state that triggers the cache bypassing optimization for one of the target *mov*s, but not the other.
4. *Lines 11-12:* Take the if-branch in case the secret value is zero, otherwise jump to the else branch (that starts at line 22). The secret value, passed as an argument to the function *asm_poor_mans_cmov*, is stored in *%rsi*.
5. *Lines 14 and 23:* The measured instructions; One in the if-branch, moving *%rdx* to the stack, and the other in the else-branch, moving a different register (*%rcx*) to the same stack location.

We will now explain how we perform and evaluate a side-channel attack on this program and present the results in comparison with Nemesis [7]. As we will discuss in more detail in section 6.1, Nemesis is a side channel attack that is based on SGX-Step as well. In contrast to our tool, it does not consider the challenges presented in section 4 and we will show that it is less precise for this reason. We use the following methodology to evaluate the side-channel accuracy of both tools on the poor man's cmov example: We generate and store random secret bits before starting the enclave. Inside the enclave, we run the poor man's cmov code in a loop and take the branches according to our pre-generated secrets. We always measure 1000 branches in one measurement and we perform 1000 of those measurements in different enclaves and show the mean percentage of correctly guessed branches as well as the standard deviation. Apart from code snippet 3, which we call the short version, we also measure a longer program that uses more padding inside both branches. We can only count measurements that captured the correct number of instructions, because otherwise we do not know where we desynchronized with the instruction stream, i.e. where we started to measure the wrong instructions. Therefore, we calculate the mean and standard deviation exclusively from correct measurements and indicate the percentage of correct measurements that were produced. If the used tool aborts, e.g. because it detects problems with the measurement, we do three retries before we abort and count this measurement as unsuccessful. Both tools, Nemesis and ours, use the interrupt method (cf. 3.3.3) to single-step the execution of the victim code and log all instruction timings. Afterwards,

Code Snippet 3 Poor Man's *cmov* Victim Code

```
 1:     .align 0x1000
 2:     .space 0x7
 3: asm_poor_mans_cmov:
 4:     movb $1, (%rdi)                  ▷ start counting instructions
 5:     test %rax, %rax
 6:     movq %rcx, -8(%rsp)
 7:     test %rax, %rax
 8:     movq %rcx, -8(%rsp)
 9:     test %rax, %rax
10:
11:     test %rsi, %rsi
12:     jnz .elseBranch
13:                                           ▷ branch for %rsi = 0
14:     movq %rdx, -8(%rsp)                   ▷ measured instruction
15:     test %rax, %rax
16:     movq %rdx, -8(%rsp)
17:     test %rax, %rax
18:     movq %rdx, -8(%rsp)
19:     movb $0, (%rdi)                   ▷ stop counting instructions
20:     ret
21:                                           ▷ branch for %rsi = 1
22: .elseBranch:
23:     movq %rcx, -8(%rsp)                   ▷ measured instruction
24:     test %rax, %rax
25:     movq %rcx, -8(%rsp)
26:     test %rax, %rax
27:     movq %rcx, -8(%rsp)
28:     movb $0, (%rdi)                   ▷ stop counting instructions
29:     ret
```

we extract the execution times of the measured instructions (in snippet 3 they are on line 14 respective 23) and sort them. We categorise the slower half as having executed the if-branch (which is aligned to be the slower *mov*) and the other half as else-branches. We do this simple categorisation because in expectation 50% of all measurements take either branch. We can evaluate if we recognized the correct branches by comparing with the pre-generated secret bits. Figure 5.12 shows our execution time measurements of the *mov*s in the two branches of the long version of the poor man's cmov example. Since those measurements form two clearly distinguishable peaks for the two branches, it is not surprising that we can detect them with high accuracy, as we will show in the next paragraph. However, the shorter version (shown in code snippet 3) is more difficult to distinguish, since there is

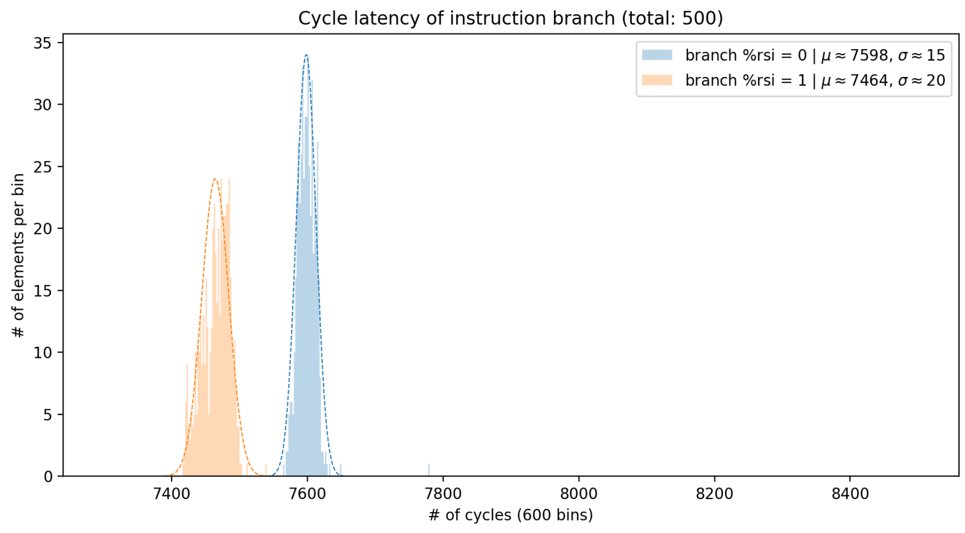


Figure 5.12: Compares the execution time of the *mov* in the true branch (*%rsi = 0*) with the *mov* in the other branch (*%rsi = 1*).

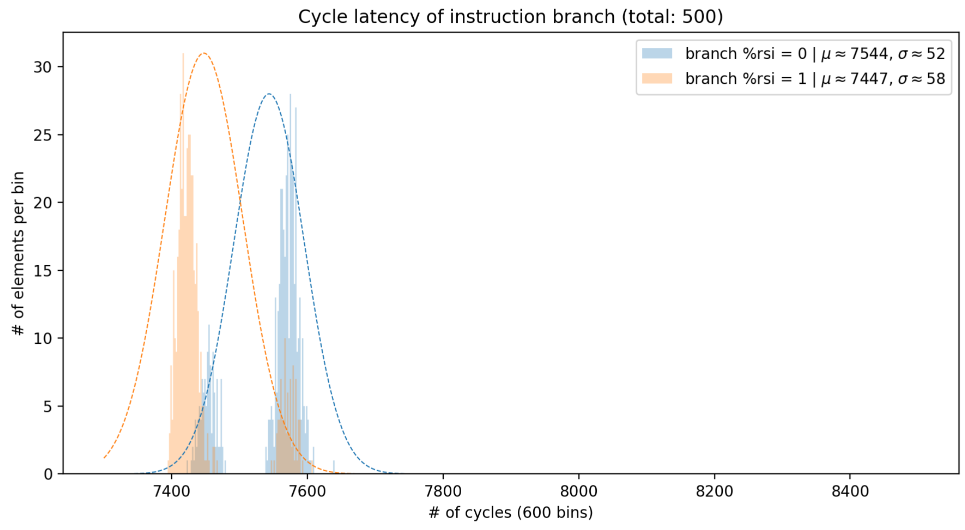


Figure 5.13: Compares the execution time of the *mov* in the true branch (*%rsi = 0*) with the *mov* in the other branch (*%rsi = 1*).

more noise for both branches as figure 5.13 shows.

Figure 5.14 shows the absolute number of correctly guessed branches in a bar diagram and table 52 provides the numerical results of this comparison in more detail. We test the short version of the poor man's cmov example – which we abbreviate with V1 – for two different offsets, which specify the alignment (line 2 for code snippet 3). This is relevant for the short version, because the branches are compact enough that they fit into a single cache line (the instructions from line 12 to 29 encode to only 52 bytes and thus fit in a 64B cache line). Therefore we measure V1 once, such that the if-clause on line 12 is aligned to 64B and the branches are in one cache line and once such that they spread across two cache lines. The middle bars in figure 5.14 show that it is more difficult to detect V1 in a single cache line than any other version, since those are the lowest bars for every combination of devices and measurement tools. The dotted line in figure 5.14 shows the threshold of 500 correct guesses, which can be achieved by randomly guessing branches. For V1 in a single cache line, our tool can still leak clearly more than half of the branches on the NUC and only slightly more than half on the laptop. Nemesis is below the threshold because it only captures 27.7% of all measurements correctly, as table 52 shows. For the ones that it captures, it is more accurate than our tool (it captures 59.5% of them compared to the 54.6% that we detect correctly on the laptop). We noticed that our prediction tends to be worse for faster enclaves[8], therefore it is possible that Nemesis misses those cases (which does not affect the mean) while we make inaccurate predictions that lower our average accuracy. In general, we see that the branch detection of our tool is better on the NUC than on the laptop: The blue bar in figure 5.14 is the highest for all versions of the victim code. We also see that we can leak the secret of the long version (called V2 in figure 5.14) very precisely: On average 98.1% of the guesses are correct on the NUC and 97.5% on the laptop (see table 52). Nemesis has an absolute number of correct guesses for the long version that is only slightly above the threshold because only 55.6% of its branch guesses are correct. We conclude that our tool reliably captures all measurements correctly and can exploit this timing side-channel with significantly higher accuracy than Nemesis in most cases.

[8]I.e. enclaves where ERSEUME and AEX take less time (cf. 4.1)

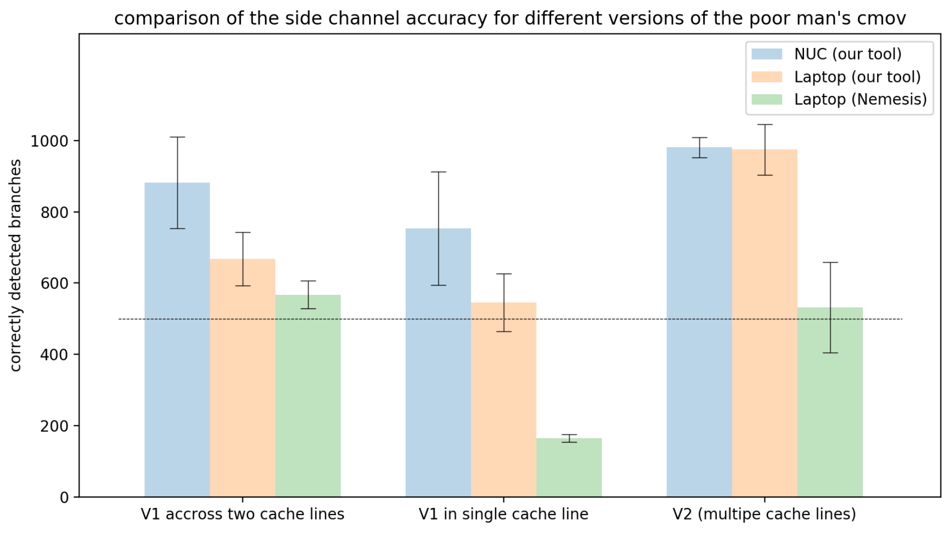


Figure 5.14: Comparison of the side-channel accuracy on the poor man's cmov examples of our tool and Nemesis [7]. We measure 1000 branches in one measurement and show the average of the number of correctly guessed branches over 1000 measurements. We look at two different versions of the poor man's cmov example: The short version, called V1, of the poor man's cmov example is shown in code snippet 3, the long one (V2) has more padding inside the loops. We test two alignments for V1, one of them such that both branches together fit into a single cache line. The dotted line shows the threshold at 500 correct guesses, that could be achieved with random guessing.

Table 52: Comparison of the side-channel accuracy on the poor man's cmov examples of our tool and Nemesis [7]. We measure 1000 branches in one measurement and show the mean percentage of correctly guessed branches for 1000 measurements. A correctly captured measurement is one that measured the correct number of instructions. The short version, called V1, of the poor man's cmov example is shown in code snippet 3, the long one (V2) has more padding inside the loops. We test two alignments for V1, one of them such that both branches together fit into a single cache line.

	Cache Lines	Correctly Captured	Mean	Standard Deviation
Our Tool (NUC)				
short version V1	2	100%	88.2	12.9
	1	100%	75.4	15.9
long version V2	> 2	100%	98.1	2.8
Our Tool (Latitude)				
short version V1	2	100%	66.8	7.5
	1	100%	54.6	8.1
long version V2	> 2	100%	97.5	7.1
Nemesis (Latitude)				
short version V1	2	94.1%	60.3	4.1
	1	27.7%	59.5	3.7
long version V2	> 2	95.6%	55.6	13.3

Chapter 6

Conclusion and Related Work

In this chapter we will first summarize the contributions of this thesis, including the improvements over the original SGX-Step framework. Furthermore, we will mention possibly interesting research directions for which our tool could be useful.

6.1 Comparing with SGX-Step

We distinguish between our fundamental changes to the SGX-Step framework and our additions to the measurement technique. Nemesis [7] (which is from the same authors as SGX-Step) goes in the same direction as our work and also modifies SGX-Step to do time measurements. However, Nemesis is less precise than our tool (cf. 5.2). All improvements that we summarise in the following are improvements compared to Nemesis as well, not only SGX-Step.

The major improvements of the SGX-Step framework are following:

1. Multiple test cases can be measured easily inside the same enclave and their plotting is automated. (4.1)
2. Better instruction serialization barriers before and after the measurement to reduce noise. (3.2.1, 3.3.3)
3. Reduced cache pollution due to constant time measurement code and delayed instruction filtering and logging (4.3, 4.4)
4. Multiple consistency checks to detect imprecise APIC timers, misconfigured tests and instructions that can execute in two parts. As well as another approach to allow thousands of test code pages while still protect against false positives. (4.5, 4.6)
5. The notion of initial, prepare and test instructions to logically structure test cases. (3.3.1)

Enhancements of the measurement technique and new features are:

1. New plot types to gain more insights about measurements:
    - 1.1. Histogram with prepare instructions (3.4.2) to prevent hidden information like the slow double peak in section 5.1.
    - 1.2. Measurements-over-time plot to display unprocessed data which helps to recognize patterns. (3.4.3, 5.1)
    - 1.3. Bar plot to investigate which instruction sequences are slower than others. (3.4.4)
2. Other measurement options that all share the same test case specifications:
    - 2.1. Measure timings outside the enclave to see differences to instructions inside the enclave. (3.3.2)
    - 2.2. Counter method to cross validate results of SGX-Step (although they have lower precision). (3.3.4)
3. Filtering of outliers in general and especially at page boundaries to get a meaningful variance and compact plots. (3.4, 4.2)
4. Demonstration of how we set flags (for operations like *cmov*) and write to memory. (4.8, 4.9, 5.1)
5. Theoretically, we can apply deconvolution to filter out the noise of AEX and ERESUME, however, in practice better zero steps measurements would be needed for this feature to be useful. (4.10)

6.2 Further research

This section is a collection of some ideas that could be interesting to further investigate.

Different Entry and Exit Times. An unanswered question is why different enclaves take varying time for AEX and ERESUME (which makes different enclaves incomparable as we saw in section 4.1). It would be interesting to investigate if those differences for example depend on which memory pages the enclave has to restore: If there are slower and faster pages, can we influence which ones are used to make different enclave measurements comparable? Or does this depend on what the enclave has to restore (e.g. the number of registers that were actually used in the enclave) and could this be used as a side channel?

Poor Man's cmov The poor man's cmov example could be investigated further: Does observing performance counters of the processor provide further evidence that the double peaks are caused by cache bypassing (e.g. when there are much less L1 cache hits for instruction measurements that have double peaks)? Are there other instructions or combinations of instructions that show double peaks? For this purpose, finding candidates with automated fuzzing could be interesting. Additionally, security relevant libraries could be searched for code sections that are vulnerable to this side channel.

Method Comparison. It would also be interesting to further compare results from the interrupt to both the counter method and measurements outside the enclave. This could also provide further insights on how exactly SGX-Step works (see paragraph 2.2.4). For example, *movnti* is only slower when we measure it with the interrupt method: Figure 6.1 shows that we can measure neither with the counter method inside the enclave nor outside a difference between *movnti* and *mov*. In general, trying to reproduce the double peak measurements outside enclaves could provide further insights on the conditions under which they occur. To achieve this, it might be necessary to simulate the effect of AEX (e.g. clearing the pipeline).

Investigate the Restored Execution Environment. The enclave restores the execution environment of the code it is running before it resumes it, i.e. the code is unaware that it was interrupted. A very interesting question is, if this makes it possible to measure details about the microarchitectural state that we cannot obtain outside enclaves? For example, we perform our time measurements by single-stepping through a long slide of instructions. It is not trivial to reproduce such measurements outside, since the code used for single-stepping would probably affect the microarchitectural state (e.g. the pipeline and buffers).

Multi-Steps. A further improvement to the framework would be to measure multi-steps, i.e. a controlled number of instructions that are executed together. This could be used to get more insights on the interaction of different operations, the state of the pipeline or even branch prediction. However, it could be more complicated than it seems to realise this. Remember that we use an imprecise timer to interrupt a single instruction. We speculated in section 2.2.4 that this works accurately because the interrupt must only arrive in the window during which the first instruction is in the pipeline. If we measure multiple instructions, hitting the right point could be more difficult, because after the pipeline is filled, a new instruction retires potentially at every cycle. However, an idea to still measure this would be to following: We pad the group of instructions that we want to measure with *nop*s, then we increase the timer such that the whole group certainly exe-

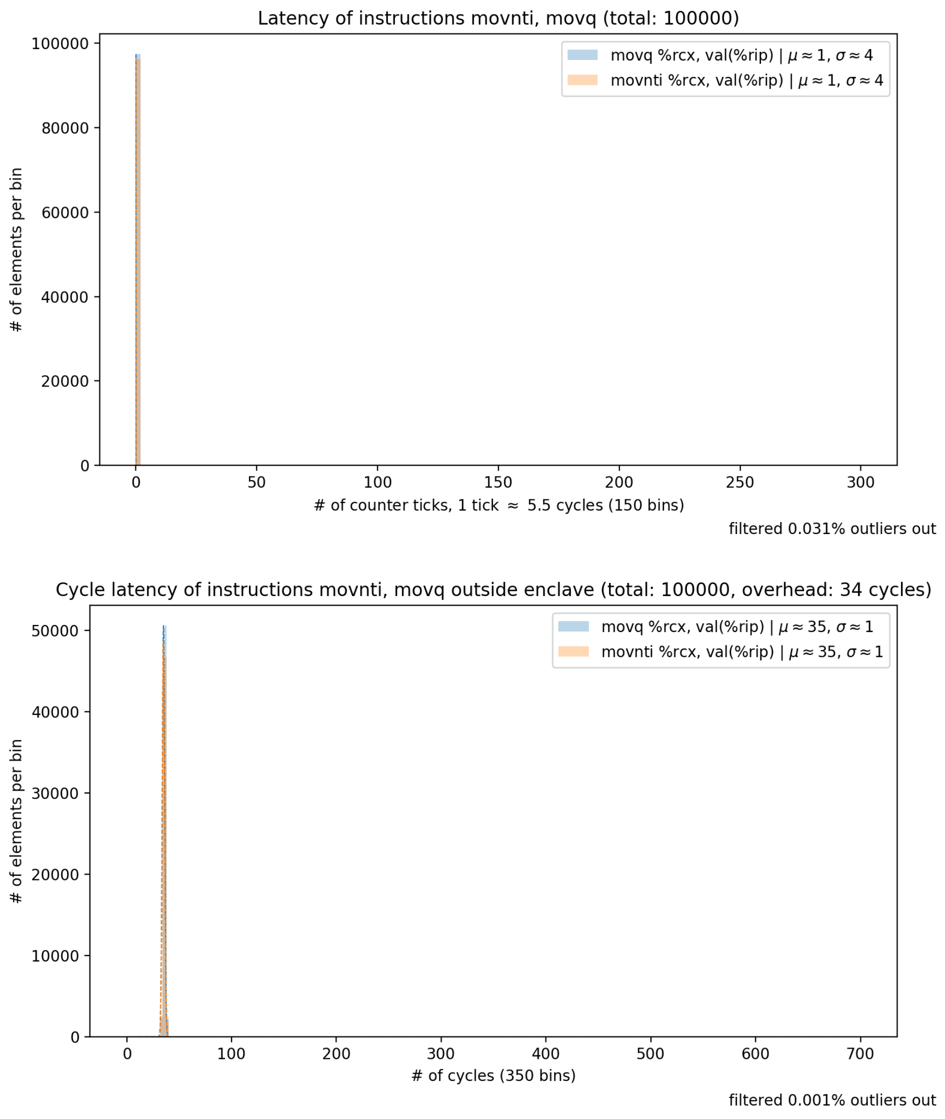


Figure 6.1: Comparison of *movnti* and *mov* that move from a register to memory. Neither with the counter method (upper plot) nor outside the enclave (lower plot) can we measure a difference like we see with the interrupt method. Measured on the NUC.

cutes. When the interrupt arrives, we switch back to single-stepping and count how many *nop* instructions were not yet executed. We might want to add an easily detectable 'stopper instruction', e.g. *rdrand*, which takes many more cycles than *nop*, to know where the padding ends and the next test starts. Using this, we could detect exactly after how many instructions the interrupt arrived.

Bibliography

[1] Choosing Histogram Bins. http://docs.astropy.org/en/stable/visualization/histogram.html. Visited on 2019-05-17.

[2] Skylake (client) - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)#Architecture. Visited on 2019-05-21.

[3] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *OSDI*, 2016.

[4] Iddo Bentov, Yan Ji, Fan Zhang, Yunqi Li, Xueyuan Zhao, Lorenz Breidenbach, Philip Daian, and Ari Juels. Tesseract: Real-Time Cryptocurrency Exchange using Trusted Hardware. *IACR Cryptology ePrint Archive*, 2017:1153, 2017.

[5] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. *CoRR*, abs/1702.07521, 2017.

[6] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *SysTEX@SOSP*, 2017.

[7] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *ACM Conference on Computer and Communications Security*, 2018.

[8] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. *International Conference*

on Architectural Support for Programming Languages and Operating Systems - ASPLOS, 48, 2013-03.

[9] GNU compilers. Intel 386 and AMD x86-64 Options. https://gcc.gnu.org/onlinedocs/gcc-4.4.7/gcc/i386-and-x86_002d64-Options.html, 2008. Visited on 2019-05-19.

[10] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[11] Agner Fog. Instruction tables. https://www.agner.org/optimize/instruction_tables.pdf, 2018. Visited on 2019-05-21.

[12] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs. https://www.agner.org/optimize/microarchitecture.pdf, 2018. Visited on 2019-05-19.

[13] Jeffrey Goldberg. Using Intel's SGX to keep secrets even safer. https://blog.1password.com/using-intels-sgx-to-keep-secrets-even-safer/, 2017-01-03. Visited on 2019-02-26.

[14] Intel Corporation. *Intel® 64 and IA-32 Architectures Developer's Manual: Volume 1*. Number 253665-060US. September 2016.

[15] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2*. Number 325383-060US. September 2016.

[16] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3D*. Number 332831-060US. September 2016.

[17] Pratheek Karnati. Data-in-use protection on IBM Cloud using Intel SGX. https://www.ibm.com/blogs/bluemix/2018/05/data-use-protection-ibm-cloud-using-intel-sgx/, 2018-05-10. Visited on 2019-02-26.

[18] David Kohlbrenner and Hovav Shacham. On the effectiveness of mitigations against floating-point timing channels. In *USENIX Security Symposium*, 2017.

[19] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX Amplifies The Power of Cache Attacks. *CoRR*, abs/1703.06986, 2017.

[20] Gabriele Paoloni. How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures. https:

//www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf, 2010-09.

[21] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 431–446, Washington, D.C., 2015. USENIX Association.

[22] Mark Russinovich. Azure confidential computing. https://azure.microsoft.com/en-us/blog/azure-confidential-computing/, 2018-05-09. Visited on 2019-02-26.

[23] Surenthar Selvaraj. Overview of an Intel Software Guard Extensions Enclave Life Cycle. https://software.intel.com/en-us/blogs/2016/12/20/overview-of-an-intel-software-guard-extensions-enclave-life-cycle, 2016-12-20. Visited on 2019-02-26.

[24] DEREK B. Simon Johnson, Dan Z. Intel® SGX: Debug, Production, Pre-release – What's the Difference? https://software.intel.com/en-us/blogs/2016/01/07/intel-sgx-debug-production-prelease-whats-the-difference, 2016-01-07. Visited on 2019-03-25.

[25] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 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, August 2018. See also technical report Foreshadow-NG [26].

[26] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report*, 2018. See also USENIX Security paper Foreshadow [25].

[27] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656, May 2015.

Appendix A

Serializing With lfence and sfence

In this appendix, we explain in detail how we can serialize instructions to reduce the noise of time measurements when *cpuid* is not available. Usually, this is not the case and it is better to use the methodology with *cpuid* described in paragraph 3.2.1. However, we actually have to use the approach shown in code snippet 4 for the counter method (chapter 3.3.4) since it runs inside enclaves.

We first assume, for simplicity, that all *slots* (on lines 3, 5, 9, and 11) of code snippet 4 are empty. I.e. we only use the instruction *rdtsc*, which reads the time stamp counter of the processor, to measure instruction execution times. After discussing the general idea, we will show how this measurement can be improved by using fences. Since the processor's time stamp counter is a 64 bit value, *rdtsc* puts the higher 32 bits into register *%edx* and the lower 32 bits into *%eax*[1]. Lines 6-7 and 12-13 store those values into variables that we can access later to log the execution time. Those variables for the first timer are pre-fetched into the cache on lines 1-2, so that the following *mov* instructions are faster.

The reason why we do not use *rdtsc* alone is because previous or later instructions could distort our measurements (out-of-order execution). Memory fences (*sfence*, *lfence* and *mfence*) can be used to ensure that all previous memory operations become visible before the next instruction executes. We look at the impact of different combinations of those fences on the measurement of an *add* instruction outside the enclave. Snippet 4 shows the assembly code for this time measurement. The slots mark places where it makes sense to insert one or more fences. To simplify the notation, we introduce the following abbreviations: S for *sfence*, L for *lfence*, M for *mfence*, and X for no fence. Then we refer for example with MXXX to the pattern that has an *mfence* in slot 1 (on line 3) and no fences in all other slots.

[1]This way, the instruction is identical on 64 and 32 bit architectures

Code Snippet 4 Measure add outside enclave

```
 1: prefetch (tsc_before_lower)
 2: prefetch (tsc_before_upper)
 3: ⟨slot 1⟩
 4: rdtsc
 5: ⟨slot 2⟩
 6: mov %eax, (tsc_before_lower)
 7: mov %edx, (tsc_before_upper)
 8: add $1, %rax                        ▷ measured instruction
 9: ⟨slot 3⟩
10: rdtsc
11: ⟨slot 4⟩
12: mov %eax, (tsc_after_lower)
13: mov %edx, (tsc_after_upper)
```

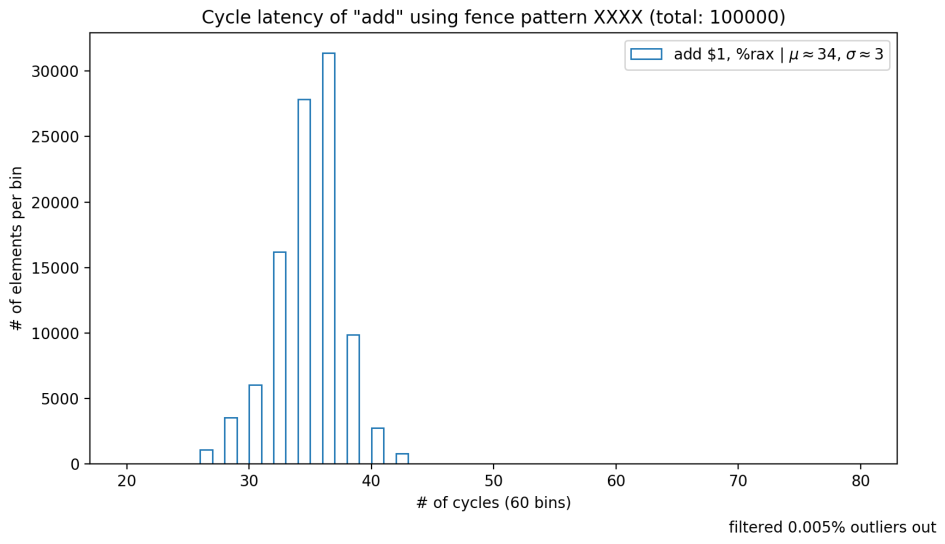


Figure 1.1: Latencies of a single add instruction when using no fences. Measured on the NUC.

Figure 1.1 shows the histogram of a measurement without any fences (XXXX). There is one bin for each cycle latency and the height of the bin represents how many measurements observed this number of cycles. Although the *add* instruction takes a single cycle, the mean ($\mu = 34$) is much larger, because we also measure moving the time stamp to memory. In fact, we measure everything from line 5 to line 9 in snippet 4.

Compared to MXXX in figure 1.2, using no fences has a higher standard deviation. The mean is also higher because other instructions in the out-of-

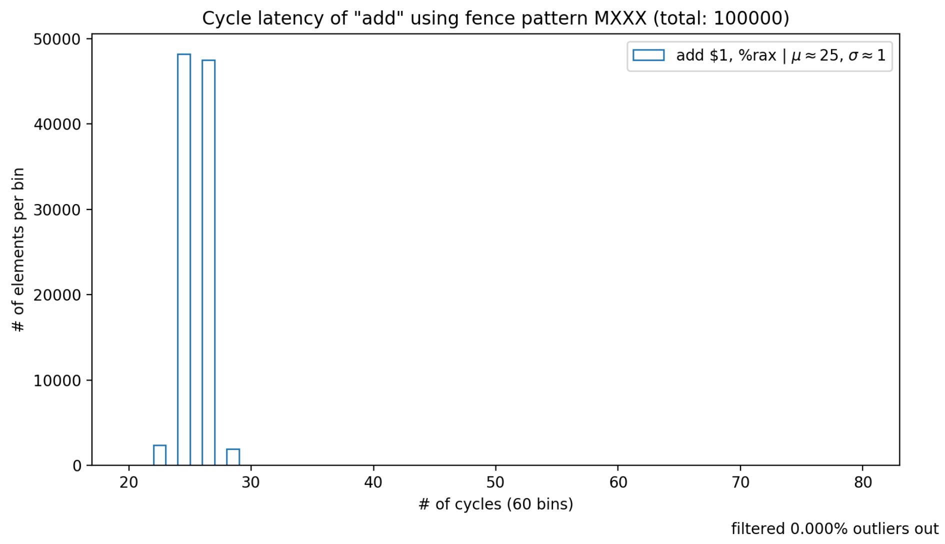


Figure 1.2: Latencies of a single add instruction when using a *mfence* before the first *rdtsc*. Measured on the NUC.

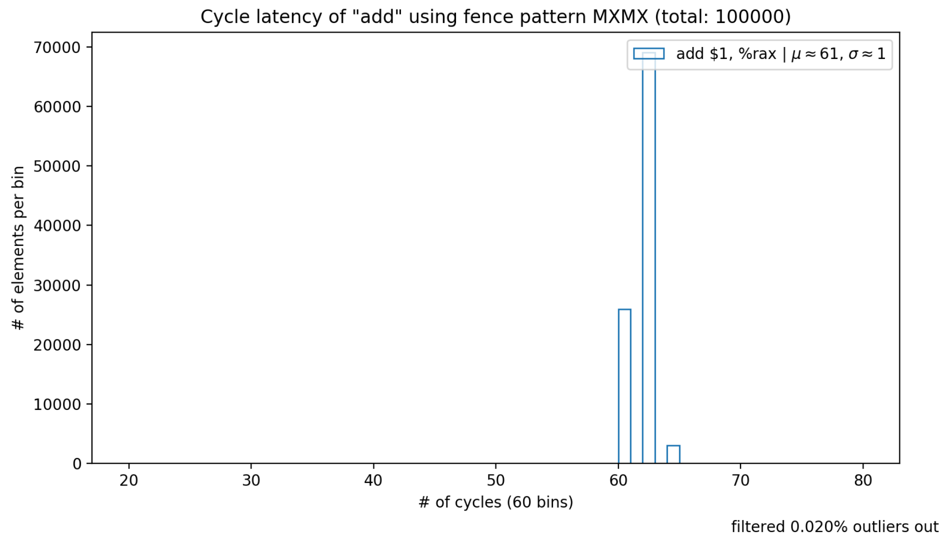


Figure 1.3: Latencies of a single add instruction when using a *mfence* before each *rdtsc*. Measured on the NUC.

order pipeline can slow down *rdtsc*. MXXX is the configuration that was used in Nemesis ([7]).

However, we found that also using a fence before the second *rdtsc* (that means using MXMX) improves the results by concentrating the measurements further to one single cycle count as figure 1.3 shows.

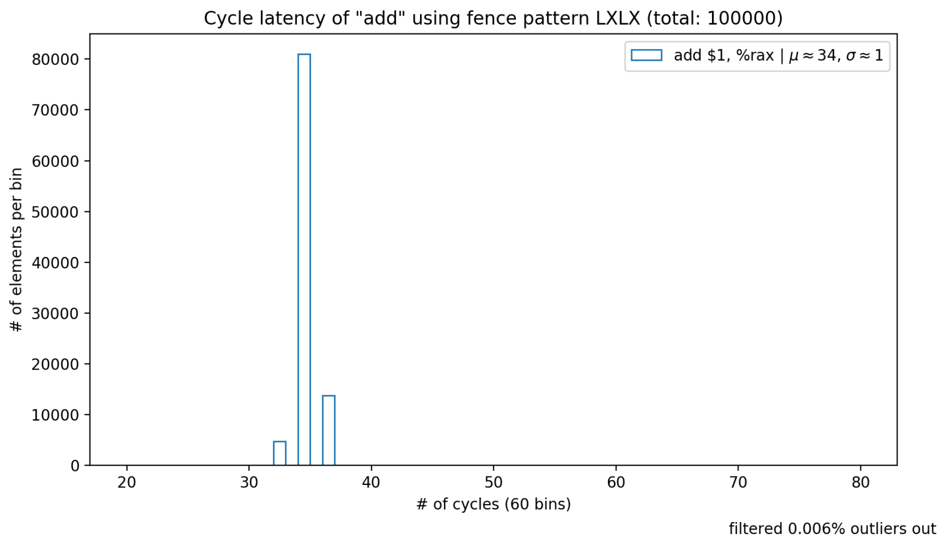


Figure 1.4: Latencies of a single add instruction when using a *lfence* before each *rdtsc*. Measured on the NUC.

According to the Intel manual [15] *mfence* 'does not serialize the instruction stream' (although on most hardware it seems to do so nonetheless). Thus, we replaced it by *lfence*, which is also the official recommendation to use before *rdtsc* in the Intel Software Developer's Manual [15]. Figure 1.4 shows the result of this, which concentrates even more measurements into a single bin. It has a lower mean, since it only serializes load instructions and thus has less overhead.

In the end, we settled to use the pattern (SL)(LS)(SL)X which uses both store and load fences in the first three slots. We order the fences this way so that the load fences are always closest to *rdtsc*, because they are the ones that actually serialize the instruction stream. Adding fences in slot 2 has the benefit that no instructions after *rdtsc* can start executing in parallel, which improves the stability of the measurement. This can be seen in the smaller standard deviation in figure 1.5 for the pattern (SL)(LS)(SL)X compared with figure 1.6 for LXLX (which even has two peaks). Those tests were performed on the laptop, because for the NUC in figure 1.7, there is hardly any difference visible compared to LXLX. This might be, because the effect of the store fences is too small to be visible because probably not many instruction and especially not many store instructions are executed around the time measurement. Nonetheless it is justified to add them because this is different in general, when more complex instructions than just a simple *add* are measured.

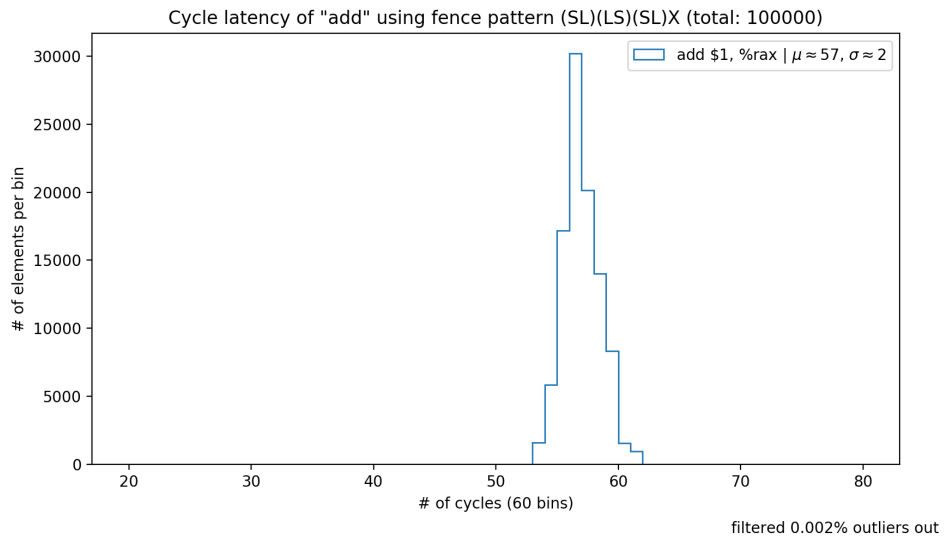


Figure 1.5: Latencies of a single add instruction when using both *lfence* and *sfence* before and after the first *rdtsc* and before the second one. Measured on the Latitude laptop.

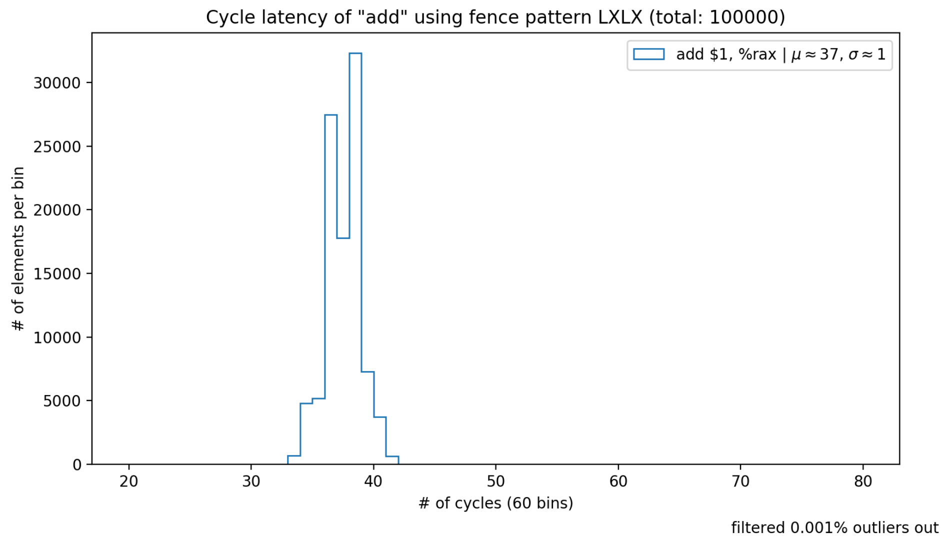


Figure 1.6: Latencies of a single add instruction when using *lfence* before each *rdtsc*. Measured on the Latitude laptop.

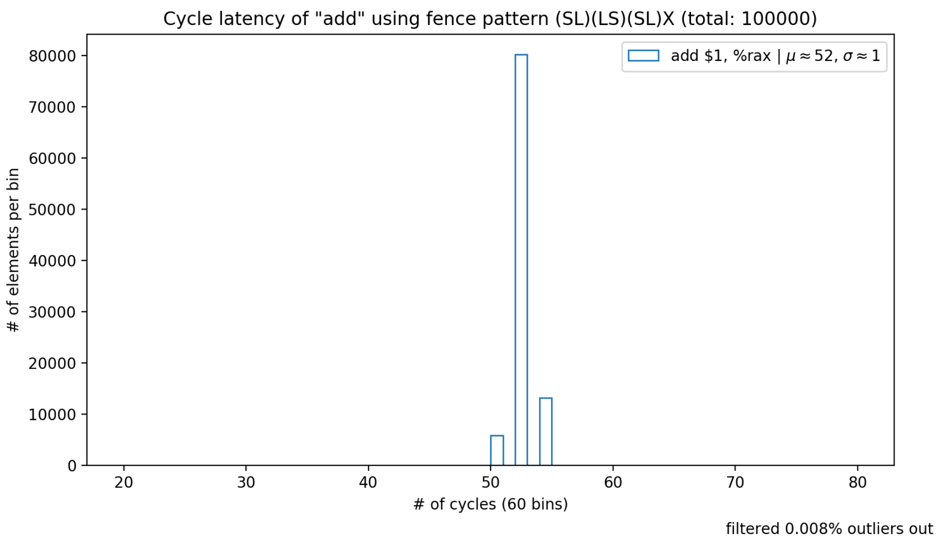


Figure 1.7: Latencies of a single add instruction when using both *lfence* and *sfence* before and after the first *rdtsc* and before the second one. Measured on the NUC.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

On Performing Accurate Time Measurements of SGX Enclave Instructions

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	First name(s):
Haller	Miro

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	Signature(s)
06.05.2019, Beinwil am See	M. Haller

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.