Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend

Ivan Puddu, Moritz Schneider, Miro Haller, Srdjan Čapkun
Department of Computer Science
ETH Zurich

***Abstract*—We introduce a new timing side-channel attack on Intel CPU processors. Our *Frontal* attack exploits the way that CPU frontend fetches and processes instructions while being interrupted. In particular, we observe that in modern Intel CPUs, some instruction's execution times will depend on which operations precede and succeed them, and on *their virtual addresses*. Unlike previous attacks that could only profile branches if they contained different code or were based on conditional jumps, the *Frontal* attack allows the adversary to distinguish between instruction-wise identical branches. As the attack requires OS capabilities to set the interrupts, we use it to exploit SGX enclaves. Our attack demonstrates that a realistic SGX attacker can always observe the full enclave instruction trace, and secret-depending branching should not be used even alongside defenses to current controlled-channel attacks. We show that the adversary can use the *Frontal* attack to extract a secret from an SGX enclave if that secret was used as a branching condition for two instruction-wise identical branches. The attack can be exploited against several crypto libraries and affects all Intel CPUs.**

I. Introduction

Today's computing world runs in the cloud. Massive datacenters maintained by cloud providers are the infrastructure upon which companies and most of the internet are increasingly relying on [8]. For many use cases renting computing resources is cost-effective and convenient. Resources can dynamically scale up when demand is high, all the while not having to maintain them. Security-wise, on the other hand, cloud computing is a much harder sell. Offloading computation and data to a third party raises questions about confidentiality and integrity. Not only could a remote attacker rent the same server and be co-located with the victim, but the provider itself could be malicious. In such a scenario, hypervisors and operating systems (OS), which usually provide isolation, can be easily compromised and thus offer little to no assurance in terms of security.

This setting has been a driving force in recent efforts to develop trusted execution environments (TEEs). While there are many TEE proposals [3], [10], [11], [14], [33], [43], [46], they are unified in their goal: providing an integrity and confidentiality oasis in an environment ruled by malicious operating systems and hypervisors. The fundamentals for this oasis are rooted in the lowest level of the computing stack: the CPU. When application security is provided through CPU primitives, the layers above need not be trusted. Among all the TEE proposals, Intel SGX [10] is the most widely deployed one, being available in almost every modern consumer CPU Intel manufactures. It protects applications by running them in so-called "enclaves". SGX authenticates and encrypts enclave's memory that crosses the CPU boundary, and blocks any other software in the system, including OS and hypervisor, from accessing enclave code and data. Nevertheless, as protected as they might be, enclaves do not execute in isolation. Enclaves share resources with other applications in the same system, particularly memory and CPU time. By design, SGX leaves the (considered malicious) OS in charge of managing these resources.

However, whenever shared resources are involved, so are side-channels. Researchers were quick to point out this shortcoming of SGX [5], [10], [17], [39], [60], casting a shadow of doubt into enclaves' ability to provide confidentiality, one of the core TEE goals. Intel acknowledged the problem but shifted the burden of protecting against side-channels to enclave developers [29]. Curbing side-channels is not trivial, and in the case of SGX, it is particularly challenging due to the role the OS plays. To manage the system resources, the OS is responsible for the enclave scheduling, memory paging, and interrupts and exceptions handling, to name a few. These OS capabilities, which the attacker controls, decrease the noise of traditional side-channel attacks [5], [34] and enable new types of side-channels, called controlled-channel attacks [60].

The first controlled-channel attacks allowed the adversary to observe enclave accesses at page granularity (4 KiB) without any noise, by merely abusing memory paging. Revoking permissions to the enclave's pages leads to page-faults, which in turn give the OS attacker a trace of every page the enclave accesses. Initial defenses that focused on detecting an abnormal number of page-faults [50] just prompted the emergence of stealthier attacks that do not rely on faults [7], [57]. In response to these attacks, Intel officially recommends SGX developers to place specific data and code within a page [27]. Controlled channels, however, do not stop at the page boundary. OS capabilities can be used to enhance cache attacks [5], [17], [39], and to extract enough information from the branch prediction unit (BPU) to give the attacker a branch granularity view of the victim [13], [25], [34]. As this undermines defenses against paging-based controlled channels, further defenses leveraging the coarse timing resolution of the attacker and the inability of BPU attacks to leak unconditional branches were proposed [34]. Nemesis [56] later showed that it is possible to time each instruction through interrupts, invalidating the assumptions on the best temporal resolution available to the attacker. Therefore, successive defenses [24] relied upon randomizing control-flow through unconditional jumps to protect enclaves.

The current understanding of the attacker's capabilities leaves the impression that as long as branches do not have observable timing differences, do not leave a different cache trace, and BPU attacks are prevented, controlled channels can be contained. As shown in the snippet of code in Listing 1,

```
static int mpi_montmul( ... ) {
    ...
    if( mbedtls_mpi_cmp_abs( A, N ) >= 0 )
        mpi_sub_hlp( n, N->p, A->p );
    else
        /* prevent timing attacks */
        mpi_sub_hlp( n, A->p, T->p );
    return( 0 );
}
```

Listing 1. Protection against timing attacks in the latest version (v2.16.6 at the time of writing) of MbedTLS. The library balances branches by having symmetric execution paths.

even wildly used crypto libraries tend to use balanced branches[1] to “prevent timing attacks.” This might seem reasonable; after all, the branches in Listing 1 would neither be observable with page attacks, since the same function is called on both paths, nor with Nemesis as both paths have the same instructions. We question this last line of defense, by increasing the resolution of the attacker once more, and demonstrating that virtually any code with control-flow secret dependencies leaks information in SGX.

Frontal attack: We show, for the first time, that when frequently interrupted, the frontend of the CPU, and in particular its fetch and pre-decode module, lead to execution time differences related to an instruction’s virtual address. Based on this observation, we construct a new attack on Intel’s SGX that we call the *Frontal* attack. Our attack allows an attacker to associate the measured instruction execution time with the alignment in the fetch window, and thus with the instructions virtual address. These leaked execution times and addresses can then be used by the attacker to infer control-flow and, therefore, branch-dependent secrets, or when combined with other attacks like Nemesis [56], to leak the execution trace of unknown binaries which are dynamically-loaded into SGX enclaves.

We focus on extracting branch-dependent secrets and show that an adversary can distinguish between two code sequences executed within SGX and hence derive the secret branch condition. Unlike in previous attacks [41], [56], which could only distinguish between sequences of different instructions, the *Frontal* attack allows the adversary to distinguish between two execution sequences even if they contain identical instructions (and even identical data). These differences are observable even when the two snippets of code reside in the same cache line and are thus not susceptible to cache side-channel attacks. We show that by using the *Frontal* attack, the adversary can extract the correct secret from the enclave with overwhelming probability ($> 90\%$). We discuss how different libraries and defenses can be exploited using this attack: the mbedTLS library, the Intel IPP library, and Zigzagger [34]. We validated our attack on all Intel microarchitectures since the introduction of Intel SGX up to the latest one, Coffe Lake Refresh, which includes hardware mitigations against various microarchitectural attacks [6], [30], [36]. We show that the attack works with high probability on all CPUs irrespective of the newest microcode updates. We further discuss which system configurations are better than others for the attacker. For instance, unlike in most other microarchitectural attacks, disabling hyperthreading helps the attacker.

Defenses: Given the resolution achieved with our attack, a more realistic SGX adversary model should be one that considers the instruction pointer to be available to the attacker at any time. Confidentiality in SGX can only be guaranteed in this model if secret dependent branching is avoided altogether, for instance, by if-conversion [9] or by writing code following data-oblivious practices [26]. These defenses are effective against any side-channel attack - including ours. However, practically deploying them is not straightforward for two reasons. First, general compiler transformations incur in high-performance overheads or require developer assistance to mark secrets [9]. Second, custom data oblivious solutions are not trivial to develop correctly and require domain-specific knowledge [26].

These practical challenges for data-oblivious code have led to several spot defenses being continuously refined based on the adversary’s capabilities. We give further proof in this paper that they are bound to be broken whenever previous assumptions about the attacker are challenged.

In summary, we make the following contributions:

- We investigate how frequent interrupts affect instruction execution times. In particular, we show a dependency between the observed execution times and their alignments within the CPU fetch window.
- We introduce the *Frontal* attack. It leverages this execution time-alignment dependency to attack Intel SGX enclaves. The *Frontal* attack leaks fine-grained control flow in branches containing the same instructions, and that only span a single cacheline. It can do so with more than 99% accuracy, depending on the target binary.
- We show several vulnerable libraries that are susceptible to the attack. We further test which CPUs are vulnerable to our attack and found that all most recent CPUs are vulnerable. Newer CPUs that include hardware mitigations against Spectre seem to be more vulnerable than older CPUs.

II. Background

SGX is a novel Trusted Execution Environment technology that introduced processor extensions, which allow for processor-supported application isolation and attestation [10]. Software executed in SGX enclaves is isolated from all other software running on the system, including the operating system (OS) and the hypervisor. Enclave memory is encrypted, and its integrity protected whenever it is moved outside the CPU. Like in classical applications, the OS remains in charge of managing the enclave’s memory through memory paging. The OS is responsible for starting and scheduling enclaves but should not be able to interfere in its execution or compromise the integrity and confidentiality of their data. SGX further supports attestation through which other enclaves or remote parties can verify enclave code and establish secure channels with enclaves. Finally, enclaves can seal data to disk using CPU generated enclave- or developer-specific keys. SGX was therefore designed to operate under the model of a local adversary, who is in full control of the OS and can schedule and interrupt the execution of enclaves.

[1] branches that contain the very same instructions on both execution paths

A. SGX-Step & Nemesis

SGX-Step [55] is an open-source framework that allows single-stepping through the execution of SGX enclaves. SGX-Step uses APIC timers to interrupt the enclave after every instruction and inserts custom routines in between the interrupt handler and the enclave resumption. It does not rely on any adversarial capability not given in the standard Intel SGX attacker model as interrupt handlers and APIC timers are controlled by the OS, which is assumed to be under the control of the adversary.

When an enclave receives an interrupt, it performs an Asynchronous Enclave Exit (`AEX`) and then jumps to the handler defined in the interrupt descriptor table (IDT) to take care of the interrupt. After the interrupt has been handled, it jumps to the address set in the asynchronous enclave pointer (AEP). The function in the AEP eventually executes the `ERESUME` instruction to resume the enclave [10]. SGX-Step installs a custom interrupt handler in user-space to gain control as soon as possible after the interrupt. It also replaces the AEP to execute custom instructions right before `ERESUME`. SGX-Step uses these modified routines to store the current cycle count just before entering the enclave and right after an AEX. To interrupt the enclave at the right time, it configures a cycle-accurate APIC timer. This timer can be configured so that the execution is interrupted after a single instruction is executed inside the enclave. These changes allow an adversary to single-step an enclave and measure timings of individual instructions (including a constant offset by the `ERESUME` and `AEX`).

The Nemesis [56] attack exploits the fact that the interrupt timings obtained through SGX-Step are correlated with the instruction type currently pending in the CPU. Since current processors execute some instructions faster than others, the adversary can make an educated guess about the type of instruction that was executed in a single step. Based on a trace of these timings, and knowledge of the binary executing in the enclave, the attacker can detect where the instruction pointer (IP) was in the enclave when the interrupt was received. Because Nemesis exploits the difference in interrupt timings, it cannot resolve the IP whenever a balanced branch is executed in the enclave, or even a branch with different instructions but with instructions that yield similar timing.

B. CPU Background: The Frontend

Although the x86 instruction set architecture (ISA) is well specified [28], the microarchitecture is typically proprietary, and its details are confidential. Intel and AMD, however, publish general overviews of the microarchitecture of their current processors. Further insights into the microarchitecture can also be obtained through the use of performance counters. Generally, the processor core can be split into three main parts: the frontend, the backend, and the memory subsystem. Here, we will focus on the frontend of the processor. For further information into the other components, we refer to [15].

The frontend of a processor is responsible for fetching and decoding instructions into a format that the backend understands. Modern Intel processors need to fetch a large number of macro-ops to feed the extremely performant out-of-order backend. The core fetches 16 bytes at once, from 16 bytes aligned blocks, also called the *fetch window*. In x86, there is an extra step during decoding where the fetched x86 instructions (macro-ops) get translated to a different internal instruction format called micro-operations (micro-ops). While the x86 ISA is well-specified [28], the internal micro-operations are considered a competitive advantage and are kept secret. The frontend of modern Intel processors contains three different paths for decoding instructions into micro-ops: the legacy decoding pipeline, the micro-op cache (also known as decoded instruction cache), and the micro-code sequencer. The latter is only used for very complex macro-ops that get decoded to more than five micro-ops. Most of the Intel SGX instructions, such as `ECREATE` or `ERESUME`, get decoded into hundreds of micro-ops. Ordinary macro-ops can either get decoded by the legacy decoding pipeline or hit the micro-op cache. The exact number of micro-ops emitted from these three components is visible through specific frontend performance counters.

III. Overview of the Frontal attack

Attacker model: We consider an attacker that wants to leak secret data from a victim SGX enclave running on a system under her control. The victim enclave has a control-flow dependency related to the secret data the attacker wants to leak. The adversary operates under the standard SGX attacker model [10]. That is, she controls the entire software stack, including the operating system (OS), on the machine in which the enclave executes. However, the CPU package is not physically compromised. We assume that the secret that the enclave holds was remotely loaded after a successful attestation. Otherwise, if the secret would be contained in the enclave code, it would be trivially available to the OS.

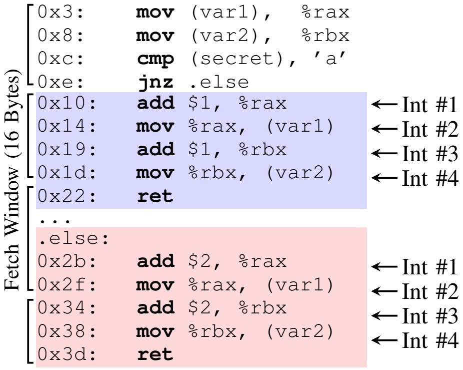
Attack overview: We introduce our attack through an example code snippet that we show in Figure 1a (C code), and Figure 1b (its x86 assembly version). On both branches, the code contains the very same instructions, and writes to the same memory addresses, albeit with different constant values. Thus, we expect its execution time to be independent of which branch is taken, and hence not to have any correlation with the “secret” input.

However, our attack shows that, when the above sequence is run within an SGX enclave, a local attacker can learn which branch was taken, and therefore derive the secret value of the branch condition. Our attack leverages two main observations. First, even if the branches have the same instructions, they are often differently aligned within the fetch windows (Listing 1b) – in our experiments, this alone did not produce observable differences in the execution times (cf. Section IV). Second, if the execution of both branches is frequently interrupted, the difference in their alignments w.r.t. the fetch windows will cause the CPU to fetch instructions at different times (Table I), resulting in a measurable difference in the execution times of the instructions and therefore of branches (cf. Section IV).

To give an intuition on why interrupts lead to a successful attack, we show which instructions are fetched by the CPU when the execution is interrupted after each instruction. There are two main factors to consider: which instructions among those already in the pipeline are retired when an interrupt is received, and how execution is resumed after an interrupt. Intel guarantees that only the first pending instruction in the reorder buffer is completed before the interrupt is handled [55]. In

```
if (secret == 'a') {
    var1 = 1 + var1;
    var2 = 1 + var2;
} else {
    var1 = 2 + var1;
    var2 = 2 + var2;
}
return;
```

(a) Secret dependent branch



```
0x3:    mov (var1),  %rax
0x8:    mov (var2),  %rbx
0xc:    cmp (secret), 'a'
0xe:    jnz .else
0x10:   add $1, %rax
0x14:   mov %rax, (var1)
0x19:   add $1, %rbx
0x1d:   mov %rbx, (var2)
0x22:   ret
...
.else:
0x2b:   add $2, %rax
0x2f:   mov %rax, (var1)
0x34:   add $2, %rbx
0x38:   mov %rbx, (var2)
0x3d:   ret
```

(b) Secret dependent branch in asm

Fig. 1. A secret-dependent branch in C and x86 assembly. Note that both branches in the assembly code fit within the same cacheline (64B). We give the virtual address of the instructions on the left. Fetch windows are 16B long and start at addresses divisible by 16.

TABLE I. IN THIS TABLE WE SHOW HOW INSTRUCTIONS ARE BATCHED INTO FETCH WINDOWS WHEN THE ENCLAVE RESUMES EXECUTION, ACCORDING TO WHICH BRANCH IS EXECUTING. IF AN INSTRUCTION CROSSES A FETCH WINDOW BOUNDARY, WE ASSUME IT IS DECODED TOGETHER WITH THE INSTRUCTIONS IN THE FOLLOWING WINDOW. THE INTERRUPTS REFER TO THE INSTRUCTIONS IN FIGURE 1B.

	If			Else			
Int #1	add	mov	add	add			
Int #2	**mov**	**add**		**mov**	**add**	**mov**	**ret**
Int #3	add			add	mov	ret	
Int #4	mov	ret		mov	ret		

out-of-order processors, other instructions might already have been executed, but none of these will be retired. The CPU architecturally commits[2] only the next pending instruction in program order. To resume execution after the interrupt is handled, the CPU needs to fetch the instruction after the last one retired. As fetch windows are statically aligned at 16 Bytes code blocks [15], the instruction committed during the interrupt will be fetched again but will have to be discarded by the frontend (unless it was the last one of a 16 Byte code block). Alignment w.r.t. fetch windows can, therefore, change which instructions are forwarded to other stages of the CPU, and ultimately populate the pipeline. To help clarify this point, for both branches of our example code, we show in Table I which instructions are fetched after every interrupt.

In principle, differences in the alignment of instructions in a fetch window should not cause any difference in the execution time of one instruction, as the same instructions will be eventually executed on both branches. However, we experimentally observe that depending on the alignment within a fetch window and the number and type of instructions present around them, some instructions consistently take longer to execute than others. In Section IV, we provide more details on which alignments of instructions produce measurable execution time differences. This observation hence allows us to associate the measured instruction execution time with the alignment in the fetch window, and therefore with the instruction virtual address (i.e., with the instruction pointer). These leaked execution times and addresses can then be used to infer executed branches (e.g., when they depend on the secret value) or, when combined with other attacks [56], to leak the execution trace (including fine-grained control flow) of unknown binaries which are dynamically-loaded into the SGX enclave.

In this work, we focus on the use of our attack in the context of secret-dependent branching. In particular, for the scenario given above in Table I when enough mov are fetched after a mov in the branch, the interrupt latency is measurably different. In our example, we measured interrupt #2 in the table to be faster if the code is executing in the "else" branch, as compared to the "if" branch, despite the fact that we are interrupting the *same instruction under the exact same system conditions*.

Let's consider again Listing 1b. By running an SGX-Step attack, we can get the timing of each instruction, stepping through them one by one. This, by itself, will not allow to differentiate which path of the branch was taken, as the branches are perfectly balanced. However, because of the observations made above, we will observe two scenarios for the 6th instruction measured, which is the instruction at address 0x14 or 0x2f, depending on the secret value. If the interrupt is "slow", we must be executing the mov at address 0x14. Conversely, if the interrupt is "fast", we must be executing the mov at address 0x30. Since the control flow of the program depends on the secret, this allows us to recover its value, and hence break the SGX confidentiality guarantees.

The snippet presented in Figure 1 produces distinguishable timing for the first mov instruction inside the branch. We were able to use the timing difference to predict the secret with $\geq 65\%$ accuracy. With three more movs after the branches (which are executed by both paths), we were able to obtain success rates $> 90\%$. The attack presented above illustrates how fully balanced branches actually produce secret dependent timings when interrupted frequently. Given that this side-channel is due to the design and behavior of the CPU Frontend, we name our attack the *Frontal* attack. In the following Sections, we will analyze the *Frontal* attack in more detail.

IV. FRONTAL ATTACK PROFILING

In this section, we provide more details and clarifications that help in understanding under which circumstances the *Frontal* Attack works. More specifically, we ask and answer the following questions: (i) are the interrupts required for the attack to be successful? (ii) what are the effects of fetch window alignment / instruction address on the attack? and (iii) which instructions produce observable timing differences?

To answer these question we perform experiments over the code snippet shown in Figure 2. Similar to code in Figure 1, this code snippet contains two perfectly symmetric branches, and the branch to be taken depends on a secret. It still consists of two perfectly balanced branches but differs in that now each branch contains 25 sequences of add-mov instructions. We chose this longer code sequence since it produces timing differences which are more clearly above the noise floor than

[2] Or discards, if it raises an exception

```
          .align (x - 0x4)
x - 0x4:     cmp (secret), 1
x - 0x2:     jnz .else
          .if:
          .rept 25
x + 0x0:     add %rax, %rbx
x + 0x3:     mov %rcx, (var1)
          .endr
x + 0x190:   ret
...
          .align y
          .else:
          .rept 25
y + 0x0:     add %rax, %rbx
y + 0x3:     mov %rcx, (var1)
          .endr
y + 0x190:   ret
```

Fig. 2. ASM Code with high attack success probability, which we use to profile the attack. The `.rept 25 ... .endr` assembler directive repeats the instructions within the block 25 times, leading to an address of `x+0x190` for the `ret` instruction.

the code in Figure 1 and therefore better illustrates timing and alignment effects under different experiment configurations. Namely, code sequences that include several `mov` instructions like the one in Figure 2 are particularly susceptible to the *Frontal* attack and allow us to extract the correct branch (i.e., branch condition secret) with over 99% accuracy, whereas with shorter sequences which contain few `mov`s (like the one in Figure 1) this accuracy drops to $\geq 65\%$. We discuss this effect in more detail later in this section.

A. Are the interrupts required for the attack to be successful?

To analyze the effect of frequent interrupts on the behavior of the processor we measure the execution time of our test code snippet (Figure 2) with and without interrupts. Both branches of this code are identical, however, like in our previous example, the fetch windows for these branches differ.

1) Outside SGX without interrupts: We first measured the overall execution time of the code snippet outside SGX without interrupts. We executed the code with two billion independent random inputs, and we observed no significant correlation between execution times and the branch that was executed (Pearson's coefficient $= -2.51 \cdot 10^{-5}$). An approximate distribution of this measurement is shown in Figure 3.

2) In SGX without interrupts: In order to exclude any effect due to SGX, we further measure the overall execution time of the code within an SGX enclave, again without interrupts. Note that SGX does not provide any way to get a precise timer (cf. Section II), so we have to measure the execution time from the untrusted app.

We perform this measurement in three different ways. First, we measure the time needed to do a function call of an enclave containing the code under measurement. This time is collected by the untrusted app by getting the clock count right before the enclave call and comparing it to the value obtained as soon as the enclave returns. We repeat this measurement in a loop from the untrusted app to account for noise in the measurement. Second, we ran a loop inside the enclave, which calls the code under measurement in each iteration. Before and after the function call the enclave performs an `ocall` to the untrusted app. This lets the untrusted app know when an iteration started and terminated, and gives it the opportunity to collect the CPU time stamp counter. Third, we use a similar setup as the second method, but instead of performing `ocalls`, the enclave samples the value of a counter stored in shared memory. A thread of the untrusted app is incrementing the counter in a loop, thus simulating the time stamp counter, albeit at a lower precision. All three methods use a independent uniform random value as the "secret" given to the code at each iteration.

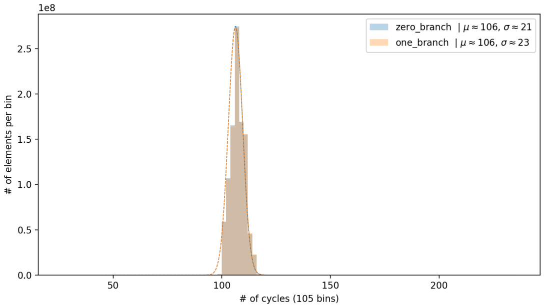


Fig. 3. Distribution of the overall execution time of the branch in Figure 2 when run outside of SGX without interrupts with 2 billion samples.

In all three cases, similar to the experiment outside of SGX, we observed no significant correlation (Pearson's coefficient $\approx 10^{-2}$ with 10^{6} runs) between the execution time and the secret provided to the enclave (e.g., the branch that was taken). This experiment confirmed that when the code is executed within SGX, both branches execute in the same, constant time.

3) In SGX with interrupts: We now investigate which effects frequent interrupts have on the execution timing of the code. We execute the same code but we interrupt it after each instruction. Upon each interrupt the CPU performs an asynchronous enclave exit (AEX), handles the interrupt, and then performs an `ERESUME` to resume the enclave execution. Such an experiment would normally require very fast and extremely precise interrupts, which is usually hard to achieve. However, in the case of a victim code running within SGX, we can use SGX-Step [55] to single step through each instruction and collect its execution time. Given these interrupts, we can not only measure the overall execution time, but also the execution time of each instruction. This means that in each run of our code, we obtain 51 measurements[3].

We then analyzed whether any of the 51 measured instruction execution times correlates with the executed branch. We observed a **strong correlation** between the timings of most of the instructions and the branch they belong to. The first 10 `mov` instructions in the branch turned out to be a stronger indicator of which branch was taken, but all the other instructions belonging to the branch showed some correlation, albeit a weaker one[4].

Like in Section III, we observed the execution time of the first `mov` in each branch to be faster or slower, depending on the branch they belong to, with the difference between them being around 100 cycles. As we will show in the following subsections, this effect will also be visible on other `mov` instructions as well as on other instructions in the code, and

[3] There are 52 instruction in Figure 2, however the first `cmp` and `jnz` get macro-fused into one instruction which cannot be split again by interrupts.

[4] Only the execution times of instructions in the branches correlated with the branches. The timings of the initial `cmp` and `jnz` instructions were independent of the executed branch.

will depend on the instruction alignment within the fetch window (i.e., instruction address).

This observation allowed us to set a timing threshold based on which we could, with up to 99.9% accuracy, determine which branch was taken, and therefore determine the secret value in the branch condition.

We stress again that the two branches are instruction-wise identical: the instructions they contain *and their inputs* are the same. This is especially important because it highlights the fact that the timing difference is due to the way the instructions are executed, and not some external system state. For instance, the difference *cannot* be due to the state of the cache, the state of the branch predictor, or in general to some speculation decisions made by the CPU. If the cause of the differences were to be due to any of these factors, we would expect two key differences. First, as we choose secrets at random, these effects would manifest with equal probability in any of the two branches. Second, we would expect the experiments in which we do not interrupt the code also to show some bias. However, we see a clear bias in one of the two branches, and the interrupt-free runs showed no correlation with the branch.

Observation 1: When code execution is frequently interrupted, the execution times of selected instructions depend on their location in the victim binary and therefore on their virtual memory address.

B. What are the effects of fetch window alignment / instruction address on the attack?

While the instructions in both branches are identical, there is one key difference between them: their virtual address. Therefore, we analyze what virtual addresses make the two branches distinguishable when frequently interrupted, and to what degree. In particular, as discussed in Section III, we also study the relationship between the alignment of the branches with respect to the fetch window and what role, if any, this plays into the success of the attack. As can be seen in Figure 2, we use the `align` compiler directive to explicitly align each branch to a given address. With `.align X` we indicate that the code following the directive starts at the next virtual address whose lower bits are equal to `X`[5]. For example, if $X = 3$ and $Y = 2$, then the `if` branch will start at address `0x13` and end at address `0x1a3`, while the `else` branch will start at address `0x1b2`.

To evaluate different alignments, we run an experiment to test if different values of `X` and `Y` in Figure 2 have any effect on the observed timing differences. We repeat the interrupt experiment described at the end of Section IV-A. That is, we send an interrupt to each instruction and then use the interrupt timing of one of the instructions in the branch as a discriminator to determine which branch was taken. We calculate the attack success as the percentage of how many times we identified the correct branch. Therefore, the attack success rate will tell us how good a certain combination of the alignments `X` and `Y` are for the attack. The higher the percentage the better a combination is for the attack, while a result close to 50% indicates that telling which branch was taken is as good as a random guess. We collect these percentages for each combination of $\{X, Y\} \in [0, 31]^2$ by running the code in Figure 2 1000 times with independently distributed uniform random secrets. We compare use timings collected for the 10th instruction (5th mov) in the timing trace. Figure 4 presents the result of our experiment. These results show clear dependency between virtual addresses and the instruction execution times.

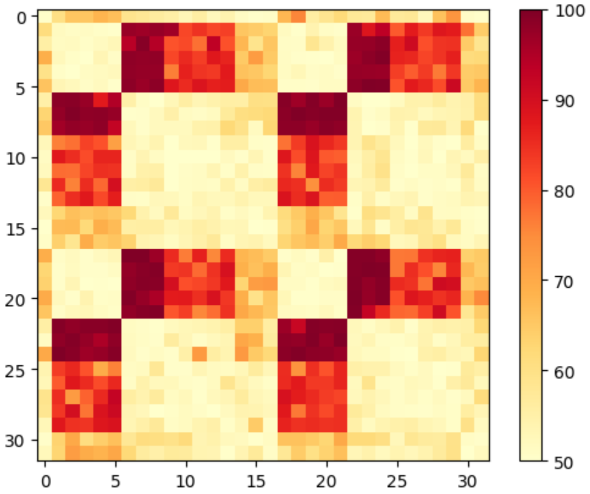


Fig. 4. Attack success rate depending on the alignment of the branches. The attack success rate is the percentage of correctly guessed branches by the attacker out of 1000 executed branches. The 10th instruction (5th `mov`) from Figure 2 is used to distinguish between both branches. The color gradient goes from red to yellow, where red boxes indicate higher attack success rates (up to 100%) and yellow ones lower success rates (down to 50%).

1) Alignment: Given this, we can further draw a dependency between the virtual address of the two branches and the attack success, as some virtual address pairs consistently result in higher attack success rate than others.

2) Modulo 16: There are four main quadrants of length 16 that are essentially identical. This hints at the fact that the behavior with respect to the alignment of the two branches repeats every 16 bytes. We verified this assumption by repeating the experiment for every value of `X` and `Y` for which `X = Y mod 16`, and for which the two branches are still contained in the same 4 kB virtual page[6]. We observed the same pattern for all values. As a consequence of this observation, when we use the term alignment, we refer to alignment modulo 16.

Observation 2.1: The attack success rate depends on the alignment *modulo 16* of the two branches.

3) Diagonals: The attack success rate on the diagonals in each quadrant is around 50%. In the diagonals, both branches are aligned to the same value `X = Y mod 16`, which shows that only differently aligned branches (modulo 16) exhibit a timing difference.

Observation 2.2: Branches and instructions with the same alignment will show the same execution times.

[5] This is equivalent to combining the two gcc asm directives `.align` $(X//2^n)$ and `.space` $(X\%2^n)$ (for the biggest n such that $2^n < X$)

[6] We did not cross the virtual page boundary because this would most likely require fetching pages that are not cached, thus introducing noise that masks the effects we are interested in measuring here.

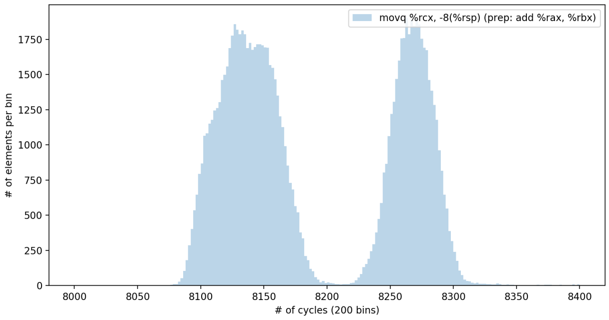


Fig. 5. Timing distribution of 100,000 `movs`.

4) Symmetry: The fourth observation emerges when looking at each of these quadrants: The attack success rates are symmetric with respect to their diagonal. Meaning that the success of the attack when the "if" branch is aligned at address `X` and the "else" branch at address `Y` is the same when the alignment of the branches is switched.

Observation 2.3: Alignments X, Y and Y, X produce the same attack success rate.

5) Shape: Finally, we focus our attention to the alignments in the heatmap in which the success rate is above 70%. As it can be observed, these success rates are grouped into rectangles. Within each of these rectangles, there are three regions of decreasing intensity. The most interesting alignments are the ones that give the higher attack success rates, as they allow to optimize the accuracy of the attack. The best results are concentrated on rectangles of size 3×5. This corresponds with the length in bytes of the two instructions within the branch in Figure 2. The `add` instruction has a length of $3B$, while the `mov` we use in Figure 2 has a length of $5B$. We will explore further the relation between the length of the instructions in the branch and the distributions of the attack success rate within the alignment in the subsection below.

Note that there are only a few structures in the CPU that are sensitive to the alignment of the instruction, and in particular, to their alignment modulo 16. On Skylake and Coffee Lake architectures, this is the *instruction pre-decode and fetch* module in the front-end of the CPU, which uses a fetch window of 16 bytes to fetch instruction from the L1 instruction cache. We cannot be entirely sure about the internal behavior of the CPU and what leads to the timing differences in the two branches. However, as discussed in Section III, the different alignment changes the way instructions are batched by the front-end and, ultimately, the timing at which they are delivered to the subsequent stages of the CPUs. The experiments presented in this section strongly suggest that these fetching differences have repercussions for the instruction's execution timing. We will discuss potential causes within the CPU that could be causing the same instruction to execute at variable timings in Section VII-A.

C. The effects of instruction alignment

To study the effects of the instruction alignment we analyze the timing distributions of a linear code sequence of repeating `add-mov`, which we fix to 100,000. Note that this is essentially an unrolled loop, which compared to a loop removes the noise that the looping instructions would introduce. We don't envision any real code to have this many repeating instructions, but by exploring the patterns that emerge from these instructions we can gather several insights about how the differences in branch alignments manifest.

We recall that the timings are collected using a slightly modified version of SGX-Step, whose changes are described in Appendix C. As in SGX-Step, the timings for each instruction include: the time to perform `ERESUME`, the time to execute the instruction, and the time required to perform `AEX`. `ERESUME`, and `AEX` prepare the CPU for the enclave execution and clean the state when returning to the untrusted app. These operations take thousands of CPU cycles to complete, and this is why, despite the fact that we are measuring a single instruction, the latency reported in the graphs are in the order of thousands of cycles. We use two figures to illustrate different aspects of the timing latency of the same run: (i) Figure 5 depicts the overall latency distribution of all the movs, and (ii) Figure 6 the distribution separated by particular virtual addresses. The analysis of these plots further allows us to discuss a third point: (iii) how these distributions are influenced by the surrounding instructions.

1) Distribution of instruction execution times: In the first graph, in Figure 5 we present the histogram of the instruction execution times, for one run, of all 100,000 executed `mov`.

The most evident feature of this distribution is that it consists of *two* Gaussian distributions. The `movs` are therefore exhibiting two different behaviors, whose distributions are, on average, around 100 cycles apart.

Observation 3.1: There are two very distinguishable behaviors of *movs*: fast and slow. The slow *movs* are, on average, around 100 cycles slower than the fast ones.

In general, we observed similar results with other instructions that access memory, such as `add` to memory. We remark here that these differences are not due to the state of the L1 data-cache. We ensure this by running the victim enclave on a dedicated physical core in the system and by always performing the same operations while handling interrupts. We further verified with the `OFFCORE_REQUESTS_ALL_REQUESTS` performance counter that no extra off-core memory transactions were being performed by the slower memory writes compared to the faster ones.

Observation 3.2: Observation 3.1 applies not only to `movs` but to all *memory writes*.

2) Instruction execution times by alignment: Regarding alignment, there is an important characteristic of the chosen instruction sequence that has not been considered in our analysis thus far. Each couple of `add-mov` in the sequence has a length of 8B, which is a multiple of 16. This implies that the `movs` *can only be aligned in two different ways modulo 16*. In general, by testing the sequence with different initial offsets, we observed `movs` at addresses between `1 and 8` to be predominately slow and `movs` at addresses `9 to 16` to be predominately fast. We highlight that the two alignments are only *predominately* fast or slow, and they exhibit timings from both behaviors, with each alignment displaying a different

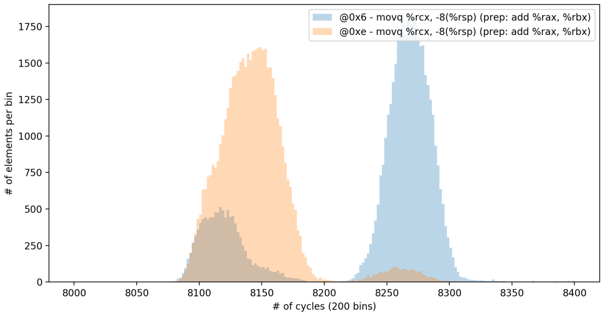


Fig. 6. Timing distribution of 100,000 `movs` split by their alignment.

intensity of the weaker one. Figure 6 shows this phenomenon for two particular alignments (`0x6` and `0xe`). As can be seen alignment `0x6` is predominately slow, but exhibits some fast timings as well. The plots for other alignments are similar, with the only thing changing being the size of the smaller peaks. We do not show them here due to space constraints.

> **Observation 3.3:** The alignment of the memory writes determines how their latency will distribute between the fast and slow behaviors.

If one branch is aligned such that the measured `mov` produces predominantly fast latency, and the other is aligned to produce predominantly slow latency then the branches are easily distinguishable, and a high success rate will be observed. If one of them has a non negligible weaker peak, like the small blue peak in figure 6, then one bit can be distinguished with high accuracy, but the other will contain some errors. If both branches have non negligible weaker peaks, both branches will be on average guessed better than random, but will also contain errors. And finally if both branches have similar predominant and weaker behaviors the success rate of the attacker will be negligible.

D. Which instructions produce observable timing differences?

Given the timing differences observed thus far, the question that we ask in this subsection is which instructions are more useful than others at revealing timing differences. Knowing which instructions produce a clearer signal allows the attacker to identify which parts of a vulnerable code are better suited as targets for the attack.

To address this point, we briefly describe what influences the shape of the weaker behaviors displayed in Figure 6. We experimentally observed that the alignment of surrounding memory writes is the key factor in determining the shape of the weak peaks. Among them, subsequent memory writes have a bigger effect than proceeding ones when determining the distribution of a target instruction. We analyze three cases in this regard: (i) all `movs` with the same alignment (modulo 16) across a measured trace, (ii) `movs` alternating between two alignments, and (iii) a trace with `movs` covering every possible alignment. Case (i) exhibits only a fast distribution, irrespective of the initial offset. Case (ii) is what we analyzed above, and depending on the alignment, we know it exhibits either a stronger fast behavior, and a weaker slow one, or vice-versa. Finally, case (iii) produces almost all combinations of behaviors depending on the alignment under measurement.

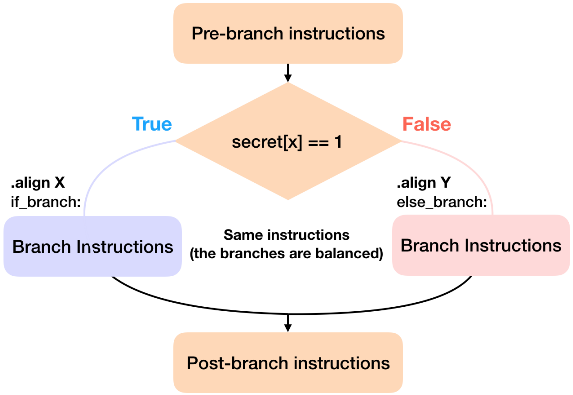


Fig. 7. Structure of a victim branch for the frontal attack.

Some alignment behaves like case (ii), others are only fast, others only slow, and finally some are equally randomly distributed between the two behaviors.

> **Observation 3.4:** The timing distribution of a memory write is not only determined by its alignment in isolation, but its intensity across the fast and slow behavior is also influenced by the number and alignment of surrounding memory instructions.

V. Frontal Attack Exploitation

A. Finding gadgets

The initial phase of the attack consists of identifying which branches of a victim enclave the attacker can target. An exploitable branch needs to be secret-dependent and properly aligned. Branches that are secret dependent can be found by static analysis, through manual inspection of the binary, and if possible dynamically, by running the victim enclave in debug mode with different inputs and comparing the resulting execution traces. Several tools that automatically perform these steps, have already been proposed, e.g., DATA [58] and Microwalk [59].Note that if these branches are not balanced, i.e., they contain a different number or different kind of instructions they are already vulnerable to a Nemesis attack [56]. However, if they are balanced, that is they contain the same number and types of instruction, they would not be vulnerable to Nemesis, but are vulnerable to the *Frontal* attack.

Identifying gadgets requires access to the victim binary. This requirement is not met if the target enclave loads its code dynamically at run-time. As hinted in Section III, the *Frontal* attack could be used in combination with Nemesis [56] to de-obfuscate a dynamically loaded binary. However, we do not explore this scenario in this paper, and simply assume that the attacker has access to the binary.

1) Optimizing Gadgets: Figure 7 summarizes different parts of a secret dependent branch that are relevant for the success of the frontal attack. There are four main elements in the figure that contribute to the attack success rate, in order of importance these are: the alignment of the two branches, the instructions in the branches, the post-branch instructions, and the pre-branch instructions. We use the observations made in Section IV to evaluate these factors.

First, we know that only specific alignments produce observable differences. Memory writes aligned at addresses

`1-8` show different predominant behaviors than writes aligned at addresses `9-16` modulo 16 (cf. Section IV-C2). Therefore, better results can be expected if the victim branches happen to have memory writes aligned at those addresses. Second, memory writes inside of the branch produce better timing differences (Observations 3.1 and 3.2) therefore, the attacker should prefer branches that contain at least one of them. The more memory writes are in the branch the easier it will be for the attacker to exploit it, as this increases the likelihood that one of them will be at an address (modulo 16) with a strong timing bias. However, the presence of memory writes in the branch instructions is not a hard requirement, as we were able to observe some biases even on branches that did not contain any memory writes. It is sufficient that memory operations are used as pre and post-branch instructions. Finally, branches that have pre or post-branch memory operations have more distinguishable timing differences (observation 3.4). This is particularly relevant when only few memory writes (e.g., only one) are present in the branch instructions.

We found that the best approach is first to use taint analysis to find all the vulnerable branches, and then use the runtime measurements of these branches to detect whether they are vulnerable or not, as described in the following.

2) Branch execution: Secrets are usually provided to enclaves from a remote verifier at run-time, after he has attested and established a secure channel with a legitimate enclave. We consider two particular cases for the attacker: the first, in which the remote verifier will provision the secret an unlimited amount of times, and the second, and most restrictive one, in which the remote verifier will provision the secret only once and the enclave only executes the branch once. We start describing the first (and simpler) case, and then describe what techniques the attacker can use to maximize her success rate when the target branch is executed only once.

If the same secret is provisioned to the enclave multiple times, she can correlate the timings of multiple runs of the same branch to reduce the noise, and increase her confidence in the result. However, naively combining all the timing measurements, won't help. As we describe in Appendix C, the timings of the same instructions for different runs have different means, and hence are shifted by a variable amount each time an enclave is re-initialized. Without accounting for these variations, correlating timing across different enclaves runs is thus problematic. However, the results can still be combined by guessing which branch was taken independently for each run, and then using majority voting for the final guess. This increases the attack success rate and allows to make an accurate guess even for target branches that give an otherwise low advantage to the attacker.

Different means for the measurements across enclave creations make it hard to attack an enclave when the secret is only one bit, and hence the branch gets executed only once. In this case, the attacker can only observe the timing of one of the two branches, and has no comparison point from the other branch. Note that because of the differences in timings across runs, the adversary can only classify branches by observing which instructions were faster than others *in the same run*. To compensate for this effect, we can use Observation 3.3, and, before the branch is executed (or after), collect the timings of instructions that have the same alignments as the instructions in the branches. This gives the attacker comparison points, and allows her to distinguish even within branches that are only executed once.

Finally, we discuss the extreme case in which the secret is provisioned only once. As compared to the previous case, the target branch will be executed only in one run, and hence the attacker can't correlate the results from different measurements. However, she can still use the technique discussed for the one bit secret to get more measurements about the target branches, albeit from instructions that do not belong to them.

B. Measurement and Analysis

There are three main factors to consider in the measurement phase of the attack. First, how many times the secret is going to be provisioned and how many times the target branch is going to be run. Second, how to deduce from the timing distributions which branch was taken. Finally, the environment in which the target enclave is running, this includes the specific system setup.

After collecting the measurements the attacker needs a strategy to assign those measurements to the branches to which they belong to. This strategy depends on what kind of bias the branch under measurement was showing. Exploitable branches show two behaviors (cf. Observation 3.1). Their exact distributions across these two behaviors determines the best strategy for the attacker. Note that, as described before, the attacker can obtain these distributions by running and profiling specific victim branches beforehand. This might require profiling on a copy of enclave binary which contains only the target branches if, for instance, the real enclave only receives the secret once.

After the target branch has been profiled, the timing analysis consists in assigning the run-time measurements to each of the two behavior classes. Given several measurements for branches (in the same run) a linear separator can usually be defined between the two behaviors, alternatively clustering algorithms can also be employed at this stage. Depending on the bias of the branches, the attacker can then make different conclusions from these classifications. Let's take as an example the results of Figure 6. Every measurement classified as slow belongs comes from the blue distribution with high probability. However, there is more uncertainty about the measurements classified as fast. In this case the attacker might choose to classify only the bits that he has a high confidence of and brute force the remaining ones. In most cases, looking at only one instruction in the branch is sufficient. However, if available, correlating measurements of multiple instructions in the branches can increase the confidence of the attacker for a particular guess.

1) System setup: The success rate of the frontal attack can be influenced by the system in which it is run. In particular, whether simultaneous multi-threading (SMT) is enabled, the system's CPU model, and its microcode version. While SMT was enabled and the virtual core co-located with the victim enclave's core was executing another binary, we were unable to reproduce the timings observed in Section IV. In general, the frontal attack is more reliable if SMT is disabled or the virtual core co-located with the victim is not executing anything. This is most likely due to the way in which the front-end handles and fetches instructions coming from different virtual cores.

```
num mont_mult(num A, num B, num R, num N) {
    ...
    if (result >= N)
        result = result - N;
    return result;
}
```

Listing 2. Secret dependent branch at the end of the Montgomery Modular Multiplication.

```
num mont_ladder(num x, num n, int k) {
    r0 = x; r1 = x*x;
    for (int i = k-1; i >= 0; i--) {
        if (n[i] == 0)
            r1 = r0 * r1; r0 = r0 * r0;
        else
            r0 = r0 * r1; r1 = r1 * r1;
    }
    return r0;
}
```

Listing 3. Montgomery Ladder. It contains a secret dependent balanced branch.

With the newest microcode, or CPUs without hardware mitigations for Spectre produce noisier results than CPUs with hardware mitigations for it. We give more details about the affected CPUs and microcode versions in Section VI. However, we noticed that the noise in the old CPUs completely disappears in some runs of the enclave. Since the attacker can profile key instructions of the victim enclave before the secret is provisioned, she can assert whether the current run is noisy or not. The attacker can collect the timings from a `mov` at different alignments and verify that the two behaviors are present and are clearly separated (e.g., 100 cycles apart on average cf. Observation 3.1). The adversary can then restart the enclave until a noise-free run is detected, and only then allow the secret to be provisioned. With this technique the adversary can also exploit older CPUs with the newest microcode updates.

C. Vulnerable Code

1) Montgomery Modular Multiplication: Montgomery modular multiplication is a fast modular multiplication algorithm and it is often used in cryptographic libraries due to its speed and minimal secret dependence. There is only a single secret dependent branch in the algorithm at the end, depicted in Listing 2. Some implementations just balance the branches by adding an else branch with a dummy subtraction. However, this naive mitigation is still vulnerable to side-channel attacks that target control flow, such as the *Frontal* attack. We found such an implementation in the mbedTLS v2.16.6 library [35] where it is used in the sliding window exponentiation. This information can be used to extract some bits of the secret key [34].

2) Montgomery Ladder: The Montgomery Ladder is a well-known algorithm to calculate the exponentiation x^n and it is widely used for elliptic curve cryptography due to its speed and side-channel resistance. Listing 3 shows a naive implementation that tries to mitigate side-channel attacks by branch balancing. The depicted implementation is vulnerable to control flow side-channel attacks such as the *Frontal* attack (and potentially also to data access side-channel attacks).

3) Zigzagger: Zigzagger [34] is a software mitigation technique against side-channel attacks that leak the instruction pointer by targeting the branch predictor state. The mitigation replaces all branches with multiple indirect jumps. These jumps modify the state of the branch predictor such that the adversary will only leak garbled data. The *Frontal* attack does not rely on the state of the branch predictor, and therefore, can still attack binaries that were protected with it. Nemesis [56] already showed that Zigzagger can be broken when the branches it protects are unbalanced, the frontal attack extends this to balanced branches.

TABLE II. LIST OF ALL THE PROCESSORS WE TESTED WITH THEIR RESPECTIVE MICROCODE VERSION AND MITIGATION AGAINST KNOWN MICROARCHITECURAL ATTACKS SUCH AS SPECTRE AND FORESHADOW.

Processor	μarch	Launched	μcode	Mitig.	Vulnerable
i7-6700HQ	Skylake	Q3'15	0xc2	μcode	yes[†]
i7-6700HQ	Skylake	Q3'15	0xd6	μcode	yes[†]
i7-7700	Kaby Lake	Q1'17	0x48	-	yes
i7-7700	Kaby Lake	Q1'17	0x8e	μcode	yes[†]
i7-9700K	C. Lake R	Q4'18	0xb8	HW	yes
i7-9700K	C. Lake R	Q4'18	0xca	HW	yes
i9-9900KS	C. Lake R	Q4'19	0xb8	HW	yes
i9-9900KS	C. Lake R	Q4'19	0xca	HW	yes
Xeon E-2278G	C. Lake R	Q2'19	0xb8	HW	yes
Xeon E-2278G	C. Lake R	Q2'19	0xca	HW	yes

[†]: Only vulnerable in some runs (see Figure 8)

4) Intel IPP Library: Several secret dependent branches were already known to be present in the Intel IPP library and can be targeted by the Frontal attack. For instance, in [59], six critical functions with secret dependent branches were identified.These functions are related to cryptographic computation and can therefore leak some secrets bits from enclaves using the library. Through manual inspection of the most recent version of the library (2.9 at the time of writing) we also found the following functions to have secret-dependent branches: `l9_cpNLZ_BNU`, `l9_cpDiv_BNU32`, `l9_ippsCmp_BN` (and basically every other big number equality comparator in the library). The branches in `l9_cpNLZ_BNU` can be used to leak part of the 64 most significant bits of two numbers being multiplied by the library. The branches and loops in `l9_cpDiv_BNU32` leak some bits of two numbers being divided with each other. Finally, `l9_ippsCmp_BN` is a function which compares two big numbers by iterating through each of their bytes one by one. At the end of the loop it executes a balanced branch to determine which number was bigger, and then it writes back the result to a location in memory. This function is particularly interesting because the binary suggests that the main loop and branches were purposely aligned to fit within a cache line, but the function is still vulnerable to the frontal attack. We believe, because of the that the branches in their binary are aligned, that they could be used to leak some bits of the secret data they operate on. In their response (cf. Appendix A) Intel specified that these functions are not used for secret dependent computation in their architectural enclaves.

VI. AFFECTED PROCESSORS

During our experiments, we noticed that microcode versions affect the timings that are exploited in the Frontal attack. We observed each microcode update to increase the number of cycles for an AEX and an EERESUME. We speculate that these are side effects of the added mitigations against microarchitectural attacks such as spectre [30] and foreshadow [6]. While we would like to test each public microcode available for each CPU, the oldest microcode version available to us was limited by two factors. First, the

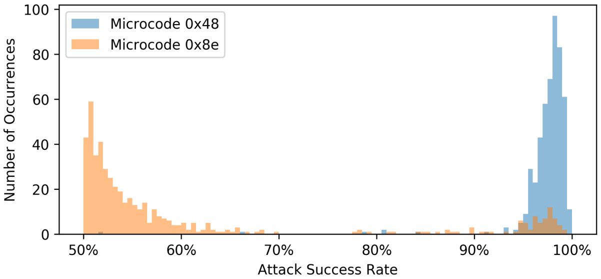


Fig. 8. The attack success rate over 500 runs each with 1000 samples for different microcode versions of the Intel Core i7-7700 processor (alignment: $X = 6, Y = 2$).

microcode version already present in the CPU cannot be downgraded. Second, the BIOS of the mainboard provides the minimum microcode version to the CPU during early boot.

We tested five different processors from the 6th generation that introduced Intel SGX up to the 9th with hardware mitigations for recent microarchitectural attacks. For each processor, we tested the minimum microcode version supplied by the mainboard and the most up to date version as of February 2020. The *Frontal* attack was successful on all tested processors. However, on older generations the microcode version significantly changes the behavior leading to (worse) success rates. A summary of the tested processors and microcode versions can be found in Table II.

Our measurements indicate that the processors can be separated into two groups with similar behavior: processors with and without hardware mitigations against various microarchitectural attacks [32]. Interestingly, newer processors with hardware mitigations built-in were more susceptible to our attack, whereas older processors with mitigations in microcode seem to add noise and thus have lower success rates on average. More in-depth analysis revealed that the most recent microcodes on processors without hardware mitigations add some randomness to our experiments. For these configurations, every run of the experiment exhibits a different behavior. Figure 8 shows the success rate for 500 separate runs each with 1000 samples. Note that most of the runs with the new microcode show a random success rate. However, some runs exhibit a clear timing difference leading to a $> 95\%$ success rate. The adversary can detect which behavior a particular run is going to exhibit by observing the timings of early `movs` aligned at particular addresses. Thus she could decide whether to attack or not before the secret is retrieved or provisioned, and relaunch the enclave until its behavior is clearly distinguishable.

VII. Discussion

A. Potential Causes

The complexity of the microarchitecture of current Intel processors makes it impossible to pin-point the cause of the timing differences to a specific component. However, we will discuss some components which we were able to decisively exclude. We start with the memory subsystem, then we investigate the execution engines, and finally we will focus on the frontend. For each potential culprit in these building blocks, we will describe an initial theory and then try to refute or confirm it using performance counters (and other measurements). Note that the performance counters are sparsely distributed over the entire core and do not exhaustively cover the entire microarchitecture. Therefore, investigation into some hypotheses is very challenging if no performance counters exist for the respective part of the processor.

1) Memory Subsystem: Observation O3.1 and O3.2 point to potential causes in the memory subsystem. Specifically the fact that the slow `mov` is around 100 cycles slower. For a current-generation processor, 100 cycles is a rather large delay that is usually only observed for accesses to external memory or the last level cache. However, performance counters refute any theory related to the memory subsystem since all performance counters related to external memory or last level cache did not show a difference between the slow and the fast `mov`.

2) Execution Engines: The execution engine gets a list of instructions from the allocation queue as input and tries to reorder and execute them as fast as possible. As far as we know, it is completely decoupled from the frontend and does not depend on any alignment since it works on decoded micro-ops. However, given Observation O2.1, we know that the alignment influences the timing difference. We thus rule out the execution engine as the root cause of the timing differences.

3) Frontend: Observation O2.1 strongly hints at the frontend as the culprit, since the fetch window is the only structure which operates at a 16 Bytes granularity, matching the 16 Bytes periodicity of the observations.

a) Micro-op Cache: The micro-op cache is a microarchitectural structure in the frontend [53] which cache holds previously decoded fetch windows and serves them to, for example, repeated jumps to the same address. On a micro-op cache hit, many cycles can be saved due to not having to decode the instructions again. Our observed timing difference might stem from hits and misses in this cache. For some interrupts, the micro-op cache might miss, and the instructions must be decoded again. For some others, it hits and immediately proceeds to the reorder buffer. However, the timing difference we observed seems excessively large for this kind of small difference in the execution path. Besides, performance counters that measure the behavior of the micro-op cache show an equivalent number of hits in the slow and the fast case. Thus, we rule out the micro-op cache as a cause.

b) Branch Prediction: Branch prediction is responsible for predicting the future control flow. The core will fetch ahead and speculatively continue to execute in the predicted path. Not only branches rely on the branch predictor, but also jumps where the target is not immediately known are predicted (e.g., the target comes from memory). The resumption of the enclave could potentially suffer from a misprediction for the instruction that must be continued and thus suffer from a delay. However, all performance counters that we measured did not show any additional mispredicts for slow or fast instructions.

In summary, while we were able to decisively refute many of the most common reasons for timing differences, none of our tests were able to identify with reasonable confidence an explanation for the timings exploited by the attack.

B. Defenses

There exist various defenses against the *Frontal* attack some of which we will discuss in this section. First and foremost, we want to stress that constant time code [9], [47] is a principled approach that thwarts every known side or controlled-channel attack. We discuss these techniques in Appendix B. As such it also remains secure against the *Frontal* attack. Nevertheless, constant time code presents a number of challenges in practice as it is hard to get right and results in a high overhead in certain applications. Therefore, in practice, many spot defenses against the known attacks have been used since they are usually easier to apply and more performant. However, most of these spot defenses are circumvented by new attacks such as the *Frontal* attack. While the behavior exploited by the *Frontal* attack stems from the underlying hardware, the defenses we discuss are on the software level. Hardware mitigations would also be possible, but due to the lengthy turn-around time for new processors that include the fix, software defenses are a lot more attractive.

Fix the Alignment: As seen in Section IV, the timing of individual instructions depends on their alignment. The *Frontal* attack only works with a rather specific alignment of the two branches. Notably, branches with identical alignment do not exhibit any observable timing difference. Therefore, aligning the two branches to the same address (modulo 16) leads to equivalent timings for both branches. However, this approach will lead to increased binary size due to the inserted unused space to align the branches.

VIII. Related Work

A. Controlled-Channel Attacks

The attacker's control over the OS enables novel noise-free deterministic side-channels [7], [57], [60] known as controlled-channels since the attacker (i.e., the OS) controls the channel. Memory paging, the scheduler, the handling of interrupts and exceptions, are a few examples of what the attacker can take advantage of – every interface between the OS and the enclaves can be leveraged in controlled channel attacks. The first SGX controlled channel attack [60] abused the permissions of the pages. They let the enclave generate a page fault for each page it tries to access by removing access or execute permissions to its own memory. The trace of page faults contains enough information to, e.g., let attackers reconstruct images processed in the enclave. Subsequent attacks made controlled channel attacks stealthier, by observing that the CPU sets the accessed and dirty bits [7], [57] in the page tables (PTs), thus allowing to monitor the enclave's execution without having to trigger page faults. However, the resolution of page-based controlled channel attacks is quite coarse, allowing the attacker to know only whether any access in a page (4 kB) was made, but not where within it.

The coarseness of PT based controlled channel attacks is an element that defenses have latched on to protect enclaves [49], [51]. These defenses either call for sensitive code to be within a page [51] or randomize the page enclave's page layout so that page accesses cannot be correlated [49]. Even Intel specifies that controlled channels can be mitigated "by aligning specific code and data blocks to exist entirely within a single page" [27]. However, the resolution of controlled channel attacks was increased through an attack exploiting legacy memory segmentation [22], which is also managed by the OS. While the attack only works under uncommon circumstances (32 bit enclaves and smaller than 1 MiB), it can observe memory accesses at 1 byte granularity.

Our attack can trace the control-flow of an enclave with instruction granularity, thus increasing the resolution of PT-based controlled channel attacks. Like other controlled channel attacks [41], [56], the *Frontal* attack relies on interrupts to observe instructions and control-flow within a page. However, it differs from them on the kind of branches that it can exploit. Nemesis [56] can distinguish between branches that have instructions with measurable timing differences, either because they have different kinds of instructions in their paths, or because they have a different number of instructions. CopyCAT [41] can track the control-flow in branches with a different number of instructions. The *Frontal* attack allows differentiating any branch, even if both paths contain the very same instructions and are hence not vulnerable to other controlled channel attacks. The only requirement for our attack is that the branch contains at least a memory store in it. Such higher resolution hence defeats previous defenses that rely on controlled channels being limited to observe only at a page resolution.

B. Microarchitectural Side-channel Attacks

Microarchitectural attacks exploit information leakage due to shared microarchitectural resources across different security domains and contexts. Among these shared resources, the ones that have been exploited the most are the cache and the branch prediction unit (BPU), but others have also been exploited. We examine side-channel attacks based on these shared microarchitectural components below.

1) BPU Attacks: The BPU records the outcome of recent branches and jumps, to aid the CPU speculation. As it is shared among contexts running in the same core, it can leak information about the control-flow of another context. The BPU was the focus of recent attacks, and particularly against SGX [13], [25], [34]. Two structures of the BPU have usually been exploited: the branch target buffer (BTB) and the pattern history table (PHT). On modern CPUs, the BTB records the virtual address (VA) of a branch instruction and the target VA last taken from it. The BTB has been exploited against SGX enclaves through Branch Shadowing [34] by shadowing an enclave branch at the same VA in the attacker context. By accurately synchronizing the attacker thread right after the victim executed a secret dependent branch, the attacker can detect what value was stored in the BTB for a victim branch. The PHT keeps the state of a finite state machine (FSM) based on the last decisions made for a branch. By monitoring the state changes in the PHT FSM, an attacker can detect the decisions made inside the enclave (or in another victim process) for a particular branch [13], [25].

BPU attacks require either SMT [25] or time multiplexing at a fine granularity between the victim and the attacker in the same physical CPU core [13], [25], [34]. These attacks are, in general, very sophisticated, and require reverse-engineering the BPU. Given how hard this is to achieve, BPU attacks are not easy to generalize to different microarchitectures and to pull off in practice [19]. These attacks are also limited to the type of branches they can exploit. For instance, they cannot detect the target of indirect jumps [34].

TABLE III. Overview and comparison of related SGX side-channel attacks. The first two columns indicate whether the attack can leak data-dependent or control-flow (CF) dependent secrets. The *Frontal* attack is the only attack that can leak the decision made for any type of branch (as long as they contain a memory store in them), even if they are based on unconditional jumps (e.g., as a mitigation against BPU attacks), or if both paths are contained within the same CL (e.g., as a mitigation against cache and controlled-channel attacks).

Attack type / Name	Data	CF	Resolution	Synchronization with Victim	Vulnerable branches
Cache [5], [17], [23], [39]	Yes	Yes	64 B (CL)	Interrupt / SMT / Multicore	If paths in different CLs
BPU [13], [25], [34]	No	Yes	Branch	Interrupt / SMT	Only conditional branches
TLB [18], [57]	Yes	No	4 KiB (Page)	SMT	If different data pages accessed based on path
False Dependency [40], [62]	Yes	No	4 B	SMT	If data $> 4B$ apart is accessed based on path
Port contention [4], [52]	No	Yes	μops	SMT	If paths issue different μops
PT Controlled-Channel [7], [22], [57], [60]	Yes	Yes	4 KiB (Page)	Page-Fault / Interrupt / SMT	If paths in different pages
Nemesis [56]	Low*	Yes	Instruction type and count	Interrupt	If paths have different instructions
CopyCat [41]	No	Yes	Instruction count	Interrupt	If paths have a different instruction count
Frontal attack	**No**	**Yes**	**Instruction VA**	**Interrupt**	**Any branch (Must have a store)**

*Leaks instruction operands (if they induce different execution time). E.g., multiplication to 1 vs. multiplication with big numbers.

As these attacks give fine-grained information to the attacker, research has also focused on defenses against them [24], [25], [34]. A holistic approach calls for flushing the BPU state across context switches [25], [34], but this would require at least a microcode update (or hardware changes) and incur high performance penalties. A promising software mitigation was proposed in [24]. This mitigation exploits the fact that these attacks cannot infer the target of unconditional indirect jumps. It transforms each conditional branch into a series of random unconditional jumps that access portions of each code block of the branch. However, even with this mitigation, only the intended code block of the original branch will execute. The *Frontal* attack can correlate executed instructions with their virtual address. It can hence leak control-flow dependent secrets even if a mitigation such as [24] is protecting enclaves from BPU attacks, or if the BPU is flushed on context switches.

2) Attacks on caches and other shared resources: Because caches are a resource shared across different contexts, an attacker thread can infer which accesses a victim recently made in another execution context by obtaining information about the cache state. While cache attacks often exploit timing variations in access latency to probe the state of the cache [16], state changes can also be detected by using instructions' side effects [12], [21]. Cache attacks target different levels of the cache hierarchy – from core-local data cache [1], [2], [5], [17], [23], [39], [44], [54], [63] and core-local instruction cache [1], [63], to the last level cache (LLC) which is shared amongst all cores [20], [48], [61]. As code and data are shared in the upper levels of cache (from L2), attacks that exploit them can leak both control-flow-dependent and data-dependent secrets [20], [48], [61]. Attacks on core-local caches require to be co-located with the victim and thus usually rely on simultaneous multithreading (SMT) or on accurate time-multiplexing. On the other hand, attacks that exploit the LLC can be run at the same time as the victim in another core.

The TLB is a shared buffer that stores the translation information from VAs to physical addresses. It can be exploited to detect whether a victim recently accessed a data memory page [18], [57]. Since the TLB is shared only among processes in the same core, it has been exploited only using SMT so far. It can leak data accesses at a 4 kB granularity. CacheBleed [62] was the first attack to demonstrate intra-CL leakage for data accesses, achieving a resolution of 8B. They exploited cache bank conflicts and write-after-read false dependencies. Since the adversary is not in the same address space, they induce a false memory dependency by making use of 4k page aliasing - where an address x is considered the same to $x + 4096$ by the hazard detection in the processor. Cache banks are only present in older Intel architectures and therefore cannot be exploited on newer CPUs. Moghimi et al. ported the CacheBleed attack to newer CPU and SGX while improving the resolution to 4B in their MemJam attack [40]. They exploit read-after-write false dependencies in the processor memory subsystem using 4k aliasing. The PortSmash [4] attack extended the resolution available to the attacker even further, by being able to detect issued microops in SGX enclaves. It works by keeping specific CPU execution ports busy and monitoring their execution latency. Execution in these ports becomes slower when another context is using them, thus leaking information about their control-flow to the attacker.

Summary: The main difference between the *Frontal* attacks and previous attacks lies in the type of branches these can leak. We compare it against all the attacks mentioned in this Section in Table III. Note that the *Frontal* attack is the only attack that is able to exploit unconditional and balanced branches, even when they are contained within the same page and CL. Previous defenses build either on the fact that controlled channel attacks cannot leak at sub-page granularity, or that BPU attacks cannot leak unconditional branches, and are hence ineffective against our attack in general enclaves.

IX. Conclusions

In this work, we observed a dependency between instructions execution time and their alignment modulo 16. We attribute these differences to the CPU frontend and its fetch and pre-decode module. We leveraged these timing dependencies to construct the *Frontal* attack, which can leak the instruction pointer of an SGX enclave at the byte level granularity. The frontal attack works against any kind of branch, as long as they contain at least a memory store. It can attack branches that are perfectly balanced, even when they are contained within one cacheline. In our evaluation, we showed that, depending on the target victim code, the frontal attack achieves a success rate of more than 99%.

We tested every modern CPU that currently supports SGX, and found them all vulnerable to this attack. We showed several commonly used libraries where these vulnerable branches are present. Finally, we discuss relevant defenses to the attack, highlighting that any kind of secret-depending branching should be avoided to guarantee confidentially in SGX enclaves.

Acknowledgements

We would like to thank Kaveh Razavi for insightful discussions about the root causes of the Frontal attack. We thank Kari Kostiainen for his feedback on early drafts of this paper.

References

[1] O. Aciiçmez, "Yet another microarchitectural attack: Exploiting i-cache," in *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, ser. CSAW 07. New York, NY, USA: Association for Computing Machinery, 2007, p. 1118. [Online]. Available: https://doi.org/10.1145/1314466.1314469

[2] O. Acıiçmez and W. Schindler, "A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on openssl," in *Topics in Cryptology – CT-RSA 2008*, T. Malkin, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 256–273.

[3] Advanced Micro Devices Inc., "AMD secure encrypted virtualization (SEV)," https://developer.amd.com/sev/, accessed: January 2020.

[4] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida Garca, and N. Tuveri, "Port contention for fun and profit," in *2019 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA, USA: IEEE, 2019, pp. 870–887.

[5] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. Vancouver, BC: USENIX Association, Aug. 2017. [Online]. Available: https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser

[6] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, p. 991–1008. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[7] J. V. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1041–1056. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck

[8] Cisco Systems, Inc., "Cisco annual internet report (20182023)," https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.pdf, 2020, accessed: May 2020.

[9] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *2009 30th IEEE Symposium on Security and Privacy*. Berkeley, CA, USA: IEEE, 2009, pp. 45–60.

[10] V. Costan and S. Devadas, "Intel sgx explained," Cryptology ePrint Archive, Report 2016/086, 2016, https://eprint.iacr.org/2016/086.

[11] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 857–874. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan

[12] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+abort: A timer-free high-precision l3 cache attack using intel TSX," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 51–67. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/disselkoen

[13] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," *SIGPLAN Not.*, vol. 53, no. 2, p. 693707, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3296957.3173204

[14] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP 17. New York, NY, USA: Association for Computing Machinery, 2017, p. 287305. [Online]. Available: https://doi.org/10.1145/3132747.3132782

[15] A. Fog, "The microarchitecture of Intel, AMD and VIA CPUs," https://www.agner.org/optimize/microarchitecture.pdf, 2020, accessed: January 2020.

[16] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018. [Online]. Available: https://doi.org/10.1007/s13389-016-0141-6

[17] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel sgx," in *Proceedings of the 10th European Workshop on Systems Security*, ser. EuroSec17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3065913.3065915

[18] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 955–972. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/gras

[19] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "Kaslr is dead: Long live kaslr," in *Engineering Secure Software and Systems*, E. Bodden, M. Payer, and E. Athanasopoulos, Eds. Cham: Springer International Publishing, 2017, pp. 161–176.

[20] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds. Cham: Springer International Publishing, 2016, pp. 279–299.

[21] R. Guanciale, H. Nemati, C. Baumann, and M. Dam, "Cache storage channels: Alias-driven attacks and verified countermeasures," in *2016 IEEE Symposium on Security and Privacy (SP)*. San Jose, CA, USA: IEEE, 2016, pp. 38–55.

[22] J. Gyselinck, J. Van Bulck, F. Piessens, and R. Strackx, "Off-limits: Abusing legacy x86 memory segmentation to spy on enclaved execution," in *Engineering Secure Software and Systems*, M. Payer, A. Rashid, and J. M. Such, Eds. Cham: Springer International Publishing, 2018, pp. 44–60.

[23] M. Hähnel, W. Cui, and M. Peinado, "High-resolution side channels for untrusted operating systems," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 299–312. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/hahnel

[24] S. Hosseinzadeh, H. Liljestrand, V. Leppänen, and A. Paverd, "Mitigating branch-shadowing attacks on intel sgx using control flow randomization," in *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, ser. SysTEX 18. New York, NY, USA: Association for Computing Machinery, 2018, p. 4247. [Online]. Available: https://doi.org/10.1145/3268935.3268940

[25] T. Huo, X. Meng, W. Wang, C. Hao, P. Zhao, J. Zhai, and M. Li, "Bluethunder: A 2-level directional predictor based side-channel attack against sgx," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 1, pp. 321–347, Nov. 2019. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8401

[26] Intel Corporation, "Guidelines for mitigating timing side channels against cryptographic implementations," https://software.intel.com/security-software-guidance/insights/guidelines-mitigating-timing-side-channels-against-cryptographic-implementations, accessed March 2020.

[27] ——, "Protection from side-channel attacks," https://software.intel.com/content/www/us/en/develop/documentation/sgx-developer-guide/top/protection-from-sidechannel-attacks.html, 2016, accessed May 2020.

[28] ——, "Intel 64 and ia-32 architectures software developer manuals," 2019.

[29] S. P. Johnson, "Intel sgx and side-channels," https://software.intel.com/content/www/us/en/develop/articles/intel-sgx-and-side-channels.html, 2017, accessed May 2020.

[30] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA, USA: IEEE, 2019, pp. 1–19.

[31] P. C. Kocher, “Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems,” in *Advances in Cryptology — CRYPTO ’96*, N. Koblitz, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113.

[32] B. Krzanich, “Advancing security at the silicon level,” https://newsroom.intel.com/editorials/advancing-security-silicon-level/#gs.yh984y, 2018, accessed March 2020.

[33] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, “Keystone: An open framework for architecting trusted execution environments,” in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys 20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387532

[34] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, “Inferring fine-grained control flow inside SGX enclaves with branch shadowing,” in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 557–574. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-sangho

[35] A. Limited, “mbedTLS (formerly known as PolarSSL),” 2015, https://tls.mbed.org/.

[36] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, “Meltdown: Reading kernel memory from user space,” in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 973–990. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[37] C. Liu, M. Hicks, and E. Shi, “Memory trace oblivious program execution,” in *2013 IEEE 26th Computer Security Foundations Symposium*. New Orleans, LA, USA: IEEE, 2013, pp. 51–65.

[38] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi, “Ghostrider: A hardware-software system for memory trace oblivious computation,” *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 87–101, 2015.

[39] A. Moghimi, G. Irazoqui, and T. Eisenbarth, “Cachezoom: How sgx amplifies the power of cache attacks,” in *Cryptographic Hardware and Embedded Systems – CHES 2017*, W. Fischer and N. Homma, Eds. Cham: Springer International Publishing, 2017, pp. 69–90.

[40] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar, “Memjam: A false dependency attack against constant-time crypto implementations,” *International Journal of Parallel Programming*, vol. 47, no. 4, pp. 538–570, Aug 2019. [Online]. Available: https://doi.org/10.1007/s10766-018-0611-9

[41] D. Moghimi, J. V. Bulck, N. Heninger, F. Piessens, and B. Sunar, “Copycat: Controlled instruction-level attacks on enclaves for maximal key extraction,” 2020.

[42] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, “The program counter security model: Automatic detection and removal of control-flow side channel attacks,” in *Information Security and Cryptology - ICISC 2005*, D. H. Won and S. Kim, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 156–168.

[43] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, “Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base,” in *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, Aug. 2013, pp. 479–498. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/noorman

[44] D. A. Osvik, A. Shamir, and E. Tromer, “Cache attacks and countermeasures: The case of aes,” in *Topics in Cryptology – CT-RSA 2006*, D. Pointcheval, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–20.

[45] C. Percival, “Cache missing for fun and profit,” 2005.

[46] S. Pinto and N. Santos, “Demystifying arm trustzone: A comprehensive survey,” *ACM Comput. Surv.*, vol. 51, no. 6, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3291047

[47] A. Rane, C. Lin, and M. Tiwari, “Raccoon: Closing digital side-channels through obfuscated execution,” in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 431–446. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/rane

[48] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, “Malware guard extension: Using sgx to conceal cache attacks,” in *Detection of Intrusions and Malware, and Vulnerability Assessment*, M. Polychronakis and M. Meier, Eds. Cham: Springer International Publishing, 2017, pp. 3–24.

[49] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, “Sgx-shield: Enabling address space layout randomization for sgx programs,” in *2017 Proceedings Network and Distributed System Security Symposium*. San Diego, CA, USA: Internet Society, 02 2017. [Online]. Available: https://doi.org/10.14722/ndss.2017.23037

[50] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, “T-sgx: Eradicating controlled-channel attacks against enclave programs,” in *2017 Proceedings Network and Distributed System Security Symposium*. San Diego, CA, USA: Internet Society, 02 2017. [Online]. Available: https://doi.org/10.14722/ndss.2017.23193

[51] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, “Preventing page faults from telling your secrets,” in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS 16. New York, NY, USA: Association for Computing Machinery, 2016, p. 317328. [Online]. Available: https://doi.org/10.1145/2897845.2897885

[52] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, “Microscope: Enabling microarchitectural replay attacks,” in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA 19. New York, NY, USA: Association for Computing Machinery, 2019, p. 318331. [Online]. Available: https://doi.org/10.1145/3307650.3322228

[53] B. Solomon, A. Mendelson, R. Ronen, D. Orenstien, and Y. Almog, “Micro-operation cache: A power aware frontend for variable instruction length isa,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 5, pp. 801–811, 2003.

[54] E. Tromer, D. A. Osvik, and A. Shamir, “Efficient cache attacks on aes, and countermeasures,” *Journal of Cryptology*, vol. 23, no. 1, pp. 37–71, 2010. [Online]. Available: https://doi.org/10.1007/s00145-009-9049-y

[55] J. Van Bulck, F. Piessens, and R. Strackx, “Sgx-step: A practical attack framework for precise enclave execution control,” in *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, ser. SysTEX17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3152701.3152706

[56] ——, “Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS 18. New York, NY, USA: Association for Computing Machinery, 2018, p. 178195. [Online]. Available: https://doi.org/10.1145/3243734.3243822

[57] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, “Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS 17. New York, NY, USA: Association for Computing Machinery, 2017, p. 24212434. [Online]. Available: https://doi.org/10.1145/3133956.3134038

[58] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl, “DATA – differential address trace analysis: Finding address-based side-channels in binaries,” in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 603–620. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/weiser

[59] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, “Microwalk: A framework for finding side channels in binaries,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC 18. New York, NY, USA: Association for Computing Machinery, 2018, p. 161173. [Online]. Available: https://doi.org/10.1145/3274694.3274741

[60] 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*. San Jose, CA, USA: IEEE, 2015, pp. 640–656.

[61] Y. Yarom and K. Falkner, “Flush+reload: A high resolution, low noise, l3 cache side-channel attack,” in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug.

2014, pp. 719–732. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[62] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: a timing attack on openssl constant-time rsa," *Journal of Cryptographic Engineering*, vol. 7, no. 2, pp. 99–112, 2017. [Online]. Available: https://doi.org/10.1007/s13389-017-0152-y

[63] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS 12. New York, NY, USA: Association for Computing Machinery, 2012, p. 305316. [Online]. Available: https://doi.org/10.1145/2382196.2382230

Appendix

A. Responsible disclosure

We notified the Intel PSIRT on February 21st 2020 about the Frontal attack. We sent them a previous version of this paper and a proof of concept for the vulnerabilities we identified. They informed us on April 22nd that their best practices [26] already invite to not have secret-dependent branching and therefore the attacks in this paper are considered out-of-scope for their SGX libraries. In particular, they stated that the balanced branches of the IPP crypto library we attack in Section V-C4 are not used for secret-dependent operations in the SGX architectural enclaves and hence do not pose any security implication.

B. Data-oblivious Execution

Resilience against side-channel attacks is often a desired security property when implementing software. This property is particularly important for libraries and applications that operate on secret and sensitive data on a system controlled by the attacker.

Side-channel attacks exploit secret dependent variations of the program execution. These variations are generally of two types: control-flow dependent and data-dependent. Control-flow secret dependencies are present whenever the control flow of an application depends on the confidential Data dependencies manifest when latency or resources utilized depend on the input data. For example when different memory accesses are performed based on some secret. Countless attacks have exploited these types of dependencies in the past [31], [45], [61], targeting in particular cryptographic libraries, as extracting secret keys handled by these libraries breaks any security guarantee built on top of them.

Data oblivious execution defends against side-channel attacks by removing the two dependencies mentioned above. This eliminates any variation in program execution that would be potentially observable by the attacker. There are two ways to obtain a data oblivious executable, first writing it directly low level assembly code, second by performing an automatic transformation at compile time from a higher level language. Note that writing the code in a higher level language in a data oblivious way, and then simply compiling it, might reintroduce data or control flow dependencies at the binary level.

Several techniques for compiling and transforming code from an arbitrary high level language to data oblivious code have been proposed [37], [38], [42], [47].

One of the most complete constant-time transformation is Raccoon [47].It removes any control flow and most data dependencies, by transforming secret dependent branches into a decoy and a real path that contain similar instructions. At run time, both paths are executed, allowing only the real one to modify memory, by carefully applying the conditional move instruction (`cmov`). Raccoon runs on SGX enclaves and uses the memory protections provided by it to ensure confidentiality against an attacker that can otherwise read arbitrary locations of memory.

C. Measurement Details

In this section, we describe the changes made to SGX-Step [55] to collect the measurements presented in this paper and present some interesting phenomena we noticed while making these changes.

As recommended by the original SGX-Step, we run the code in its own isolated core to reduce interference from other processes and the kernel scheduler. We were running the kernel with watchdogs disabled to avoid that any other kernel function gets scheduled while we are single-stepping through the enclave. We executed SGX-Step with the userspace interrupt handler.

The modifications we made to the tool were aimed at decreasing the noise and stabilizing the timing results. We noticed that code executing outside the enclave would often impact the variance of the collected latency. To reduce this variance, we made three changes. The first was in the `aep_cb_func`, which is called every time an interrupt happens. We made the function constant time and made sure that no function calls where performed in it. We tried to reduce the cache footprint as much as possible as well to make sure that no unnecessary data is evicted from it.

Second, to stabilize the interrupts and make them more precise, we explicitly serialize the instruction stream before setting the APIC counter and eventually entering the enclave. Before this change, we sometimes observed multi-steps even with very tight interrupt intervals. Interesting, while debugging for multi-steps, we observed that we were never able to interrupt in between fused macro-instructions, these seem to be treated atomically by the CPU.

Third, we filter out the first measurement of a new page. This is because the first measurements of a new code page tend to have a high variance and be difficult to predict. This is probably due to the fact that the new code page is not in the cache, and somehow the CPU is not prefetching them in time. Note that in a real attack, the adversary may want to select gadgets that are not at a page boundary as she cannot afford to throw away measurements.

Finally, there was one source of noise that was always present and simply seemed to act as a constant shift across measurements. Across enclave creations the distributions of the measured latency had a variable mean. That is, measuring the same enclave, but on different runs, gives different results. However, the results were always consistent within the same run. While the shifting was always between 0 and 200 cycles, this value is still so large that sometimes the timing of `nop` instructions could be longer that the timings of a multiplication (across different runs). Given this shifting in the measurements we concluded that instructions timings are not comparable across different runs.

D. Outside Intel SGX

A question remains on whether these effects manifest only while executing code inside an SGX enclave or whether they are present also while running a program outside of the SGX. Since we cannot send interrupts fast enough during a normal execution, we decided to simulate the effect of the interrupts by modifying the code in Figure 2 such that each `mov` triggers an exception. We handle the exception and measure the time it took to execute it, and then resume the program execution from the instruction after the one that triggered the exception. Note that exceptions are handled very similarly to interrupts, with the key difference that the instruction that is currently executing retires when an interrupt is triggered, while it needs to be discarded when an exception is raised.

The timing differences between instructions in the two branches were less pronounced than when the code was run within SGX. Nonetheless, we were able to observe a (small) correlation between the exception handling time of single instructions and the branch being executed. This correlation hints that the effects are not only present when interrupting enclaves, but would manifest also when interrupting applications if we had a fast enough interrupt timer.