A Security RISC: Microarchitectural Attacks on Hardware RISC-V CPUs

Lukas Gerlach, Daniel Weber, Ruiyi Zhang, Michael Schwarz
CISPA Helmholtz Center for Information Security

***Abstract*—Microarchitectural attacks threaten the security of computer systems even in the absence of software vulnerabilities. Such attacks are well explored on x86 and ARM CPUs, with a wide range of proposed but not-yet deployed hardware countermeasures. With the standardization of the RISC-V instruction set architecture and the announcement of support for the architecture by major processor vendors, RISC-V CPUs are on the verge of becoming ubiquitous. However, the microarchitectural attack surface of the first commercially-available RISC-V hardware CPUs still needs to be explored.**

This paper analyzes the two commercially-available off-the-shelf 64-bit RISC-V (hardware) CPUs used in most RISC-V systems running a full-fledged commodity Linux system. We evaluate the microarchitectural attack surface and introduce 3 new microarchitectural attack techniques: Cache+Time, a novel cache-line-granular cache attack without shared memory, Flush+Fault exploiting the Harvard cache architecture for Flush+Reload, and CycleDrift exploiting unprivileged access to instruction-retirement information. We also show that many known attacks apply to these RISC-V CPUs, mainly due to non-existing hardware countermeasures and instruction-set subtleties that do not consider the microarchitectural attack surface. We demonstrate our attacks in 6 case studies, including the first RISC-V-specific microarchitectural KASLR break and a CycleDrift-based method for detecting kernel activity. Based on our analysis, we stress the need to consider the microarchitectural attack surface during every step of a CPU design, including custom ISA extensions.

1. Introduction

Microarchitectural attacks have been a threat to system security for multiple years. In recent years, a multitude of side channels have been discovered on x86 [27], [86] and ARM [59], [31]. These side channels have been used to attack cryptographic implementations [64] or spy on user behavior [67]. Still, traditional side-channel attacks, such as cache attacks [102] have often been ignored in CPU designs. Thus, developers must ensure that the software is resistant to side channels [46]. While side-channel-resistant code is common for cryptographic algorithms, it is often infeasible to protect general software.

A large body of mitigation proposals ranges from surgical changes, which, e.g., prevent the unprivileged use of specific instructions [102], to complete redesigns of microarchitectural elements such as caches [99], [74], [89]. Although such mitigations cannot prevent leakage for all applications, they at least decrease the leakage rate or increase the complexity of potential attacks. Moreover, several of these proposals come with minimal or no performance overhead when implemented in hardware [41], [71], [99]. CPU, OS, and browser vendors gradually adopt such approaches, e.g., by preventing unprivileged access to high-resolution timers [94] and performance counters, removing such functionality entirely from the hardware [39], or ensuring that instructions do not have operand-specific execution times [47]. However, drastic changes to the instruction set architecture are unlikely for backward compatibility.

In 2010, the RISC-V instruction set architecture was announced. Due to its open-source license and flexibility, RISC-V cores have found adaptation outside the academic field. With the freezing of the user- and kernel-space ISA in 2019 [7], companies started producing publicly sold RISC-V cores. In addition to RISC-V CPUs for embedded systems and IoT, RISC-V-based mobile phones [85] and laptops have been announced [100]. The collaboration between academia and industry, a CPU designed from scratch, and the freedom to modify the microarchitecture in vendor-specific ways pose an opportunity to mitigate known microarchitectural attacks. Especially for mitigations with next to no overhead, RISC-V CPUs could become a more secure alternative to older ISAs, such as x86 and ARM.

In this paper, we ask the following research question:

Does the microarchitectural attack surface on COTS RISC-V CPUs differ from x86 and ARM CPUs, and does this affect possible microarchitectural attacks and defenses?

To answer this question, we analyze the microarchitectural attack surface on the SiFive U74 [84], and the XuanTie C906 [88] announced in 2019 and 2020, respectively. Both CPUs are among the first mass-produced 64-bit RISC-V CPUs. These CPUs have been available for end users since 2021 and are still the most recent RISC-V CPUs available on the market. They are explicitly marketed for consumer devices since they run a full-fledged multi-user Linux distribution with a graphical user interface. While several RISC-V boards are available, these CPU cores and their derivatives are used in most boards running Linux, making them highly representative of the currently available RISC-V CPUs. In contrast to previous microarchitectural attacks on RISC-V [29], [4], we focus on readily-available CPUs running Linux in a default configuration, not on a simulated or synthesized core. Ahmadi et al. [4] introduces software side-channel attacks on RISC-V processors and evaluates Prime+Probe on an FPGA board. Our work, however, performs a systematic evaluation of the microarchi-

tectural attack surface and finds novel attacks specific to RISC-V CPUs. In addition, we focus on commercially-available hardware RISC-V CPUs and show that custom vendor extensions add additional attack surface.

Based on our analysis of the microarchitectural attack surface, we present 3 new microarchitectural attack techniques on RISC-V CPUs: Cache+Time, Flush+Fault, and CycleDrift. Cache+Time is a new cache side-channel attack that exploits speculative instruction prefetching of existing RISC-V CPUs in combination with the standardized `fence.i` instruction that flushes the instruction cache. Cache+Time is the counterpart to Evict+Time [69], but with cache-line granularity. Hence, Cache+Time is the first cache-line-granular cache attack without shared memory. Flush+Fault is a Flush+Reload variant on Harvard-style instruction caches, i.e., instruction caches that are fully separated from data caches, used on these RISC-V CPUs. While, Flush+Reload on x86 and ARMv8 exploits that unified instruction and data caches exist and accessing instructions via a memory load directly reveals their cache state. We avoid the dependency on such a unified cache for measuring the reload by abusing simple code gadgets or misaligned code in the victim. With Flush+Fault, an attacker times a jump into the victim code that either returns or leads to a fault, providing the same information and granularity as Flush+Reload on x86. Due to the stricter alignment requirements and fewer memory operands on RISC-V than on x86, this variant can be mounted reliably. CycleDrift allows attackers to infer side-channel information about code executed in different security domains, e.g., the kernel or other applications. RISC-V provides the number of retired instructions and the number of CPU cycles for the entire CPU core to unprivileged attackers. Hence, by looking at the differences between cycles, instructions, and wall time, CycleDrift provides insights into (inaccessible) executed code, even across security domains.

In addition to introducing these new attacks, we compare them with microarchitectural attacks on platforms such as x86 and ARM. We demonstrate well-known cache attacks that rely only on timing measurements and memory accesses, such as Prime+Probe [70] and Evict+Reload [36]. In contrast to previous attacks on x86 [63], [34] and ARM [59], [39], we achieve success rates of up to $100\,\%$, due to the different cache architectures of the C906 and U74. Due to the deterministic replacement policy on the C906, an attacker can even improve Prime+Probe to determine the *number* of victim accesses to the monitored cache set. Based on the non-standard cache-flush instruction `dcache.civa` present on the C906, we also demonstrate the first Flush+Reload [102] attack on a commercial-of-the-shelf (COTS) hardware RISC-V CPU, and a variant of the Flush+Flush attack [35], [91], both with a success rate of $100\,\%$, i.e., a single measurement is sufficient to correctly identify whether a cache line is cached.

To demonstrate the real-world impact of our attacks, we present 6 case studies. We first demonstrate Flush+Fault on the square-and-multiply code found in the RSA implementation of MbedTLS 1.3.10, which is commonly used to demonstrate side-channel attacks [43], [55], [26]. We show the first microarchitectural KASLR breaks using two different techniques on RISC-V CPUs. One exploiting the RISC-V-specific `rdinstret` instruction on the U74, and one exploiting timing similar to KASLR breaks on Intel and AMD [33], [57]. Both KASLR breaks achieve a success rate of $100\,\%$ in less than $800\,\mathrm{ms}$, which is comparable to microarchitectural KASLR breaks on x86 [12]. In addition to breaking KASLR, the retired-instruction counter used in CycleDrift circumvents the Zigzagger branch-shadowing mitigation [55], and acts as an oracle whether an inaccessible file exists inside a write-only (drop-box) folder. Similarly, we use CycleDrift to detect asynchronous code execution in the kernel, such as interrupt handlers. Previous work showed that detecting these interrupts allows spying on user input [67], [18], [58] or fingerprinting websites [58]. Finally, we mount Evict+Reload, Flush+Reload, and Prime+Probe on the AES implementation of OpenSSL 1.0.1e, another common target for demonstrating side-channel attacks [9], [69], [35], [60], [11]. We achieve close to $100\,\%$ accuracies for all of these attacks.

For all discussed attacks, hardware-mitigation proposals exist [99], [74], [89], [75], [96], [17], [37], [41]. However, all of our attacks—both new and existing—are successful on both CPUs, indicating that there are no integrated hardware mitigations. Additionally, due to specific design and implementation decisions, our proof-of-concept implementations on the RISC-V CPUs are more reliable than on x86 and ARM CPUs. Even more concerning is that non-standard instructions have been added to the CPUs, fostering such attacks. Our findings support the hypothesis that most mitigations proposed in recent work are beyond what hardware manufacturers are willing to implement. We hope that our insights into the attack surface of early models of RISC-V COTS CPUs lead to more secure CPUs in the future.

Contributions. The main contributions of this paper are:

1) We evaluate the microarchitectural attack surface on the C906 and U74 RISC-V cores, identifying new attack techniques and confirming the applicability of attacks known from x86 and ARM.
2) We introduce 3 attack techniques on RISC-V: Cache+Time, Flush+Fault, and CycleDrift, enabling powerful cache attacks and instruction-level side channels.
3) We show the first RISC-V microarchitectural KASLR break, and demonstrate the implications of unprivileged access to retired instructions across security domains.
4) We reproduce known cache attacks, such as Flush+Reload, Prime+Probe, Evict+Reload, to enable the comparison with well-studied ISAs such as x86 and ARM. We show their effectiveness on RISC-V by extracting keys from vulnerable AES and RSA implementations with nearly $100\,\%$ accuracy.

Responsible Disclosure. We disclosed our findings to T-head and SiFive. T-head is currently looking into ways of disabling unprivileged access to interfaces we use in our paper. SiFive is analyzing the results presented in our paper. Our experiments and case studies are open source: https://github.com/cispa/Security-RISC

2. Background

2.1. RISC-V

RISC-V is a free and open instruction set architecture (ISA) maintained by the RISC-V foundation. The reduced size of the ISA and the load-store architecture simplifies CPU design compared to CISC architectures such as x86. RISC-V provides 32, 64, and 128-bit addressing modes, making it adaptable for use cases from IoT to high-performance computing. The ISA consists of an unprivileged [97] and a privileged part [98]. RISC-V supports up to 4 privilege levels, machine, hypervisor, supervisor, and user mode, with the hypervisor mode still being finalized. The operating system typically runs in supervisor mode, and applications run in user mode. Major compilers, such as GCC and LLVM, and operating systems, such as Linux, support the RISC-V ISA. Additionally, several open-source cores are available [6], [14]. These cores can be used as soft cores synthesized on FPGAs for development and research purposes. Moreover, CPUs in silicon, such as the SiFive U74 [83] or the XuanTie C906 [88], are already available. Single-board computers, such as the Sipeed Nezha or Sipeed Lichee RV, are equipped with a C906 CPU, run a full-fledged Debian system, and can be bought off the shelf.

2.2. CPU Microarchitecture

The CPU microarchitecture describes the specific hardware implementation of an ISA. Generally, events in the microarchitecture are invisible to the programmer, as they are not defined at an ISA level and are only present for, e.g., optimization purposes. In the following, we introduce microarchitectural elements relevant to this paper, including the cache, address translation, and prediction subsystems.

Cache. The cache is a memory buffer located between the main memory and the CPU. A cache speeds up memory access by keeping recently used copies of data closer to the CPU. Typically, caches are split into instruction (I-Cache) and data caches (D-Cache). Caches are organized into cache lines indexed by virtual or physical memory addresses. Most caches use cache sets that group multiple cache lines.

Address Translation. The CPU performs page-table walks to translate virtual to physical addresses. The results of the address translation are stored in a separate cache, the Translation Lookaside Buffer (TLB). TLBs are typically organized in multiple levels.

Branch Prediction. Branch predictors are used to speed up instruction fetches. When a branch instruction is executed, significant overhead occurs due to stalling, as the branch direction or target is unknown and cannot be fetched until the instruction is resolved. To mitigate branch overhead, the CPU guesses whether the branch is taken based on the history of recently taken and not taken branches. In addition, the destination of the branch can be predicted.

2.3. Microarchitectural Attacks

Microarchitectural attacks exploit effects introduced by the microarchitecture. Microarchitectural side-channel attacks leak data from the microarchitecture using a side channel exploited in software. The attacks relevant to this paper can be classified by the exploited microarchitectural element: the cache, address translation, or branch predictors.

Cache Attacks. Caching introduces the potential for side channels, as there is a timing difference depending on whether values are cached. Cache attacks can be classified by their methods to control and leak the cache state.

Flush+Reload [102] uses cache maintenance instructions to remove values in shared memory from the cache. An attacker can infer if the location has been accessed by performing a timed load on the shared memory. Flush+Reload is used for attacks on cryptographic implementations [64] and to spy on user behavior [67], and as a building block for transient-execution attacks [13]. Flush+Flush [35], is a variant that exploits the flush instruction's timing behavior.

An alternative, if cache maintenance instructions are unavailable [59], [39], is Evict+Reload [36]. Instead of removing a cache line by flushing, the target cache line is removed by repeatedly accessing addresses in the same cache set. Eviction sets can be built efficiently using the cache-set mapping [59], [34], or by relying on side channels [93].

Prime+Probe [70] enables attacking victim processes that do not share memory with the attacker. Similarly to Evict+Reload, a cache set is evicted. In addition, the attacker measures the access time on the evicted cache set to infer whether the monitored location was accessed. As an attacker needs to know cache sets of interest to probe, Prime+Probe-based attacks include a phase in which the cache is monitored for attackable event-dependent accesses [63].

Attacks on the Memory Subsystem. As the TLB acts as a cache for address translation, it can also be used for microarchitectural attacks. The timing difference of cached and uncached addresses in the TLB has been used to attack cryptographic implementations [30], or to break KASLR [42], [33], [76], [57]. Contrary to cache attacks, hardware mitigations for the TLB are not well explored.

Prediction-based Attacks. Branch predictors induce a timing difference depending on whether they predict a branch correctly or incorrectly. If the addresses of the victim and attacker processes map to the same entry in a shared branch predictor, they influence each other. Branch-prediction side channels [3] abuse secret-dependent delays by mistraining a branch predictor. Such attacks allow, e.g., to extract keys from the OpenSSL RSA implementation [2]. Lee et al. [55] introduced a similar approach called branch shadowing. By observing the mispredictions in the attacker process, the direction of victim branches can be inferred.

Transient-execution Attacks. Transient execution attacks rely on the execution of the predicted instruction stream. Gonzales et al. [29] simulated Spectre attacks on the open-source BOOM core in the FireSim simulator. Similarly, Fuchs et al. [25] develop a test suite for Spectre attack on RISC-V. The currently available RISC-V processors do

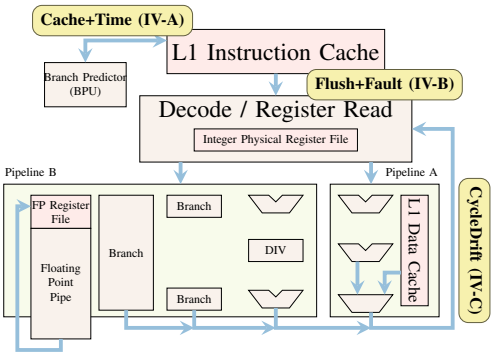

Figure 1: Block diagram of the U74 core. The diagram also contains our attack primitives next to the microarchitectural elements they exploit and a reference to the section where they are described in more detail.

not support speculative execution and are thus not vulnerable to classic transient execution attacks.

3. Systematic Side-Channel Analysis

This section systematically analyzes and categorizes the microarchitectural side channels on the C906 and U74 CPUs. Unlike previous work [65], [4], we cover a wide range of microarchitectural elements. Figure 1 shows an overview of the more-complex U74 core with its documented microarchitectural elements. The figure also contains references to the sections where our new attack primitives based on the analysis of this section are described in detail. Moreover, we only use COTS hardware and software with the default configuration. Therefore, our results apply to real systems found in the wild. While we use the C906 and U74 in our experimental setup, these CPUs already cover most 64-bit Linux-capable RISC-V devices [16] available at the time of this research. As RISC-V CPUs are expected to gain popularity, this underlines the necessity of analyzing the microarchitectural attack surface.

Requirements. Mounting microarchitectural attacks on RISC-V CPUs has the following requirements, which we systematically explore in a bottom-up manner.

$\mathcal{R}1$ *High-resolution Timer.* Most microarchitectural attacks require a high-resolution timer. Although previous work covers several of these primitives for x86 [102], [79], [77] and ARM [59], [39], there is no analysis on RISC-V.

$\mathcal{R}2$ *Cache Maintenance.* In contrast to x86 and ARM, RISC-V does not specify instructions for cache maintenance. Hence, flush-based primitives, such as Flush+Reload, are only possible with unprivileged custom vendor extensions. Moreover, there is no analysis of cache structures and replacement strategies on RISC-V CPUs. However, this knowledge is required to build effective eviction sets for eviction-based attacks, such as Prime+Probe. In addition, we analyze the address translation cache (TLB), which is required for attacks such as TLBleed [30].

$\mathcal{R}3$ *Novel Performance Optimizations and Interfaces.* While most RISC-V cores use well-known performance optimizations, such as caches and TLBs, there are also optimizations not used in this form on x86 or ARM, such as speculative branch-target prefetching. As these optimizations can introduce new attack vectors, they must also be analyzed. In addition, RISC-V can expose interfaces to unprivileged users unavailable on other CPU architectures. Moreover, these interfaces are only supported in some widely-used compilers.

Experimental Setup. We evaluate two different processors, the T-head C906 and SiFive U74. To evaluate the C906, we use two Allwinner D1 boards, the Sipeed Lichee RV and the Sipeed Nezha. Both boards use the C906 RISC-V CPU and only differ in their amount of main memory (512 MB and 1 GB, respectively). The C906 provides the `rv64imafdcv` instruction set extensions, i.e., it supports the base RISC-V instruction set, including the extensions for a general-purpose ISA, as well as compressed and vector instructions. Both boards run Debian 12 with kernel 5.14.0-rc4-nezha (Nezha) and 5.4.61 (Lichee RV). To evaluate the U74, we use the StarFive VisionFive board. The U74 implements the `rv64gc` instruction set extension and contains 8 GB of main memory. The VisionFive board runs Ubuntu 22.04.1 LTS with the 5.17.5-visionfive kernel. All proofs of concept use unprivileged code execution on the unmodified operating system.

3.1. Systematic Evaluation of Timers

In this section, we tackle requirement $\mathcal{R}1$ and evaluate timing primitives available to unprivileged attackers. Table 3 (Appendix A) summarizes our results. We distinguish between *hardware timers* directly accessible using assembly instructions and *software timers* that involve the OS.

Hardware Timers. The RISC-V ISA provides two different performance counters for measuring time that can be accessed via the pseudo instructions `rdcycle` and `rdtime` [97]. Both counters are accessible to unprivileged users. The RISC-V ISA manual explicitly states: *We mandate these basic counters be provided in all implementations as they are essential for basic performance analysis, adaptive and dynamic optimization, and to allow an application to work with real-time streams.* We verify that these counters are available on the C906 and U74 and can be used to measure small timing differences. The `rdcycle` counter has a resolution of 1 cycle on both tested CPUs. The `rdtime` counter provides timing measurements with a lower resolution of 45 ns. Still, the counter increments in steps of 1. For both counters, it is not relevant if they are read via the pseudo instructions or directly using the `csrr` with the respective counter.

In addition to these explicit timing-related instructions, the RISC-V ISA provides an unprivileged performance counter that tracks the number of retired instructions. With that, an attacker can implicitly time program execution. In the best case, where the attacker knows the code and every instruction takes one CPU cycle, this method results in a timer with a 3 ns resolution on the C906. On the U74, this method achieves a higher resolution of 1 ns due to

the increased throughput of the CPU pipeline. This counter updates after every retired instruction. It can be read using the `rdinstret` or the `csrr` instruction.

Software Timers. In addition to the hardware counters defined in the RISC-V ISA, it is also possible to rely on OS-provided timers. POSIX defines the `clock_gettime` system call, which returns a high-resolution timestamp. This system call has also been used for microarchitectural attacks in previous works [59]. It is also available on Debian, providing a time stamp with a resolution of 1 ns as well. As a fallback solution, it is possible to implement a counting thread as used in related work [59], [77]. A counting thread can become necessary if the microarchitectural attack is not implemented in native code but in a restricted environment such as a browser [79], [30]. Due to the added loop overhead, the resolution of the timer is lower and dependent on the CPU performance. On the C906, we achieve a resolution of 2 ns. On the U74, we achieve a 1 ns resolution. While the resolution is sufficient in both cases to mount microarchitectural attacks, a counting thread has the disadvantage of requiring a dedicated core to execute on.

3.2. Cache Maintenance

In this section, we tackle requirement $\mathcal{R}2$ for the data and instruction cache and systematically analyze the cache design and possible cache maintenance primitives.

Cache Design. The C906 and U74 CPUs have dedicated first-level caches for data (D-Cache) and instructions (I-Cache), which are both 32 kB in size. Both the I-Cache and the D-Cache have 64 B cache lines. The I-Cache is a 2-way set-associative cache with 256 cache sets. The D-Cache is a 4-way set-associative cache with 128 cache sets. As on most CPUs, both caches are virtually indexed and physically tagged [87]. The U74 adds a second-level shared data and instruction cache. The L2 cache is 128 kB in size, has a line size of 64 B and is 8-way set associative. In contrast to complex cache replacement policies on x86 [1] and ARM [59], the C906 uses a deterministic first-in-first-out (FIFO) strategy [87]. This replacement policy is used for the D- and I-Cache. The U74 uses a PLRU replacement policy in the L1 D-Cache and random replacement in both the L1 I-Cache and the shared L2 cache. We verified that a more complex stratey than tree PLRU is used by implementing a perfect tree-PLRU eviction and observing that it is insufficient to find eviction sets.

Cache-maintenance Instructions. While the RISC-V instruction set does not specify any cache-maintenance functions, vendors are free to implement such functions. The only function in the base instruction set that can–depending on the actual implementation–be used for cache maintenance as a side effect is the unprivileged `fence.i` instruction. The RISC-V ISA manual [97] states: *A simple implementation can flush the local instruction cache and the instruction pipeline when the FENCE.I is executed*. We verify that `fence.i` shows this behavior on both CPUs. Executing the `fence.i` instruction flushes the entire instruction cache. The execution time is 914 cycles and 28 cycles on the C906 and U74, respectively.

The C906 CPU also provides a set of non-standard cache-maintenance instructions [88, §7.3.2] as shown in Table 4. While most instructions are privileged, two relevant instructions can be used in user space. The `dcache.civa` can be used as the `clflush` instruction on x86. It takes a virtual address and flushes the specified cache line from the data cache. Likewise, the `icache.iva` instruction precedes an equivalent action for the instruction cache. Both instructions require 4 cycles to execute. All other flush instructions flush by physical address, cache set, and way, or the entire cache, are only available in the OS. We exploit these unprivileged flush instructions in Section 4.2 for Flush+Fault and, for comparing to other architectures, to reproduce known attacks in Section 4.4 and Section 5.6.

Cache Eviction. With the FIFO cache-replacement strategy used in the C906, it is sufficient to access as many addresses falling into the same cache set as there are cache ways. Thus, executing code on 2 addresses mapping to the same cache set is sufficient to evict a cache line from the I-Cache. Similarly, accessing 4 addresses mapping to the same cache set is sufficient to evict a cache line from the D-Cache. The only requirement is that bits 6 to 12 for the D-Cache and 6 to 13 for the I-Cache, are identical for addresses in the eviction set. Evicting a cache line from the D-Cache takes 25 cycles. Due to the straightforward eviction-set generation and deterministic replacement policy, we achieve an eviction rate of 100 %. For the U74, the method by Gruss et al. [34] finds an efficient eviction set for the L1 D-cache. Our eviction set performs 168 memory accesses and takes 1671 cycles to evict a cache line with an F-Score of 0.997. We use cache eviction to reproduce known attacks in Section 4.4 and Section 5.6 to allow comparison to other architectures.

3.3. TLB

This section analyzes the TLB design and provides efficient TLB eviction strategies, tackling $\mathcal{R}2$.

TLB Design. Both analyzed CPUs use 39-bit addresses for virtual memory (Sv39). The C906 caches address translations in two separate 10-entry fully-associative TLBs for data and instructions [87]. The U74 uses separate 40-entry L1 TLBs for data and instructions and a shared 512-entry L2 TLB. As on x86 and ARM, there is no unprivileged instruction to flush TLB entries or the entire TLB.

TLB Eviction. Our experiments show that accessing as many addresses on different pages as there are entries in the first-level TLB reliably evicts the TLB. With a page size of 4 kB, an attacker only requires a 40 kB buffer to evict the entire TLB C906 and 160 kB on the U74. The TLB eviction takes 1396 cycles on the C906 and 866 cycles on the U74, with an F-Score of 1. We use TLB eviction to build a side channel in Section 4.4.

3.4. Branch Prediction

In this section, we tackle requirement $\mathcal{R}3$ for prediction-based optimizations in the C906 and U74. Both CPUs have an in-order pipeline and do not support speculative or out-of-order execution. However, they use speculative prefetching and decoding of instructions following branches. While the U74 allows the deactivation of speculative instruction fetching [84, §7.6], it is enabled in the provided Linux image. Both CPUs have 3 branch predictors: a branch-direction predictor for conditional jumps, a branch-target predictor for jumps and calls, and a return-address predictor for function returns. The U74 also has an indirect jump-target predictor. Predictors are shared between user and kernel mode.

The *branch history table* (BHT) is used to predict conditional branch instructions. The BHT on the C906 has a capacity of 16 kB and uses the gshare prediction algorithm [88, §7.1.2] with a 14-bit global history register (GHR). The instruction-fetch unit prefetches the predicted branch After the branch is retired, a writeback to the predictor updates the corresponding GHR entry. On the U74, a 3.6 KiB BHT is used together with a proprietary prediction algorithm [84, §3.2.7]. No CPU provides functions to flush the BHT in user mode.

The *branch jump target* (BJT) buffer contains 16 entries and predicts the target of direct jumps. On the C906, it is fully associative [88, §7.1.3]. An entry is indexed using bits 16 to 32 of the current program counter. If a BJT entry is available, the jump results can be immediately predicted by adding the current program counter to the offset stored in the BJT entry. The instruction fetch unit can then prefetch the jump target address result without waiting for the jump to be resolved. User and supervisor mode do not have access to BJT maintenance functions [88, §16.1.7.3].

The *return address predictor* (RAP) predicts the return address when returning from a function using a stack. The RAP contains 4 addresses on the C906 and 6 on the U74. An *indirect-jump target predictor* is used on the U74 to predict indirect jumps. It contains 8 entries and is not manageable from machine or user mode.

In Section 4.4, we show that the branch predictors allow mounting known branch-prediction attacks.

3.5. Performance Counters

In this section, we tackle requirement $\mathcal{R}3$, showing that the RISC-V ISA provides unprivileged access to specific performance counters unavailable on x86 or ARM.

The RISC-V ISA guarantees access to 3 unprivileged performance counters. The `cycle` counter counts CPU cycles and is available via the unprivileged `rdcycle` instruction or by reading the performance counter register with `csrr` directly. The `time` counter tracks wall clock time and map it to an internal time register. The counter can be read with the `rdtime` instruction or directly from the control register. The `instret` counter counts the number of retired instructions and can be read with the `rdinstret` instruction or by reading the corresponding control register directly. The access to all mentioned counters takes 6 cycles, independent of whether they are read via the corresponding pseudo instruction or the performance counter register using the `csrr` instruction. While other counters exist on all tested systems, they are not enabled in user mode per default, meaning an unprivileged attacker cannot access them.

We exploit the unprivileged performance counters in our CycleDrift primitive (cf. Section 4.3) that we showcase in the case studies in Section 5.2, Section 5.3, and Section 5.4.

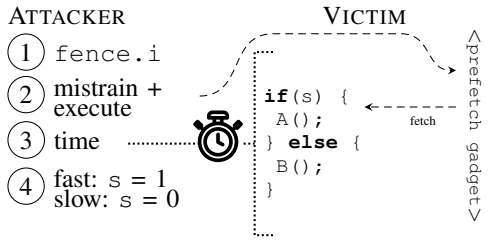

Figure 2: Cache+Time: An attacker flushes the I-Cache (①), mistrains and executes a speculative prefetch gadget [13] (②) to cache a secret-dependent code path (`A()`), and times the victim's runtime (③). Based on the runtime, the attacker knows if the prefetched code path was executed (④).

4. Microarchitectural Attack Primitives

In this section, we introduce our new attack primitives Cache+Time, Flush+Fault, and CycleDrift. Cache+Time is an I-Cache attack specific to the RISC-V architecture exploiting I-cache flushing in conjunction with speculative prefetching. Flush+Fault shows that it is possible to mount attacks on the I-Cache in split-cache architectures as present on the C906. CycleDrift exploits the differences between the exposed retired instructions and cycle counters and is specific to the RISC-V architecture. We also reproduce and extend known microarchitectural attack primitives, such as Prime+Probe, Flush+Reload, Evict+Reload, Flush+Flush, and branch-prediction attacks.

4.1. Cache+Time

In this section, we introduce a new cache-attack primitive, Cache+Time. Cache+Time exploits branch prediction in combination with the possibility to flush the I-Cache on RISC-V. Cache+Time has cache-line granularity without requiring shared memory.

Overview. The basic idea is to turn Evict+Time [69] around. For Evict+Time, an attacker evicts a cache line of interest and measures if the execution time of the application changes. If the execution time increases, the victim uses the target cache line; otherwise, it is not. For Cache+Time, the attacker flushes the entire instruction cache and caches a cache line of interest. If the execution time of the target changes due to this caching, the attacker knows that the victim uses the cache line. Cache+Time leverages two building blocks. First, Cache+Time exploits that on all

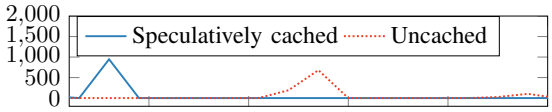

Figure 3: Cache+Time histogram of the victim execution time on the U74. The runtime of speculatively-prefetched code is lower compared to normal execution.

analyzed RISC-V CPUs, an unprivileged attacker can flush the entire instruction cache (cf. Section 3.2), which is not possible on x86 or ARM. Second, Cache+Time exploits that the predictors on the C906 and U74 can be tricked into fetching arbitrary cache lines from the victim address space into the cache using a simple speculative prefetch gadget [13], [81], [50]. Figure 2 shows an outline of Cache+Time. In this example, the attacker wants to exploit a secret-dependent branch instruction. For this, the attacker flushes the instruction cache (①), and uses a speculative prefetch gadget in the victim's application to load a cache line of the `A` function into the I-Cache (②). The attacker can then measure the execution time of the victim (③). The victim executes faster if the function `A` is called, as this function is the only one cached. If there is a speed-up, the attacker learns that the secret `s` in the branch condition is 1 (④).

Threat Model. We assume that the victim application contains a secret-dependent execution flow. We do not require any shared memory between attacker and victim applications. The only requirement is a speculative-dereference gadget that the victim executes. The victim can be either a privileged or an unprivileged application. The attacker controls an unprivileged application.

Evaluation. We evaluate the attack on an artificial implementation of square and multiply. For the prefetch gadget, we rely on the branch-history table for misprediction. Based on an attacker-induced misprediction of the branch, the square function is either cached or not. In line with previous work [80], the misprediction is not perfect. For our gadget, we achieve a misprediction rate of approximately 14 % on the C906 and 100 % on the U74. The higher misprediction rate on the U74 can be explained with the more complex branch predictor, as discussed in Section 3.4. Figure 3 shows the execution time of the target branch. If the attacker does not induce misspeculation, the execution time is always slow, as the function code is flushed from the I-Cache. Otherwise, the execution time is faster, as seen in the histogram's peak on the left side.

Related Attacks. Cache+Time is related to Spectre [52] and Evict+Time [69]. While the C906 and U74 predict the outcome of branch instructions (cf. Section 3.4), they do not support execution of the predicted instruction stream. Instead, the predicted instructions are fetched into the I-Cache. Hence, existing Spectre-style attacks [52], [13] requiring instruction execution during speculation are impossible. In contrast to Evict+Time, Cache+Time achieves cache-line granularity, as it does not require eviction to force entries out of a cache set. Instead, the entire I-Cache is flushed, and

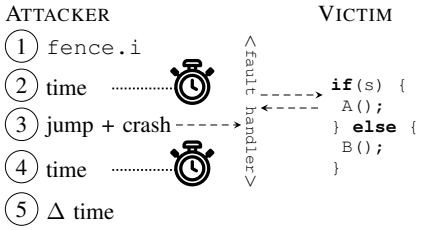

Figure 4: Crash based Flush+Fault: The attacker flushes the entire I-Cache (①), and times (②, ④) a jump to the victim address containing the target cache line (③), while handling the successive fault. Based on the timing difference (⑤) of the fault handling (③) an attacker infers the cache state of the target cache line.

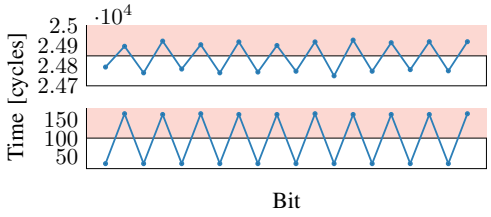

Figure 5: Comparison between the noise of the crash-based (top) and the return-based Flush+Fault (bottom).

the targeted cache line is prefetched using the speculative prefetch gadget. As `fence.i` flushes the entire I-Cache, Cache+Time does not require shared memory. Cache+Time is specific to RISC-V CPUs, as x86 and ARM do not allow an unprivileged user to flush the entire I-Cache.

4.2. Flush+Fault

In this section, we introduce Flush+Fault, a variant of Flush+Reload that can be used without unified caches.

Overview. State-of-the-art Flush+Reload attack scenarios against victim code pages rely on the victim executing the target cache line, bringing it into the CPU cache. This relies on the assumption that the I- and D-caches are either unified or synchronized, which is the case for most Intel CPUs L2 and L3 caches. However, this synchronization is not always given for the targeted RISC-V CPUs, with some of them employing a Harvard split-cache architecture. To overcome this challenge and enable Flush+Reload-style attacks for code pages on these CPUs, we introduce Flush+Fault. The main idea is to replace Flush+Reload's data-load step (*reload*) with a faulting jump into the shared victim code or an immediate return from the victim code. We refer to these cases as fault-based Flush+Fault and return-based Flush+Fault, respectively. As both cases execute, and therefore load, the victim code, their timing differs slightly when the victim code is in the CPU's instruction I-cache.

Figure 4 shows the steps of Flush+Fault. First, the attacker brings the I-cache into a controlled state ① by flushing it using the `fence.i` instruction. Next, the attacker takes a timestamp ② and jumps to an address ③ that

maps to the targeted cache line in the victim. The jump has to target an address that leads to a fault. This can either be provoked by jumping to an invalid combination of bytes or jumping to an instruction sequence containing a faulting instruction still in the targetted cache line. For example, the attacker can zero the `a0` register and jump to the instruction `lb t1, 0(a0)`, thus inducing a fault. The attacker handles the fault ③, e.g., by registering a signal handler and takes the second timestamp ④. By comparing the delta between the two timing measurements ⑤, an attacker can deduce whether code in the target cache line was executed. This is because code already cached in the I-cache loads slightly faster and thus also induce the fault slightly faster, leading to a lower timestamp delta. After each measurement, the attacker issues multiple calls from the same jump instruction to a dummy location outside the target cache line. This ensures that the branch predictor never predicts and thus prefetches the target cache line. This is important as such a prefetch would cache the target cache line and effectively remove our information leakage.

As crash handling induces overhead, we also present an optimized return-based technique inspired by return-oriented programming [82]. If the target cache line contains a `ret` instruction, the attacker can jump directly to it in ③. This leads to an immediate return of the call without any further side effects. Return-based Flush+Fault results in low-overhead measurements but relies on a (potentially unaligned) `ret` instruction in the target cache line. Both the U74 and the C906 support the compressed (C) instruction-set extension. Hence, they allow unaligned instructions, increasing the number of potential `ret` instructions.

Evaluation. Figure 5 visualizes the difference in noise between the two discussed techniques when leaking a signal of alternating ones and zeroes. The plotted values are accumulated using the median of 1000 measurements executed on the C906. The context switch performed by the signal handling in the crash-based technique results in higher timing variations leading to a noisier signal. Nevertheless, even if it performs worse than the optimized techniques, the fault-based Flush+Fault is still highly precise when measurements are accumulated. We achieve a recall and precision of 1.0 over 100 different bits using the fault-based Flush+Fault technique. This also results in an F-score of 1.

4.3. CycleDrift

In this section, we introduce CycleDrift, an attack primitive exploiting the unprivileged access to the number of retired instructions.

Overview. In contrast to x86 and ARM, RISC-V provides an unprivileged counter for retired instructions, `rdinstret`. This introduces a side channel, as instructions taking more than one cycle lead to differences between retired instructions and CPU cycles. An attacker that reads both the `rdcycle` and the `rdinstret` counter can determine how many such instructions have been retired. These instructions are defined per CPU core and include statistics about instructions from different applications and different security domains, such as the operating system. Hence, this side channel can spy on kernel code (cf. Section 5.5). In addition, we verify that even machine mode code is susceptible to CycleDrift. We choose a simple environment call into the platform-specific firmware running in machine mode to get the vendor ID (`SBI_EXT_BASE_GET_MVENDORID`) and a more complex one to output a character to the console (`SBI_EXT_0_1_CONSOLE_PUTCHAR`). We see a clear timing difference when executing the two instructions to get the ID and outputting a character. On the C906, we measure 613 and $85\,109$ cycles, respectively (average over 1000 measurements). On the U74, we measure 963 and $85\,507$ cycles, respectively. Moreover, we also see a difference in the retired instructions. On the C906, 265 and 4505 instructions retire, respectively (again the average over 1000 measurements). On the U74, 267 and 7050 instructions retire, respectively.

As the RISC-V ISA does not specify the latency of instructions, we benchmark the RISC-V instruction set on the C906 and the U76. Our results show that, among others, instructions using the FPU consume more than one cycle while still retiring as a single instruction. Table 5 (Appendix A) shows the instruction execution timings on both the U74 and C906 processors.

Threat Model. As both the `rdcycle` and `rdinstret` are unprivileged and do not distinguish between user and kernel space, CycleDrift enables an unprivileged attacker to target unprivileged or privileged applications. We assume the targeted code contains secret-dependent control flow that can be distinguished by the ratio of retired instructions and spent CPU cycles.

Evaluation. We evaluate CycleDrift on a padded square-and-multiply implementation. The implementation in Listing 1 (Appendix A) hides the cycle difference between the different branches by adding `nop` instructions. When observing the number of retired instructions, an attacker can distinguish which branch was taken as the multiplication retires as a single instruction, whereas the 3 `nop` instructions used to pad the else branch retires as 3 instructions. The cycles stay equal, with 18 and 14 instructions per iteration on the C906 and U74, respectively. However, the retired instructions differ on both CPUs, by 2 and 3 on the U74 and C906, respectively. Hence, the user-accessible `rdinstret` instruction opens up additional attack surfaces and allows an attacker to bypass mitigations intended to conceal the timing behavior of cryptographic implementations.

4.4. Known Attacks and New Variants

This section evaluates the applicability of state-of-the-art cache, TLB, and branch-predictor attacks on the U74 and C906. All measurements use a sample size of $n = 1\,000\,000$ and evaluate the F-score and temporal resolution. Appendix A contains the histograms for the attacks.

Flush+Reload [102]. Flush+Reload allows an attacker to monitor accesses to a D-Cache or I-Cache line residing in shared memory. The attack requires a precise timer and an instruction that flushes a cache line. Both of these

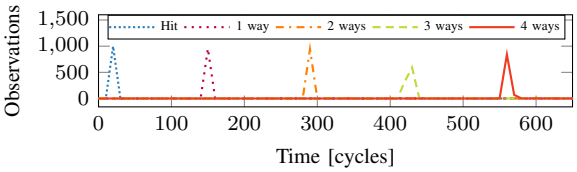

Figure 6: Prime+Count histogram on the C906. The cache hit (densely-dotted blue) is for reference. The other timings are for an increasing number of evicted cache ways: 1 (dotted purple), (dash-dotted orange), 3 (dashed green), and 4 ways (solid red). Due to the perfect eviction and reverse-probe step, these cases can be clearly distinguished.

requirements are present on the C906 chip. On the C906, an attacker can use the unprivileged `dcache.civa` and `icache.iva` instructions to flush a virtual address from the D or I-Cache. Alternatively, the `fence.i` instruction can be used to flush the entire I-Cache on both the C906 and U74, which we refer to as Fence+Reload. As on x86 [102] and ARM [59], Flush+Reload requires shared memory between the attacker and victim. All timing primitives from Section 3.1 are usable. An attacker achieves cache-line granularity with a temporal resolution of 339 ns on the C906 with Flush+Reload and 276 ns on the C906, and 120 ns on the U74 with Fence+Reload. Flush+Reload and Fence+Reload have a perfect F-score of 1.

Flush+Flush [35]. With Flush+Flush, we can see the timing difference between flushing a modified and a non-modified cache line, as shown on x86 by Van Bulck et al. [91]. While both the `dcache.civa` and `icache.iva` instructions induce timing differences depending on the cache state, this is not the case for the `fence.i` instruction, making Flush+Flush exclusive to the C906. Flush+Flush has a perfect F-score of 1, with a temporal resolution of 334 ns. In contrast to x86 [35], Flush+Flush cannot distinguish a cache hit from a cache miss for a non-modified cache line.

Evict+Reload [36]. The advantage of Evict+Reload over Flush+Reload is that no unprivileged flush instruction is required, as the target address is evicted, making Evict+Reload viable on both the C906 and U74. For eviction, we rely on the eviction primitive from Section 3.2. As the eviction only requires knowledge of virtual addresses, building the eviction set is simple since the virtual address of the target cache line is known. Evict+Reload has a perfect F-score of 1 and a resolution of 397 ns on the C906, and an F-score of 0.997 with a resolution of 1355 ns on the U74.

Prime+Probe [70]. For Prime+Probe, we use the same eviction set for both the prime and the probe step. On the C906, this results in a perfect F-score of 1 and a temporal resolution of 392 ns. Due to the PLRU replacement strategy, Prime+Probe on the U74 has an F-score of 0.999 and a temporal resolution of 584 ns.

Prime+Count. Due to the deterministic replacement policy of the cache on the C906, a cleverly-chosen prime and probe step allows counting the number of victim accesses in the probe set. We refer to this variant as Prime+Count. Prime+Count provides more information than the binary classification of whether a victim accessed an address falling into the target cache set. As the replacement policy is known to be least-recently used, an attacker can access the eviction set in *reverse order* for the probe step. By doing this, the attacker learns *how many* addresses of the prime set have been evicted by the victim. Given a prime set $\mathcal{P} = \{p_0, p_1, p_2, p_3\}$. The victim accesses have the newest timestamps and evict the addresses first accessed in the prime set, e.g., the first victim access evicts p_0, the fourth victim access evicts p_3. By reversing the access order in the probe step, the attacker maximizes the number of cache hits on the prime set, ensuring that the probe step evicts the victim data last. For a single victim access, probing in the order p_3, p_2, p_1, p_0 only results in a cache miss for the last access, i.e., to p_0. In contrast, probing in the same order as the prime step results in 4 cache misses.

Figure 6 shows the execution time of this reverse-probe step. The number of victim accesses that fall into the targeted cache set is distinguishable. This property can be used for a covert channel, as a single probe step can distinguish 5 different symbols (0 to 4 evicted ways) for the 4-way cache of the C906. The spatial resolution is one cache set, i.e., 256 B, and the temporal resolution is 392 ns. This attack is not possible on the U74 but transfers to all CPUs that use a fully-deterministic last-recently-used replacement policy.

TLB Eviction. An attacker that can evict the TLB can spy on address translations performed by a victim process. We use the eviction strategy found during our systematic analysis in Section 3.3 to evict entries from the TLB. We achieve perfect eviction with an F-score of 1.

Simple Branch Prediction Analysis. As described by Acıiçmez et al. [2], an attacker can influence the branch predictor in the attacker application to affect the predictions of the victim, introducing timing differences. We verify that this attack is also possible on the analyzed CPUs. In the attacker application, we train a branch to consistently predict the same direction, e.g., taken. As a result, a secret-dependent branch in the victim that matches the trained direction is faster. By measuring the execution time of either the victim or the branches in the attacker, an attacker can infer whether the trained branch direction matches the one of the victim's execution. We extract the victim branch direction with an F-score of 0.94 and 1 over 1 000 000 samples on the C906 and U74, respectively.

Branch Shadowing. The BHT can also be used in branch-shadowing attacks [21], leaking the secret if the victim program executes a secret-dependent branch. After the execution of a secret-dependent branch in the victim program, the BHT contains whether the victim branch was taken or not. The attacker executes a jump with a fixed direction that maps to the same BHT entry as the branch predicted based on the victim's secret. By measuring the timing of the branch in the attacker process, an attacker can infer whether the victim took the branch. We extract the branch information over 1 000 000 samples with an F-score of 0.94 (C906) and 1 (U74).

TABLE 1: Papers using similar case studies for evaluation.

Case Study	Paper
Square and Multiply attack on MbedTLS	[77], [101], [43], [55], [26], [72], [61]
AES T-Table attack on OpenSSL	[9], [69], [36], [48], [59], [35], [19], [92], [60], [11], [73]
KASLR break	[42], [49], [22], [53], [12], [76], [81], [57], [95]
Interrupt Detection	[67], [18], [58], [103]

5. Case Studies

We show the applicability of our evaluated attacks in 6 case studies. These studies demonstrate the attack primitives introduced in Section 4. The case studies are chosen in line with previous papers to make our results comparable. Table 1 lists papers that used similar case studies on other architectures, enabling comparison with our results. In Section 5.6 we attack the OpenSSL 1.0.1e AES T-Table implementation with Flush+Reload, Prime+Probe, and Evict+Reload. Section 5.1, uses Flush+Fault to attack a vulnerable RSA implementation as found in MbedTLS 1.13.10. We show the first KASLR breaks on RISC-V using timing and retired instructions in Section 5.2. In two case studies, we demonstrate the security implications of the unprivileged `rdinstret` instruction by bypassing branch shadowing mitigations in Section 5.3 and leaking the contents of a read-protected folder in Section 5.4. Section 5.5 uses CycleDrift to infer kernel activity, detecting network interrupts.

Methodology. We structure our case study by attack primitive. Firstly we evaluate the novel primitives discovered in Sections 4.1 to 4.3 in Sections 5.1 to 5.5. We then reproduce the well-known cache attack primitives from Section 4.4 by performing AES T-Table attack in Section 5.6.

5.1. Flush+Fault on Square and Multiply

This case study demonstrates Flush+Fault to recover the control flow of a victim, thereby leaking the private key of an RSA implementation as found in MbedTLS version 1.3.10. We attack MbedTLS, as this library is a common attack target that researchers use to demonstrate their side-channel attacks [77], [101], [43], [55], [26], [72], [61], thus allowing better comparisons with related work. More specifically, our attack leaks a 2048-bit RSA private key by attacking the square-and-multiply implementation of the shared library.

Overview. Our attack works by exploiting that the victim square-and-multiply implementation leaves different traces in the CPU cache depending on the secret bits of the private key. More precisely, our attacker, sharing the I-Cache cache with the victim, monitors a cache line that is only cached when the current secret bit of the exponent is a '1' bit. Thus, we can infer that the bit was '1' if we see a cache hit and that the bit was '0' otherwise. As the C906 does not have shared instruction and data caches, we have to rely on Flush+Fault instead of the normal Flush+Reload. As the U74 does not provide an unprivileged flush instruction, we rely on the fence instruction as in Fence+Reload to mount Flush+Fault.

Threat Model. We assume an unprivileged attacker running on the same system as the victim decryption routine.

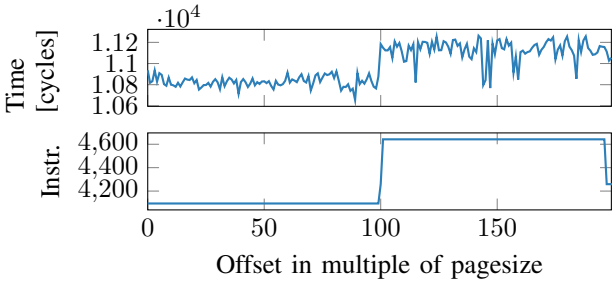

Figure 7: Access time (top, C906) and retired instructions (bottom, U74) for mapped and unmapped kernel pages

The attacker is given oracle access to the victim routine, i.e., can decrypt arbitrarily often. Furthermore, we assume that the attacker and the victim share their I-Cache.

Evaluation. We evaluate our attack by leaking the entire 2048-bit private key from the shared library using the `ret` instruction-based Flush+Fault. Figure 17 shows the signal on the C906 for the first 20 bits of the key, obtained by decrypting the same message 100 times and plotting the minimal measurement for each key bit. Faster access times, corresponding to '1' bits, can be clearly distinguished from the slower access times corresponding to '0' bits. Using the signal from these 100 iterations, we leak all 2048 bits of the RSA key with an accuracy of $99.9\,\%$ (n = 100) on both the C906 and U74. More precisely, our experiments leak on average 2046.3 (on the U74) and 2046.1 (on the C906) key bits. The remaining two bits can easily be brute-forced by an attacker by testing $\binom{2048}{2} \approx 2^{21}$ combinations.

5.2. KASLR Break

This case study presents the first microarchitectural KASLR break on RISC-V. Our KASLR break exploits the side-channel leakage from the page-table walk to infer whether a kernel address is physically backed. While fine-grained per-function randomization for kernel code is in development [54], the current coarse-grained KASLR only randomizes the starting position of kernel code [12]. Thus, to derandomize the kernel location, it is sufficient to detect where a known kernel function, e.g., the start of the kernel image, is mapped [12].

Overview. Both analyzed CPUs use 39-bit virtual addresses, which use 3 levels of page tables to translate virtual to physical addresses [98]. Both CPUs show different timing behavior depending on the page-table level on which a page fault is resolved. If an entire range of virtual addresses is not mapped, the page-table walk can already abort at an early level; there are no further levels for which permissions must be checked. In contrast, if a virtual address is physically backed, the page-table walk has to walk all levels and can only abort at the final page-table level due to the permission check. We measure the time it takes to execute a load instruction on kernel memory and handle the fault using a signal handler. The timing determines whether a kernel virtual address is physically backed. Additionally, on the

U74, the aborts on different page-table levels lead to a different number of retired instructions.

Threat Model. We assume an unprivileged attacker and no kernel-level mitigations against microarchitectural KASLR breaks, such as KAISER [32], LAZARUS [28], or FLARE [12]. This is a realistic assumption, as all these countermeasures are implemented for x86 systems using specific properties of the x86 architecture.

Evaluation. In line with microarchitectural KASLR breaks on x86, we scan over the entire kernel address space to identify mapped and unmapped pages. Trying to load data from a mapped page (and handling the resulting fault using a signal handler) results in a higher access time (and in the case of the U74, more retired instructions) due to the longer page-table walk. Figure 7 shows the different behavior of mapped and unmapped pages on the C906 and U74. On the C906, testing one page takes $1456\,\mu s$ and reveals whether the page is physically backed with an accuracy of $100\,\%$. On the U74, we implement two different KASLR breaks using access times and retired instructions. Testing one page takes $125\,\mu s$ independent of the used method. A full KASLR break, testing all 512 possible locations of the kernel [12], [56] takes $745.4\,\mathrm{ms}$ on the C906 and $54\,\mathrm{ms}$ on the U74. We note that this could be further optimized by aborting when the kernel is found instead of always testing all locations. Both methods have $100\,\%$ accuracy.

5.3. Zigzagger Bypass

This case study shows that the Zigzagger [55], [40] branch-shadowing mitigation can be bypassed using the `rdinstret` instruction. Zigzagger is a compiler-based mitigation that obfuscates conditional branches by replacing them with a series of indirect (decoy) jumps and conditional moves. The main idea is that an indirect jump does not leak its target via the branch predictor.

Overview. We show that the availability of the performance counter for retired instructions, specified in the RISC-V ISA [97], bypasses the Zigzagger mitigation. We use a ported version of the Zigzagger example from SGX-Step [90] (cf. Listing 2 in Appendix A). We run all possible branches of the Zigzagger code, profiling the number of retired instructions. Different arguments to the Zigzagger code result in differences in the retired instruction count. Even constant-time versions of the Zigzagger mitigation [40] have differences in instruction counts [66]. Similarly to Moghimi et al. [66], we show that these differences can be exploited via a side channel.

Threat Model. We assume an unprivileged attacker that can access the retired-instructions performance counter, e.g., via the unprivileged `rdinstret` instruction. The victim code under attack uses the Zigzagger mitigation. As previous work showed, an unprivileged attacker can preempt a victim [37], [8]. Thus, in line with recent work [15], we assume that the attacker and victim take turns.

Evaluation. The 3 different cases for invoking the protected function have a distinct number of retired instructions on the C906 and the U74, making them distinguishable. Due to the deterministic architectural information, the attack has a perfect F-score of 1 for distinguishing all branch combinations. Using a timing-based side channel decreases the F-score to 0.99 in the tested simple Zigzagger code. For more complex code, e.g., with variable-latency instructions, we expect more noise in the latency domain, implying less precise results. The attack has a temporal resolution of $111\,\mathrm{ns}$ on the C906 and $551\,\mathrm{ns}$ on the U74.

Comparison to CopyCat [66]. Our attack achieves the same goal as CopyCat on Intel CPUs. While CopyCat also exploits the number of retired instructions to break the Zigzagger mitigation, CopyCat requires a privileged attacker and targets code running inside the SGX trusted execution environment. Our unprivileged version of this attack on RISC-V works on a broader range of targets.

5.4. Leaking Contents of a Drop-Box Folder

In this case study, we demonstrate that the unprivileged `rdinstret` instruction can be exploited to leak the files inside a write-only (drop-box) folder. Such a folder has its permissions configured so that a user can copy items to the folder but cannot see the contents of the folder, which are only visible to folder owners [5]. As the `rdinstret` instruction also counts the number of retired instructions of the kernel, and file-handling code in the kernel is not implemented using constant-time algorithms, we can see differences when opening non-existing files and files with mismatching permissions.

Overview. In our setup, we use such a drop-box folder by setting the permissions so that only the owner of the folder has read access to the folder. This prevents anyone else from listing the directory content and only allows the creation of files in the folder. By configuring the drop-box directory such that added files inherit the directory's owner, all users can write to the folder, but only the owner can read its contents. We show that using the `rdinstret` instruction, it is possible to detect whether a given file is stored inside the drop-box folder. An attacker tries to open it directly through the `openat` system call to detect if a file exists within the drop-box folder. If the requested file exists, the operating system continues with the permission check, executing more instructions. If the file does not exist, the operating system can directly abort.

Threat model. We assume the attacker can access a drop-box folder and read the unprivileged `instret` performance counter. The attack does not require any privileges or any specific libc version.

Evaluation. The number of retired instructions increases from 3647 to 4491 on the C906 and 4266 to 5117 on the U74 if a file exists. The number of instructions is extremely stable since no microarchitectural effects with unpredictable timings, such as caching, are involved in the attack. We rely on two measurements to rule out potential interrupts and rescheduling of the attacker. With this setup, we achieve an F-score of 1. Each access takes $2.7\,\mathrm{ms}$ on the C906 and $0.33\,\mathrm{ms}$ on the U74. As a comparison, we evaluate the same attack with the runtime (`rdcycle`), which leads to

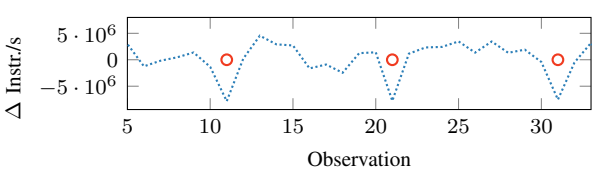

Figure 8: Network interrupt detection on the C906. The red dots indicate when a network request was sent to the board.

a noisier attack due to caching effects. When using timing, the F-score is 0.7 on the U74 and 0.2 on the C906.

5.5. Interrupt Detection via CycleDrift

This case study shows that CycleDrift detects asynchronous operating-system events, such as interrupts. Previous work showed that detecting interrupts is a privacy risk, as they can be used to spy on entered text [67], [18], [58] or to fingerprint visited websites [58], [103].

Overview. As discussed in Section 4.3, the `rdinstret` instruction also counts the number of retired instructions of the kernel. Hence, by looking at this number of retired instructions over a constant attacker-chosen time interval, an attacker can infer asynchronous events on the current CPU core. Similar to Zhang et al. [103], we use the number of retired instructions within a fixed coarse-grained time interval of 1 s. The attacker periodically records the number of retired instructions while executing instructions requiring a minimal amount of cycles. As the attacker code is known and deterministic, the number of retired instructions only depends on *other* code executed on the same CPU core. If an interrupt is executed on the CPU core, the number of retired instructions decreases, depending on the number of high-latency instructions in the interrupt handler.

Threat Model. We assume the attacker can execute native code on the CPU that handles interrupts. This is a realistic assumption, as the attacker can change the program affinity. The attacker does not require any privileges or a high-resolution counter as in previous work [78].

Evaluation. Figure 8 shows the difference of retired instructions to the average number of retired instruction within 5 s on the C906. We send network requests to the C906, marked as red circles in the plot. When the board receives such a network request, an interrupt is triggered and handled by the kernel. In the plot, this can be seen as a downward spike. This method detects interrupts reliably.

5.6. AES T-Table Attacks

In this section, we show that Evict+Reload and Prime+Probe attacks described in Section 3.2 on the AES T-tables implementation can be successfully mounted on both the C906 and U74. We additionally evaluate a Flush+Reload attack on the C906, as the unprivileged data-cache flushing is unavailable on U74. Although this AES implementation

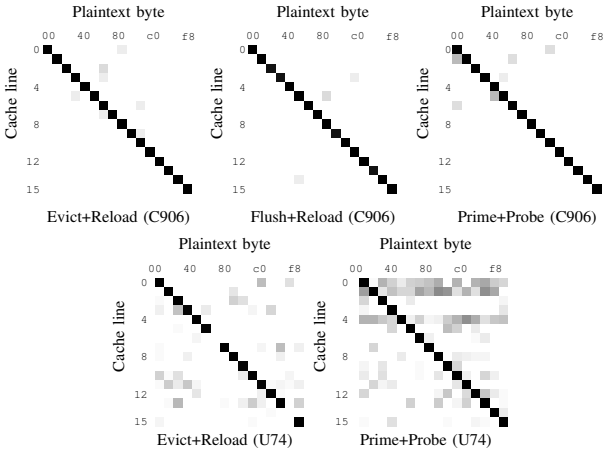

Figure 9: Cache hits on the first T-table on C906 (300 encryptions) and on U74 (1000 encryptions). Darker means more cache hits. k_0 = `0x00`.

is deprecated by OpenSSL, it is a well-known standard benchmark to show the performance of cache attacks [9], [69], [36], [48], [59], [35], [19], [92], [60], [11], [73].

Overview. In line with previous work, we use the vulnerable OpenSSL version 1.0.1e. In the T-table-based AES implementation, accesses to the T-table purely depend on the plaintext and secret key. Specifically, T-table cache-line accesses are made to entries $T_j[p_i \oplus k_i]$ with $i \equiv j \bmod 4$ and $0 \leq i < 16$ at the first round of encryption. Therefore, cache line $T_j[p_i \oplus k_i]$ is accessed in 100 % of the encryptions. For the other p_j, the remaining addresses are accessed in 92.5 % of the encryptions. Therefore, such an access pattern allows attackers to use cache attacks to derive values for $p_i \oplus k_i$, and the specific value of k_i with a chosen plaintext. As with prior work [35], we assume that an attacker knows the T-table addresses. We do not perform a full secret-key-recovery attack, as we only illustrate how traditional cache attacks can be reproduced on RISC-V CPUs.

Threat Model. We assume that the targeted cryptographic library uses a vulnerable AES T-table implementation. Furthermore, if Flush+Reload or Evict+Reload are used, the T-table must reside in shared memory. If Prime+Probe is used, the T-table can reside in non-shared memory. We do not assume any privileges for the attacker.

Evaluation. Figure 9 compares the used cache-attack primitives on both CPUs. The access patterns of T-table entries are generated by randomly chosen plaintexts and a fixed key value `0x00`. Similar results have been shown on Intel, AMD, and ARM devices [36], [35], [59], [92]. With only 300 encryptions, the cache line expected to be accessed in 100 % of the encryptions is clearly distinguishable on the C906. The attacker needs more encryptions to reproduce attacks on the U74 due to a shared L2 cache and the random replacement strategy. The attacks can derive the secret value, making the attack primitives usable in realistic scenarios.

TABLE 2: Systematic analysis of microarchitectural attacks and attack primitives on the C906 and U74, including proposed hardware countermeasures that could prevent them.

	Primitive	C906	U74	Hardware Countermeasures
Generic	High-resolution timer	✓	✓	Privileged timer [37], Fuzzy time [41], low-resolution timers [68]
	TLB eviction	✓	✓	Partitioned TLB [30], Random-Fill TLB [17]
	Branch Pred. Evict	✓	✓	Secure branch predictors [104]
	Performance counters	✓	✓	Privileged counters [38]
Attacks	Flush+Reload/Flush+Fault	✓	✗	Privileged flush [102]
	Flush+Flush	~	✗	Constant-time flush [35]
	Evict+Reload	✓	✓	Randomized caches [99], [74], [89], cache partitioning [75], [96], [62]
	Prime+Probe	✓	✓	Randomized caches [99], [74], [89], cache partitioning [75], [96], [62]
	Cache+Time	✓	✓	Cache partitioning [75], [96], [62]
	CycleDrift	✓	✓	Privileged counters [38]

~ cannot distinguish cache hit from miss for non-modified cache line

6. Discussion

This section discusses design decisions of the RISC-V ISA and the specifics of the C906 and U74 that enable or foster microarchitectural attacks. We argue that several proposals for hardware mitigations against such attacks would have been relatively easy to integrate into these CPUs.

RISC-V ISA Design. The RISC-V ISA is not specific regarding implementation details [97]. Therefore, the base instruction set does not include functions that interact with the microarchitecture, such as cache-maintenance instructions. The ISA reserves an opcode range for custom instructions to implement such functionality.

Still, the ISA mandates unprivileged instructions that provide insight into the microarchitecture. Although performance counters are not configurable in user space, a subset of them is always accessible [97]. This contrasts the design of x86 and ARM, in which performance counters are privileged and can only be made unprivileged by the operating system. Linux provided unprivileged read access to some performance counters for some time. However, since these interfaces have been exploited for microarchitectural attacks [59], [10], they are only available to privileged users on many distributions, or at least limited to the scope of one's own application. The RISC-V ISA manual argues that access to these counters is mandatory for performance analysis and optimizations [97]. Other new designs completely remove the option for unprivileged performance counter [38] and cycle-accurate timer access.

We see two unprivileged instructions as especially problematic: the `rdcycle` and `rdinstret` instructions. These two instructions allow reading the timestamp counter and the number of retired instructions and are available to unprivileged attackers. The timestamp counter has been used for nearly all microarchitectural attacks and also in this paper. Thus, when designing a new ISA, it would have been advisable to provide at least an option to prevent access to the timestamp counter for unprivileged applications, as is the case for x86 [45] and ARM [59]. Even better, this instruction should be privileged by default or not available for unprivileged code, as for the Apple M1 [39]. Additionally, `rdinstret` provides the number of retired instructions of the entire CPU core to an unprivileged attacker. As shown in Section 5.2, the `rdinstret` instruction allows for a reliable KASLR break. In addition, the amount of retired instructions leaks information from a higher-privileged security domain, i.e., the kernel, to a lower-privileged security domain, i.e., the user space. In Section 5.5, this is exploited to leak interrupt behavior to a user-space application. While such leakage is possible based on timing primitives, it is significantly amplified and made architectural by the `rdinstret` instruction. Therefore, this instruction deliberately weakens the isolation boundary between the user and kernel space. We expect that this instruction can also become problematic in virtualized contexts.

Individual CPU Design. Several design decisions in the C906 and U74 are problematic regarding microarchitectural attacks. First, the addition of the custom unprivileged flush instructions on the C906. Adding privileged flush instructions as present on the U74 is unproblematic from a security perspective, and it can be necessary for specific functionality, such as interaction with memory-mapped devices. However, we do not see the need for an unprivileged flush instruction. Unprivileged flush instructions have been exploited for different microarchitectural attack primitives [102], [51], [35], [20]. The decision to make *some* flush instructions privileged suggests that at least some security implications were considered when designing the custom instruction set extension. As shown by the ARMv7 [59] and Apple M1 [39] CPUs, it is possible not to expose any flush instruction to the user space. Second, the choice of the cache-replacement policy on the C906 and to a lesser extent on the U74 are problematic for microarchitectural security. With the fully-deterministic LRU policy on the C906, eviction-based attacks are reliable and noisefree. Modern policies often use (pseudo)randomness in the replacement strategy [1]. This (pseudo)randomness is introduced for performance reasons [1], but has the side effect of complicating eviction-based attacks [59], [34], [93]. Hence, using a (pseudo)random replacement policy as on ARM [59] could impede attacks without adding significant complexity to the implementation or decreasing the performance. While the U74 uses a random replacement policy, it only does so in the L1 I-Cache and L2 Cache. As the L1 I-Cache can be flushed using the `fence.i`, the random replacement policy in the U74 only mitigates cross-core attacks on the shared L2 Cache. Third, although limited, the branch prediction implemented on the C906 and U74 still is problematic. From previous attacks, such as Spectre [52], it should be clear that speculation is problematic for security, and mechanisms to control and abort speculation [44], [52], [13] should be provided when adding speculation to a CPU.

Hardware Mitigations. Table 2 shows the results of our systematic side-channel evaluation on the C906 and U74. For some attacks and attack primitives, hardware countermeasures have been proposed in the past. A commonly proposed countermeasure is to make instructions, such as flush, privileged [71] or execute in constant time [35]. Eviction-based side channels can be impeded by using randomization [17], [99], [74], [89], [44], [104] or partitioning [75], [96], [62]. These countermeasures prevent or significantly impede our discussed attacks. Many of these countermeasures do not have any performance overhead [37], [41], [71] or even improve performance in some scenarios [99]. Also, the implementation complexity of these mitigations is realistic and well-evaluated for a small to medium-sized CPU such as the C906 and U74. Still, no such mitigation is implemented, showing that there needs to be more focus on security for CPU implementations. We hope that future work can show that the proposed hardware countermeasures can significantly improve the security of RISC-V CPUs.

Other Boards and Cores. All currently available boards running a full Linux distribution use either the C906 or one of the Freedom cores from SiFive [24]. For this paper, we use all hardware cores that are currently (March 2023) buyable and run a full-fledged 64-bit Linux distribution. Other existing RISC-V cores either only exist as FPGA designs or lack the capabilities to run a full-fledged Linux distribution. Using hardware cores has the advantage that findings are valid for all instances. In contrast, FPGA-based RISC-V cores might have differences with respect to side channels caused by the synthetization process.

7. Conclusion

This paper analyzed the XuanTie C906 and SiFive U74, two of the first COTS 64-bit RISC-V CPUs. In our systematic evaluation of their design, we discovered 3 new attack techniques based on RISC-V-specific design and implementation decisions. We showed that known attacks, such as Flush+Reload (or the new variant Flush+Fault) and Prime+Probe apply to these CPUs. In 6 case studies, we showed that we can leak keys from AES and RSA implementations, break KASLR, leak file names from inaccessible write-only folders, bypass the Zigzagger mitigation, and detect interrupt handling. We stress that reducing the attack surface by hardening an ISA and the microarchitecture should be part of CPU designs to ensure the security of future devices.

Acknowledgment

We want to thank our shepherd and the anonymous reviewers for their comments and suggestions. We also want to thank Andreas Kogler for fruitful discussions. This work was supported in part by Semiconductor Research Corporation (SRC) Hardware Security Program (HWS).

References

[1] A. Abel and J. Reineke, "Reverse engineering of cache replacement policies in intel microprocessors and their evaluation," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[2] O. Acıiçmez, c. K. Koç, and J.-p. Seifert, "On the Power of Simple Branch Prediction Analysis," in *AsiaCCS*, 2007.

[3] O. Acıiçmez, J.-P. Seifert, and c. K. Koç, "Predicting secret keys via branch prediction," in *CT-RSA*, 2007.

[4] M. M. Ahmadi, F. Khalid, and M. Shafique, "Side-channel attacks on risc-v processors: Current progress, challenges, and opportunities," *arXiv preprint*, 2021.

[5] Apple, "macOS Sierra: Set permissions for items on your Mac," 2019. [Online]. Available: https://support.apple.com/kb/PH25287

[6] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The rocket chip generator," *EECS Berkley*, 2016.

[7] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for risc-v," *University of California Berkeley*, 2014.

[8] C. Ashokkumar, R. P. Giri, and B. Menezes, "Highly efficient algorithms for aes key retrieval in cache access attacks," in *EuroS&P*, 2016.

[9] D. J. Bernstein, "Cache-Timing Attacks on AES," 2005. [Online]. Available: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[10] S. Bhattacharya, C.-m.-t.-n. Maurice, S. Bhasin, and D. Mukhopadhyay, "Template Attack on Blinded Scalar Multiplication with Asynchronous perf-ioctl Calls," *Cryptology ePrint Archive, Report 2017/968*, 2017.

[11] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks," in *USENIX Security Symposium*, 2020.

[12] C. Canella, M. Schwarz, M. Haubenwallner, M. Schwarzl, and D. Gruss, "KASLR: Break It, Fix It, Repeat," in *AsiaCCS*, 2020.

[13] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *USENIX Security Symposium*, 2019, extended classification tree and PoCs at https://transient.fail/.

[14] C. Celio, D. A. Patterson, and K. Asanović, "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor," Tech. Rep., 2015.

[15] M. Dai, R. Paccagnella, M. Gomez-Garcia, J. McCalpin, and M. Yan, "Don't mesh around: Side-Channel attacks and mitigations on mesh interconnects," in *USENIX Security Symposium*, 2022.

[16] Debian Wiki, "ASIC implementations, i.e. "real" CPU chips," 2022. [Online]. Available: https://wiki.debian.org/RISC-V#ASIC_implementations.2C_i.e._.22real.22_CPU_chips

[17] S. Deng, W. Xiong, and J. Szefer, "Secure TLBs," in *ISCA*, 2019.

[18] W. Diao, X. Liu, Z. Li, and K. Zhang, "No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis," in *S&P*, 2016.

[19] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX," in *USENIX Security Symposium*, 2017.

[20] C. Easdon, M. Schwarz, M. Schwarzl, and D. Gruss, "Rapid Prototyping for Microarchitectural Attacks," in *USENIX Security*, 2022.

[21] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Covert channels through branch predictors: a feasibility study," in *HASP*, 2015.

[22] ——, "Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR," in *MICRO*, 2016.

[23] A. Fog, "The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers," 2016.

[24] R.-V. Foundation, "RISC-V Exchange: Available Boards," 2022. [Online]. Available: https://riscv.org/exchanges/boards/

[25] F. A. Fuchs, J. Woodruff, S. W. Moore, P. G. Neumann, and R. N. Watson, "Developing a Test Suite for Transient-Execution Attacks on RISC-V and CHERI-RISC-V," 2021.

[26] C. P. García, S. Ul Hassan, N. Tuveri, I. Gridin, A. C. Aldaya, and B. B. Brumley, "Certified side channels," in *USENIX Security Symposium*, 2020.

[27] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware," *Journal of Cryptographic Engineering*, 2016.

[28] D. Gens, O. Arias, D. Sullivan, C. Liebchen, Y. Jin, and A.-R. Sadeghi, "LAZARUS: Practical Side-Channel Resilient Kernel-Space Randomization," in *RAID*, 2017.

[29] A. Gonzalez, B. Korpan, J. Zhao, E. Younis, and K. Asanović, "Replicating and Mitigating Spectre Attacks on a Open Source RISC-V Microarchitecture," in *Third Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2019.

[30] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in *USENIX Security Symposium*, 2018.

[31] M. Green, L. Rodrigues-Lima, A. Zankl, G. Irazoqui, J. Heyszl, and T. Eisenbarth, "AutoLock: Why Cache Attacks on ARM Are Harder Than You Think," in *USENIX Security Symposium*, 2017.

[32] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is Dead: Long Live KASLR," in *ESSoS*, 2017.

[33] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR," in *CCS*, 2016.

[34] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," in *DIMVA*, 2016.

[35] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," in *DIMVA*, 2016.

[36] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches," in *USENIX Security Symposium*, 2015.

[37] D. Gullasch, E. Bangerter, and S. Krenn, "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice," in *S&P*, 2011.

[38] M. Handley, "M1 Exploration - v0.70," 2021.

[39] L. Hetterich and M. Schwarz, "Branch Different - Spectre Attacks on Apple Silicon," in *DIMVA*, 2022.

[40] S. Hosseinzadeh, H. Liljestrand, V. Leppänen, and A. Paverd, "Mitigating Branch-Shadowing Attacks on Intel SGX using Control Flow Randomization," *arXiv:1808.06478*, 2018.

[41] W.-M. Hu, "Reducing Timing Channels with Fuzzy Time," *Journal of Computer Security*, 1992.

[42] R. Hund, C. Willems, and T. Holz, "Practical Timing Side Channel Attacks against Kernel Space ASLR," in *S&P*, 2013.

[43] 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," in *CHES*, 2020.

[44] Intel, "Speculative Execution Side Channel Mitigations," 2018, revision 3.0.

[45] ——, "Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide," 2019.

[46] Intel Corporation, "Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations," 2020. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html

[47] ——, "Data Operand Independent Timing Instruction Set Architecture (ISA) Guidance," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html

[48] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES," in *S&P*, 2015.

[49] Y. Jang, S. Lee, and T. Kim, "Breaking Kernel Address Space Layout Randomization with Intel TSX," in *CCS*, 2016.

[50] B. Johannesmeyer, J. Koschel, K. Razavi, H. Bos, and C. Giuffrida, "Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel," in *NDSS*, 2022.

[51] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.

[52] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *S&P*, 2019.

[53] J. Koschel, C. Giuffrida, H. Bos, and K. Razavi, "TagBleed: Breaking KASLR on the Isolated Kernel Address Space Using Tagged TLBs," in *EuroS&P*, 2020.

[54] M. Larabel, "FGKASLR Patches Revised A 10th Time For Improving Linux Kernel Security," 2022. [Online]. Available: https://www.phoronix.com/news/FGKASLR-Linux-v10

[55] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing," in *USENIX Security Symposium*, 2017.

[56] Z. Li, "Support KASLR for RISC-V," 2020. [Online]. Available: https://lwn.net/Articles/815891/

[57] M. Lipp, D. Gruss, and M. Schwarz, "AMD Prefetch Attacks through Power and Time," in *USENIX Security*, 2022.

[58] M. Lipp, D. Gruss, M. Schwarz, D. Bidner, C.-m.-t.-n. Maurice, and S. Mangard, "Practical Keystroke Timing Attacks in Sandboxed JavaScript," in *ESORICS*, 2017.

[59] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache Attacks on Mobile Devices," in *USENIX Security Symposium*, 2016.

[60] M. Lipp, V. Hadžić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, "Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors," in *AsiaCCS*, 2020.

[61] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "PLATYPUS: Software-based Power Side-Channel Attacks on x86," in *S&P*, 2020.

[62] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *HPCA*, 2016.

[63] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *S&P*, 2015.

[64] X. Lou, T. Zhang, J. Jiang, and Y. Zhang, "A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography," *ACM CSUR*, 2021.

[65] P. Mata and N. Rao, "Flush-reload attack and its mitigation on an fpga based compressed cache design," in *International Symposium on Quality Electronic Design*, 2021.

[66] D. Moghimi, J. V. Bulck, N. Heninger, F. Piessens, and B. Sunar, "CopyCat: Controlled Instruction-Level Attacks on Enclaves for Maximal Key Extraction," in *USENIX Security Symposium*, 2020.

[67] J. Monaco, "SoK: Keylogging Side Channels," in *S&P*, 2018.

[68] Mozilla, "performance.now resolution," 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Performance/now

[69] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: the Case of AES," in *CT-RSA*, 2006.

[70] C. Percival, "Cache Missing for Fun and Profit," in *BSDCan*, 2005.

[71] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *USENIX Security Symposium*, 2016.

[72] I. Puddu, M. Schneider, M. Haller, and S. Čapkun, "Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend," in *USENIX Security Symposium*, 2021.

[73] A. Purnal, F. Turan, and I. Verbauwhede, "Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks," in *CCS*, 2021.

[74] G. Saileshwar and M. Qureshi, "MIRAGE: Mitigating conflict-based cache attacks with a practical fully-associative design," in *USENIX Security Symposium*, 2021.

[75] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and efficient fine-grain cache partitioning," in *Proceedings of the annual international symposium on Computer architecture*, 2011.

[76] M. Schwarz, C. Canella, L. Giner, and D. Gruss, "Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs," *arXiv:1905.05725*, 2019.

[77] M. Schwarz, D. Gruss, S. Weiser, C. Maurice, and S. Mangard, "Malware Guard Extension: Using SGX to Conceal Cache Attacks," in *DIMVA*, 2017.

[78] M. Schwarz, M. Lipp, D. Gruss, S. Weiser, C. Maurice, R. Spreitzer, and S. Mangard, "KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks," in *NDSS*, 2018.

[79] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript," in *FC*, 2017.

[80] M. Schwarzl, P. Borrello, A. Kogler, K. Varda, T. Schuster, D. Gruss, and M. Schwarz, "Robust and scalable process isolation against spectre in the cloud," in *ESORICS*, 2022.

[81] M. Schwarzl, T. Schuster, M. Schwarz, and D. Gruss, "Speculative Dereferencing of Registers: Reviving Foreshadow," in *FC*, 2021.

[82] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *CCS*, 2004.

[83] SiFive, "HF105 Datasheet," 2022. [Online]. Available: https://sifive.cdn.prismic.io/sifive/d0556df9-55c6-47a8-b0f2-4b1521546543_hifive-unmatched-datasheet.pdf

[84] ——, "U74 core complex manual," 2022. [Online]. Available: https://sifive.cdn.prismic.io/sifive/ad5577a0-9a00-45c9-a5d0-424a3d586060_u74_core_complex_manual_21G3.pdf

[85] Sipeed, "RISC-V 64bit chip (C910) run Android 10," 2022. [Online]. Available: https://twitter.com/SipeedIO/status/1457529282134089734

[86] J. Szefer, "Survey of microarchitectural side and covert channels, attacks, and defenses," *Journal of Hardware and Systems Security*, vol. 3, no. 3, pp. 219–234, 2019.

[87] T-Head, "C906," 2022. [Online]. Available: https://www.t-head.cn/product/c906

[88] T-Head Semiconductor, "Xuantie c906 r1s0 user manual," 2022. [Online]. Available: https://dl.linux-sunxi.org/D1/Xuantie_C906_R1S0_User_Manual.pdf

[89] Q. Tan, Z. Zeng, K. Bu, and K. Ren, "PhantomCache: Obfuscating Cache Conflicts with Localized Randomization," in *NDSS*, 2020.

[90] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control," in *Workshop on System Software for Trusted Execution*, 2017.

[91] 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," in *USENIX Security Symposium*, 2017.

[92] S. Van Schaik, C. Giuffrida, H. Bos, and K. Razavi, "Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think," in *USENIX Security Symposium*, 2018.

[93] P. Vila, B. Köpf, and J. Morales, "Theory and Practice of Finding Eviction Sets," in *S&P*, 2019.

[94] L. Wagner, "Mitigations landing for new class of timing attack," 2018.

[95] Y. Wang, R. Paccagnella, E. He, H. Shacham, C. W. Fletcher, and D. Kohlbrenner, "Hertzbleed: Turning power side-channel attacks into remote timing attacks on x86," in *USENIX Security Symposium*, 2022.

[96] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, 2007.

[97] A. Waterman and K. Asanović, "The RISC-V Instruction Set Manual, Vol. I: Unprivileged ISA, Version 20191213," 2019.

[98] A. Waterman, K. Asanović, and J. Hauser, "The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Document Version 20211203," 2021.

[99] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "ScatterCache: Thwarting Cache Attacks via Cache Set Randomization," in *USENIX Security Symposium*, 2019.

[100] Xcalibyte, "Roma Laptop Pre-order," 2022. [Online]. Available: https://xcalibyte.com.cn/en/roma-preorder/

[101] Y. Xiao, M. Li, S. Chen, and Y. Zhang, "Stacco: Differentially Analyzing Side-channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves," in *CCS*, 2017.

[102] Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security Symposium*, 2014.

[103] R. Zhang, T. Kim, D. Weber, and M. Schwarz, "(M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels," in *USENIX Security*, 2023.

[104] T. Zhang, T. Lesch, K. Koltermann, and D. Evtyushkin, "STBPU: A Reasonably Secure Branch Prediction Unit," in *DSN*, 2022.

Appendix

Evaluation of Timing Primitives

Table 3 shows the resolution and increment of all available timing primitives on the C906 and U74. In addition, we list whether the primitives are accessible to an unprivileged attacker and can be exploited in microarchitectural attacks.

TABLE 3: C906 and U74 timing primitives (Debian).

Timer	Unprivileged		Exploitable		Resolution		Increment	
	C906	U74	C906	U74	C906	U74	C906	U74
`rdcycle`	✓	✓	✓	✓	1 ns	1 ns	1	1
`csrr reg, cycle`	✓	✓	✓	✓	1 ns	1 ns	1	1
clock_gettime	✓	✓	✓	✓	1 ns	1 ns	1	1
counting thread	✓	✓	✓*	✓*	2 ns	1 ns	1	1
`rdinstret`	✓	✓	✓†	✓†	3 ns	1 ns	1	1
`csrr reg, instret`	✓	✓	✓†	✓†	3 ns	1 ns	1	1
`rdtime`	✓	✓	✓	✓	45 ns	45 ns	1	1
`csrr reg, time`	✓	✓	✓	✓	45 ns	45 ns	1	1

* requires an additional CPU core for the timer † not a generic timer, but can be used for certain attacks

TABLE 4: Cache maintenance instructions on the devices.

	Instruction	Unpriv.	Target	Addressing	Exec. Time
C906	dcache.civa	✓	D-Cache	Virtual address	4 cycles
	dcache.ciall	✗	D-Cache	Entire cache	-
	dcache.cipa	✗	D-Cache	Physical address	-
	dcache.cisw	✗	D-Cache	Way and set	-
	icache.iva	✓	I-Cache	Virtual address	4 cycles
	icache.iall	✗	I-Cache	Entire cache	-
	icache.ipa	✗	I-Cache	Physical address	-
	fence.i	✓	I-Cache	Entire cache	914 cycles
U74	clflush.D.L1	✗	D-Cache	Entire cache/Virtual address	-
	cldiscard.D.L1	✗	D-Cache	Entire cache/Virtual address	-
	fence.i	✓	I-Cache	Entire cache	28 cycles

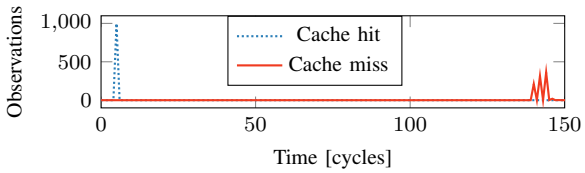

Figure 10: Flush+Reload histogram (C906) for the D-Cache showing cache hits (dotted blue) and misses (solid red).

Cache Maintenance Instructions

Table 4 shows the cache maintenance instructions available on both the C906 and U74. For each instruction, we list whether it can be accessed from an unprivileged context, the cache it targets and the addressing mode that is used.

Padded Square and Multiply

Listing 1 shows a padded square and multiply implementation that cannot be distinguished by timing but only by the number of retired instructions.

```
1 int sqmult(int bit, int a, int b) {
2     a *= a;
3     if(bit) a *= b;
4     else asm volatile("nop; nop; nop");
5     return a;
6 }
```

Listing 1: Padded square and multiply with constant cycle behavior but differing in the amount of retired instructions

Zigzagger Bypass

In this section, we provide the RISC-V port of the Zigzagger examples from Lee et al. [55]. Listing 2 shows the code we used for the case study in Section 5.3. RISC-V does not provide a conditional move instruction. We emulated it using arithmetic and bitwise operations. As with the `cmov` instruction on x86, our implementation is side-channel resistant.

Histograms

We provide the histograms for the attacks in Section 4.4.

```
1  .macro cmov r1, r2, cond
2      seqz t2, cond
3      li t3, 0xFFFFFFFFFFFFFFFF
4      mul t2, t2, t3
5      xor t3, r2, r1
6      and t2, t2, t3
7      xor r1, r1, t2
8  .endm
9  zigzag_bench:
10             beqz a0, zigzag_bench_ret
11 block0:     la t1, block1
12             la t0, block2
13             cmov t1, t0, a0
14 block0_j:   j zz1
15 block1:     nop
16             la t1, block5
17 block1_j:   j zz2
18 block2:     la t1, block3
19             la t0, block4
20             cmov t1, t0, a1
21 block2_j:   j zz3
22 block3:     nop
23             la t1, block5
24 block3_j:   j zz4
25 block4:     nop
26 block5:     nop
27             addi a0, a0, −1
28             j zigzag_bench
29 zz1:        j block1_j
30 zz2:        j block2_j
31 zz3:        j block3_j
32 zz4:        jr t1
33 zigzag_bench_ret:
34             ret
```

Listing 2: RISC-V port of the Zigzagger example [55]

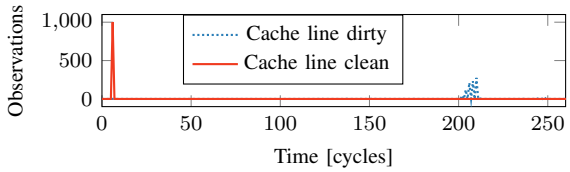

Figure 11: Flush+Flush histogram (C906). Left (solid red) is the execution time of `dcache.civa` for a cached clean cache line, on the right (dotted blue) for a modified (dirty) line.

Flush+Reload. Figure 10 shows a histogram for accesses to uncached and cached target memory on the C906.

Flush+Flush. Figure 11 shows Flush+Flush on the C906. The execution time is 6 cycles for flushing a non-modified cache line (including measurement overhead), and 207 cycles for a modified cache line.

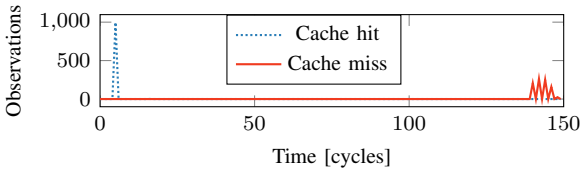

Figure 12: Evict+Reload histogram (C906) with cache hits (dotted blue) and cache misses (solid red) via eviction.

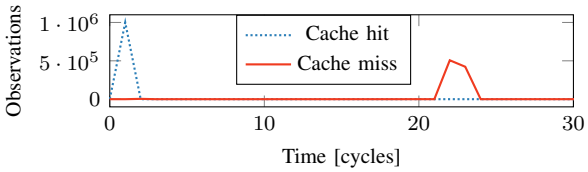

Figure 13: Evict+Reload histogram (U74) for cache hits (dotted blue) and cache misses (solid red) via eviction.

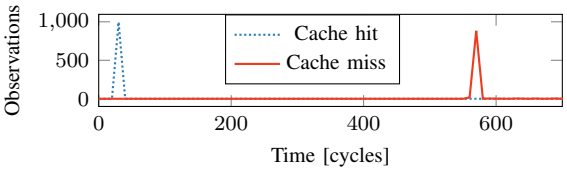

Figure 14: Prime+Probe histogram (C906) for cache hits (dotted blue) and cache misses (solid red) via eviction.

Evict+Reload. Figure 12 shows the access times to an accessed (cached) and an evicted (uncached) cache line on the C906. Figure 13 shows the the measurement on the U74.

Prime+Probe. Figure 14 shows the access times to an accessed (cached) and an evicted (uncached) cache line on the C906. Figure 15 shows the measurement on the U74.

TLB Eviction. Figure 16 shows a histogram for accesses to a memory location where the address is cached in the TLB (hit) or requires a page-table walk (miss).

Flush+Fault Attack Trace

Figure 17 shows a timing trace of Flush+Fault on a Square-and-Multiply RSA implementation. The values of the key bits are distinguishable by a simple threshold.

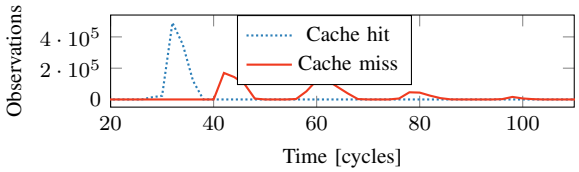

Figure 15: Prime+Probe histogram (U74) for cache hits (dotted blue) and cache misses (solid red) via eviction.

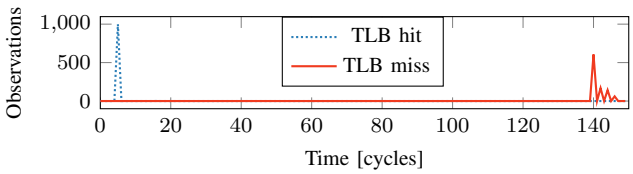

Figure 16: Histogram for TLB hits (dotted blue) and TLB misses (solid red) enforced via TLB eviction on the C906.

Figure 17: Flush+Fault: 20 RSA key bits (C906). '1' bits are above, '0' bits below 100 cycles.

TABLE 5: Instruction latencies on the U74 and the C906 processor. Timings with '*' indicate that the latency can vary depending on the operands.

Instruction	U74	C906	Instruction	U74	C906
nop	0.5	1	divu	14*	3
mv	1	1	divuw	7*	3
add	1	1	divw	70*	3
sub	1	1	mul	1	4
subw	1	1	mulh	1	4
addi	1	1	mulhsu	1	4
addiw	1	1	mulhu	1	4
addw	1	1	mulw	1	2
slt	1	1	rem	15*	3
slti	1	1	remu	13*	3
sltu	1	1	remuw	7*	3
sltiu	1	1	remw	69*	3
lui	1	1	fadd.s	2	3
and	1	1	fclass.s	1	3
xor	1	1	fdiv.s	9	7
or	1	1	feq.s	1	3
andi	1	1	c.add	1	1
xori	1	1	c.addi	1	1
ori	1	1	c.addiw	1	1
sll	1	1	c.addw	1	1
srl	1	1	c.and	1	1
slli	1	1	c.andi	1	1
slliw	1	1	c.li	1	1
sllw	1	1	c.lui	1	1
srli	1	1	c.mv	1	1
srliw	1	1	c.nop	0.5	1
srlw	1	1	c.or	1	1
srai	1	1	c.slli	1	1
sraiw	1	1	c.srai	1	1
sra	1	1	c.srli	1	1
sraw	1	1	c.sub	1	1
fence	1	5	c.subw	1	1
div	16*	3	c.xor	1	1

Instruction Latencies

Table 5 illustrates the latencies of the RISC-V base instruction set. To measure the latency of the instructions, we repeat each instruction 1024 times in a fully-unrolled loop written in assembly, i.e., there is no other instruction in between. The operands are set to random values before the measurement is started. We measure execution time using the `rdcycle` instruction. Additionally, we measure the constant overhead of the time measurement itself by measuring an empty block. By subtracting the overhead from the measurement and dividing the execution time by 1024, we infer the execution time of each instruction. We consider this instruction to have a non-constant execution time, if its timing differes over multiple runs. Simmilar to x86 CPUs [23], this is the case for the division and remainder instructions on the U74.