Systematic Assessment of Cache Timing Vulnerabilities on RISC-V Processors

Cédrick Austa[1 [0009−0002−1223−3285]], Jan Tobias Mühlberg[1 [0000−0001−5035−0576]], and Jean-Michel Dricot[1 [0000−0002−8539−9940]]

Université Libre de Bruxelles, Av. Roosevelt 50, 1050 Bruxelles, Belgium
{cedrick.austa,jan.tobias.muehlberg,jean-michel.dricot}@ulb.be

Abstract. While interest in the open RISC-V instruction set architecture is growing, tools to assess the security of concrete processor implementations are lacking. There are dedicated tools and benchmarks for common microarchitectural side-channel vulnerabilities for popular processor families such as Intel x86-64 or ARM, but not for RISC-V. In this paper we describe our efforts in porting an Intel x86-64 benchmark suite for cache-based timing vulnerabilities to RISC-V. We then use this benchmark to evaluate the security of three commercially available RISC-V processors, the T-Head C910 and the SiFive U54 and U74 cores. We observe that the C910 processor exhibits more distinct timing types than the other processors, leading to the assumption that code running on the C910 would be exposed to more microarchitectural vulnerability sources. In addition, our evaluation reveals that 65.9% of the vulnerabilities covered by the benchmark exist in all processors, while only 6.8% are absent from all cores. Our work, in particular the ported benchmark, aims to support RISC-V processor designers to identify leakage sources early in their designs and to support the development of countermeasures.

Keywords: RISC-V architecture · Cache timing side channel · Microarchitectural vulnerability · Security · Benchmark.

1 Introduction

Modern processors include many performance-enhancing features, such as caching, paging, out-of-order execution and speculative executions, which improve computer system performances. However, these features can be exploited to create side channels.

A *side channel* is an unintended communication channel between two entities, where one party observes information that is inadvertently leaked by the functioning of the other party's system [1]. This communication channel can be exploited either actively, by interacting with the system and observing how it reacts, or passively, by observing behavioral changes. A processor microarchitecture is a processor-specific logic implementation of an instruction set architecture (ISA) [2]. While the latter is the definition of the instruction set requirements, e.g., the supported instruction set and data types, the registers, and the privilege

modes, the microarchitecture describes the specifics of an ISA implementation, e.g., the instruction pipeline stages, the implementation of the cache and other optimization features. Microarchitectural side-channel vulnerabilities stem from and exploit effects of operations performed on a specific processor microarchitecture, which are not documented in the ISA. Examples of such vulnerabilities are cache-based side-channels which are used to extract information by observing operations on the cache or to build more sophisticated attacks, e.g., Meltdown vulnerability [3]. This category of vulnerabilities has been extensively studied for Intel processors [4, 5, 6], AMD processors [6, 7, 8, 9], and ARM processors [10, 11], and tools have been made available to evaluate their security.

The recent and open-source RISC-V ISA, which has gained a lot of popularity amongst industry and researchers, did not receive that much attention. Though, we know that RISC-V implementations are not exempt from microarchitectural vulnerabilities [12, 13]. E.g., Gerlach et al. [12] already demonstrated that some differences may be observed between vulnerabilities on the Intel x86-64 architecture and the RISC-V architecture. Moreover, since RISC-V is an open-source ISA, different RISC-V hardware implementations are made available by CPU vendors, which leads to processor-specific vulnerabilities. As a result, the RISC-V architecture needs its own security toolkit to evaluate existing and future RISC-V processors. Recently, Thomas et al. [14] provides automatic detection of architectural vulnerabilities but benchmarks or test suites to assess the microarchitectural attack surface of a given RISC-V implementation are currently missing. In this paper we address this shortcoming. With a focus on cache-based timing vulnerabilities, we ported a benchmark [4, 15] to target RISC-V and evaluate it on three commercially available processors.

Contributions. We ported and extended a benchmark from Intel x86-64 [4] to detect cache-based side-channel vulnerabilities for the RISC-V ISA. The original benchmark is based on a theoretical three-step model proposed in Deng et al. [15] and is made of two software components: the first allows to observe the different cache timing types on a processor, while the second allows to determine cache-based timing vulnerabilities on the L1 data cache of this processor. By allowing for different cache hierarchies, different cache set associativities, and different cache eviction strategy parameters, our ported benchmark is more flexible and easier to adapt to new RISC-V implementations. Our contributions are:

- porting and translating original work from Deng et al. [4] to RISC-V processors;
- refactoring the benchmark to support new RISC-V processor configurations;
- evaluating cache-based vulnerabilities on the three commercially available RISC-V processors, the T-Head C910, and the SiFive U74 and U54;
- identifying and interpreting timing differences between memory operations;
- identifying L1 data cache-based vulnerabilities across all implementations.

With our evaluation, we show that the translated benchmark can be used to successfully identify cache timing leakage sources and potential vulnerabilities on RISC-V processors. By doing so, we demonstrate the utility of such a benchmark

for RISC-V chip designers to identify vulnerabilities in their designs for which countermeasures should be implemented. Our results show that 39 out of the 88 cache-based timing vulnerabilities highlighted in Deng et al. [4] are present in all implementations. We make our benchmark available under an open-source license: `https://github.com/ReSP-Lab/risc-v-sidechannel-benchmark`

Even though such an evaluation was never performed on RISC-V implementations, we do not consider that following confidential disclosure procedures with the manufacturers are warranted: (i) most of the implementations we evaluate are used in development boards that are mainly dedicated to hobbyists and researchers, (ii) the benchmark on which we based our work as well as the covered vulnerabilities are not new, (iii) the vulnerabilities are rather involved and not easily exploitable in practice, and (iv) subsets of these vulnerabilities have already been reported by Gerlach et al. [12] for the same or similar processors.

2 Background

Below we provide background on caches and caching behavior in modern processors, we outline how caches can be abused in side-channel attacks, and we summarize Deng et al. [4]'s work as the foundation of our research.

2.1 System Caches and Caching

To optimize performance, when a processor requires data from (slow) main memory it first checks the (fast) caches if this data is already available. If the data is in the cache, the cache *hits*; otherwise, the cache *misses* and data is fetched from main memory and placed into the cache. *Caching* exploits *temporal locality* and *spatial locality* [2]: As a consequence of temporal locality, data is loaded into the cache from main memory when a cache miss occurs. This happens in *cache blocks* that include data from adjacent addresses, thus exploiting spatial locality.

Cache Hierarchy. To decrease the *miss penalty*, i.e., the latency overhead due to cache misses, a *multi-level cache hierarchy* is often used. The highest level and the lowest level in the hierarchy respectively are the L1 cache and the *last-level cache (LLC) cache*. The lower the cache level in the hierarchy, the larger are both the cache size and the latency to access data in this cache.

Cache Specifications. The performance impact of a cache depends on cache capacity C, block size b, and the degree of associativity N [2]. Let $B = C/b$ be the number of cache blocks in a cache. The cache is *N-way set associative* when these blocks are grouped into S sets of N ways, i.e., block locations in the set, with $S = B/N$ and $1 \leq N \leq B$. The number of ways per set determines how memory addresses in main memory are mapped to cache locations: If $N = 1$, any memory address is mapped to only one block; such a cache is called a *direct mapped* cache. If $N = B$, any memory address can be mapped to every blocks in the cache; such a cache is called a *fully associative* cache.

Cache Replacement Policy. When a miss occurs while the cache is fully populated, a cache block needs to be evicted to load the target block into the cache set. For a N-way set associative cache with $N > 1$, a *cache replacement policy* is necessary to select which block to evict. Examples of cache replacement policies are the *random* replacement policy (evicting a random block), the first-in first-out (FIFO) replacement policy (evicting the oldest block), or the least recently used (LRU) policy. In LRU, to avoid tracking the last use of each way in a set, approximations of this policy, so-called pseudo-LRU (PLRU) policies, are used.

Cache Coherence Protocol. In multicore processors, cache coherence protocols are used to track cache block states among private and shared caches. By doing so, each core reads the same content from a cache block at any time between two write operations. The two main categories of cache coherence protocols are *directory-based* protocols and *snooping* protocols. The former track cache block states using a directory, in a centralized way where the state of a cache block is known by accessing the directory. The latter track cache block states using a snooping bus and the state of a cache block is stored locally in the cache and each cache block state alteration is broadcasted on the bus for other listeners. Examples of states are the followings: M(odified), O(wned), E(xclusive), S(hared), or I(nvalid). The choice of a cache coherence protocol has impact on performance and scalability.

2.2 Cache-based Timing Vulnerabilities

Cache-based microarchitectural attacks exploit the cache status as a leaking information source. Information is collected either by observing timing differences on cache block accesses [3, 16] or by measuring power consumption differences on cache block content change [17, 18]. *Cache-based timing* attacks exploit the observed latency between cache hits and cache misses, e.g. the access time increases whenever cache blocks to load are lower in the cache hierarchy. Attackers can interact with the cache using a specific memory operation and measure the operation time or latency to infer the victim's cache state. According to Su et al. [19], a succeeding cache attack depends on the following three conditions:

1. a relation should exist between a change in the cache state and target sensitive information;
2. at least one cache in the cache hierarchy is shared between both the attacker program and the victim program;
3. both programs sharing the cache can infer changes to each other's cache state by monitoring their own cache state.

To illustrate how the cache state may be exploited, we describe selected well-known cache-based timing attacks below.

Prime+Probe [20]. This first cache-based timing attack works on cache sets as follows: (i) the attacker fills one or more sets with its own cache blocks (*prime*), (ii) victim cache blocks are accessed, and (iii) the attacker reloads its own cache blocks in each set and measures timing (*probe*). In this attack, the cache probing is longer if attacker cache blocks were evicted by victim cache blocks.

Evict+Time [20]. This attack, which was introduced with Prime+Probe, consists in the following steps: (i) the attacker runs the victim program and measures timing, (ii) the attacker evicts a victim cache block (*evict*), and (iii) the attacker runs the victim program again and measures timing (*time*). If the evicted cache block was accessed by the victim, the execution time is longer to load the victim cache block.

Flush+Reload [21]. To perform this attack, (i) the attacker flushes a victim cache block (*flush*), (ii) the victim program eventually accesses the victim cache block, and (iii) the attacker reloads the victim cache block and measures timing (*reload*). If the victim accessed the flushed cache block, the reload operation time is shorter for the attacker. A variant of this attack, *Evict+Reload*, was proposed by Gruss et al. [22] to avoid the flush instruction requirement.

Flush+Flush [23]. This attack uses steps (i) and (ii) from the Flush+Reload attack. For step (iii) the attacker flushes the victim cache block again and measures timing. If the victim accessed their cache block, the flush operation takes longer than if they did not.

2.3 Deng et al. [4] Benchmark Suite

Observable timing types. In their work, Deng et al. [4] study timing differences between memory operations depending on the cache level in which data is located, depending on the cache block data state (i.e., *dirty* or *clean*) and whether the attacker is running their program on the same core as the victim. As a result, 66 different timing types are considered instead of only the two traditional timing types for cache-based timing vulnerabilities (i.e., *fast* or *slow*). Even though some of these timing types were already exploited in the literature, e.g., both Flush+Reload [21] and Flush+Flush [23] rely on at least two of these 66 timing types, their work emphasizes the existence of such distinct timing types. To obtain the 66 timing types (i.e., 22 timing types per operation) the following cases are considered:

- either a read, write or flush operation is used to access data (3 operations);
- data is present either on the local L1/L2/L3 cache or on a remote L1/L2/L3 cache and the cache block in which data is present is either in dirty or clean state ($6 \times 2 = 12$ cases), **or** data is present both in either the local L1, L2 or L3 cache and either the remote L1, L2 or L3 cache in clean state ($3 \times 3 = 9$ cases), **or** data is only present in the DRAM (1 case).

Table 1: L1 cache timing-based vulnerabilities [4]. The *No.* column assigns each type of vulnerability a number range. The *Attack Strategy* column gives a common name for each set of vulnerabilities and each vulnerability types that would be exploited in an attack in a similar manner. Inv. means invalidation.

No.	Attack Strategy
1-4	Cache Collision
5-8	Flush + Reload
9-10	Reload + Time
11-14	Flush + Probe
15-16	Flush + Time
17-20	Cache Coherence Flush + Reload
21-28	Cache Coherence Prime + Probe
29-32	Cache Coherence Evict + Time
33-36	Bernstein's Attack
37-38	Evict + Probe
39-40	Prime + Time
41-42	Evict + Time
43-44	Prime + Probe

(a) Third step as memory access operation.

No.	Attack Strategy
45-46	Cache Collision Inv.
47-50	Flush + Flush
51-52	Flush + Reload Inv.
53-54	Reload + Time Inv.
55-58	Flush + Probe Inv.
59-60	Flush + Time Inv.
61-64	Cache Coherence Flush + Reload Inv.
73-76	Cache Coherence Evict + Time Inv.
77-80	Bernstein's Inv. Attack
81-82	Evict + Probe Inv.
83-84	Prime + Time Inv.
85-86	Evict + Time Inv.
87-88	Prime + Probe Inv.

(b) Third step as invalidation operation.

Table 2: Test configurations for the L1-D cache benchmarks. For memory accesses: `read` and `write` operations. For invalidations: `flush` and `write` operations.

Test config.	**Step 1**	**Step 2**	**Step 3**	**Run**
RF_RF_RF_TS	read/flush	read/flush	read/flush	time-slicing
RF_RF_RF_SMT	read/flush	read/flush	read/flush	SMT
RF_RF_W_TS	read/flush	read/flush	write	time-slicing
RF_RF_W_SMT	read/flush	read/flush	write	SMT
RF_W_RF_TS	read/flush	write	read/flush	time-slicing
RF_W_RF_SMT	read/flush	write	read/flush	SMT
RF_W_W_TS	read/flush	write	write	time-slicing
RF_W_W_SMT	read/flush	write	write	SMT
W_RF_RF_TS	write	read/flush	read/flush	time-slicing
W_RF_RF_SMT	write	read/flush	read/flush	SMT
W_RF_W_TS	write	read/flush	write	time-slicing
W_RF_W_SMT	write	read/flush	write	SMT
W_W_RF_TS	write	write	read/flush	time-slicing
W_W_RF_SMT	write	write	read/flush	SMT
W_W_W_TS	write	write	write	time-slicing
W_W_W_SMT	write	write	write	SMT

A first part of their benchmark suite implementation generates histograms to identify the different timing types existing on a target. Each timing type which may be distinguished from the others in this histogram corresponds to a timing leakage source which can be exploited by an adversary to monitor cache usage.

Three-step model. Deng et al. [4] propose an improvement for a theoretical model [15] describing cache timing vulnerabilities as sequences of three steps (i.e., three memory-related operations), each modifying the state of a target cache block: state initialization, state alteration, and timing observation. From each step, one of 17 possible cache block states should be reached for the target cache block. The memory-related operations which can be used for each step correspond to the `read`, `write` and `flush` operations, depending on the resulting cache block state.

To validate their three-step model and detect all possible L1-D cache-based timing vulnerabilities on concrete processors, a subset of the previous timing types are used in a benchmark suite. In particular, benchmarks are automatically generated to evaluate the 88 vulnerabilities Deng et al. [4] identified as "strong" out of 4913 possible vulnerabilities. These 88 vulnerabilities are summarized in Table 1. For more details, the reader is re-oriented towards Deng et al. [4].

Moreover, to evaluate the existence of each vulnerability on a processor, their benchmark suite considers 16 test configurations for each vulnerability. These configurations aim to reach the same three cache block states for a vulnerability by applying different memory-related operation; cache block access can be performed with `read` or `write` operations, while cache block invalidation can be performed with `flush` or `write`. These configurations also compare timing differences observed whenever a victim and an adversary are located on the same physical core and whenever either *time-slicing* or *simultaneous multithreading (SMT)* is used. With some of these configurations, Deng et al. [4] identified vulnerabilities relying on cache coherence. Table 2 lists these test configurations.

3 Threat Model, Benchmark Suite, Experimental Setup

The goal of our work is to develop and evaluate a benchmark to assess the security of RISC-V implementations regarding cache-based timing vulnerabilities. Below we define our attacker model, and present the ported benchmark suite and our experimental setup.

3.1 Threat Model

We assume an attacker model where the adversary is able to gain unprivileged access to a system of interest and execute code on that system, and that this code can access hardware performance counters of the system, in particular clock cycle counters to measure memory operation latencies. Moreover, we only consider vulnerabilities which are evaluated in the benchmark (cf. Section 2.3, Table 1). We also make the assumptions that the conditions listed in Section 2.2

are satisfied: the existence of a link between cache states and victim secrets, and a shared and mutually observable cache state between victim and attacker code.

Under these conditions, we can rely on measurements from hardware performance counters to determine if the system is vulnerable or not. If no vulnerability is found that way—i.e., by relying on accurate clock cycle counters—the chances for an adversary to find vulnerabilities by relying on less accurate timers, such as POSIX timers, would be null. However, vulnerabilities found that way may not always result in exploitable vulnerabilities in practice. Verifying exploitability would typically be specific to a victim program and goes beyond the scope of this work. Also the inclusion of additional sources of information leakage, hardware side channels or microarchitectural side channels other than caches, is beyond the scope of this work.

3.2 Benchmark Suite Implementation & Porting

Regarding the porting to RISC-V, we decided (i) to segment the source code into smaller dedicated header and source files to ease for the code maintenance; (ii) to move all inline assembly snippets into inline assembly functions in a dedicated file to reduce the number of redundant code lines and, hence, to ease source code maintenance and corrections; (iii) to dedicate a header file for each target which defines all target specifications, related parameters (e.g., for Gruss et al. [24] eviction algorithm), and the (optional) related inline assembly `flush` function, to ease the support implementation for a new target; (iv) to allow the user to define which cache levels are exist on the target so that we can use the benchmark more constrained or developed targets.

In order to facilitate the evaluation of cache-based timing vulnerabilities, huge pages of 2MB are used to find conflicting addresses. Since most processors use virtually indexed physically tagged caches, this allows us to avoid taking virtual address translation into account. While this approach is relevant for embedded systems, it is less realistic for scenarios involving a fully-fledged operating system. Yet, since the goal of this work is not to evaluate the difficulty to build eviction sets on the targets, we consider this a minor limitation. In this context, eviction is not performed by building conflict sets, as it is the case in the original benchmark suite [4], but by using the eviction algorithm presented by Gruss et al. [24]. Deng et al. [4] only covered processors that use an LRU policy on L1 and L2 caches, which means that only sequential access to N cache blocks is required to evict a cache set. The authors built conflict sets only to evict the L3 cache of their targets. When considering targets that use different cache replacement policies per cache level, this method would require us to build a specific conflict set for each of these cache levels. Instead, the eviction algorithm proposed by Gruss et al. [24] allows us to deal with different replacement policies than the LRU policy by adapting the algorithm parameters, and conflicting addresses are easy to list since the set index is straightforward to obtain using huge pages on our targets, which has performance benefits. Nevertheless, this approach requires to define adequate algorithm parameters. For our study we manually explored different

Table 3: Evaluated system-on-chip cache specifications. Both SiFive cores use a directory-based cache coherence protocol, while the C910 core uses a snooping cache coherence protocol.

Core	L1-D cache			L2 cache		
	Size	Ways	Replacement	Size	Ways	Replacement
C910	64kB	2	FIFO	1MB	16	FIFO
U54	32kB	8	random	2MB	16	PLRU
U74	32kB	8	random	2MB	16	random

sets of parameters and settled on parameters that produce stable results with small error margins in our evaluation.

3.3 Evaluation Targets & Experimental Setup

For the experiments, three single-board computers (SBCs) with RISC-V application cores were used. These SBCs are the BeagleV-Ahead [25] (SoC: TH1520, core: C910), the BeagleV-Fire [26] (SoC: U54-MC, app. core: U54) and the HiFive Unmatched [27] (SoC: FU740, app. core: U74). Their cache specifications are given in Table 3. Note that since the cache replacement policy for the L2 cache from U54 is not documented, we made the assumption it was a PLRU policy. Experiments were performed with the Ubuntu images available for these SBCs (images from the SBC provider or from Ubuntu Boards).

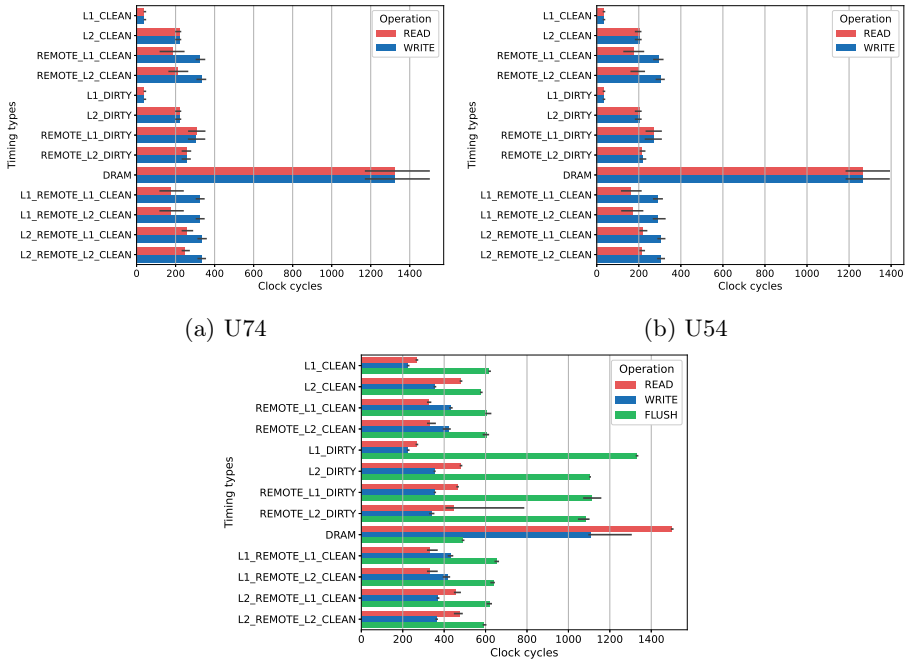
First, none of the selected SBCs benefits from an L3 cache. Thus, 27 timing types are not considered during the evaluation of memory operations, resulting to the measurement of only 39 out of the 66 timing types proposed by Deng et al. [4]. Moreover, previously selected images allow the user-mode to use the `rdcycle` instruction which are used to obtain high accuracy timings for the benchmarks. Unfortunately, by using these images, flush instructions are not available except for the C910 processor which leaves it available to user-mode programs. As a result, only 26 timing types can be measured for the evaluations of the U54 and U74 processors. However, to be able to measure DRAM latency for each operation, flush instruction is replaced by L2 eviction when necessary.

Finally, as proposed by Deng et al. [4], we measure latencies on 8 cache lines from distinct cache sets at a time, in order to minimize false negatives. Regarding false positives, Deng et al. [4] proposed to isolate cores to reduce software noise. We decided to not isolate cores to have a more realistic situation, since our configuration is not running anything else than the operating system and our benchmarks. Moreover, we only observe a small differences when isolating cores; we still observe differences between distinct evaluations with isolation, probably due to last level cache conflicts with the operating system and our process, and due to random cache policies.

4 Results

4.1 Timing Types

Measurements. The observed timing types from our tests on the three evaluated cores are given in Figure 1. The most frequent clock cycle latency from all tests is presented as a bar plot and a percentile interval of 95% is superposed to the plot as an error bar. Timing type labels refer to a subset of the 22 timing types per operation described in Section 2.3. Obviously, timing types related to L3 cache are not considered.



(a) U74 (b) U54

(c) C910

Figure 1: Most frequent clock cycle latencies over 10000 tests, per timing type and per target. Label prefixes `Lx_` or `REMOTE_Lx`, refer to the cache block location in the cache level `x` on local or remote core. Label suffixes `_CLEAN` or `_DIRTY`, refer to the cache block state before the memory operation, respectively.

Observations and Preliminary Discussion. From Figure 1, the observed clock cycle latencies differ between most of the evaluated timing types, for all cores. For both SiFive cores, we see four groups of timing types which seem to show to observable different latencies for the write operation, while five groups

show observable differences for the read operation. We assume the observed latency overheads appearing for the write operation come from the directory-based cache coherence protocol which ensures that other existing cache block copies are invalidated on remote caches. Regarding the T-Head core, four groups of timing types can be observed, both for the write and read operations. Nevertheless, we see differences between the read and write operations due to the use of a snooping cache coherence protocol. In particular, a MESI protocol is used between L1-D caches with a bypassing mechanism and a MOESI protocol is used to ensure coherence between L1-D caches and the L2 cache. Our interpretation is that overheads appear either due to the cache coherence, to the snoop buffer, to store (or write) buffers, or to the load-store unit present between cores and their L1 cache. However, timing type latencies observed for the flush operation are more heterogeneous than for the read and write operations. In particular, latencies are larger when cache block has to be written back into main memory (cache block is dirty), and latencies decrease when cache block is located in lower levels of the cache hierarchy. The minimum latency observed for this operation is when cache block is already written back into main memory.

From the previous observations, we learn that the read operation is the main leakage source on both SiFive cores, while the dominant leakage source on the T-Head is the flush operation. We also learn that the C910 core exposes more potentially vulnerable behavior than the SiFive cores: first due to the differences observed between all memory operations and, second, due to the availability of the `flush` instruction.

4.2 L1 Data Cache Vulnerabilities

Measurements. The evaluation of the three targets for the 88 strong vulnerabilities identified by Deng et al. [4] is provided by Figure 2. On Figure 2, the presence of a vulnerability for a specific test configuration, as described in Table 2, for a given target is indicated by a marker. In addition, two categories indicate whether the vulnerability is present in all evaluated CPUs or absent from all of them.

Cache timing vulnerability scores (CTVSs) [4] are used to quantify how a target is vulnerable by providing the ratio of vulnerabilities which are observable on a target. These CTVSs are given in Table 4. In addition, the ratio of vulnerabilities successfully observed on a target (i.e., a success ratio), only taking into account all valid test configurations for the latter, is provided too in Table 5 and Table 6. Valid test configurations depend on the support of flush instruction in user mode or on the support of SMT, which leads to different maximum number of cases per target. Thus, for each table, the success ratio for test configurations shared between all targets or exclusive to some targets (e.g., depending on the `flush` instruction support) is given. This success ratio is obtained by dividing the number of successfully observed vulnerabilities for the target with the total number of cases for the considered category.

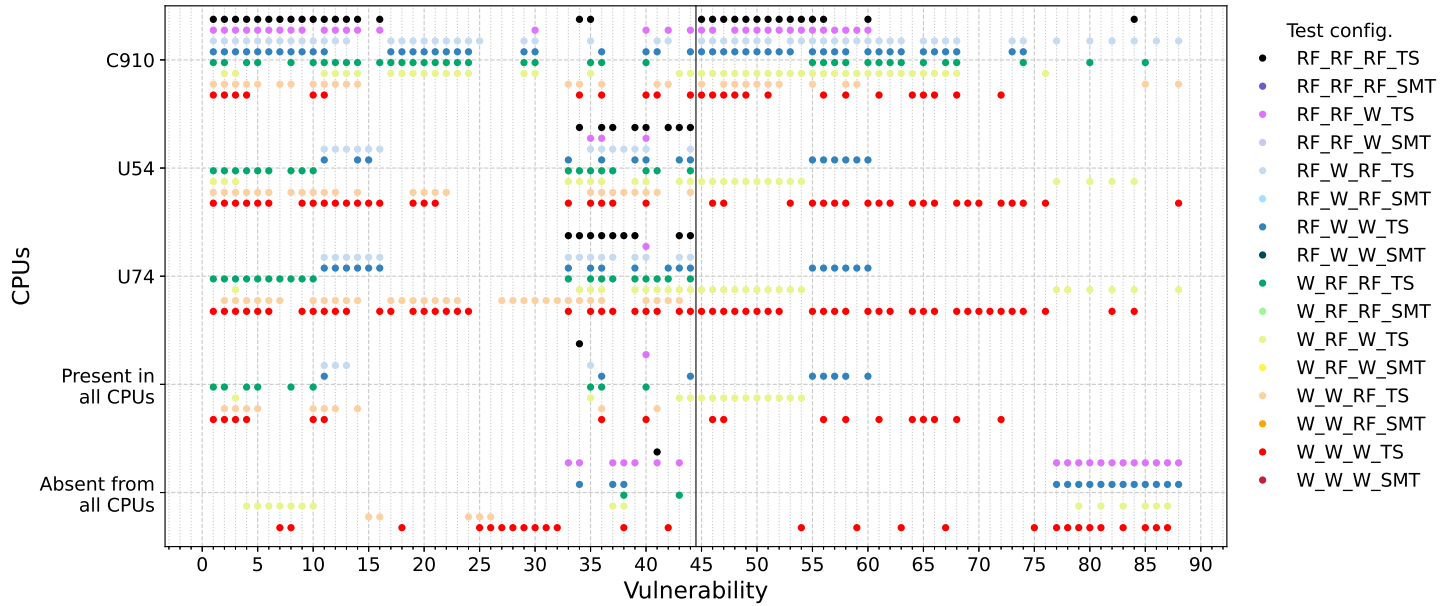


Figure 2: Evaluated timing types per vulnerability on target RISC-V SoCs: vulnerabilities on the left part have a read or write access as a third step, while vulnerabilities on the right have an invalidation as a third step. To read this plot, choose a target on the vertical axis. Then, for a vulnerability located on the horizontal axis, determine which test configuration allows to detect the vulnerability for the target. Example: vulnerability #1 uses an invalidation as a first step. From all targets, markers from the top half dedicated to this vulnerability are only present for the C910 core. The top half corresponds to test configurations starting with `RF_`; they are test configurations where the first step is performed either with a `read` operation (memory access) or with a `flush` operation (invalidation). As a result, we deduce that since the flush operation is only accessible from U-mode on the C910, vulnerability #1 will only be marked on the C910 for this set of test configurations. We can also note, for example, that the `W_RF_RF_TS` test configuration, allows to observe vulnerability #1 on all targets.

Table 4: Cache timing vulnerability score.

	C910	**U54**	**U74**	**All**	**None**
CTVS	70/88	64/88	77/88	58/88	6/88

Table 5: Ratio of vulnerabilities successfully observed for different running configurations: whether an attacker is present or not and whether the victim and the attacker run on the same core (time-slicing or hyper-threading) or not. Note that none of the targets support SMT.

Core	Victim, attacker Same core		Victim, attacker Different cores	Victim only
	Time-slicing	SMT		
C910	36/157	0	17/38	34/93
U54	70/157	0	24/38	54/93
U74	89/157	0	26/38	62/93
All	23/157	0	14/38	28/93
None	52/157	0	8/38	24/93

(a) Shared test configurations.

Core	Victim, attacker Same core		Victim, attacker Different cores	Victim only
	Time-slicing	SMT		
C910	122/233	0	31/52	65/131

(b) Exclusive test configurations.

Observations and Preliminary Discussion. A first observation that can be made from Figure 2 is regarding the vulnerabilities which are absent from all boards. In particular, absent vulnerabilities relate to observable timing differences by evicting or invalidating addresses out of the sensitive memory locations, which conflict with addresses in the sensitive memory locations of interest. This result can be interpreted as follows: (i) most of the vulnerabilities relying on invalidation are on the C910 core and SiFive cores only rely on remote `write` operations to invalidate cache blocks, in our implementation; (ii) no significant timing difference is observed for the `write` operation on SiFive cores between

Table 6: Ratio of vulnerabilities successfully observed for different test configurations and memory operations. *Inv.* means invalidation.

Core	Local Read	Local Write	Remote Write Inv.	Flush Inv.
C910	34/96	24/96	29/96	0
U54	62/96	45/96	41/96	0
U74	75/96	52/96	50/96	0
All	24/96	16/96	25/96	0
None	8/96	32/96	44/96	0

(a) Shared test configurations.

Core	Local Read	Local Write	Remote Write Inv.	Flush Inv.
C910	51/80	49/80	48/80	70/176

(b) Exclusive test configurations.

`REMOTE_L1_CLEAN` and `REMOTE_L2_CLEAN`, making vulnerabilities #79, #81, and #83, which rely on the eviction of remote-only memory location, less effective.

Then, from Table 4, it also appears that the U74 core exhibits more vulnerabilities than the U54. We could interpret this as the former being more vulnerable or less secure than the latter. However, from the more detailed view provided by Figure 2, we learn that the presence of these vulnerabilities on one of these cores is a result of the specific benchmark run as the vulnerabilities are only observed for one or two test configurations. This is supported by the similarities between both microarchitectures, in particular for the L1 cache, and Figure 1 which does not highlight any significant difference between observable timing types. Thus, we assume that if some vulnerabilities identified on the U74 core are false positives, the number of vulnerabilities absent from all evaluated targets should increase. Depending on the previous assumption, the U74 would be either more or less (in case previous assumption is correct) vulnerable than the C910, but further research is needed to assess this.

Even though the C910 might present more vulnerabilities than both SiFive cores, it appears from Table 5 and Table 6 that, on shared test configurations, less configurations led to observable timing differences on the C910. It seems that, if the C910 presents more vulnerabilities, both SiFive cores present more ways to exploit each of their vulnerabilities. Finally, it also clearly appears that allowing the `flush` instruction provides ways to exploit these vulnerabilities which are not present on SiFive cores, from user mode.

4.3 Discussion

From our results in Section 4.1 and Figure 2 we learn that the T-Head C910, featuring a U-mode `flush` instruction, has more observable latency differences than the other cores. These differences provide additional attack surface and potentially more exploitable vulnerabilities. An effective way to improve the security the C910 would be to restrict access to `flush` to the M-mode (cf. [12]).

Following our interpretation of cycle latencies and CTVS, both SiFive cores, the U54 and U74, expose the same set of vulnerabilities which is smaller than that of the T-Head core. The absence of the `flush` instruction in U-mode definitively plays an important role for test configurations relying on a local invalidation step. An attacker who gains control over the kernel or firmware may still be able to invoke `flush` in M-mode. However, these attacks are beyond the scope of our threat model and we therefore did not study timing of `flush` instructions on these cores. Yet, on shared test configurations (cf. Table 5 and Table 6) both SiFive cores score substantially higher numbers vulnerabilities than the T-Head core. This may be due to the difference of cache coherence protocols or other microarchitectural design differences.

Both parts of the benchmark suite, the evaluation of timing types and the evaluation of L1-D cache vulnerabilities, provide valuable information about timing types an adversary may exploit a RISC-V processor. The histograms should help the designer to identify leakage sources in their design and to make choices

to improve its resilience against cache-based timing vulnerabilities. Then, identifying the test configurations for which an implementation present a given vulnerability help to assess the security of their design and to determine which implemented countermeasures and design choices really have an impact on their design security, e.g., restricting the access to `flush` instructions.

Limitations. Our benchmark does not consider virtual addresses and S-mode executions in general. Thus, our results are representative for embedded systems but are incomplete regarding scenarios that involve an operating system. Depending on how virtual addresses are managed, finding conflicting addresses in a cache set may be more difficult in these scenarios. Moreover, we did not study systems with an L3 cache yet, which may, depending on the availability of `flush`, present even more timing differences than the cores studied. Our benchmark is also dependent on knowledge of the target cache specifications (cf. Table 3). Evaluating implementations with missing specifications may lead to poor results. Finally, we only use the eviction algorithm from Gruss et al. [24], which may be limited depending on cache replacement policies or the supported access pattern detection features.

5 Related Work

Kelsey et al. [28] first introduced the idea of using the cache hit ratio to perform attacks on substitution boxes (S-boxes) from symmetric encryption algorithms. Based on this idea, Page [29] implements a first example of cache attack on DES. Bernstein [16] reported the extraction of an AES key from a remote computer by observing larger timings on S-boxes due to cache block eviction, and Percival [30] describes how memory between threads could be used both as a covert channel and as a side channel to attack RSA.

New ways to leverage cache timing differences are still being proposed. Gruss et al. [22] introduce cache template attacks which consist on a phase dedicated to the cache profiling with respect to an event of interest, followed by an exploitation phase which infers event occurrences by monitoring the cache. Yan et al. [31] illustrates that cache coherence protocols, more specifically directory-based protocols, can be considered an exploitable source of timing leakage. Briongos et al. [32] introduces Reload+Refresh which exploits the cache replacement policies and determines if a cache block is accessed by the victim.

For eviction-based attacks, different approaches to perform eviction or to build eviction sets exist, including eviction-based remote Rowhammer [33] and a parameterizable eviction strategy by Gruss et al. [24]. The latter strategy is the one used in our work. Similarly, Liu et al. [34], Song et al. [35] and Vila et al. [36] present formal definitions and methods to build minimal eviction sets. These approaches could be incorporated in our work by relying on conflict sets to perform the eviction, as an alternative to our method (see Section 3.2) and as originally performed by Deng et al. [4].

More recently, Lipp et al. [37, 38] and Kogler et al. [18] propose a software-based power side-channel attacks which measure power consumption due to operations on the cache or to the bit flips occurring on cache replacement to extract keys or to break address space randomization. If similar software interfaces allowing to monitor the power consumption would be available on RISC-V processors, timing measurement could be replaced in our benchmark with power consumption monitoring operations as an attempt to detect cache misses, similarly to what Bertoni et al. [17] proposed.

Lyu et al. [39], Su et al. [19] and Shen et al. [40] survey cache-based side channel attacks and countermeasures. Purnal et al. [41] analyze the effectiveness of randomization to protect caches and Deng et al. [11] analyze the effectiveness of secure caches on ARM processors. Our benchmark can be used to validate hardened RISC-V designs that incorporate these countermeasures.

Le [13] study the security of the RISC-V architecture and Gerlach et al. [12] propose benchmarks and cases to evaluate and demonstrate vulnerabilities in selected RISC-V boards, including some of the 88 vulnerabilities from Deng et al. [4]. Thomas et al. [14] provide a fuzzing tool to detect architectural vulnerabilities in RISC-V implementations. Work in this direction is complementary to our approach, providing additional points of evidence for the presence or absence of vulnerabilities and building confidence in existing designs.

Finally, Oleksenko et al. [42] propose a model-based black-box approach to detect speculative leakage sources in CPUs by observing traces. Fabian et al. [43] propose a framework, supporting formal models for composite speculation mechanisms, which is used inside a program analysis tool to detect speculative vulnerabilities in code snippets. Barthe et al. [44] present a testing framework to assess given program security based on a description language dedicated to leakage models of microarchitectural optimizations. In comparison, our work is a model-based white-box approach focusing on the identification of vulnerabilities, which could inform leakage models of formal approaches to program security. Since we use a white-box approach, implementation specifications help to obtain accurate results by adequately adapting the benchmark parameters to the target. However, our work is predominantly useful for processor developers, enabling early evaluation of a processor implementation for cache-based timing leakage.

6 Conclusion

In this paper we reported on porting a comprehensive benchmark suite for cache-based timing vulnerabilities to RISC-V. We evaluate the benchmark on three commercially available RISC-V cores and present our findings, showing diverging leakage profiles across the processors with 65.9% of vulnerabilities present across all three processors and 6.8% of vulnerabilities being absent from all cores. In addition, 37.5% of the vulnerabilities are present for at least one shared test configuration across all three processors. Our benchmark and evaluation artifacts are available under an open-source license.

We anticipate that the ported benchmark will be useful for researchers as well as commercial developers of RISC-V implementations to evaluate leakage patterns of processors, to inform developers of potentially diverging security risks across different implementations, and to guide the development of strong testing and verification tools for RISC-V processors as well as for software compiled for RISC-V. As our benchmark suite helps to identify the different timing types and leakage patterns, our work can also support the development of concrete and formal leakage models. We have taken care and refactored the benchmark specifically to allow for easy re-configuration towards microarchitectures with different cache hierarchies, different cache set associativies, and different cache eviction strategies, which should greatly improve reusability of our artifacts.

In future work, we will evaluate the security of processors from other vendors and the exploitability of our findings for S-mode and U-mode. Regarding exploitability, we will provide proofs of concept based on the benchmark for specific applications with different risk profiles. Furthermore, we will evaluate countermeasures against cache timing vulnerabilities, e.g., cache partitioning and secure caches. We will also investigate the impact of different cache-coherence models on the security of open-source RISC-V cores with our benchmark. The security of implementations of Trusted Execution Environments against cache-based timing vulnerabilities could be another target of future work. Finally, we seek to extend the benchmark towards similar vulnerabilities, e.g., translation-lookaside buffer or transient execution vulnerabilities.

Acknowledgements. This research is supported by the CyberExcellence program of the Wallon region of Belgium under GA #2110186.

References

[1] Jakub Szefer. *Principles of Secure Processor Architecture Design*. Springer International Publishing, 2019. DOI: 10.1007/978-3-031-01760-5.

[2] Sarah L. Harris. *Digital design and computer architecture: RISC-V edition*. Ed. by David Money Harris. Includes index. Cambridge: Morgan Kaufmann, 2022. 1564 pp.

[3] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[4] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. "A Benchmark Suite for Evaluating Caches' Vulnerability to Timing Attacks". In: *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. ACM, Mar. 2020. DOI: 10.1145/3373376.3378510.

[5] Fabian Rauscher et al. “A Systematic Evaluation of Novel and Existing Cache Side Channels”. English. In: *Network and Distributed System Security Symposium (NDSS) 2025*. Network and Distributed System Security Symposium 2025 : NDSS 2025. Feb. 2025. DOI: 10.14722/ndss.2025.23253. URL: https://www.ndss-symposium.org/ndss2025/.

[6] Antoon Purnal and Ingrid Verbauwhede. “Cache Side-Channel Attacks on Existing and Emerging Computing Platforms”. eng. PhD thesis. 2023.

[7] Xida Ren et al. “I See Dead µops: Leaking Secrets via Intel/AMD Micro-Op Caches”. In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 2021, pp. 361–374. DOI: 10.1109/ISCA52012.2021.00036.

[8] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “Cross Processor Cache Attacks”. In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ASIA CCS ’16. ACM, May 2016. DOI: 10.1145/2897845.2897867.

[9] Moritz Lipp et al. “Take A Way: Exploring the Security Implications of AMD’s Cache Way Predictors”. In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. ASIA CCS ’20. New York, NY, USA: Association for Computing Machinery, 2020, 813–825. DOI: 10.1145/3320269.3384746.

[10] Moritz Lipp et al. “ARMageddon: Cache Attacks on Mobile Devices”. In: *25th USENIX Security Symposium*. USENIX Association, Aug. 2016, pp. 549–564. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp.

[11] Shuwen Deng et al. *Evaluation of Cache Attacks on Arm Processors and Secure Caches*. 2021. DOI: 10.48550/ARXIV.2106.14054.

[12] Lukas Gerlach et al. “A Security RISC: Microarchitectural Attacks on Hardware RISC-V CPUs”. In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023, pp. 2321–2338. DOI: 10.1109/SP46215.2023.10179399.

[13] Anh-Tien Le. “Research of RISC-V Out-of-order Processor Cache-Based Side-Channel Attacks-Systematic Analysis, Security Models and Countermeasures”. PhD thesis. 2023.

[14] Fabian Thomas et al. “RISCVuzz: Discovering Architectural CPU Vulnerabilities via Differential Hardware Fuzzing”. In: (2024).

[15] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. “Analysis of Secure Caches Using a Three-Step Model for Timing-Based Attacks”. In: *Journal of Hardware and Systems Security* 3.4 (Nov. 2019), pp. 397–425. ISSN: 2509-3436. DOI: 10.1007/s41635-019-00075-9.

[16] Daniel J. Bernstein. “Cache-timing attacks on AES”. In: 2005. URL: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[17] G. Bertoni et al. “AES power attack based on induced cache miss and countermeasure”. In: *International Conference on Information Technology: Coding and Computing (ITCC’05) - Volume II*. IEEE, 2005, 586–591 Vol. 1. DOI: 10.1109/itcc.2005.62.

[18] Andreas Kogler et al. “Collide+Power: Leaking Inaccessible Data with Software-based Power Side Channels”. In: *USENIX Security*. 2023.

[19] Chao Su and Qingkai Zeng. "Survey of CPU Cache-Based Side-Channel Attacks: Systematic Analysis, Security Models, and Countermeasures". In: *Security and Communication Networks* 2021 (June 2021). Ed. by Petros Nicopolitidis, pp. 1–15. ISSN: 1939-0114. DOI: 10.1155/2021/5559552.

[20] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: The Case of AES". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 1–20. DOI: 10.1007/11605805_1.

[21] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[22] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches". In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 897–912. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss.

[23] Daniel Gruss et al. "Flush+Flush: A Fast and Stealthy Cache Attack". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2016, pp. 279–299. DOI: 10.1007/978-3-319-40667-1_14.

[24] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript". In: July 2016, pp. 300–321. DOI: 10.1007/978-3-319-40667-1_15.

[25] BeagleBoard.org Foundation. *BeagleV-Ahead*. English. Last visit: 2025-02-21. URL: https://www.beagleboard.org/boards/beaglev-ahead.

[26] BeagleBoard.org Foundation. *BeagleV-Fire*. English. Last visit: 2025-02-21. URL: https://www.beagleboard.org/boards/beaglev-fire.

[27] SiFive. *HiFive Unmatched*. English. Last visit: 2025-02-21. URL: https://www.sifive.com/boards/hifive-unmatched.

[28] John Kelsey et al. "Side channel cryptanalysis of product ciphers". In: *Computer Security — ESORICS 98*. Springer Berlin Heidelberg, 1998, pp. 97–110. DOI: 10.1007/bfb0055858.

[29] D. Page. *Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel*. Cryptology ePrint Archive, Paper 2002/169. 2002. URL: https://eprint.iacr.org/2002/169.

[30] Colin Percival. "Cache missing for fun and profit". In: (Aug. 2005). URL: https://www.daemonology.net/hyperthreading-considered-harmful/.

[31] Mengjia Yan et al. "Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2019. DOI: 10.1109/sp.2019.00004.

[32] Samira Briongos et al. "RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020,

pp. 1967–1984. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/briongos.

[33] Yoongu Kim et al. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors". In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 2014, pp. 361–372. DOI: 10.1109/ISCA.2014.6853210.

[34] Fangfei Liu et al. "Last-Level Cache Side-Channel Attacks are Practical". In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 605–622. DOI: 10.1109/SP.2015.43.

[35] Wei Song and Peng Liu. "Dynamically Finding Minimal Eviction Sets Can Be Quicker Than You Think for Side-Channel Attacks against the LLC". In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, Sept. 2019, pp. 427–442. URL: https://www.usenix.org/conference/raid2019/presentation/song.

[36] Pepe Vila, Boris Kopf, and Jose F. Morales. "Theory and Practice of Finding Eviction Sets". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2019, pp. 39–54. DOI: 10.1109/sp.2019.00042.

[37] Moritz Lipp et al. "PLATYPUS: Software-based Power Side-Channel Attacks on x86". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 355–371. DOI: 10.1109/SP40001.2021.00063.

[38] Moritz Lipp, Daniel Gruss, and Michael Schwarz. "AMD Prefetch Attacks through Power and Time". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 643–660. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/lipp.

[39] Yangdi Lyu and Prabhat Mishra. "A Survey of Side-Channel Attacks on Caches and Countermeasures". In: *Journal of Hardware and Systems Security* 2.1 (2017), pp. 33–50. DOI: 10.1007/s41635-017-0025-y.

[40] Chaoqun Shen, Congcong Chen, and Jiliang Zhang. "Micro-architectural Cache Side-Channel Attacks and Countermeasures". In: *Proceedings of the 26th Asia and South Pacific Design Automation Conference*. ASPDAC '21. ACM, Jan. 2021, pp. 441–448. DOI: 10.1145/3394885.3431638.

[41] Antoon Purnal et al. "Systematic Analysis of Randomization-based Protected Cache Architectures". In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2021, pp. 987–1002. DOI: 10.1109/sp40001.2021.00011.

[42] Oleksii Oleksenko et al. "Hide and Seek with Spectres: Efficient discovery of speculative information leaks with random testing". In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2023, pp. 1737–1752.

[43] Xaver Fabian, Marco Guarnieri, and Marco Patrignani. "Automatic detection of speculative execution combinations". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2022, pp. 965–978.

[44] Gilles Barthe et al. "Testing side-channel security of cryptographic implementations against future microarchitectures". In: *Proceedings of the 2024*

on ACM SIGSAC Conference on Computer and Communications Security. 2024, pp. 1076–1090.