



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

Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Čapkun, *ETH Zurich*

<https://www.usenix.org/conference/usenixsecurity21/presentation/puddu>

This paper is included in the Proceedings of the
30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the
30th USENIX Security Symposium
is sponsored by USENIX.



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 timing differences that arise from how the CPU frontend fetches and processes instructions while being interrupted. In particular, we observe that in modern Intel CPUs, some instructions' 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 had known branch targets, 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 further demonstrates that secret-dependent branches 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. We successfully tested the attack on all the available Intel CPUs with SGX (until 10th gen) and used it to leak information from two commonly used cryptographic libraries.

1 Introduction

Today's computing world runs in the cloud. Massive data centers maintained by cloud providers are the infrastructure upon which companies and most of the internet are increasingly relying [1]. For many use cases, renting computing resources is cost-effective and convenient. Resources can dynamically scale up when demand is high, 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 can 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 [2, 3, 4, 5, 6, 7, 8], 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. Intel SGX [2] is the most widely deployed among all the TEE proposals, being available in almost every modern consumer CPU Intel manufactures. It protects applications by running them in *enclaves*. SGX authenticates and encrypts enclaves' memory accesses that cross the CPU boundary and blocks any other software in the system, including OS and hypervisor, from accessing enclaves' code and data. Nevertheless, as protected as they might be, enclaves do not execute in isolation. Enclaves share resources with other applications running in the same system, particularly memory and CPU time. By design, SGX leaves the (untrusted) 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 [2, 9, 10, 11, 12], casting 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 [13]. 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 interrupt and exception handling, to name a few. These OS capabilities, which the attacker controls, decrease the noise of traditional side-channel attacks [9, 14] and enable new types of side-channels, called controlled-channel attacks [12].

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

```

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.

accesses. Initial defenses that worked on the assumption that the attacker would need to trigger page-faults [15], just prompted the emergence of stealthier attacks that observe page metadata set by the CPU [16, 17]. In response to these attacks, Intel officially recommends SGX developers place sensitive data and code within a page [18]. Controlled channels, however, do not stop at the page boundary. OS capabilities can be used to enhance cache attacks [9, 10, 11] and extract enough information from the branch prediction unit (BPU) to give the attacker a branch granularity view of the victim [14, 19, 20]. As this undermines defenses against paging-based controlled channels, further defenses leveraged the coarse timing resolution of the attacker and the inability of BPU attacks to leak the target of unconditional branches [14]. Nemesis [21] 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 [22] 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, even widely used crypto libraries tend to use balanced branches¹ 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 attacker’s resolution further and demonstrating that virtually any binary with control-flow secret dependencies leaks information in SGX.

Frontal attack: For the first time, we show that when interrupts are frequently issued, instructions’ execution time is correlated to their virtual address and that the fetch and pre-decode module of the CPU frontend plays a role in this correlation. Based on this observation, we construct a new attack against Intel SGX that we call the *Frontal* attack. Our

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

attack allows an attacker to associate a measured instruction’s execution time with its offset in the *instruction fetch window* and thus with the instruction’s virtual address. The attacker can then use these leaked execution times and addresses to infer control-flow and, therefore, branch-dependent secrets.

We focus on extracting branch-dependent secrets, showing that an adversary can distinguish between two code sequences executed within SGX and hence, derive the secret branch condition that led to their execution. Unlike previous attacks [21, 23], 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 probability up to 99% on our test binaries. We discuss how two different libraries, the mbedTLS library [24] and the Intel IPP [25] Cryptography library, can be exploited using our attack. Showing that, with just a few runs, the attacker can recover the condition of the executed victim branches with high confidence (> 99.9%), and that with a single trace it is possible to recover a full RSA key within seconds on 65% of the runs (out of 1000). We validated our attack on all available Intel microarchitectures since the introduction of Intel SGX (up to Comet Lake at the time of writing). We show that the attack works with high probability on all tested CPUs irrespective of their microcode version. 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 [26] or by writing code following data-oblivious practices [27]. 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 high-performance overheads or require developer assistance to mark secrets [26]. Second, custom data oblivious solutions are not trivial to develop correctly and require domain-specific knowledge [27].

These practical hurdles for data-oblivious code have led to several spot defenses being continuously refined based on the adversary’s capabilities. We give further evidence in this paper that these 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 the dependency between execution time and virtual address 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 in our synthetic binaries.
- We exploit two commonly used cryptographic libraries using the Frontal attack: the Intel IPP Cryptography library, and the mbedTLS library. We further test which CPUs are vulnerable to our attack and found that all available CPUs with SGX at the time of writing (up to 10th gen) are vulnerable. Newer CPUs that include hardware mitigations against Spectre [28] seem to be more vulnerable than older CPUs. We responsibly disclosed the findings to the affected vendors.

2 Background

SGX-Step & Nemesis SGX-Step [29] 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 interrupt-vector entry defined in the interrupt descriptor table (IDT) to handle 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 [2]. 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 the execution time of individual instructions (including a constant offset by the `ERESUME` and `AEX`).

The Nemesis [21] 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 can only infer the instruction type, it cannot resolve the IP whenever a balanced branch is executed in the enclave.

CPU Background: The Frontend Although the x86 instruction set architecture (ISA) is well specified [30], the microarchitecture is typically proprietary, and its details are confidential. 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 [31].

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. A modern Intel core fetches 16 bytes at once [31], from 16 bytes aligned blocks, also called the instruction *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).

3 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 their 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 [2]. That is, they control the entire software stack, including the operating system (OS), on the machine in which the enclave executes. Since the attacker controls the OS, we assume they can disable any CPU core to reduce noise or prevent the scheduler from running tasks on a particular core. 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 (x86 assembly). The code fits in a single cacheline and has a branch whose target depends on a secret value. On both branches, the code contains the very same instructions and writes to the same memory addresses. Thus, we expect

```

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 ← Int #1
0x14: mov %rax, (var1) ← Int #2
0x19: add $1, %rbx ← Int #3
0x1d: mov %rbx, (var2) ← Int #4
0x22: ret
...
.else:
0x2b: add $2, %rax ← Int #1
0x2f: mov %rax, (var1) ← Int #2
0x34: add $2, %rbx ← Int #3
0x38: mov %rbx, (var2) ← Int #4
0x3d: ret

```

(b) Secret-dependent branch in asm

Figure 1: A secret-dependent branch in C and x86 assembly. Both branches in the assembly code fit within the same cache-line (64B). The virtual address of the instructions is reported on the left. Note that while the branches are instruction-wise identical, their instructions get grouped differently by the fetch window (which always start at multiples of 16B).

its execution time to be independent of which branch is taken and hence not to have any correlation with the secret input.

However, when the above sequence is run within an SGX enclave, our attack shows that 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 aligned differently within the fetch windows (Listing 1b) – in our experiments, this alone did not produce observable differences in the execution times (cf. Section 4). 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 1), resulting in a measurable difference in the execution times of the instructions and therefore of the branches (cf. Section 4).

To give an insight into 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 oldest pending instruction in the re-order buffer is retired² before the interrupt is handled [29]. In out-of-order processors, other instructions might have already

²Or discarded, if it raises an exception

	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		

Table 1: Here 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.

been executed, but none of these will be retired. To resume execution after the interrupt is handled, the CPU needs to fetch the instruction sequence starting at the current program counter. However, while the program counter can, in general, have any value, fetch windows are statically aligned at 16 bytes code blocks [31]. Assume that the program counter falls 5 bytes after the start of the fetch window. Those initial 5 bytes will be fetched only to be then discarded by the frontend. Thus out of 16 bytes fetched, only 11 are usable. Now assume that the same instruction sequence begins 10 bytes after the start of the fetch window. Instead of 11 bytes as before, there are only 6 bytes that can be decoded, meaning we now need two fetch windows (and hence two cycles) to decode the same number of instructions as we did before in just one fetch window. Alignment w.r.t. fetch windows can, therefore, change the order in 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 1 which instructions are fetched after every interrupt.

In principle, given the same system conditions, a particular instruction should exhibit the same time distribution at different virtual addresses. 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 4, 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). 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 1 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 again consider Listing 1b. By running SGX-Step, we can time all instructions by stepping through them one by one. As a consequence 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 “slower” (compared to the others measured), we must be executing the `mov` at address 0x14. Inversely, if the interrupt is “faster”, we must be executing the `mov` at address 0x2f. 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 timings for the first `mov` instruction inside the branch. We were able to use the timing difference to predict the secret with $\geq 65\%$ accuracy. By adding 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 our attack in more detail.

4 Frontal Attack Profiling

In this section, we provide more detail and clarification to 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 the fetch window alignment / instruction address on the attack? and (iii) which instructions produce observable timing differences?

To answer these questions 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 depending 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 that are more clearly above the noise floor than 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 secret branch condition with an accuracy of at least 99%, whereas with shorter sequences that contain few `movs` (like the one in Figure 1), this accuracy drops to $\geq 65\%$. We discuss this effect in more detail later in this section.

4.1 The Role of Interrupts

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.

```

        .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

```

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

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.

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 2), so we have to measure the execution time from the untrusted app.

We perform this measurement using three different methods. All methods use the same code snippet in a loop, but they differ in how the measurement is collected and where the loop is executed. We do this to filter out any effects of the enclave entry and exit operations. First, we measure a whole enclave call from the untrusted app. Multiple measurements are collected by having a loop in the untrusted app. Second, we run the loop entirely inside the enclave and collect the iteration execution time with two `ocalls` to the untrusted app. The two `ocalls` are done at the beginning and the end of each loop iteration. 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 increments the counter in a loop, thus simulating the time stamp counter, albeit at a lower precision. All three methods use an independent uniform random value as the “secret” given to the code at each iteration.

For all three methods, similar to the experiment outside of SGX, we observed no significant correlation (Pearson’s

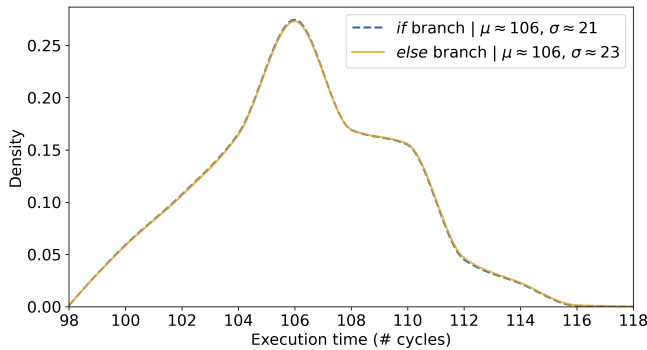


Figure 3: Distribution of the overall execution time of the branches in Figure 2 when run outside of SGX without interrupts (computed from $2 * 10^9$ samples).

coefficient $\approx 10^{-2}$ with 10^6 runs) between the execution time and the secret provided to the enclave.

In SGX with interrupts We now investigate which effects frequent interrupts have on the execution time of the code. We execute the same code snippet as before 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 [29] 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 correlate 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

As in Section 3, we observed the execution time of the first `mov` in each branch to be faster or slower, depending on the branch it belongs to, with a difference between the slower and faster `mov` of around 100 cycles. This observation allowed us to set a timing threshold with which we could, with up to 99.9% accuracy, determine which branch was taken, and therefore determine the secret branch condition.

We stress again that the two branches are instruction-wise

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

⁴The timings of the initial `cmp` and `jnz` were independent of the executed branch - only instructions within the branches were correlated with the secret.

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 to also 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 secret.

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.

4.2 Relationship to Virtual Addresses

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 3, we also study how the relationship between the alignment of the branches with respect to the fetch window affects 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 4.1. 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, and thus what the secret was. We then calculate the attack success as the percentage of correctly identified secret bits. 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 an alignment combination is for the attack, while a result close to 50% indicates that predicting 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

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

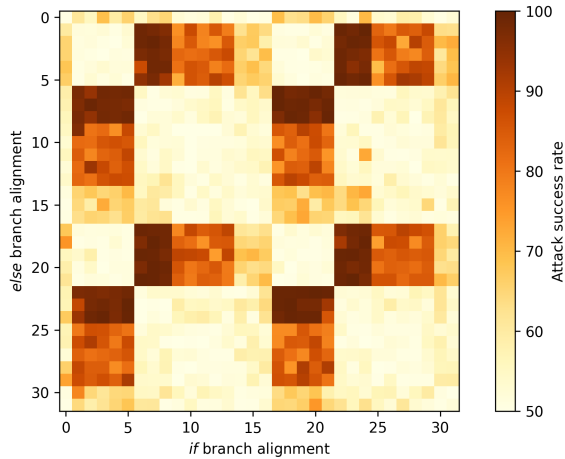


Figure 4: Attack success rate depending on the alignment of the branches. The attack success rate is the percentage of correctly guessed branches by the attacker out of 1000 executed branches. The 10th instruction (5th `mov`) from Figure 2 is used to distinguish between both branches. The color gradient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

code in Figure 2 1000 times with uniformly random secrets. We use the timings of the 10th instruction (5th `mov`) to discriminate between the branches. Figure 4 presents the result of our experiment. These results show a clear dependency between virtual addresses and the instruction execution times.

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 the two branches are still contained in the same 4 kB virtual page.⁶ We observed the same pattern for all the quadrants of length 16 in this test. 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.

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 $X = Y \bmod 16$.

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

⁶We 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 that we are interested in measuring.

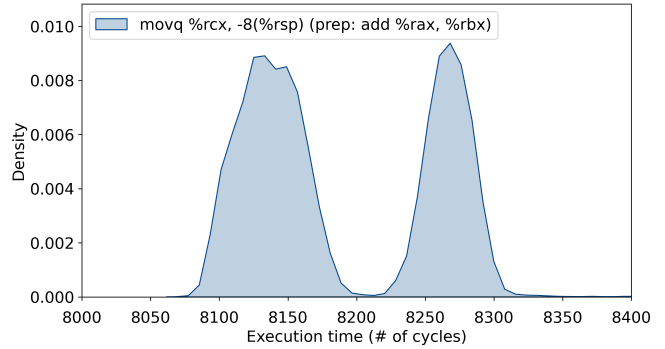


Figure 5: Timing distribution of a `mov` to the stack when executing it in a trace containing 100,000 repeated `add-mov` instructions (unrolled).

Symmetry 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.

Shape Finally, we focus our attention on the alignments in the heatmap in which the success rate is above 70%. 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 $5B$. Unfortunately, this rule does not trivially generalize with more complex instruction size combinations.

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, one of them is the *instruction pre-decode and fetch* module in the frontend 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 3, the different alignment changes the way instructions are batched by the frontend 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 time. We will discuss potential causes that could lead the observed variable timings in Section 7.

4.3 The Effects of Instruction Alignment

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

The timings are collected using a slightly modified version of SGX-Step, whose changes are described in Appendix A.3. The timing of each instruction includes 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 latencies 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.

Distribution of instruction execution times In Figure 5 we present the distribution of the instruction execution times, estimated from all the 100,000 executed `mov`. The most evident feature of this distribution is that it consists of a *bimodal* Gaussian distribution. The `movs` are therefore exhibiting two different distribution modes, whose peaks are, on average, around 100 cycles apart. We refer to the mode with the lower average and the one with the higher average as the *fast mode* and *slow mode*, respectively.

Observation 3.1: The timing distribution of the `movs` follows a bimodal distribution. The peaks of the two distribution modes are around 100 cycles apart.

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.

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

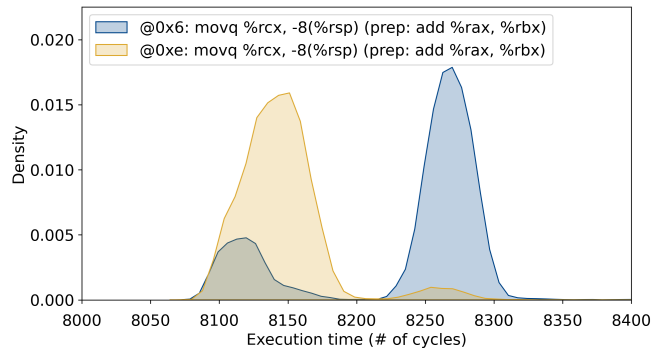


Figure 6: Timing distribution of the `movs` from Figure 5 grouped by their virtual address alignment.

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 modulo 16 in two different ways. 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 usually they exhibit timings from both distribution modes. We can think of each instruction at a given alignment to have a certain intrinsic probability p to exhibit the fast mode and probability $1 - p$ to exhibit the slow mode every time it executes. Different alignments have a different value of p . Figure 6 shows this phenomenon for two particular alignments (`0x6` and `0xe`). As can be seen, alignment `0x6` is predominately slow, but some of its timings exhibit the fast mode as well. The plots for other alignments are similar, with the only difference 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 distribution modes.

The value of p relates to the attack success rate. Say that one branch is aligned such that the measured `mov` has $p \geq 0.9$, and the other is aligned to have a $p \leq 0.1$ then the branches are easily distinguishable, and a high success rate will be observed. If one of them has $0.3 \geq p \geq 0.7$, and the other a very small or very high p , as is the case for the distributions in Figure 6, then one bit can be distinguished with high accuracy, but the other will contain some errors. If the branches have a $p \approx 0.3$ and say $p \approx 0.7$, then both branches will be on average guessed better than random, but will also contain errors. And finally if both branches have a similar p the success rate of the attacker will be negligible.

4.4 Requirements and Limitations

In our experiments, we only observed timing differences in branches which contain *memory writes*. Thus, at least a memory write must be present for the side-channel to emerge. All the other conditions being equal, other memory write instructions we tested (variations of `mov` to different addresses and arithmetic instructions that write back to memory), excluding the `push` instruction, exhibited the very same behaviors as described so far. Notably, instructions that are surrounded by other memory writes also show a timing difference, albeit usually smaller. Furthermore, the timing distribution of a memory write is not only determined by its alignment in isolation, but it is also influenced by the number and alignment of surrounding memory instructions. For instance, the more memory writes in the branch (or even right after it), the more distinguishable the distributions will be, increasing the probability of success of the attack. Another element we were able to characterize, relates to the vicinity of the memory instructions with each other. Particularly, when writes are executed in a loop, the attack success probability is higher if the loop executes only a few instructions (around 10) in between writes, and the fewer, the better, for the attack.

It is worth noting that simultaneous multi-threading (SMT) was a big source of noise in our experiments. When the core co-located with the victim is executing a CPU-heavy workload, we were unable to observe any significant timing difference. In general, the Frontal attack is more reliable if SMT is disabled or the virtual core co-located with the victim is idle. We speculate that this is most likely due to how the frontend handles and fetches instructions coming from different virtual cores, but possibly also to the resulting lower interference in the memory subsystem.

5 Frontal Attack Exploitation

The Frontal attack exploits control-flow secret dependencies. Therefore, the first step of the attack is to identify target code paths in the victim binary which execute secret-dependent branches. Several techniques have been proposed to automate finding such code paths [32, 33]. Among these code paths, as discussed before, the attacker should choose those that contain at least one memory write. Until now, we mostly focused on balanced branches, but unbalanced branches are also distinguishable with our attack. As unbalanced branches can be exploited with other attacks as well, we focus on more challenging balanced branches in our example exploits below. Balanced branches are not rare in compiled code. In fact, we found two code patterns that commonly lead to this type of branches: slightly different return statements, and inlined function calls with different parameters.

In the following, we give examples of vulnerable branches satisfying the conditions above in two libraries: the Intel IPP Cryptography library [25], and the mbedTLS library [24].

We note that since a secret-dependent code path must be present, branch-prediction attacks can also exploit the binaries vulnerable to the Frontal attack. For instance, the examples we present below, when compiled with `gcc` are also vulnerable to branch-shadowing attacks [14]. However, when compiling the mbedTLS library with the compiler from [22], all the branches are translated to indirect unconditional jumps, which are hitherto not vulnerable to any known BPU attack. On the other hand, we verified that even when using [22] the branch targets are unchanged and have in general different alignments, thus remaining vulnerable to the Frontal attack. The attacks described in this section were performed on an Intel i9-9900KS CPU with the latest microcode available at the time of writing (0xca).

5.1 Intel IPP Cryptography Library

The Intel IPP Cryptography library is a cryptographic library optimized for Intel CPUs and advertised as constant-time [25]. However, through manual inspection we identified several secret dependent branches in its most recent version (2.9 at the time of writing). Among these, the `l9_ippsCmp_BN` function compares two big numbers represented as arrays of integers by iterating through each element of the array. The function then terminates when a different array entry is found. It can take three different exit paths, depending on whether the first input is smaller, bigger, or equal to the second. The *smaller-than* and *bigger-than* paths are instruction-wise identical, while the *equal* path contains the same instructions as the others but in a different order. Given that the different order of instructions of the equal vs. unequal paths can be inferred with other attacks, we focus on distinguishing the *smaller-than* vs. *bigger-than* paths with the Frontal attack. With branch-prediction mitigations in place, other known attacks do not allow to leak this information, as all the paths fit in a single cache-line. The exit paths contain a `mov` to memory, which we target in our attack. We did not observe any timing difference on this instruction alone, despite the fact that the paths start at different alignments, this is expected as the memory write is executed only once. However, by inlining the function in an enclave that performs a loop of at least 9 memory writes after the IPP function call, we obtained the distributions shown in Figure 7. The figure shows two distributions that differ in their modality. The timing distribution of the `mov` in the *smaller-than* path has a single peak around 9400 cycles. On the other hand, the `mov` in the *bigger-than* path exhibits two modes, a small one around 9300 cycles, and a predominant one at 9525 cycles, and is thus usually slower to execute than the `mov` in the *smaller-than* path. Consequently, if a measured `mov` timing is “slow” it must mean that the *bigger-than* path was executed (3% false positive). Overall, by using this comparison repeatedly with a secret bitstring as input, we were able to accurately recover 25% of the secret’s bits (with 1000 function calls).

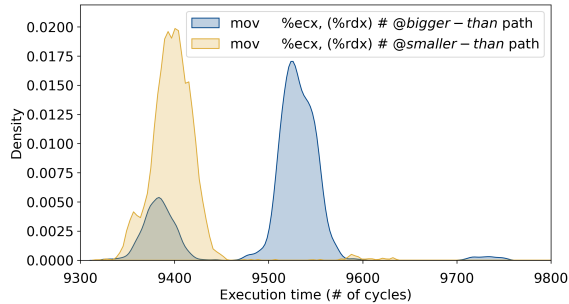


Figure 7: Timing distributions of two different `mov`s in the IPP Cryptography library’s `19_ippsCmp_BN` function (each estimated from 3000 samples). The function executes a secret dependent comparison, which can result in two balanced paths being taken: the *bigger-than* or *smaller-than* path. Each path contains a differently-aligned `mov` in it, whose distribution is shown in the figure.

5.2 Montgomery Modular Multiplication

The Montgomery modular multiplication (MM) is a fast MM algorithm often used in cryptographic libraries due to its efficiency and minimal secret dependence. There is only a single secret-dependent branch in the algorithm: a conditional subtraction that is done at the end of the multiplication. MM is used to perform modular exponentiation, and knowing whether the subtraction was done or not leaks some bits of a secret key used in the exponentiation [34]. Some implementations, including `mbedtls` as of version 2.16.6, just balance the branches by adding an `else` branch with a dummy subtraction in it (cf. Listing 1). However, this naive mitigation is still vulnerable to side-channel attacks that target control-flow secret dependencies, such as the Frontal attack. We compiled the `mbedtls` library with the `gcc -O3` flag and used it inside an enclave that performs a modular exponentiation (as the MM function is not directly exposed in the library’s API). The `O3` flag inlines functions when possible, so instead of performing two function calls, as shown in Listing 1, the binary contains two identical copies of the `mpi_sub_hlp` function. The branch condition determines which of these two gets executed. The `mpi_sub_hlp` function contains a loop with two memory writes. The loop repeats a number of times proportional to the size of the modulus of the multiplication. In Listing 2 in the Appendix, we give the assembly code generated by the compiler for the loop we exploit. Since the two loops were aligned differently, they exhibited different timing distributions, as shown in Figure 8. While the differences were not as big as seen in our controlled tests (most likely due to the fact that several instructions are executed in between consecutive memory writes), they were enough to differentiate the branches. Using Welch’s *t*-test, we correctly classified 83% (511 out of 616) subtraction calls (whether they were dummies or not) with 99.9% confidence with just 16 repetitions of an exponentiation with the same inputs.

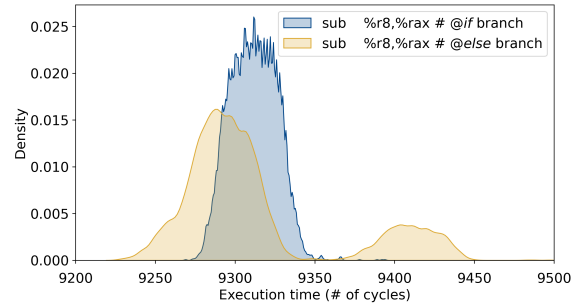


Figure 8: Comparison of the real subtraction (*if* branch) and dummy subtraction (*else* branch) branches in the `mbedtls` MM implementation. The two branches are identical, and both include a `for` loop that executes two memory writes (cf. Listing 2). The graph shows the distribution of the 11th instruction in the `for` loop (a `reg to reg` subtraction), highlighting that as long as memory writes are present, surrounding loop instructions produce different distributions based on their alignment as well. The distributions were estimated from 1000 function calls, each of which has 6 loop iterations, resulting in 6000 measurements per instruction.

5.3 Leaking RSA Keys

We demonstrate a full end-to-end attack leveraging the Frontal attack by exploiting the function that generates a new random RSA key pair (`mbedtls_rsa_gen_key`) in `mbedtls` v2.16.6. This function has several secret-dependent branches. The one we target is executed during the computation of $\text{gcd}(e, (p-1)(q-1))$, where e is the RSA public exponent and p and q are two RSA primes. Leaking $(p-1)(q-1)$ allows to easily compute the RSA private key (as together with $n = pq$ we can solve for p and q and then compute $d = e^{-1} \bmod \lambda(n)$). Control-flow leakage from the `gcd` implementation has been thoroughly studied [35, 36, 37] but it only leads to partial information recovery without fine grained execution traces [35]. The binary `gcd` implemented in `mbedtls` has a main loop that removes the trailing zero bits to its operands and then has a balanced branch in which a subtraction and a `shift-right` is performed. To recover the RSA private key it is sufficient to leak two pieces of information: the output of the function that counts the number of trailing zero-bits and the path taken in the balanced branch. We leak the trailing zero-bits by counting the number of instructions executed in the respective function, as demonstrated in [23, 37]. The result of the balanced branch is leaked with the Frontal attack. Similar to the MM attack described above, the branches need to contain inlined function calls for the attack to work. To achieve this, we modified the signature of the `int mbedtls_mpi_shift_r(...)` function in `bigint.c` to `inline int mbedtls_mpi_shift_r(...)`. Note that different compiler versions might lead to the compiler inlining this function on its own, thus producing a vulnerable binary. With this function inlined, the branches

Processor	μ arch	Launched	μ code	Mitig.	Vulnerable
i7-6700HQ	Skylake	Q3'15	0xc2	μ code	yes [†]
i7-6700HQ	Skylake	Q3'15	0xd6	μ code	yes [†]
i7-7700	Kaby Lake	Q1'17	0x48	-	yes
i7-7700	Kaby Lake	Q1'17	0x8e	μ code	yes [†]
i7-9700K	Coffee Lake R	Q4'18	0xb8	HW	yes
i7-9700K	Coffee Lake R	Q4'18	0xca	HW	yes
i9-9900KS	Coffee Lake R	Q4'19	0xb8	HW	yes
i9-9900KS	Coffee Lake R	Q4'19	0xca	HW	yes
i9-10900K	Comet Lake	Q2'20	0xca	HW	yes
Xeon E-2278G	Coffee Lake R	Q2'19	0xb8	HW	yes
Xeon E-2278G	Coffee Lake R	Q2'19	0xca	HW	yes

[†] Only vulnerable in some runs (see Figure 9)

Table 2: List of all the processors we tested with their respective microcode version. The *Mitig.* column indicates whether the mitigation against known microarchitectural attacks such as Spectre and Foreshadow is implemented in hardware (HW) or μ code.

both contain the loop shown in Listing 3 in the Appendix, leading to a differently aligned memory write depending on the branch taken. This loop within the branches is usually executed 32 times, giving us a fairly high number of memory writes to profile. We collect and use the information from the distribution of each instruction in the loop in order to recognize which branch is being executed. The overall timing distributions are omitted here due to lack of space, but in short some instructions look like Figure 8, while others more like Figure 7. This means that we can classify the branch whenever any instruction’s timing is in the “slow” mode of Figure 7 or whenever an instruction’s timing is in the tail of the distributions of Figure 8. We executed 1000 runs, fixing the exponent to $e = 65537$, and generating a new pseudo-random key in each run. Note that since a new key is generated on each run, we cannot correlate the executions of multiple runs. In each execution, the attacked branch was executed 1018 times on average ($std = 25.40$), and on average we could not classify 89 ($std = 92.35$, median = 55) branches. This means that on average we would need to brute force 89 bits to recover the secret key. In practice, we noticed that since the exponent is orders of magnitude smaller than $(p - 1)(q - 1)$, early iterations of the secret branch are very likely not taken. Leveraging this information, we perform several guesses of the key starting from the last unclassified iteration. We assign this iteration as ‘taken’ and check if this results in a correct key. If not, we assign the next iteration as taken as well and repeat. This greedy approach worked on 65% of the runs and allowed us to recover the key of those runs in a matter of seconds.

6 Affected Processors and Configurations

We tested five different processors from the 6th generation, which introduced Intel SGX, up to the 10th which has hardware mitigations for recent microarchitectural attacks [38]. We give the details of the CPUs tested in Table 2. For each

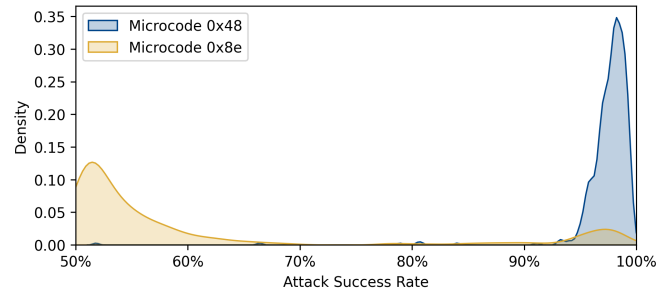


Figure 9: Distribution of the attack success rate with different microcode versions of an Intel Core i7-7700 CPU - across 500 runs per microcode. For each run, we estimate the attack success rate as the percentage of branches the attacker guessed correctly among 1000 executed branches from Figure 2, with alignment $X = 6, Y = 2$.

processor, we tested the minimum microcode version supplied by the mainboard and the most up to date version as of February 2020. Each CPU was tested by computing the attack success rate for various alignments as in Figure 4. The Frontal attack was successful on all tested CPUs and microcodes.

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. 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 increase the number of cycles used for AEX and ERESUME and add some randomness to our experiments. For these configurations, every run of the experiment exhibits a different behavior. Figure 9 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 they could decide whether to attack or not before the secret is retrieved or provisioned, and relaunch the enclave until its behavior is clearly vulnerable.

7 Potential Causes

The complexity of the microarchitecture of current Intel processors makes it very challenging 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.

Memory Subsystem Observation 3.1 and 3.2 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 `movs`.

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 2.1, we know that the alignment influences the timing difference. We thus rule out the execution engine as the root cause of the timing differences.

Frontend Observation 2.1 strongly hints at the frontend as the culprit, since the fetch window is one of the only structures which operates at a 16 Bytes granularity, matching the 16 Bytes periodicity of the observations.

The micro-op cache is a microarchitectural structure in the frontend [39] that 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. Whereas, 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 `movs`. Thus, we rule out the micro-op cache as a cause.

Branch prediction is responsible for predicting the future control flow. The core will fetch ahead and speculatively continue to execute in the predicted path. Branches and jumps where the target is not immediately known (e.g., the target comes from memory) both rely on the branch predictor to guess which instruction will be executed next. Hence, the resumption of the enclave could potentially suffer from a misprediction on the current enclave instruction and therefore

suffer from a delay. However, all performance counters that we measured did not show any additional mispredictions for slow or fast instructions.

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 observed timings exploited by the attack.

8 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 data-oblivious code [40, 26] is a principled approach that thwarts every known side or controlled-channel attack. We discuss these techniques in Appendix A.2. As such it also remains secure against the Frontal attack. Nevertheless, data-oblivious code presents several challenges in practice, as it is hard to get right and results in 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 simple defense we discuss is at the software level. Hardware mitigations would also be possible, but due to the lengthy turn-around time for new processors, software defenses are more attractive.

As seen in Section 4, the execution time of individual instructions depends on their alignment. Particularly, branches with identical alignment do not exhibit any observable timing difference. Therefore, aligning the two branches to the same address (modulo 16) leads to indistinguishable timing distributions for both branches. We evaluated the overhead in terms of binary size and performance of this approach on three common libraries: `libc`, `OpenSSL`, and `mbedtls`. We used `GCC v7.5.0` with the compile flag `-falign-jumps=16` - this flag aligns all branch targets to `0x10`, thwarting our attack. The highest size overhead (3.73%) was on one of the binaries generated for `libc`, this however was the only outlier as all the other binaries had an overhead of less than 0.5%. For comparison, compiling with `-O3` added on average 14% compared to `-O2`. To evaluate performance, we use `libc-bench`⁷ for `libc` and the benchmarks that come with the libraries for `mbedtls` and `OpenSSL`. The `strstr` test in `libc-bench` had the highest overhead at 30%, and `libc` overall had an average overhead of 1%. Depending on the evaluated cryptographic function `mbedtls` had overheads ranging from 4% to -5.5%, while `OpenSSL` from 3% to -4%, showing that for some cryptographic functions' implementation the defense even provides performance boosts.

⁷<https://www.etalabs.net/libc-bench.html>

9 Related Work

We compare our attack and related ones in Table 3. In short, the main differences lie in the type of branches that are vulnerable to the various attacks. Previous defenses build either on the fact that controlled channel attacks cannot leak at sub-page granularity or that BPU attacks cannot leak the target virtual address of unconditional branches. In general, these defenses are ineffective against our attack since we exploit a fundamentally different mechanism. In the following, we describe the differences between the Frontal attack and other related attacks in more detail.

9.1 Controlled-Channel Attacks

The attacker’s control over the OS enables novel noise-free deterministic side-channels [12, 16, 17] known as controlled-channels since the attacker 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. In [12], Xu et al. modify page permissions so that the CPU generates a page fault for each page the enclave tries to access. 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 [17, 16] 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 onto, to protect enclaves [48, 49]. These defenses either call for sensitive code to be within a page [48] or randomize the 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” [18]. However, the resolution of controlled channel attacks was increased through an attack exploiting legacy memory segmentation [47], 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 [21, 23], 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 [21] 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. Copy-Cat [23] 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.

9.2 Microarchitectural Side-channel Attacks

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

BPU Attacks The BPU records the outcome of recent branches and jumps, to aid the CPU speculation. As it is shared among different execution 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 [14, 19, 20]. BPU attacks require either SMT [20] or time multiplexing at a fine granularity between the victim and the attacker in the same physical CPU core [20, 19, 14]. 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 [50]. These attacks are also limited to the type of branches they can exploit. For instance, they cannot leak the target virtual address of indirect jumps [14]. As these attacks give fine-grained information to the attacker, there have been a few defenses proposed against them [14, 20, 22]. Most notably, some defenses call for a holistic approach by flushing the BPU across context switches [14, 20]. Other defenses propose spot defenses such as replacing every branch with indirect jumps [22]. BPU attacks are particularly related to the Frontal attack, as they both exploit secret-dependent branches. However, as the Frontal attack exploits a fundamentally different mechanism, any spot-defense against BPU attacks is not effective against our attack.

Attacks on caches and other shared resources Because caches are a resource shared across different execution contexts, an attacker thread can infer which accesses a victim recently made in another 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 [51], state changes can also be detected by using instructions’ side effects [52, 53]. Cache attacks target

Attack type / Name	Data	CF	Resolution	Synchronization with Victim	Vulnerable branches
Cache [41, 9, 10, 11]	✓	✓	64 B (CL)	Interrupt / SMT / Multicore	If paths in different CLs
BPU [14, 19, 20]	✗	✓	Branch	Interrupt / SMT	If paths virtual addresses are known
TLB [42, 16]	✓	✗	4 KiB (Page)	SMT	If different data pages accessed based on path
False Dependency [43, 44]	✓	✗	4 B	SMT	If data > 4B apart is accessed based on path
Port contention [45, 46]	✗	✓	μ ops	SMT	If paths issue different μ ops
PT Controlled-Channel [12, 17, 16, 47]	✓	✓	4 KiB (Page)	Page-Fault / Interrupt / SMT	If paths in different pages
Nemesis [21]	✗*	✓	Instruction type and count	Interrupt	If paths have different instructions
CopyCat [23]	✗	✓	Instruction count	Interrupt	If paths have a different instruction count
Frontal attack	✗	✓	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.

Table 3: 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 indirect 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).

different levels of the cache hierarchy – from core-local data cache [54, 55, 56, 57, 58, 9, 10, 41, 11] and core-local instruction cache [54, 57], to the last level cache (LLC) which is shared amongst all cores [59, 60, 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 [59, 60, 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 virtual addresses (VA) to physical addresses. It can be exploited to detect whether a victim recently accessed a data memory page [42, 16]. 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 [43] was the first attack to demonstrate intra-CL leakage for data accesses, achieving a resolution of 8B. It 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 as $x + 4096$ 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. [44] ported the CacheBleed attack to newer CPU and SGX while improving the resolution to 4B in their MemJam attack. They exploit read-after-write false dependencies in the processor memory subsystem using 4k aliasing. The PortSmash [45] 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.

10 Conclusions

In this work, we observed a dependency between instructions execution time and their alignment modulo 16. We attributed these differences to the CPU frontend and its fetch and pre-decode module. We leveraged these time 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 it contains at least a memory write. It can attack perfectly balanced branches, even when they fit within one cacheline. We showed that the Frontal attack achieves a success rate of more than 99%, depending on the target victim code. We tested every modern CPU microarchitecture that currently supports SGX (up to 10th gen) and found them all to be vulnerable to our attack. We demonstrated the practicality of our attack by exploiting two commonly used cryptographic libraries, mbedTLS, and the Intel IPP Cryptography library. We discussed relevant defenses to the attack, such as aligning all branch targets to the same offset modulo 16. While we show that this defense has tiny size and performance overheads, we stress that, in general, secret-depending branching should be avoided to guarantee confidentiality in SGX enclaves.

Availability

A proof of concept of the attack is available online at https://github.com/dn0sar/frontal_poc.

Acknowledgements

We would like to thank Kaveh Razavi for insightful discussions about the root causes of the Frontal attack and Kari Kostiaainen for his feedback on early drafts of this paper. We thank our shepherd Yuval Yarom and the anonymous USENIX reviewers for their valuable suggestions.

References

- [1] Cisco Systems, Inc. *Cisco Annual Internet Report (2018–2023)*. <https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-cl1-741490.pdf>. Accessed: May 2020.
- [2] V. Costan and S. Devadas. *Intel SGX Explained*. Cryptology ePrint Archive, Report 2016/086.
- [3] S. Pinto and N. Santos. “Demystifying Arm TrustZone: A Comprehensive Survey”. *ACM Computing Surveys* (2019).
- [4] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song. “Keystone: An Open Framework for Architecting Trusted Execution Environments”. *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys ’20)*.
- [5] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. “Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base”. *22nd USENIX Security Symposium (USENIX Security ’13)*.
- [6] V. Costan, I. Lebedev, and S. Devadas. “Sanctum: Minimal Hardware Extensions for Strong Software Isolation”. *25th USENIX Security Symposium (USENIX Security ’16)*.
- [7] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. “Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software”. *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP ’17)*.
- [8] Advanced Micro Devices Inc. *AMD Secure Encrypted Virtualization (SEV)*. <https://developer.amd.com/sev/>. Accessed: January 2020.
- [9] F. Brasser, U. Müller, A. Dmitrienko, K. Kostianen, S. Capkun, and A.-R. Sadeghi. “Software Grand Exposure: SGX Cache Attacks Are Practical”. *11th USENIX Workshop on Offensive Technologies (WOOT ’17)*.
- [10] A. Moghimi, G. Irazoqui, and T. Eisenbarth. “CacheZoom: How SGX Amplifies the Power of Cache Attacks”. *Cryptographic Hardware and Embedded Systems – CHES 2017*.
- [11] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. “Cache Attacks on Intel SGX”. *Proceedings of the 10th European Workshop on Systems Security (EuroSec ’17)*.
- [12] Y. Xu, W. Cui, and M. Peinado. “Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems”. *2015 IEEE Symposium on Security and Privacy*.
- [13] S. P. Johnson. *Intel SGX and Side-Channels*. <https://software.intel.com/content/www/us/en/develop/articles/intel-sgx-and-side-channels.html>. Accessed May 2020.
- [14] 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”. *26th USENIX Security Symposium (USENIX Security ’17)*.
- [15] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. “T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs”. *Proceedings 2017 Network and Distributed System Security Symposium (NDSS ’17)*.
- [16] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bind-schaedler, H. Tang, and C. A. Gunter. “Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX”. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS ’17)*.
- [17] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. “Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution”. *26th USENIX Security Symposium (USENIX Security ’17)*.
- [18] Intel Corporation. *Protection from Side-Channel Attacks*. <https://software.intel.com/content/www/us/en/develop/documentation/sgx-developer-guide/top/protection-from-sidechannel-attacks.html>. Accessed May 2020.
- [19] D. Evtushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev. “BranchScope: A New Side-Channel Attack on Directional Branch Predictor”. *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’18)*.
- [20] 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* (2019).
- [21] J. Van Bulck, F. Piessens, and R. Strackx. “Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic”. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS ’18)*.
- [22] S. Hosseinzadeh, H. Liljestrand, V. Leppänen, and A. Paverd. “Mitigating Branch-Shadowing Attacks on Intel SGX Using Control Flow Randomization”. *Proceedings of the 3rd Workshop on System Software for Trusted Execution (SysTEX ’18)*.
- [23] D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar. “CopyCat: Controlled Instruction-Level Attacks on Enclaves”. *29th USENIX Security Symposium (USENIX Security ’20)*.
- [24] A. Limited. *mbedTLS (formerly known as PolarSSL)*. <https://tls.mbed.org/>. Accessed March 2020.
- [25] Intel Corporation. *Cryptography for Intel Integrated Performance Primitives Developer Reference*. <https://software.intel.com/content/www/us/en/develop/documentation/ipp-crypto-reference/top.html>. Accessed October 2020.
- [26] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter. “Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors”. *2009 30th IEEE Symposium on Security and Privacy*.

- [27] 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.
- [28] 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". *2019 IEEE Symposium on Security and Privacy*.
- [29] J. Van Bulck, F. Piessens, and R. Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". *Proceedings of the 2nd Workshop on System Software for Trusted Execution (SysTEX'17)*.
- [30] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer Manuals*.
- [31] A. Fog. *The microarchitecture of Intel, AMD and VIA CPUs*. <https://www.agner.org/optimize/microarchitecture.pdf>. Accessed: January 2020.
- [32] 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". *27th USENIX Security Symposium (USENIX Security '18)*.
- [33] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar. "MicroWalk: A Framework for Finding Side Channels in Binaries". *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*.
- [34] C. D. Walter and S. Thompson. "Distinguishing Exponent Digits by Observing Modular Subtractions". *Topics in Cryptology – CT-RSA 2001*.
- [35] A. Cabrera Aldaya and B. B. Brumley. "When one vulnerable primitive turns viral: Novel single-trace attacks on ECDSA and RSA". *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2020).
- [36] O. Aciçmez, S. Gueron, and J.-P. Seifert. "New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures". *Cryptography and Coding 2007*.
- [37] A. Cabrera Aldaya, A. J. C. Sarmiento, and S. Sánchez-Solano. "SPA vulnerabilities of the binary extended Euclidean algorithm". *Journal of Cryptographic Engineering* (2017).
- [38] B. Krzanich. *Advancing Security at the Silicon Level*. <https://newsroom.intel.com/editorials/advancing-security-silicon-level/#gs.yh984y>. Accessed March 2020.
- [39] 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* (2003).
- [40] A. Rane, C. Lin, and M. Tiwari. "Raccoon: Closing Digital Side-Channels through Obfuscated Execution". *24th USENIX Security Symposium (USENIX Security '15)*.
- [41] M. Hähnel, W. Cui, and M. Peinado. "High-Resolution Side Channels for Untrusted Operating Systems". *2017 USENIX Annual Technical Conference (USENIX ATC '17)*.
- [42] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks". *27th USENIX Security Symposium (USENIX Security '18)*.
- [43] Y. Yarom, D. Genkin, and N. Heninger. "CacheBleed: a timing attack on OpenSSL constant-time RSA". *Journal of Cryptographic Engineering* (2017).
- [44] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar. "MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations". *International Journal of Parallel Programming* (2019).
- [45] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García, and N. Taveri. "Port Contention for Fun and Profit". *2019 IEEE Symposium on Security and Privacy*.
- [46] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher. "MicroScope: Enabling Microarchitectural Replay Attacks". *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*.
- [47] J. Gyselink, J. Van Bulck, F. Piessens, and R. Strackx. "Off-Limits: Abusing Legacy x86 Memory Segmentation to Spy on Enclaved Execution". *Engineering Secure Software and Systems* (2018).
- [48] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. "Preventing Page Faults from Telling Your Secrets". *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16)*.
- [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". *Proceedings 2017 Network and Distributed System Security Symposium (NDSS '17)*.
- [50] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. "KASLR is Dead: Long Live KASLR". *Engineering Secure Software and Systems* (2017).
- [51] Q. Ge, Y. Yarom, D. Cock, and G. Heiser. "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware". *Journal of Cryptographic Engineering* (2018).
- [52] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen. "Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX". *26th USENIX Security Symposium (USENIX Security '17)*.
- [53] R. Guanciale, H. Nemati, C. Baumann, and M. Dam. "Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures". *2016 IEEE Symposium on Security and Privacy*.
- [54] O. Aciçmez. "Yet Another MicroArchitectural Attack: Exploiting I-Cache". *Proceedings of the 2007 ACM Workshop on Computer Security Architecture (CSAW '07)*.
- [55] D. A. Osvik, A. Shamir, and E. Tromer. "Cache Attacks and Countermeasures: The Case of AES". *Topics in Cryptology – CT-RSA 2006*.
- [56] O. Aciçmez and W. Schindler. "A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and Its Demonstration on OpenSSL". *Topics in Cryptology – CT-RSA 2008*.

- [57] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. “Cross-VM Side Channels and Their Use to Extract Private Keys”. *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*.
- [58] E. Tromer, D. A. Osvik, and A. Shamir. “Efficient Cache Attacks on AES, and Countermeasures”. *Journal of Cryptology* (2010).
- [59] Y. Yarom and K. Falkner. “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack”. *23rd USENIX Security Symposium (USENIX Security '14)*.
- [60] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. “Flush+Flush: A Fast and Stealthy Cache Attack”. *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '16)*.
- [61] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. “Malware Guard Extension: Using SGX to Conceal Cache Attacks”. *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '17)*.
- [62] P. C. Kocher. “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems”. *Advances in Cryptology — CRYPTO '96*.
- [63] C. Percival. *Cache missing for fun and profit*. <http://css.csail.mit.edu/6.858/2014/readings/ht-cache.pdf>. 2005.
- [64] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. “GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation”. *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*.
- [65] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. “The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks”. *Information Security and Cryptology - ICISC 2005*.
- [66] C. Liu, M. Hicks, and E. Shi. “Memory Trace Oblivious Program Execution”. *2013 IEEE 26th Computer Security Foundations Symposium*.

A Supplemental information

A.1 Responsible disclosure

We notified the Intel PSIRT on February 21 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 22 that their best practices [27] already suggest avoiding secret-dependent branching, and therefore our attack is 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 5.1 are not used for secret-dependent operations in the SGX architectural enclaves and hence do not pose any security implication. The vulnerability shown in Listing 1 was reported to the mbedTLS team, which promptly fixed it. The vulnerability was also described in a 2017 paper [14] and was still unknown to the developers.

A.2 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 confidential information. Data dependencies manifest when latency or resources utilized depend on the input data. For example, when memory accesses at different addresses are performed based on some secret. Countless attacks have exploited these types of dependencies in the past [62, 63, 59], 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 in 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 [40, 64, 65, 66, 26]. One of the most complete constant-time transformation for SGX is Raccoon [40]. 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 SGX’s memory protections to ensure confidentiality against an attacker that can otherwise read arbitrary locations of memory.

A.3 Measurement Details

We made several changes from stock SGX-Step [29], primarily aiming to reduce measurement noise as much as possible. In terms of functionality, we added the possibility to measure performance counters alongside instructions’ timings. We identified four major sources of noise: the OS, variability in the APIC timer, unpredictability of shared resource state, and enclave creation offset noise. We discuss how we addressed each of these in the following paragraphs.

The OS is a source of noise as it needs to run the scheduler on each core to decide which tasks to execute. If the scheduler runs in between the start of a measurement and its

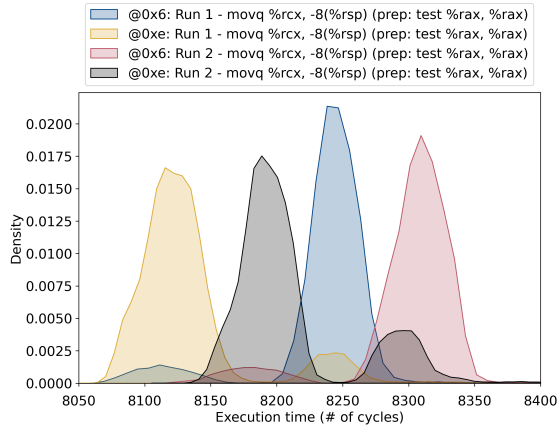


Figure 10: Distribution of the instructions of Figure 6 across different runs, split by their alignments. This figure highlights how different enclave runs exhibit a shift in the mean of their instructions’ distribution and hence distributions are not directly comparable between runs.

end, the measurement will inevitably be longer. Moreover, running any OS function while we single-step can sometimes evict part of the enclave memory from the cache, thus forcing the enclave to fetch it again when it is resumed. This also happens when the scheduler executes on the sibling core. As recommended in the original SGX-Step framework, we run the code in its own isolated core to reduce this noise. However, this alone stops neither the scheduler nor the other cores from interrupting the isolated core. We observed that disabling watchdogs at boot and disabling the graphical user interface tends to reduce the noise produced by the OS, albeit it does not eliminate it completely.

In stock SGX-Step, the APIC timer is set in the `aep_cb_func`. The `aep_trampoline` then executes and resumes the enclave. Various conditions can create variability between the time in which the APIC timer is set and the time at which the enclave resumes. For instance, sometimes, the `aep_trampoline` code page or some of the data it uses might not be present in the cache. We addressed this variability by setting the APIC timer from the `aep_trampoline` function with a value passed from the `aep_cb_func` function and by serializing the instruction stream (using `CPUID`) just before setting the timer. Interestingly, while debugging for this source of noise, we observed that we were never able to interrupt in between fused macro-instructions as these seem to be treated atomically by the CPU, as also observed in [23].

The third source of noise stems from the difference in the microarchitectural state in-between measurements. While we could not completely eliminate this source of noise as we have no direct view of the microarchitecture, we were able to reduce it significantly. The most effective change in this regard was obtained by linearizing the code of the `aep_cb_func` so that there is no mis-speculation in

between single-steps and the function always has the same cache footprint. Even with this change, we observed that instructions that cross a virtual page boundary remained noisy. To account for this, we remove these measurements from the trace when possible. Note that the attacker can easily tell if an instruction crosses the page boundary as the access bit of the new page is set by the CPU.

Finally, while validating these changes, we noticed a source of noise across enclave creations, whose effects we illustrate in Figure 10. The figure shows the measurement of the `movs` from Figure 6 across enclave creations. As can be seen, the distributions remain bimodal, but the position of the modes across creations changes. However, the relative position between the modes stays the same: the `mov` at alignment `0x6` is slower than that at `0xe` on both runs. While we never observed modes shifting more than 200 cycles, this shift is still large enough such that, for instance, the distribution of `nop` instructions could overlap with the distribution of multiplication instructions from different runs. Given this shifting between enclave creations, we concluded that instructions’ timings *are only comparable within the same enclave*.

```

loop_start:
    mov    (%rcx,%rdx,8),%rax
    xor    %r9d,%r9d
    cmp    %rsi,%rax
    setb   %r9b
    sub    %rsi,%rax
    mov    %rax,(%rcx,%rdx,8)
    mov    (%rdi,%rdx,8),%r8
    mov    %r9,%rsi
    cmp    %r8,%rax
    adc    $0x0,%rsi
    sub    %r8,%rax
    mov    %rax,(%rcx,%rdx,8)
    add    $0x1,%rdx
    cmp    %rdx,%rbp
    jne    loop_start

```

Listing 2: Exploited for loop in the `mbedtls` library’s `mpi_montmul` function (compiled on `gcc 7.5.0` with `-O3`).

```

loop_start:
    mov    (%rax),%rcx
    sub    $0x8,%rax
    mov    %rcx,%rdx
    shl   $0x3f,%rcx
    shr   $0x1,%rdx
    or    %rdi,%rdx
    mov    %rcx,%rdi
    mov    %rdx,(%rax+8)
    cmp    %rax,%rsi
    jnz   loop_start

```

Listing 3: Exploited for loop in the `mbedtls` library’s `rsa_gen_key` function of the `mbedtls` library.