Branch Different - Spectre Attacks on Apple Silicon

Lorenz Hetterich and Michael Schwarz

CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

Abstract. Since the disclosure of Spectre, extensive research has been conducted on both new attacks, attack variants, and mitigations. However, most research focuses on x86 CPUs, with only very few insights on ARM CPUs, despite their huge market share. In this paper, we focus on the ARMv8-based Apple CPUs and demonstrate a reliable Spectre attack. For this, we solve several challenges specific to Apple CPUs and their operating system. We systematically evaluate alternative high-resolution timing primitives, as timers used for microarchitectural attacks on other ARM CPUs are unavailable. As cache-maintenance instructions are ineffective, we demonstrate a reliable eviction-set generation from an unprivileged application. Based on these building blocks, we demonstrate a fast Evict+Reload cross-core covert channel, and a Spectre-PHT attack leaking more than 1500 B/s on an iPhone. Without mitigations for all Spectre variants and the rising market share of ARM CPUs, more research on ARM CPUs is required.

1 Introduction

With the discovery of Spectre [16] and Meltdown [20], a new class of so-called transient-execution attacks has been introduced [7]. Follow-up works discovered several such attacks classified into Spectre-type and Meltdown-type [7] attacks. Spectre-type attacks exploit speculative execution, a performance optimization found in most CPUs [16]. Meltdown-type attacks exploit vulnerabilities in the exception handling during out-of-order execution [20]. In both cases, transiently-executed instructions, i.e., instructions without architectural effect, have temporary access to data inaccessible in the architectural program flow.

While these transient executions are not visible on an architectural level, they may leave microarchitectural traces such as a modified cache state. Hence, to make the transiently-accessed data visible, these attacks rely on microarchitectural side channels to convert the microarchitectural state to an architectural state. Most implementations rely on the cache as a microarchitectural element to encode the leaked data, and on cache side channels to retrieve them [20, 16, 7]. Cache side channels are well researched and robust [38]. Even before transient-execution attacks, cache side channels have been used to attack implementations of cryptographic algorithms [27, 38, 21, 22], covertly transmit data in the cloud [37, 25], or spy on user behavior [11, 19].

Although both cache side channels and transient-execution attacks have been researched for multiple years, the main focus is still on x86 CPUs. While x86 undoubtedly plays a big role in computers and servers, ARM chips become more and more popular in PC chips. By the end of 2021, ARM has an estimated market share in the PC chip market of 8 %, mostly due to Apple's M1 chip [32]. Moreover, x86 does not play a large role in mobile devices such as smartphones. These devices are typically powered by ARM CPUs. In this market, Apple CPUs are leading with a market share of 23.4 % in 2022.[1] Still, there is next to no research on these CPUs designed by Apple.

In this paper, we analyze Apple ARMv8 CPUs for their susceptibility to Spectre and cache attacks. We rely on 3 different CPUs designed by Apple, the A10 and A11 used in iPhones, and the M1 used in Macbooks and the Mac mini. Our evaluation runs on the stock operating system, i.e., iOS and macOS. In contrast to previous cache attacks on the A10 [12], we do not require a bootrom exploit or a jailbreak. We show that cache attacks are possible from unprivileged applications.

While cache attacks have been shown on other ARMv8 CPUs [19], we face several new challenges on Apple CPUs. Compared to other ARMv8 CPUs, the ISA is limited. For example, the timestamp counter cannot be used in user space, and the flush instructions cannot be used to flush arbitrary addresses. Both are essential building blocks that have been used for side-channel attacks on ARM [19]. Moreover, there is no documentation on the CPU, experiments on the A10 and A11 are tedious as they have to be executed on smartphones, and the operating system is more limited in terms of low-level functionality than, e.g., Linux or Android.

To enable microarchitectural attacks, we investigate different timing primitives on Apple CPUs. For this, we evaluate system registers available to unprivileged users and analyze system libraries. We demonstrate that there are stable timers that can be used by unprivileged users. However, while they can measure microarchitectural states, their resolution is insufficient for efficient attacks. Thus, we rely on a handcrafted counting thread, similarly as previously shown on x86 CPUs [30], achieving a nano-second resolution.

In addition to the timing primitive, we analyze cache-maintenance instructions on Apple CPUs. We show that, although available, the cache-flush instructions cannot be used in attack settings. Except for the low-power mode of the A10, the flush instructions silently fail, leaving the target data cached. Hence, we show how state-of-the-art eviction-set-generation algorithms can be modified to work on Apple CPUs. Our eviction sets can be generated in less than a second and do not require knowledge of physical addresses. We demonstrate the efficacy of our eviction sets by building a cross-core Evict+Reload covert channel transmitting 3 kB/s with error rates below 13 %.

By combining our building blocks, we show a Spectre-PHT and Spectre-BTB implementation on two iPhones and an M1 Mac mini. Our Spectre attack leaks

[1] `https://www.statista.com/statistics/216459/global-market-share-of-apple-iphone/`

up to 1500 B/s, which is in the same range as the fastest proof of concepts on x86 [31]. Our proof-of-concept implementations also demonstrate that efficient Spectre attacks can be mounted without requiring a detailed understanding of the branch-prediction structures. We show that memory barriers, which are the recommended mitigation by ARM [4], do not behave the same on all tested CPUs, impeding the efficient mitigation of Spectre gadgets. Moreover, there are no hardware or software mitigations against our same-address-space, in-place Spectre-BTB implementation.

We release our entire code as open source:
`https://github.com/cispa/BranchDifferent`.

Contributions. The main contributions of this work are:

1. We systematically evaluate unprivileged timing primitives on Apple CPUs.
2. We study cache maintenance primitives and provide fast and effective eviction-set generation from unprivileged code.
3. We demonstrate that unprivileged cache side-channel attacks on iOS are feasible by implementing a cross-core Evict+Reload covert channel.
4. We show that Spectre-type attacks are possible on Apple CPUs with a proof-of-concept attack leaking up to 1500 B/s.

Structure. The paper is organized as follows. Section 2 covers the required background knowledge. Section 3 discusses the building blocks on Apple CPUs. Section 4 demonstrates a cross-core Evict+Reload cache covert channel on Apple devices. Section 5 describes our Spectre proof of concept. Section 6 evaluates proposed timers, cache maintenance methods, and the Spectre proof of concept. Section 7 discusses mitigations on Apple CPUs. Section 8 concludes the paper.

2 Background and Related Work

Caches Modern CPUs feature a hierarchy of set-associative caches with N levels. Each cache level has S cache sets each consisting of W ways. Every way stores a single cache line of a fixed size. When a memory location is cached, the cache set is determined by the address, while the cache way is decided by a cache-replacement strategy. The cache line is additionally tagged with the physical address of the memory location. On a cache lookup, all ways of the target set are checked by comparing the stored tag to the address. If the tag matches, the cache line can be used and the memory access is done. Otherwise, the next level in the hierarchy is checked. If a matching cache line is found (cache hit) no access to main memory is performed, otherwise (cache miss) the value has to be fetched from main memory. On multi-core CPUs, the hierarchy usually features at least one level of small but fast caches that are private to each core and a level with a bigger but slower shared last-level cache (LLC). A cache level is inclusive if its content is a subset of the next-higher cache level. In inclusive cache hierarchies, cache lines not present in the LLC are not present in any cache.

Cache Attacks Cache attacks exploit the timing differences between accessing memory that is cached, and memory that is not cached. The best-known cache attack is Flush+Reload [38], which relies on read-only shared memory between attacker and victim. On x86 CPUs, Flush+Reload uses the unprivileged `clflush` instruction to remove the targeted data from all cache hierarchies. If the victim accesses the target data, it is again cached. The attacker can measure the access time to the target data, based on that time infer whether the victim accessed the data. This suffices to recover cryptographic keys [38, 11] or spy on users [19].

To measure the cache state, an accurate timer is used to time memory accesses and distinguish cache hits from cache misses. x86 provides the unprivileged `rdtsc` instruction to obtain a high-resolution timestamp. If this instruction is unavailable, e.g., as an attacker runs in a restricted environment, a counting thread has been shown as an alternative timing primitive [19, 30, 28, 9]. Depending on the environment and the implementation, the resolution of such a counting thread is in the same range as the native timestamp counter [28].

On x86, an unprivileged user can remove any accessible memory location from the cache hierarchy using `clflush`. In contrast to x86, the ARM instruction set does not necessarily provide an unprivileged flush instruction. A slower alternative to flushing is eviction. By accessing a set of addresses mapping to the same cache set, a so-called eviction set, the target cache line is evicted from the cache. Depending on the replacement strategy implemented in the processor, the addresses are accessed multiple times in special patterns to achieve good eviction rates [19]. Using eviction instead of flushing, Flush+Reload can be modified to Evict+Reload. Moreover, a vendor can also decide to prevent unprivileged access to the timestamp counter. As a result, cache attacks on ARM devices are more challenging [19, 8]. Green et al. [10] also showed that cache attacks on some ARM devices are harder than anticipated due to Autolock, a performance optimization locking cache lines in the LLC if they are present in a core-private cache. Lipp et al. [19] demonstrate cache attacks on some ARM devices using system calls to access otherwise privileged performance counters. To maintain the cache state, they rely on flushing if available and on eviction otherwise. As Android exposed virtual to physical address mappings at the time of their research, finding eviction sets was straightforward as the physical address determines the cache set. Haas et al. [12] also demonstrate cache side-channel attacks on an Apple A10 CPU. However, they rely on privileged code to do so.

Transient Execution To improve the performance, modern CPUs rely on out-of-order and speculative execution. These performance optimizations allow executing instructions in a different order than specified in the application to reduce pipeline stalls. However, to ensure correctness, instructions retire in application order, i.e., architecturally, it seems that the instructions are executed in the order specified in the application. The umbrella term for out-of-order and speculative execution is *transient execution*, and instructions executed during transient execution are called *transient instructions* [20, 16, 7]. Transient instructions that are wrongly executed, e.g., due to a previous misprediction of the control flow, are discarded. Similarly, exceptions during transient execution are not raised

architecturally but only result in a pipeline flush. However, microarchitectural state changes, e.g., cache states, are not reverted. Transient-execution attacks exploit these microarchitectural traces to leak data that is not accessible during normal architectural program execution. Transient-execution attacks are categorized into Spectre-type attacks, which exploit control- or data-flow mispredictions [16], and Meltdown-type attacks, which exploit vulnerabilities in delayed exception handling during out-of-order execution [20].

Spectre Spectre [16] is a transient-execution attack that exploits speculative execution. To avoid pipeline stalls on, e.g., branches, CPUs try to predict the outcome of branches based on previous observations. For correct predictions, the CPU successfully avoids stalling, resulting in an improved performance. However, for wrong predictions, the CPU executes a code path that would not be executed during architectural execution. Such a code path can, e.g., be an out-of-bounds access of a data structure. A *Spectre gadget* is a special piece of code that encodes such illegitimately accessed data into a microarchitectural state. An attacker relies on side channels to bring this microarchitectural state to the architectural state, ultimately leaking the data. Even though several techniques for encoding the data exist [5, 29, 36, 18], most Spectre attacks rely on cache covert channels. Spectre attacks are classified by the target predictor [7]. The variants that received the most attention are Spectre-PHT and Spectre-BTB. Spectre-PHT (also known as Spectre Variant 1) exploits the pattern history table used for predicting whether a conditional branch is taken or not [7]. Spectre-BTB exploits the branch-target buffer predicting the target of indirect branches.

3 Building Blocks

In this section, we introduce the building blocks required for Spectre attacks on Apple CPUs. We focus on Spectre-PHT with a cache-based covert channel. The reason is that Spectre-PHT is widespread [15], not mitigated in hardware [4], and not mitigated via automated software workarounds [34]. However, these building blocks can also be used for different Spectre variants, as we describe in Section 7. The main building blocks required are as follows.

1. **Accurate timing**: To distinguish cache hits from cache misses, we require a high-resolution timing source. Previous work [19, 12] often relies on platform-specific instructions or APIs that are not available to unprivileged users on Apple CPUs running iOS or macOS.
2. **Cache maintenance**: To continuously probe a cache line, we need an efficient way to remove certain cache lines from the cache. Previous work [19] typically relies on the flush instructions which does not work in unprivileged code on Apple CPUs. Also, the mapping of virtual to physical addresses [19, 12] is not available, preventing the direct calculation of eviction sets.
3. **Speculative execution and mistraining**: For a successful Spectre attack, we must mistrain a predictor and obtain a long enough transient-execution window to leak information with a Spectre gadget.

Table 1: Devices used for testing.

	iPhone 7	**iPhone 8 Plus**	**M1 Mac mini**
CPU	Apple A10 Fusion	Apple A11 Bionic	Apple M1
OS version	iOS 14.3	iOS 14.2	macOS 11.2.1

In the remaining paper, we use an iPhone 7, iPhone 8 Plus, and M1 Mac mini as listed in Table 1. All these devices feature an ARM-based CPU designed by Apple and run the stock Apple operating system, i.e., iOS on the iPhones and macOS on the M1. All building blocks are evaluated on all of the devices.

3.1 High-resolution Timing

Distinguishing cache hits from cache misses by timing a memory access requires a high-resolution timing source with a resolution of several nanoseconds. Based on a systematic analysis of available timers, we identify and evaluate three possible timing sources discussed in more detail in this section. First, we investigate *system control registers* provided by the CPU. Amongst other functionality, these registers provide different timing sources. Second, we analyze *library functions* provided by the operating system. Such functions sometimes rely on undocumented syscalls or instructions that can be used for precise timing measurement. Third, we implement a *dedicated counting thread* to emulate a high-resolution timer. This approach has also been used in restricted environments for mounting microarchitectural attacks [30, 9].

System Control Registers On ARMv8, system control registers can be read using the `mrs` instruction. While some registers can only be read by privileged users, others are accessible by unprivileged users. We identify two promising registers: the system counter registers (`CNTPCT_ELx`, `CNTVCT_ELx`) [3] as well as performance counters (`PMCCNTR_ELx`) [2]. However, reading the performance counters as unprivileged user results in an illegal-instruction exception on all tested devices. This is in contrast to previous work [19], which used this performance counter via a syscall to mount cache attacks on Android-based ARM devices. We could not find similar unprivileged system calls on Apple devices, hence we cannot use this known high-resolution counter. As unprivileged user, only `CNTPCT_EL0` and `CNTVCT_EL0` are accessible. Apart from a possible fixed offset, `CNTVCT_EL0` is the same counter as `CNTPCT_EL0` [3]. Our evaluation of the counter resolution (cf. Section 6) shows that while the counter is stable, its resolution is not sufficient to reliably distinguish cache hits from misses. However, it might still be useful for attacks that distinguish larger timing differences, or when combining it with amplification methods [31].

Library Functions Library functions provide another source of accurate timing. They may use undocumented system calls or instructions to access a high-

resolution timer. We analyze system libraries on iOS and macOS for timing-related functions. According to Singh [33], `mach_absolute_time` is the function with the highest resolution on macOS. We also identify `clock_get_time` as an alternative to `clock_gettime` used in previous research. However, further analysis shows that both functions internally use the `CNTVCT_EL0` system register (cf. Figure 7 in Appendix B). Hence, these functions do not provide a higher resolution than directly accessing the system register. Consequently, they are not accurate enough to be used as building blocks.

The `clock_gettime` syscall used in previous work [19] is only available on macOS and not on iOS. In contrast to Android, macOS only provides a microsecond resolution instead of a nanosecond resolution. Although the `CLOCK_MONOTONIC_RAW` should provide nanosecond resolution, it again falls back to `CNTVCT_EL0`.

Counting Thread As an alternative to using an existing timer, we create our own timer by incrementing a shared variable in a background thread. Previous work also used such counting threads for microarchitectural attacks [19, 30]. As in previous work [30], we handcraft the counting thread in Assembly to ensure that we achieve the highest-possible update frequency. Using the counting thread, we can distinguish cache hits from cache misses reliably, as evaluated in Section 6.

3.2 Cache Maintenance

To remove cache lines from the cache, several flush instructions are available on the ARMv8 architecture. They differ in whether they flush or just invalidate, target instructions (`IC`) or data (`DC`) and which levels of caches are flushed. Also, some determine the cache line to flush by virtual address, while others flush by cache set and way. According to the ARM manual [1], they may or may not be available to unprivileged users. On ARM CPUs where these instructions are available, Lipp et al. [19] demonstrate that they can be used for cache attacks. Specifically, they rely on `DC CIVAC`. This instruction flushes data by virtual address from all CPU cache levels, similar to the `clflush` instruction on x86 CPUs. While this instruction is available to unprivileged users on Apple CPUs, it does not flush the target address. Similarly, none of the other flush instructions raises an illegal-instruction exception. However, they also fail to flush the targeted cache line. In Section 6, we evaluate the effects of the instructions, showing that they are not ignored but also do not work as expected.

As an alternative to flushing, eviction can remove cache lines from the cache by accessing multiple addresses mapping to the cache set of the target cache line. With access to physical addresses, generating an eviction set is straightforward, as parts of the physical address determine the cache set. However, there is no unprivileged way to read the mapping from virtual to physical addresses on iOS or macOS. Thus, we cannot calculate eviction sets as shown previously by Lipp et al. [19]. Lipp et al. [19] relied on the `/proc/self/pagemap` file, which exposed this information on older Android versions. This Linux-specific (pseudo) file is not available on iOS or macOS.

```
uint64_t cachemiss(char* page){
  /* page is a fresh page */
  memory_access(page + rand()
    % (PAGE / 2));
  return probe(
    page + PAGE / 2 +
    rand() % (PAGE / 2));
}
```

```
uint64_t cachehit(char* page){
  memory_access(page);
  memory_access(page);
  return probe(page);
}
```

Listing 1: Code to produce cache hits and misses without flushing or eviction.

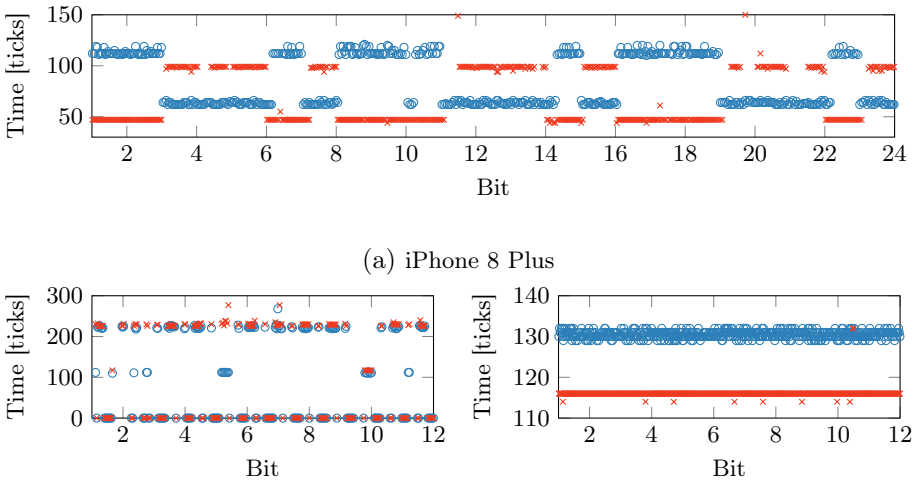
Our eviction is based on the fast eviction-set generation using group testing introduced by Vila et al. [35]. This approach starts from a large set of addresses that likely form an eviction set and reduces it until a minimal eviction set is reached. The original implementation is only available for x86 and does not work on ARM CPUs. The main reason is the usage of low-level x86 functions, e.g., `clflush` and `rdtsc`, which are unavailable on ARM. Moreover, it only supports the eviction strategies of Intel CPUs. However, these strategies are not efficient on ARM due to the different cache-replacement strategy [19, 12].

As the eviction-set generation relies on timing to distinguish cache hits from misses, it calibrates a threshold initially. Without a flush instruction available, we would have to resort to eviction for the calibration, resulting in a chicken-and-egg problem. We can easily measure cache hits by accessing the same address multiple times. Accesses after the first access are most likely L1 cache hits. To reliably measure cache misses without flushing or eviction, we rely on the property that a newly-allocated page is not cached. To ensure that the page is physically backed and the translation is cached in the TLB, we access one cache line of the page. Accessing any of the remaining cache lines results in a cache miss. As generating multiple cache misses on a page can trigger the hardware prefetcher [13], we only measure one other cache line on the page before freeing the buffer again. Listing 1 shows the code for measuring cache hits and misses without relying on flushing or eviction. For measuring the time, we rely on a counting thread, as it provides the highest resolution (cf. Section 6.1).

4 Fast Covert Channel

In this section, we rely on our building blocks to implement a fast cross-core Evict+Reload covert channel. As our Spectre PoC attack uses Evict+Reload to transfer information from the microarchitectural to the architectural state, this covert channel provides an approximate upper bound for the leakage rate in a cross-core scenario. We evaluate the covert channel on the devices listed in Table 1.

Setup Sender and receiver are both unprivileged applications running in parallel on different CPU cores. They both map the same read-only shared file into



(a) iPhone 8 Plus

(b) iPhone 7

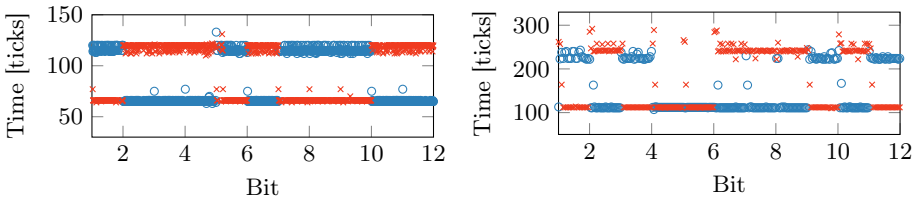
(c) M1 Mac mini

Fig. 1: Covert channel repeatedly transmitting `01110001` with 48 bit/s. Timing on data 1 cache line (red) and data 0 cache line (blue).

their virtual address space. The data transmission uses two cache lines in this shared file to transmit data. Depending on the bit to send, one of the two lines is repeatedly accessed while the other one is not accessed. Relying on two cache lines already provides a simple form of error detection. With this encoding, we can detect if either the sender or receiver is not scheduled, a common cause for errors in covert channels [26]. The receiver alternatingly mounts an Evict+Reload attack on both cache lines.

Evaluation On the iPhone 8 Plus, the covert channel works well with low transfer rates as seen in Figure 1. As the receiver was started at an arbitrary point in the transmission sequence, the received bit sequence is offset (11100010). On the iPhone 7, the covert channel does not work reliably as pictured in Figure 1. The figure shows that many measurements are zero. This is due to the limited amount of available cores: On the iPhone 7, only two cores can be active at the same time and no simultaneous multithreading is available. Since our setup requires three threads in parallel, namely one for sending, one for the timer and one to mount the Evict+Reload attack, two cores do not suffice. As a result, we cannot accurately measure memory access times, as the counting thread is not always scheduled.

On the M1 Mac mini (Figure 1) as well as for increased transfer rates on the iPhone 8 Plus, we observe an interesting phenomenon: We measure cache hits even though the cache line is not accessed by the sender. One reason is that ARM devices can have non-inclusive caches [19, 10] which is also the case for the M1 as



(a) iPhone 8 Plus at 2400 bit/s (b) M1 Mac mini at 24 000 bit/s

Fig. 2: Optimized covert channel repeatedly transmitting `01110001`. Timing on data 1 cache line (red) and data 0 cache line (blue).

reported by Handley [13]. On non-inclusive cache hierarchies, a cache line might be present in a private cache of a different core even if it is not present in the shared LLC. Instead of fetching the value from memory, it can be served from the private cache instead. Additionally, with AutoLock [10], there is a performance optimization on ARM CPUs. AutoLock prevents a cache line present in a core-private cache from being evicted from shared cache levels. While these effects are problematic for cache attacks [10], we can counteract them as we control both sender and receiver. After transmitting a bit by repeatedly accessing one of the cache lines, the sender evicts that cache line from its private cache. This makes sure that the receiver does not measure cache hits caused by cross-core cache lookups or cache lines locked by AutoLock. With this modification, the phenomenon disappears and we can increase the transfer rate (cf. Figure 2).

On the M1 Mac mini, we can increase the transfer rate up to 24 000 bit/s as shown in Figure 2. At this transfer rate, we achieve a bit-error rate of 12.67 % without any error correction in place. This results in a true capacity of 10 840 bit/s, which is slightly faster than the fastest Evict+Reload covert channel on ARM demonstrated by Lipp et al. [19]. On the iPhone 8 Plus, we achieve a transfer rate of 2400 bit/s with an error rate of 7.84 %, resulting in a true capacity of 1448 bit/s. As shown by Maurice et al. [26], an error-free transmission via such cache-based channels is possible by using error correction.

5 Spectre Exploit

In this section, we describe our Spectre-PHT proof-of-concept implementation using a cache side channel to extract the leaked data. The attack runs as unprivileged applications and leaks up to 1500 B/s on the tested devices (Table 1).

Threat Model For our proof of concept, we assume an attacker can reliably trigger the execution of a Spectre gadget in the victim. In line with previous work [29, 6, 31, 23], we rely on a bit-wise leakage gadget. Finding such Spectre gadgets in existing code is orthogonal to our work [16, 17, 24, 29]. Hence, we follow best practices for the evaluation and inject our own gadget into the victim

```
1 void victim(size_t index){
2   int shift = (index % INDICES_PER_BYTE) * BITS;
3   index = index / INDICES_PER_BYTE;
4   if(index < *array_size) {
5     mem_access(array2 + ((array1[index] >> shift) & (VALUES - 1)) *
      ENTRY_SIZE);
6   }
7 }
```

Listing 2: The Spectre gadget of the PoC allows leaking 1, 2, 4, or 8-bit values.

application [16, 17, 24, 29]. Listing 2 shows a generic gadget that we use for the evaluation. This gadget allows selecting the number of bits to leak per invocation. Generally, the gadget can either be in the same process, e.g., in a sandbox environment or a different process. In the case of different processes, we also assume shared memory for the array indexed by the leaked value such that the leakage can be recovered using Evict+Reload.

Attack The attacking code can be divided into four steps:

1. **Mistraining**: To induce a misprediction, we need to mistrain the branch predictor in a way that it predicts the bound check in our Spectre gadget to be true even though the index is out of bounds.
2. **Leaking a value**: With the mistrained branch predictor, we provide an out-of-bound index to access a normally inaccessible value and encode it into the cache. We rely on the same cache-encoding technique as previous work [16].
3. **Retrieving the leaked value**: As the leaked value is only visible in the microarchtiectural state, we rely on a cache covert channel, namely Evict+Reload, to make it visible in the architectural state.
4. **Repeat**: We reset the microarchitectural state by evicting the cache set used for the encoding. With this fresh state, we can repeat the attack for every target bit.

Mistraining and Leaking a Value For the mistraining, we rely on in-place mistraining [7]. We call the gadget with an in-bound value 9 times to bias the prediction to predict that the provided index is always in bounds. We refer to these invocations of the gadget as training calls. In the 10^{th} call to the gadget, we provide an out-of-bounds index to access memory located after the array. The illegitimately accessed data is encoded in the cache by accessing a memory location that depends on the leaked value [16]. The in-place mistraining has two advantages. First, potential hardware countermeasures cannot prevent this mistraining strategy [7]. Second, we do not have to reverse engineer the intricate details of the branch-prediction structures to find colliding virtual addresses. The mistraining happens at exactly the same virtual address as the misprediction, resulting in a portable and stable way of exploiting Spectre-PHT.

```
for(int i = 40; i >= 0; i--) {
  // leak_index every 10 iterations, training_index otherwise
  size_t x = (!(i % 10)) * (leak_index - training_index)
    + training_index; // avoid branches
  cache_remove(array2); // remove from cache
  victim(x); // training (in-bound) or attack (out-of-bounds) call
}
```

Listing 3: Mistraining and out-of-bounds call to leak a value speculatively.

The training calls each encode the values from the target array (`array1`) into the cache state (`array2`). Since we train on a null byte in `array1`, `array2[0]` is always cached. For the leakage, a cache line of `array2` is accessed that depends on the leaked value. To increase the speculative execution window, we evict the size of the array (`array_size`) that is used for the bound check from the cache. Listing 3 shows the loop that calls the Spectre gadget. To deal with the non-perfect misprediction rate, we try to leak every value 4 times.

Retrieving the Leaked Value and Repeat To retrieve the leaked value from the cache state, we probe all entries of `array2` and remember indices where a cache hit was measured. A cache hit at a non-zero index directly reveals the leaked value. A cache hit on index zero does not directly provide information, as this index is always cached due to the training calls. However, if there is no other cache hit, we infer that the leaked value must be '0'. To reset the cache state of `array2`, we require an eviction set for every entry that we probe. These eviction sets are generated once and then used for every repetition.

Section 6 evaluates our proof-of-concept implementation.

6 Evaluation

In this section, we evaluate our implementation. We analyze the building blocks, i.e., the timing sources and cache-maintenance functions (cf. Section 3). We evaluate the Spectre-PHT proof-of-concept implementation, demonstrating leakage rates of up to 1500 B/s. All evaluations are done on three devices (2 iPhones, 1 Mac mini), as shown in Table 1.

6.1 High-Resolution Timing

Accurate timing allows distinguishing cache hits from cache misses and serves as a powerful primitive for cache side channels. Section 3.1 introduces three different methods for accurate timing, based on directly reading system-control registers, via syscalls, and using a counting thread. In this section, we analyze the resolution of these different timing sources. We show that a dedicated counting thread is the most reliable method for distinguishing cache hits from misses on all tested devices.

System Control Registers The system control register of the system timer can be accessed directly via inline assembly (cf. Listing 4 in Appendix A). We do this to benchmark the timer on both iPhones and the M1 Mac mini. The measured frequency for all three devices is approximately 25 MHz (40 ns per increment). This measurement aligns with the 24 MHz (41.67 ns per increment) reported by the system counter frequency register `CNTFRQ_EL0`. This resolution is in the same range as the difference between cache hits and misses. However, based on when the register is read, misclassifications can happen in both directions, i.e., cache hits can be classified as misses and vice versa. While this is tolerable for the Spectre attack, it is not tolerable for the eviction-set generation. In this process, the timing has to reliably distinguish cache hits from misses to ensure that the algorithm converges. Using this system counter as a timer, we are unable to find eviction sets due to these misclassifications. This indicates that the timer is not accurate enough for measuring single events.

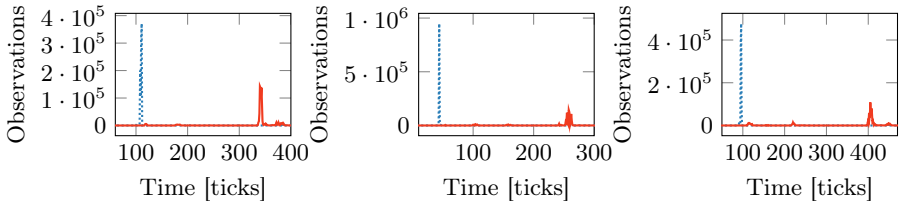
Library Functions Section 3.1 introduces two candidates for accurate timing through library functions: `clock_get_time` and `mach_absolute_time`. However, the analysis of these functions (cf. Figure 7 in Appendix B) shows that internally, they rely on reading the system counter. We also evaluate this empirically, showing the expected update frequency of approximately 25 MHz. Hence, these functions are no improvement over directly accessing the system control register.

Counting Thread We implement the counting thread purely in inline assembly to ensure the highest-possible update frequency (cf. Listing 5 in Appendix A). To measure the resolution, we evaluate the number of increments per second. The counting thread achieves approximately 800 MHz (1.25 ns per increment) on the iPhone 8 Plus, 2.4 GHz (0.42 ns per increment) on the iPhone 7, and 3 GHz (0.33 ns per increment) on the M1 Mac mini. In addition to the update frequency, we evaluate how well the timer is suited for distinguishing cache hits and misses. To evaluate this, we measure individual cache hits and misses to verify that we can distinguish them. For ensuring that an address is a cache hit or miss, we use the method shown in Listing 1 that we also use to calibrate the eviction-set generation. Using this method, we ensure that we do not inadvertently see the effects of imperfect eviction.

Figure 3 shows that we can clearly distinguish most cache hits from misses. However, in fewer than 1 % of the measurements, we measure an access time of zero for both cache hits and misses. This happens if the counting thread is not scheduled and, thus, the counter is not incremented. As zero is never a valid access time, we can mark measurements of zero as invalid instead of cache hits to decrease the number of false positives. We observe less than 0.6 % false positives and negatives on both iPhones, excluding invalid measurements of zero.

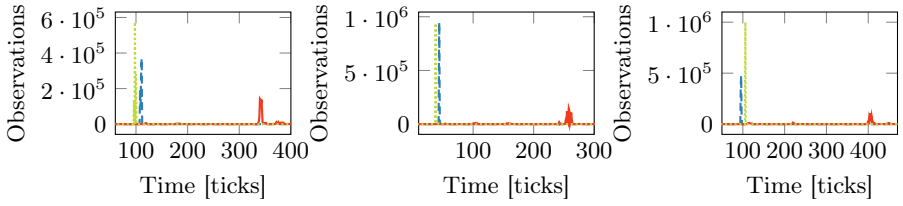
6.2 Cache Maintenance

In this section, we evaluate the native cache-flush instructions and cache eviction on Apple CPUs.



(a) iPhone 7 (b) iPhone 8 Plus (c) M1 Mac mini

Fig. 3: Counting thread cache hit (blue dotted) and miss (red solid) timings.



(a) iPhone 7 (b) iPhone 8 Plus (c) M1 Mac mini

Fig. 4: Cache hit (blue dashed), miss (red solid), and flushed (green dotted) histogram using the counting thread.

Flushing We evaluate the flush instructions by timing 1 000 000 accesses to cached cache lines and flushed cache lines using the counting thread as the timing source. While we expect a timing difference between cache hits and cache misses, the average access times for flushed and non-flushed cache lines are the same, as shown in Figure 4. However, if we only measure 200 accesses on the iPhone 7, we see a timing difference for accessing flushed and cached cache lines (Figure 5). Increasing the run time of the evaluation code by prepending a busy-wait loop reveals that flushing does not work if the execution time of the code exceeds approximately 25 ms. This is likely caused by the processor switching from low-power to high-power cores and flushing only working on low-power cores. Thus, the native flush instructions cannot be used for generic cache attacks on Apple CPUs. Interestingly, this silent-fail behavior is undocumented. The flush instructions can throw an exception, e.g., if the target address is not readable or if the instruction is not enabled for unprivileged users. Based on the documentation, the instruction is successful if no such exception is thrown.

Eviction For the eviction, we use the eviction-set-generation algorithm from Section 3.2. For both iPhones, an eviction set size of size 16 works reliably. For the M1 Mac mini, the set contains 32 addresses. We do not require minimal eviction sets for our Evict+Reload attack as this only affects the leakage rate but not the success of the attack. Thus, trading some performance for greater

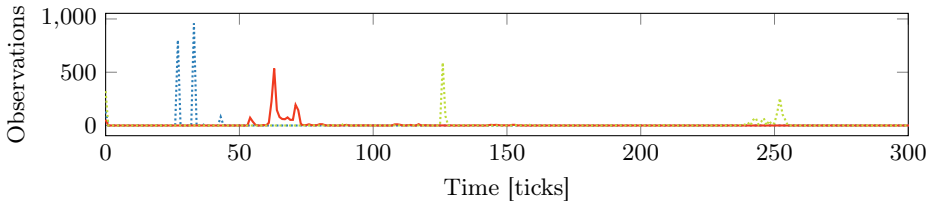
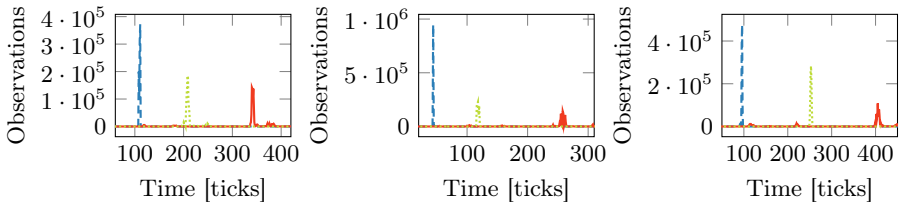


Fig. 5: Cache hit (blue dashed), miss (red solid), and flushed (green dotted) histogram using the counting thread on an iPhone 7 presumably on low-power core.



(a) iPhone 7 (b) iPhone 8 Plus (c) M1 Mac mini

Fig. 6: Cache hit (blue dashed), miss (red solid), evicted (green dotted).

reliability by increasing the number of addresses is reasonable. Figure 6 shows the access times for data in the cache, not in the cache, and evicted. Accessing evicted data is faster than data not in the cache, as it might not have been evicted from all caches. Still, the timing difference is large enough to distinguish evicted data from cached data reliably.

Eviction takes on average 904 ticks on the iPhone 7, 820 ticks on the iPhone 8, and 987 ticks on the M1 Mac mini. For each device, we measure one million evictions and calculate the eviction rate by dividing the number of cache misses by the number of valid measurements. We achieve eviction rates above 98.5 % on all devices, as shown in Table 2.

Table 2: Eviction rates tested with 1 000 000 measurements.

Device	Threshold	Cache Hits	Cache Misses	Invalid	Eviction Rate
iPhone 7	100	14	996 968	3018	99.999 %
iPhone 8 Plus	80	13 868	976 748	9384	98.600 %
M1 Mac mini	140	6	999 223	771	99.999 %

Table 3: Spectre PoC leakage and error rate.

Device	Leakage Rate	Error Rate	Setup Time
iPhone 7	1522 B/s	6.18 %	804 ms
iPhone 8 Plus	1590 B/s	4.89 %	852 ms
M1 Mac mini	1109 B/s	13.66 %	957 ms

6.3 Spectre-PHT Proof of Concept

In this section, we evaluate the Spectre-PHT proof of concept (cf. Section 5). For the evaluation, we use a Spectre gadget leaking 4 bit per invocation on both iPhones and 2 bit per invocation on the M1 Mac mini. Those values resulted in the highest leakage rate in our testing. To evaluate throughput and error rate, we leak 10 kB of data. As the setup time has a significant impact on the total runtime, we measure the setup time additionally. The setup time mainly consists of creating the eviction sets used for the cache encoding. On all devices, the setup time is slightly below 1 s. Note that the setup time is independent of the size of the secret and thus always in the same range.

The results are summarized in Table 3. We measure leakage rates above 1000 B/s on all devices, with up to 1590 B/s on the iPhone 8 Plus. Our leakage rates are in the same range as state-of-the-art Spectre proof-of-concept implementations on x86 [31]. These results not only show that our building blocks are robust but also that Apple CPUs are not inherently more secure against Spectre than Intel or AMD CPUs.

7 Discussion

Other Spectre Variants For our proof of concept, we focus on Spectre-PHT, as it is difficult to mitigate and a widespread issue [19, 15]. However, our building blocks can also be combined for other Spectre variants. We experimentally verify that our proof of concept also works with a Spectre-BTB [16] gadget, which mispredicts the destination of an indirect branch. As our code is open source, it can be used to test a variety of transient-execution attacks on Apple CPUs. Testing other Spectre variants is simply a matter of replacing the victim gadget.

Mitigations To prevent Spectre-PHT attacks, the state-of-the-art mitigation technique on all CPUs is to add memory fences between conditional jumps and subsequent memory accesses [14, 4]. ARM introduced a new barrier instruction `CSDB`. This barrier is also used in higher-level functions provided by ARM[2], such as `load_no_speculate_fail` that load a value from a bounded buffer without speculatively accessing values out of bounds. However, we experimentally verify that on both iPhones, the `CSDB` instruction has no effect on speculative execution. Adding `CSDB` to the Spectre-PHT gadget does not mitigate the exploit. On

[2] `https://github.com/ARM-software/speculation-barrier`

the M1 Mac mini, `CSDB` is available and stops the leakage in our proof of concept. ARM also suggests to use a Data Synchronization Barrier (`DSB`) together with an `ISB`. The `ISB` instruction prevents exploitation on both iPhones and the M1 Mac mini. Data Memory Barriers (`DMB`) or `DSB` alone do not stop speculative execution.

Other Spectre variants, such as Spectre-BTB, can be mitigated in an automated way on x86 using retpolines [34]. However, this workaround does not work on ARM CPUs [4]. Hence, our Spectre-BTB proof of concept can currently not be mitigated on Apple CPUs.

8 Conclusion

We demonstrated a reliable Spectre attack on the ARMv8-based Apple CPUs with leakage rates up to 1500 B/s. Our attack solves several challenges specific to Apple CPUs and their operating system. We showed that no unprivileged high-resolution timer is available but that a counting thread is highly reliable for microarchitectural attacks. We demonstrated a reliable eviction-set-generation implementation to enable cache attacks despite the unavailability of cache-maintenance instructions. The incomplete software workarounds for Spectre variants on these CPUs combined with the rising market share shows that more research on ARM, and especially Apple CPUs, is required.

9 Acknowledgments

We thank the anonymous reviewers for their valuable feedback and suggestions that helped to improve the paper. Furthermore, we thank the Saarbrücken Graduate School of Computer Science for their funding and support of Lorenz Hetterich.

References

1. ARM: ARM Architecture Reference Manual ARMv8. ARM Limited (2013)
2. ARM: Arm coresight performance monitoring unit architecture. [Online]. Available: `https://developer.arm.com/documentation/ihi0091/a-a` (November 2020)
3. ARM: Learn the architecture: Generic timer. [Online]. Available: `https://developer.arm.com/documentation/102379/0000/The-processor-timers` (January 2021)
4. ARM: Whitepaper Cache Speculation Side-channels: (2018), `https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/download-the-whitepaper`
5. Bhattacharyya, A., Sandulescu, A., Neugschwandt ner, M., Sorniotti, A., Falsafi, B., Payer, M., Kurmus, A.: SMoTherSpectre: exploiting speculative execution through port contention. In: CCS (2019)
6. Bhattacharyya, A., Sandulescu, A., Neugschwandtner, M., Sorniotti, A., Falsafi, B., Payer, M., Kurmus, A.: SMoTherSpectre: exploiting speculative execution through port contention. arXiv:1903.01843 (2019)

7. Canella, C., Van Bulck, J., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., Gruss, D.: A Systematic Evaluation of Transient Execution Attacks and Defenses. In: USENIX Security Symposium (2019), extended classification tree and PoCs at https://transient.fail/.
8. Easdon, C., Schwarz, M., Schwarzl, M., Gruss, D.: Rapid Prototyping for Microarchitectural Attacks. In: USENIX Security (2022)
9. Gras, B., Razavi, K.: ASLR on the Line: Practical Cache Attacks on the MMU. In: NDSS (2017)
10. Green, M., Rodrigues-Lima, L., Zankl, A., Irazoqui, G., Heyszl, J., Eisenbarth, T.: AutoLock: Why Cache Attacks on ARM Are Harder Than You Think. In: USENIX Security Symposium (2017)
11. Gruss, D., Spreitzer, R., Mangard, S.: Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security Symposium (2015)
12. Haas, G., Potluri, S., Aysu, A.: itimed: Cache attacks on the apple a10 fusion soc. Cryptology ePrint Archive (2021)
13. Handley, M.: M1 Exploration - v0.70 (2021)
14. Intel: Intel analysis of speculative execution side channels (2018), `https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf`
15. Johannesmeyer, B., Koschel, J., Razavi, K., Bos, H., Giuffrida, C.: Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel. In: NDSS (2022)
16. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre Attacks: Exploiting Speculative Execution. In: S&P (2019)
17. Koruyeh, E.M., Khasawneh, K., Song, C., Abu-Ghazaleh, N.: Spectre Returns! Speculation Attacks using the Return Stack Buffer. In: WOOT (2018)
18. Lipp, M., Gruss, D., Schwarz, M.: AMD Prefetch Attacks through Power and Time. In: USENIX Security (2022)
19. Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., Mangard, S.: ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security Symposium (2016)
20. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading Kernel Memory from User Space. In: USENIX Security Symposium (2018)
21. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-Level Cache Side-Channel Attacks are Practical. In: S&P (2015)
22. Lou, X., Zhang, T., Jiang, J., Zhang, Y.: A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. ACM CSUR (2021)
23. Loughlin, K., Neal, I., Ma, J., Tsai, E., Weisse, O., Narayanasamy, S., Kasikci, B.: DOLMA: Securing Speculation with the Principle of Transient Non-Observability. In: USENIX Security Symposium (2021)
24. Maisuradze, G., Rossow, C.: ret2spec: Speculative Execution Using Return Stack Buffers. In: CCS (2018)
25. Maurice, C., Neumann, C., Heen, O., Francillon, A.: C5: Cross-Cores Cache Covert Channel. In: DIMVA (2015)
26. Maurice, C., Weber, M., Schwarz, M., Giner, L., Gruss, D., Alberto Boano, C., Mangard, S., Römer, K.: Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS (2017)
27. Osvik, D.A., Shamir, A., Tromer, E.: Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA (2006)

28. Schwarz, M., Maurice, C., Gruss, D., Mangard, S.: Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In: FC (2017)
29. Schwarz, M., Schwarzl, M., Lipp, M., Gruss, D.: NetSpectre: Read Arbitrary Memory over Network. In: ESORICS (2019)
30. Schwarz, M., Weiser, S., Gruss, D., Maurice, C., Mangard, S.: Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA (2017)
31. Schwarzl, M., Borrello, P., Kogler, A., Varda, K., Schuster, T., Gruss, D., Schwarz, M.: Dynamic process isolation. arXiv:2110.04751 (2021)
32. Shah, A.: Apple is beginning to undo decades of Intel, x86 dominance in PC market. The Register (2021), `https://www.theregister.com/2021/11/12/apple_arm_m1_intel_x86_market/`
33. Singh, A.: Mac OS X Internals: A Systems Approach: A Systems Approach. Addison-Wesley (2016)
34. Turner, P.: Retpoline: a software construct for preventing branch-target-injection (2018), `https://support.google.com/faqs/answer/7625886`
35. Vila, P., Köpf, B., Morales, J.: Theory and Practice of Finding Eviction Sets. In: S&P (2019)
36. Weber, D., Ibrahim, A., Nemati, H., Schwarz, M., Rossow, C.: Osiris: Automated Discovery of Microarchitectural Side Channels. In: USENIX Security (2021)
37. Wu, Z., Xu, Z., Wang, H.: Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud. ACM Transactions on Networking (2014)
38. Yarom, Y., Falkner, K.: Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium (2014)

A Code

```
uint64_t read_system_counter() {
  uint64_t result;
  asm volatile("MRS %[result], CNTPCT_EL0": [result] "=r" (result));
  return result;
}
```

Listing 4: Reading the system counter control register.

Listing 4 shows the code for directly accessing the system control register of the system timer. Listing 5 shows the code used for the counting thread. This code is executed in a thread running on its own CPU core.

B Library Functions

Figure 7 shows the call graph of library functions providing high-resolution timestamps. All functions internally use the `CNTVCT_EL0` system register.

```
1 static void* counter_thread(void* arg) {
2   asm volatile(
3     "LDR x10, [%[counter]]\n" // load counter once
4     "loop:\n"                 // while(true) {
5     "ADD x10, x10, #1\n"      //     increment counter
6     "STR x10, [%[counter]]\n" //     store counter to memory
7     "B loop\n"                // }
8     :: [counter] "r" (arg) : "x10", "memory");
9 }
```

Listing 5: The counting thread used for accurate timing.

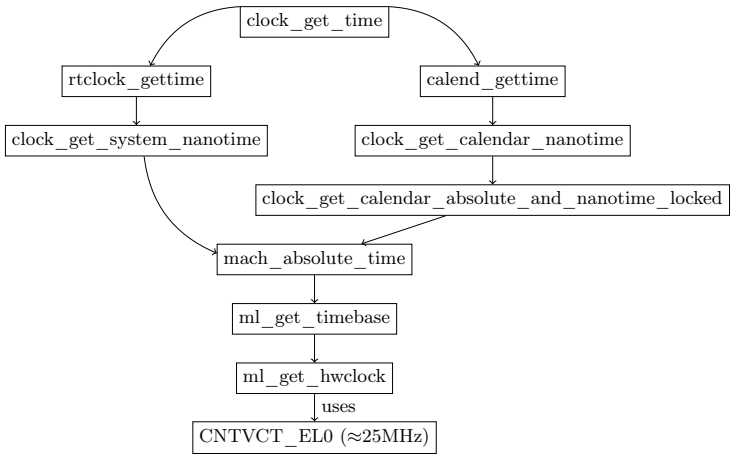


Fig. 7: Call hierarchy of timer library functions.