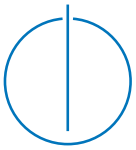
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

A Closer Look at Cache Replacement Policies on ARM

Robert Imschweiler



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

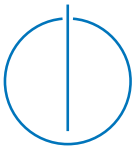
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

A Closer Look at Cache Replacement Policies on ARM

Untersuchung von Cache Replacement Policies auf ARM

Author:	Robert Imschweiler
Examiner:	Prof. Claudia Eckert
Supervisors:	Kilian Zinnecker, M.Sc., Andreas Seelos-Zankl, M.Sc.
Submission Date:	December 10, 2024



I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, December 10, 2024 Robert Imschweiler

Abstract

Modern CPUs rely heavily on caches to speed up memory accesses. Caches are shared resources, and their behavior has been shown to leak information between processes. This side-channel can, for instance, be exploited to retrieve private cryptographic keys or passwords. A large group of these side-channel attacks rely on the targeted eviction of data and instructions from the cache. The efficiency of this eviction increases through knowledge about the implementation details of the cache. The cache replacement policies are particularly relevant since they directly affect the targeted eviction. While the implementation details of these policies are usually not publicly available, they are crucial for assessing the vulnerability of processors and systems to cache attacks. In this work, we thus study cache replacement policies on ARMv8-A CPUs and infer their functionality from careful observation of the cache behavior. Previous research has proposed multiple effective approaches for the x86 architecture. We select two existing frameworks, combine and port them to the ARMv8-A architecture, and add support for hardware debugging probes. With this setup, we infer the replacement policy of the ARM Cortex-A76 L1 data cache, study the pseudo-random replacement policy of the ARM Cortex-A55 L1 data cache, and develop approximations of the currently unknown replacement policy employed by the ARM Cortex-A76 L2 cache. The results show that our framework is capable of revealing implementation details of replacement policies found on ARM CPUs, which establishes a foundation for in-depth risk analysis and for developing next-generation cache replacement policies with increased resilience against cache attacks.

Contents

1 Introduction

Cache side-channel attacks leverage cache behavior to retrieve information about other users of the cache, for example, software processes running on a CPU that uses this cache. This information might be confidential and would typically not be accessible to the attacker. While their inception dates back more than two decades [1, 2], cache side-channel attacks gained widespread recognition in 2018 with the discovery of Meltdown [3] and Spectre [4]. Research has shown various ways to exploit the memory hierarchy as a side-channel—not only on the x86 architecture but also on ARM CPUs [5]. Assessing the vulnerability of a system to cache side-channel attacks thus is crucial for risk analysis. However, microarchitectural details allowing an in-depth risk analysis are usually closed-source intellectual property (IP), even for RISC-V CPUs. To gain a better understanding of the security-relevant aspects, many previous works employed reverse-engineering methods.

Despite the incomplete knowledge of cache implementations, there are cache attack primitives conceptually working on all common cache systems. Two examples are PRIME+PROBE [6] and EVICT+RELOAD [7]. To conduct a PRIME+PROBE attack, an attacker would use a process on the target system to fill target cache sets with their own data. The attacker would then wait and probe whether the victim process accesses memory, which fills the target cache sets and thus evicts the attacker's data. For an EVICT+RELOAD attack, the attacker would use a process on the target system to evict parts of a shared memory object, for example, a shared library, from the cache. The attacker would then wait for the victim process to execute and would then access the targeted shared memory again to check whether it already is in the cache again—which would indicate that the victim accessed it in the meantime.

In order to fill or evict a cache set efficiently, the attacker needs in-depth insight into the microarchitectural behavior of the cache. The cache replacement policy is one of the most relevant properties in this context. In case a cache set is already filled with cache lines, but a new line is to be inserted, one of the old lines needs to be evicted. The cache replacement policy is an algorithm deciding which line should be evicted in such a case. For example, the replacement policy could use a First-in-First-out (FIFO) queue to manage the lines in a set. Real-world replacement policies can differ fundamentally and be significantly more complex. See [8] for an example of how a policy could systematically handle different distances between cache accesses to the same memory address.

The more knowledge attackers have on the details of the cache replacement policy, the better they can manipulate the cache. To secure devices and assess risks regarding microarchitectural attacks, it is therefore important to understand these details and

the effort required to reverse-engineer them. In this work, we thus study replacement policies on selected modern ARM CPUs and investigate how to reverse-engineer them and how they might be exploited for cache attacks such as PRIME+PROBE or EVICT+RELOAD. Our contributions include:

- The adaptation of existing approaches to reverse-engineer cache replacement policies on x86 to ARM CPUs. In particular, we made use of *nanoBench* [9] and *cachequery* [10].
- The enhancement of these approaches by employing a hardware debugging device for direct access to the cache contents. We used a Lauterbach TRACE32 debugger to inspect the L1d cache on the ARM Cortex-A55 CPU and were thus able to directly compile a list containing the evicted elements in the correct order.
- The evaluation of these methods on selected targets: the Radxa Rock 5B and the Avnet RZBoard V2L. The ARM Cortex-A55 and -A76 served as our target CPUs.

As a result, we were able to gain the following insights.

- We determined the specific variant of the PLRU policy implemented for the ARM Cortex-A76 L1d cache.
- After various tests, we conclude that the ARM Cortex-A55 L1d cache most likely uses a pseudo-random replacement policy, which either uses a Pseudorandom Number Generator (PRNG) with a very long period or is affected by other factors that hinder a successful reverse-engineering significantly. We discuss this result by reviewing research about the security of pseudo-random replacement policies.
- We developed and tested a hypothetical model for the replacement policy of the ARM Cortex-A76 L2 cache, whose policy is yet to be revealed. In our tests, this policy simulation came close to the behavior of the hardware and thus can serve as a good foundation for future research.

2 Caches, Attacks, and Replacement Policies

In the following, we briefly introduce the relevant aspects of computer architecture, caches, and cache attacks necessary for understanding the work we will present later.

2.1 Memory Hierarchy

To overcome the so-called von Neumann bottleneck [11], the memory system of all common computer architectures employs multiple caches between CPU and RAM. This is commonly referred to as the memory hierarchy. There are usually up to three cache levels inside this hierarchy, with the lower levels closer to the CPU and the higher ones near the main memory. Each cache is named according to its level, so in the case of three cache levels, there would be L1, L2, and L3 caches. The L1 cache is usually split into an instruction and a data cache, denoted as L1i and L1d. The L1 cache is typically core-private, so a system with four CPU cores would have four instances of L1i and L1d caches each.

As the distance between CPU and cache grows, the access time increases accordingly. On the other hand, the size of the cache also increases. For the ARM Cortex-A76 processor, for instance, the size of the L1 cache is 64KB (per core), and the size of the L3 cache can be as large as 4MB [12].

2.1.1 Cache Structure

The caches we investigate within the scope of this work are set-associative. They operate on cache elements, called cache lines, which are 64-byte large memory blocks. The cache is divided into sets of an implementation-defined but fixed size. This size is specified by the associativity or the number of ways of this cache. Every line can only go into exactly one set of a specific cache. Whenever a line is loaded into a cache level, the cache controller extracts the index of the corresponding cache set from the line's address, as shown in Figure 2.1. Depending on the number of sets, more or fewer bits of the address are used for indexing.

Note that this implies the following: assume that there are two caches, L1 and L2, with L1 having 512 and L2 having 1024 sets. This results in 9 index bits needed to find the target set in the L1 cache and 10 bits needed to determine the set index for the L2 cache. Consequently, two lines can map to the same L1 set but to different L2 sets. However, if two lines map to the same L2 set, they automatically map to the same L1 set in this scenario. See Figure 2.2 for an illustration.

tag	set index	offset in line
#(address bits) - set index + offset	⌈log 2 #sets⌉	⌈log 2 |line|⌉

Figure 2.1: Schematic overview of how the cache controller uses the bits of the address of a cache line [13].

tag	L2 set index e.g. 10 bit → 1024 sets	offset
tag	L1 set index e.g. 9 bit → 512 sets	offset

Figure 2.2: Visualization of set index subsequences. Two addresses can map to the same L1 set but to different L2 sets in case the most significant bit of the L2 set index differs.

Since multiple cache lines can map to the same set in a set-associative cache, the "tag" (as shown in Figure 2.1) is used to differentiate cache lines and ensure that the correct line is used for a given address.

VIPT vs. PIPT Modern CPUs often use virtual memory addressing, where virtual addresses are mapped to physical addresses by the memory management unit. Two processes running on a CPU might use the same virtual addresses but different physical memory. An implementation of a cache needs to decide how virtual and physical addresses should be handled. Higher cache levels than the L1 cache typically follow the Physically Indexed, Physically Tagged (PIPT) scheme. They use the physical address to retrieve both the index of the target cache set and the tag. For L1 caches, however, the Virtually Indexed, Physically Tagged (VIPT) scheme can be favorable. It uses the physical address only for the tag. The set index bits are taken from the virtual address. This allows for parallel indexing and address translation. However, it also requires that the set index bits are either completely included in the page offset bits and thus unaffected by the address translation, or the cache needs to take care of aliasing issues where a physical address corresponds to different virtual addresses. See [14] and [15] for reference and further details.

2.2 Memory Consistency and Cache Coherence

Multiple cache levels and CPU cores require concepts to guarantee that data in memory is not corrupted. In the following, we briefly discuss relevant aspects of modern CPU design that provide a basic understanding of this topic's complexity.

Out-of-order Execution Modern CPUs do not only try to parallelize instruction execution by using pipelining (≈ "start the next instruction before the previous one has finished") and a superscalar architecture (≈ "have multiple execution units"). The CPU also reorders instructions and executes them out-of-order. This can, for instance, help hide the latency of memory operations. However, it still has to be guaranteed that no instruction dependencies are violated. For example, a read from memory that happens after a write to memory in program order must not be reordered in a way that it reads the memory value before the write instruction finishes. If that were to happen, this would be called a "RAW" (read after write) data hazard. The insights presented here and more information can be found in [15].

Memory Consistency Models Reordering load and store instructions is limited by the memory model of the CPU architecture. ARMv8 is defined as weakly-ordered [16, 17]. This is especially relevant when dealing with shared caches. In general, the order of memory operations can be forced using barrier instructions. These instructions may also be required to enforce that previous memory instructions have been completed before further instructions are executed [17]. This aspect is highly relevant for the implementation of the cache-analyzing algorithms we discuss in this thesis.

Memory Write Optimizations If a cache implements a write-through policy, every cache line is immediately written back to the next level or the main memory on modification. A write-back policy implementation only modifies the copy of the line in the cache. Especially write-through caches, but also write-back implementations, employ write buffers. When a dirty cache line needs to be written back to the next level/RAM, this does not always happen immediately. To reduce latency, the memory stores are temporarily put in a write buffer and are either executed in parallel or on demand when the data is needed by the main memory. [15] provides the aspects mentioned here and further details.

Cache Inclusion Policy In a system with multiple cache levels, there are three policies for managing the interaction between different levels: inclusive, exclusive, and non-inclusive. An inclusive policy guarantees that every cache line that is present in a lower-level cache is also held in a higher-level cache. This is not the case for the exclusive policy, where a cache line is always present in exactly one cache level. Policies are called non-inclusive if contents of the lower-level caches may but do not have to be present in higher-level caches. Assume that a cache line is not present in any cache at the moment. When a core now accesses this cache line, it is loaded into the core-private L1 cache of this core and possibly also in the (shared) higher cache levels, depending on the cache inclusion policy. In the case of an inclusive cache, whenever a cache line is evicted from the L2 cache, it also has to be evicted from the L1 cache because the L1 cache always has to be a subset of the L2 cache in this scenario.

***AutoLock* [18]** *AutoLock* is an undocumented feature of ARM CPUs uncovered by Green et al.. The authors detected that *AutoLock* prevents cache lines from being evicted from inclusive caches as long as the line is present in any other cache with which the target cache is inclusive. They state, however, that this does not affect core-private caches.

Memory Coherence With multiple CPU cores and the existence of core-private and shared caches, the cache system needs to manage cache lines that are loaded into multiple caches. The Modified Exclusive Shared Invalid (MESI) protocol is commonly used to accomplish this synchronization work [15].

2.3 ARMv8-A

ARMv8 is a processor architecture developed by Arm Holdings. ARMv8 has been introduced as the successor to ARMv7 in 2011. There are several processor profiles, according to different areas of application: A-, R-, and M-profile [19]. In this thesis, we will focus on ARMv8-A, or ARMv8 in short. The most notable change brought by ARMv8-A is the 64-bit capability. ARMv8-A supports two execution states: AArch32 and AArch64. The former offers 32-bit support with enhanced A32 and T32 instruction sets, and the latter employs the new A64 instruction set [20]. ARM processors are specified as Reduced Instruction Set Computer (RISC); the A64 instruction set specifies a fixed width of 32 bits for all instructions [20].

2.3.1 Memory Fences and Cache Maintenance Instructions

First, we need to introduce the Point of Coherency (PoC) and Point of Unification (PoU) [17]. In simple terms, the PoC is unaffected by coherency protocols as the common access point for any memory location. This is typically the main memory (RAM). The PoU defines the point where data and instruction caches come together. This is usually the L2 cache, as it is only common for the L1 cache to be divided into L1i and L1d caches. The PoU is especially relevant for JIT compilers or self-modifying software where the programmer needs to take care that the updated instructions are propagated from the L1d to the PoU and then re-fetched to the L1i.

Table 2.1 gives an overview of cache maintenance instructions as they will become relevant within this thesis. Developers can use these operations to manually manage cache contents and, for example, force the cache to write-back ("clean") a modified line to memory. Note that some of these instructions are only accessible with elevated privileges (ARM Exception Level (EL) 1 or higher) or need to be configured appropriately to be used from user space. Table 2.2 lists the important memory barrier instructions for this thesis. The (optional) barrier variant, which can be specified for the `DSB` instruction, will, within the scope of this thesis, always be `sy`. This variant ensures that the instruction

Instruction	Short Description
`DC CVAC, <reg>`	Clean by virtual address to Point of Coherency
`DC CVAU, <reg>`	Clean by virtual address to Point of Unification
`DC CIVAC, <reg>`	Clean and invalidate by virtual address to Point of Coherency
`IC IALLU`	Invalidate all to Point of Unification
`IC IVAC, <reg>`	Invalidate by virtual address to Point of Coherency
`IC IVAU, <reg>`	Invalidate by virtual address to Point of Unification
`DC CISW, <reg>`	Clean and Invalidate data cache by set/way.

Table 2.1: Selected Cache Maintenance Instructions [17].

Instruction	Short Description
`DSB <variant>`	Data Synchronization Barrier, guarantees that all preceding memory accesses are completed
`ISB [sy])`	Instruction Synchronization Barrier, (re-)fetches all subsequent instructions

Table 2.2: Selected Memory Barrier Instructions [17].

acts as a "full system barrier" on both load and store operations [17]. The `ISB` instruction has `sy` as the only optional variant, which is why it can be omitted.

2.4 Cache Attacks

Cache attacks use the cache behavior as a side-channel to gather information about other processes running on the same system. The victim process might, for example, access memory locations depending on a secret value. This is the case for software implementations of AES, for instance, where the lookup in the S-box table depends on the value of the secret key. By monitoring the cache behavior, the attacker gains insight into cache accesses made by the victim. Those might form a pattern and thereby reveal information that should not be accessible to other system users.

Flush-based Cache Attacks

Flush-based cache attacks rely on efficient, targeted removal of data and instructions from the cache hierarchy. Therefore, they need user-space access to relevant cache maintenance instructions, which may be limited or, depending on the architecture, may not exist in the first place. Additionally, flush-based attacks require shared memory between the attacker and the victim.

Eviction-based Cache Attacks

Eviction-based cache attacks work by causing conflicts with the cache accesses of another process by working with addresses that use the same cache set(s). An example is PRIME+PROBE [6]. This attack primitive requires the attacker to evict the contents of target set(s) with their own data, wait for the victim process to execute, and then check if re-accessing the data used for eviction results in a cache hit or miss. In case of a miss, this would indicate that the victim process might have used specific data, which then caused the attacker's data to be evicted from the observed cache sets.

Flush- vs. Eviction-based Cache Attacks

Flush-based cache attacks are faster, but they rely on cache maintenance instructions, which may not be available or accessible, and they require the attacker and victim processes to have shared memory. Eviction-based attacks are more generic and thus harder to defend against. However, they are also more complex to implement and might face new countermeasures based on techniques such as index randomization, where set indices no longer have a fixed mapping to the corresponding bits of the (virtual) memory address [21].

2.5 Cache Replacement Policies

In Subsection 2.1.1, we already introduced how cache lines are loaded into the cache, or rather into the specific set inside the cache to which the line corresponds. However, since there are significantly more lines than there are slots in the cache, more lines map to the same set than would actually correspond to the associativity (= set size). As a result, the cache might need to insert an incoming line into a set where all slots are already occupied. Consequently, the cache controller needs to employ some algorithm to decide which cache line should be evicted to make room for the incoming one. There are two popular algorithms commonly used for this kind of problem in Computer Science: FIFO and Least Recently Used (LRU) queues. Figure 2.3 and Figure 2.4 provide a simple visualization of those two algorithms.

2.5.1 Properties of Cache Replacement Policies

Replacement policies aim to minimize cache misses and maximize cache hits. However, the ideal algorithm to achieve this depends heavily on the access pattern. When a program scans through an array, for example, it may not be useful to favor recent cache entries because they are unlikely to be needed again soon. For other workloads where the same data is used recurrently with temporal locality, the cache should keep recent data available.

In [8], the authors analyze the distance with which cache elements are re-referenced. This can be *near-immediate* for small working sets and *distant* for scans. Based on those

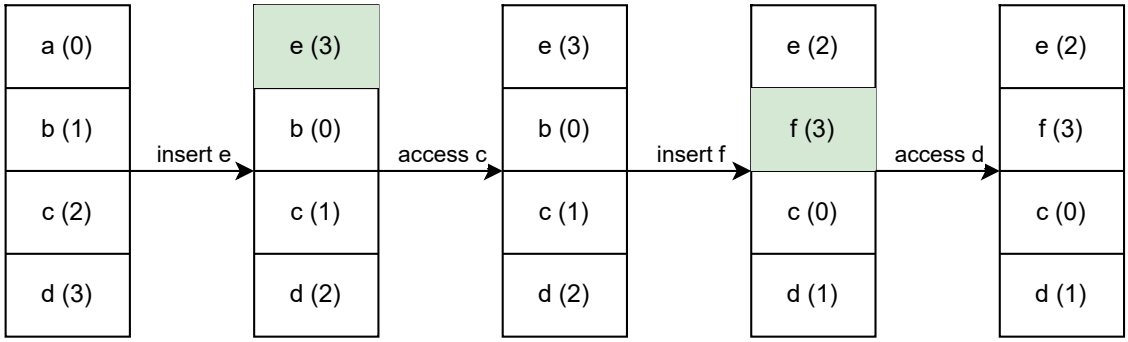


Figure 2.3: Visualization of the FIFO algorithm. There is an insertion index indicating the position in the queue. The first element, i.e., the one with the lowest insertion index, gets evicted.

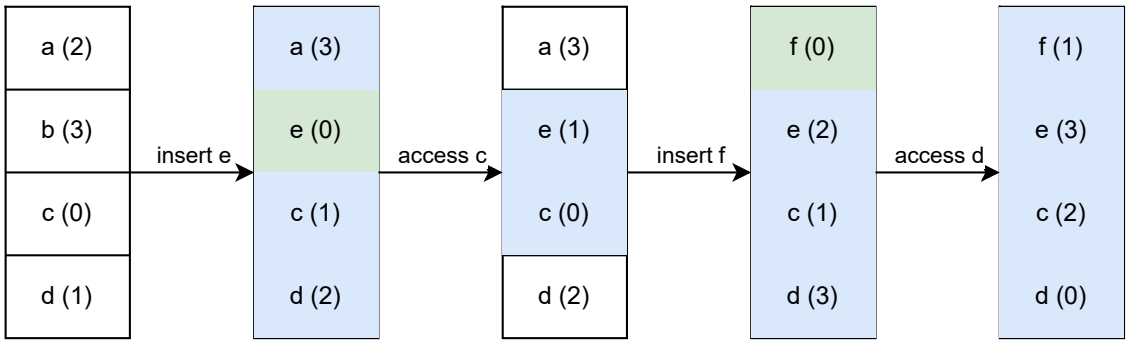


Figure 2.4: Visualization of the LRU algorithm. There is an age number associated with each element. The element with the oldest age gets evicted. Every time an element is accessed, its age number is updated, as well as the age numbers of other elements as needed.

definitions, they define four types of workloads:

- Recency-friendly Access Patterns: recent elements are to be favored because the re-reference distance is short. A stack may be a typical example of this.
- Thrashing Access Patterns: cyclic accesses of cache elements "thrash" the cache whenever their re-reference interval is larger than the cache.
- Streaming Access Patterns: elements are never re-used, the re-reference interval is infinite, and there is no locality. Under this condition, no replacement policy leads to cache hits.
- Mixed Access Patterns: tasks with locality are interrupted by scan activities. There is a mix of near-immediate and distant re-referencing. In case the cache is too small to fulfill both requirements, the policy should try to keep the near-immediate recurring elements and not evict them in favor of those whose re-referencing interval is large.

This leads to two main properties of replacement policies: *scan-resistance* and *thrash-resistance* [8]. While the former prevents working set elements from being evicted by scans, the latter tries to keep elements of a cyclic access pattern that does not entirely fit in the cache.

Adaptive Replacement Policies and Set Dueling Based on [22], Wong [23] describes that, in contrast to usual cache replacement policies, adaptive ones employ two policies and decide dynamically which one is best suited for the current situation. This is done by concurrently using them on distinct portions of the available cache sets, also called *dedicated sets*—we may also call them *leader sets*. The remaining sets, referred to as *follower sets*, then use the policy that has been determined to perform better.

2.5.2 LRU and PLRU

Especially LRU is often used in practice for replacement policies. This and the following explanation of the Pseudo-LRU (PLRU) algorithm are based on the documentation by Grund and Reineke [24]. True LRU would be too complex for an implementation in hardware, especially for larger associativity values. Therefore, hardware designers use approximations of LRU, called Pseudo-LRU (PLRU). Among PLRU algorithms, the tree-based variant is a common choice. Figure 2.5 visualizes an example of a possible variant of this concept. In hardware, the arrows of the tree would be implemented by using bit values 0 and 1 to determine the "direction" to follow. The concept of tree-based PLRU algorithms leaves some ambiguities, which lead to slightly different behavior of hardware implementations. Grund and Reineke use the terms *Tree-fill* and *Sequential-fill* to describe whether filling empty or invalid slots always follows the tree arrows, as shown in Figure 2.5, or whether such slots are filled sequentially, as shown in Figure 2.6, disregarding the tree. It is important to keep in mind that these are not the only two possibilities in which specific PLRU variants can differ.

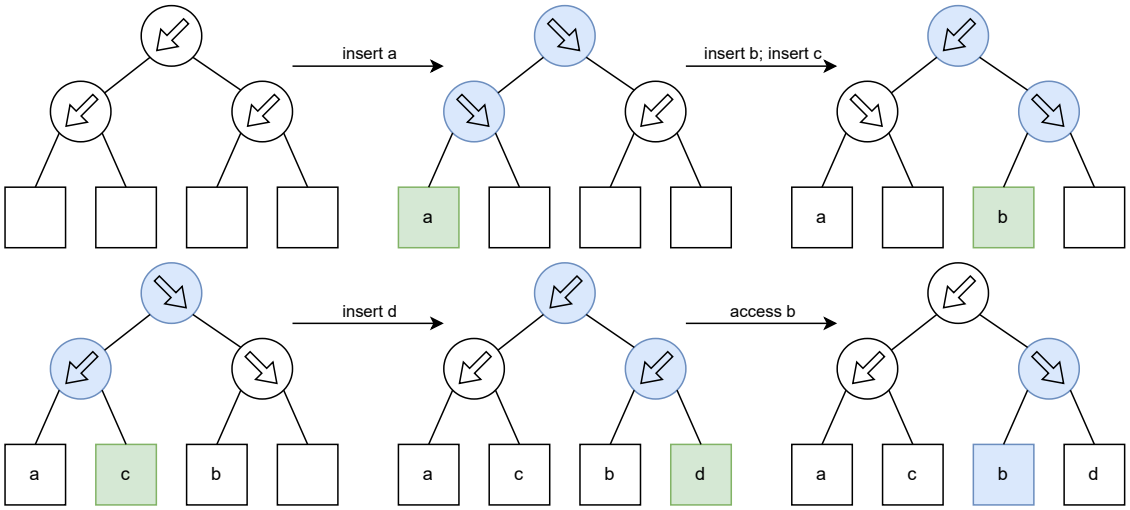


Figure 2.5: Visualization of a tree-based PLRU algorithm (tree-fill variant). Every insertion follows the arrows to the target slot. The direction of the arrows on this path is then inverted. The same happens on accesses, except for the case that an arrow already points in the opposite direction.

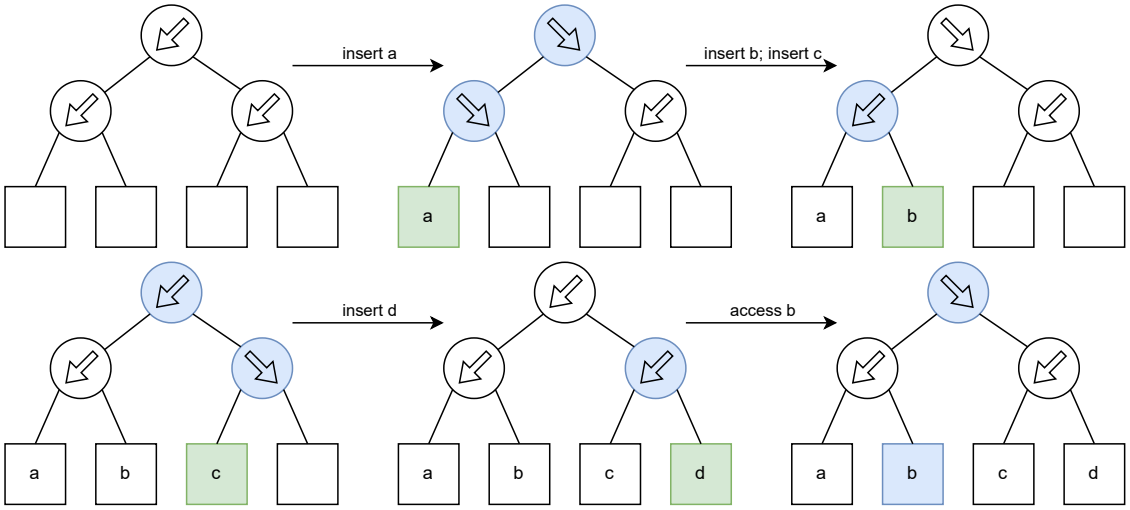


Figure 2.6: Visualization of a tree-based PLRU algorithm (sequential-fill variant). Insertion into empty/invalid slots happens sequentially. The direction of the arrows on the insertion path is still inverted in this visualization.

2.5.3 LIP Variants [22]

The authors in [22] start by splitting the definition of a replacement policy into two parts: the *victim selection policy* and the *insertion policy*. While the first selects the cache element that should be evicted, the second specifies the position the newly inserted element should take in the queue. LRU, for example, specifies that a new element is inserted at the position of the most recently used element in the queue. This is what the authors call *MRU Insertion*. In contrast to this design, they propose the LRU Insertion Policy [22] (LIP) policy, which inserts a new element at the position of the least recently used element in the queue. This element will only be promoted to the MRU position if it causes a cache hit while residing in the LRU position of the queue. A "recency stack" then holds the elements promoted to the MRU position. This policy thus does not use the same aging mechanism as traditional LRU. LIP results in fresher elements being more likely to be evicted. While this counteracts effects from thrashing or cyclic patterns, it may not adapt well to changes in the working set. The authors introduce Bimodal Insertion Policy [22] (BIP) to deal with this issue and use aging and an infrequent choice of the LRU instead of the MRU position for insertion.

Since LIP/BIP both are not designed for workloads where the classic LRU policy would perform best, the authors propose Dynamic Insertion Policy [22] (DIP) as a dynamic policy with set dueling to determine the best-suited policy for the current situation among LRU and BIP.

2.5.4 RRIP Variants [8]

Based on the re-referencing time or distance, the authors in [8] define the concept of Re-Reference Interval Prediction [8] (RRIP) and, building on this, the Static RRIP [8] (SRRIP) and Dynamic RRIP [8] (DRRIP) replacement policies. Both algorithms are scan-resistant, with DRRIP also being thrash-resistant. SRRIP works by predicting which cache elements will be re-referenced in the near-immediate future and thus should be favored. This prediction can be based on whether there has been a hit on a specific element or on the frequency with which this element has been accessed. Thus, each cache line is associated with a predicted re-referencing value of a certain fixed size (e.g., 2 bits). DRRIP extends SRRIP with a dynamic approach using set dueling, thereby deciding whether new cache elements should initially be expected to be re-referenced sooner or later.

2.5.5 Other Policy Variations

Intel mentioned the use of a policy called Quad-Age LRU [25] (QLRU). There is no further official documentation, but there have been attempts to reverse-engineer this policy and possible variants in literature [26], [27]. According to Abel [26], the QLRU variants mainly differ in the *hit promotion policy* and the *insertion age*. Abel mentions that the previously described SRRIP policy can be viewed as a variant of the QLRU if

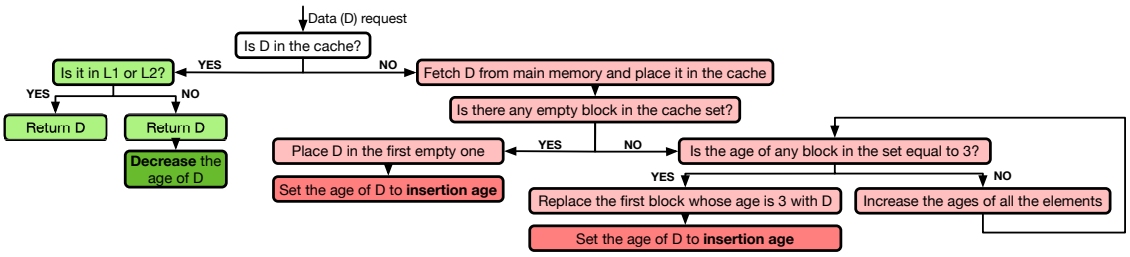


Figure 2.7: Visualization of QLRU by Briongos [27].

implemented with a 2-bit state, thus allowing four different ages (= "quad-age") [26]. Briongos [27] mentions that the QLRU variants they discovered only differ in the insertion age. Even when set dueling is applied, the different "modes", as the authors call them, only differ in the value of the insertion age. Figure 2.7 shows a visualization of the QLRU principle by Briongos.

Simpler variants of PLRU-like algorithms include Most Recently Used [28, 29] (MRU) and Not Recently Used [30] (NRU). The authors of [29], for example, describe MRU as a policy using one status bit per cache element. It is set to one whenever an element is accessed, except for the case where this would result in all status bits of this set being set to one. In that scenario, all the other status bits of the set are reset to zero. In case of a cache miss, the first element (according to some ordering) with a status bit value of zero is replaced.

While introducing RRIP in [8], the authors mention that a one-bit RRIP is effectively an NRU policy. They describe NRU as a PLRU variant with one status bit (they call it "nru-bit") per cache element, which describes whether the element has *not* been recently used. In case of a cache insertion or a hit, the value of the nru-bit is set to zero. Whenever there is a cache miss, the first element (according to some ordering) with an nru-bit value of one is replaced. In case all nru-bits of a cache set are zero, they are all set to one so that the eviction candidate can be determined as usual.

2.5.6 Cache Replacement Policies on ARM

Research shows that Intel CPUs often implement some variant of LRU/PLRU/QLRU or SRRIP policies [5, 27, 31, 32]. ARM CPUs are known to employ pseudo-random replacement policies [5, 33–35], along with different other policy variants [5, 34, 35], e.g., PLRU [5, 35, 36]. Some ARM CPUs even allow configuring the replacement policy [34]. The Technical Reference Manual (TRM) of the ARM Neoverse CMN-700 Coherent Mesh Network explicitly specifies that "optionally, CMN-700 supports an enhanced LRU (eLRU) cache replacement policy that you can enable by setting a bit in the configuration register." [37] However, details of the employed policies are not documented. Even if, for example, PLRU is specified, this does not provide any implementation details. Additionally, ARM introduces new policies, such as the "Dynamic biased replacement policy" of the Cortex-A76 [36], which is not further specified in the TRM. Also, the

concrete implementation of the pseudo-random replacement policies remains unknown to the best of our knowledge.

3 Existing Approaches to Cache Replacement Policy Reversing

There are several existing approaches to reversing cache replacement policies on x86 architectures, mostly designed for Intel architectures. The following sections give an overview of the targeted hardware, the reversed policies, and the requirements of the employed techniques.

3.1 Measurement-based (Abel & Reineke)

With `chi` [38], "a measurement-based cache hierarchy inference tool", Abel and Reineke implemented a methodology to reverse cache replacement policies. They published their approach over the course of several publications [31, 39–41]. Their main objective is portability to different x86-based platforms. They therefore prefer but do not depend on using Performance Monitoring Units (PMUs) for measuring memory access times.

They formalize the view on cache replacement policies by creating an equivalence relation over the cache behavior. Thereby, two caches are called "observationally equivalent" iff they show the same number of cache misses for any given address sequence. If that is not the case, they call the analyzed caches "observationally different". Based on this definition, they define their goal as finding the correct parameters for a *cache template* such that it is observationally equivalent to an observed cache on a given target.

In order to identify cache replacement policies, the authors introduce the concept of a *permutation vector* $\Pi = \langle \Pi_0, \ldots, \Pi_{A-1}, \Pi_{miss} \rangle$ (for associativity A) which shows how a cache set has been updated by the access of the i^{th} element of the set or by a cache miss in this set. Table 3.1 shows the permutation vectors for the LRU and FIFO replacement policies. The vectors can be seen as a representation of an abstract queue containing the cache elements in the order in which they would be evicted, i.e., the last element of the queue is the first to be evicted and vice versa. It can be observed that for LRU, the most recently accessed element always moves to the top of the queue. In contrast, for the FIFO policy, accessing cache elements does not change their order in the queue. Π_{miss} is generally set to $(A-1, 0, 1, \ldots, A-2)$, which means that in case of a cache miss, the new element will be prepended to the existing queue of elements, and the last element in this queue will be evicted. The authors claim that this description of the miss behavior holds for all replacement policies they encountered [31].

The permutation vectors allow for the definition of *permutation policy templates*, which can represent the behavior of replacement policies. Again, there is an equivalence

LRU policy	FIFO policy
$\Pi_0^{LRU} = (0,1,2,3,4,5,6,7)$	$\Pi_0^{FIFO} = (0,1,2,3,4,5,6,7)$
$\Pi_1^{LRU} = (1,0,2,3,4,5,6,7)$	$\Pi_1^{FIFO} = (0,1,2,3,4,5,6,7)$
$\Pi_2^{LRU} = (2,0,1,3,4,5,6,7)$	$\Pi_2^{FIFO} = (0,1,2,3,4,5,6,7)$
$\Pi_3^{LRU} = (3,0,1,2,4,5,6,7)$	$\Pi_3^{FIFO} = (0,1,2,3,4,5,6,7)$
$\Pi_4^{LRU} = (4,0,1,2,3,5,6,7)$	$\Pi_4^{FIFO} = (0,1,2,3,4,5,6,7)$
$\Pi_5^{LRU} = (5,0,1,2,3,4,6,7)$	$\Pi_5^{FIFO} = (0,1,2,3,4,5,6,7)$
$\Pi_6^{LRU} = (6,0,1,2,3,4,5,7)$	$\Pi_6^{FIFO} = (0,1,2,3,4,5,6,7)$
$\Pi_7^{LRU} = (7,0,1,2,3,4,5,6)$	$\Pi_7^{FIFO} = (0,1,2,3,4,5,6,7)$

Table 3.1: Permutations Π_i showing the state of a given cache set after accessing the i^{th} element (associativity 8) [31].

relation over the permutation policies. Iff their permutation vectors match for every $i \in \{0,\ldots,A-1\}$ and *miss*, two *permutation policies* are called "observationally equivalent". The authors emphasize that the set of permutation policies is small enough to be feasible for inference in practice, given that the associativity is restricted to "realistic" values. Before being able to start working on the caches, the authors need to introduce one more abstraction, *logical cache set states*. They provide a mapping between the logical position in a cache set (the "way") and the stored element. Thereby, some of the following considerations can be made regardless of the physical position of a given element in the cache set.

The replacement inference algorithm [31] identifies the permutations Π_i and thus constructs the permutation vector. From a high-level view, for every Π_i, the algorithm:

- first initializes the chosen cache set with a known logical cache set state by accessing a sequence of addresses which all cause a cache miss in this set and thus are being inserted into it such that the resulting logical cache set state is $[a_0,\ldots,a_{A-1}]$ (A being the associativity and a_i being the i^{th} address; the addresses have been accessed in reverse order).
- then accesses a_i, which triggers permutation Π_i.
- finally identifies the resulting logical cache set state. For every a_k with $k \in \{0,\ldots,A-1\}$, the procedure iteratively ($j = A-1$ down to $j = 0$) generates up to j cache misses and then checks whether the access to a_k yields exactly one more cache miss. If there is none, then j needs to be decreased because a_k must have already been accessed during the eviction of the j previous elements. Since this is an iterative procedure and the logical cache set state needs to be restored every time, the whole algorithm needs to be repeated until the position of every a_k has been determined and the permutation Π_i is identified.

The authors then adapt the aforementioned algorithm to make it more robust and usable in practice, where the measurements of the used performance counter library

PAPI [42] alone could affect the results significantly. Also, they do not require exclusive access to a CPU core—thus, other processes could also change the cache state. The most relevant adaptations of their algorithm are:

- measuring N cache sets simultaneously instead of only one. This means that in the last step of the algorithm, the targeted number of cache misses for every element a_k needs to be N instead of 1. Since the algorithm now works on N cache sets in an interleaving way, the authors note that this also prevents the test program from being affected too much by pattern-based memory prefetching.
- employing "pointer chasing", by which the authors mean that each accessed memory address contains a pointer to the next memory address to access and so on, forming a chain of pointers. According to the authors, these pointer chains help to reduce the impact of non-blocking caches and out-of-order execution.

The authors explicitly do not employ some other simplifications to minimize interference, such as using flush instructions to establish the initial state of the cache set or using one CPU core exclusively without competing with other programs. As mentioned above, they highly value portability and claim that their results show that this does not come at the expense of poor inference precision.

As a result of their work in [31], for example, the authors managed to reverse LRU approximations on Intel and AMD CPUs, some of which had been unknown before.

3.2 Learning Cache Replacement Policies using Register Automata

Rueda 2013 [43] shows that the approach described in Section 3.1 does not work for all replacement policies seen in practice. The problem with permutation vectors is that they need to be able to represent all possible states of the policy as a permutation of the elements in the cache set.

For a tree-based PLRU, this starts to become difficult since there can be the same "order" of leaves in the tree but with different direction directives in the nodes above. This can be mitigated by what Rueda calls "normalization of PLRU", where the list of elements in the permutation vector is always sorted in a way that the direction directives in the tree nodes have a fixed known state and thus are no longer required. Regarding the semantics of permutation vectors, it is sensible to create a sorting according to the order in which the elements would be evicted, with the first element chosen first. The mitigation described for the tree-based PLRU policy is not possible for the MRU policy because there are too many states to be represented by a vector of the cache set size.

However, Rueda does not present a working approach to infer replacement policies on hardware. The scope of the author's work is restricted to

- testing the creation of automatons by the LearnLib [44] software, based on simulated replacement policies.

Figure 3.1: CacheQuery system design by Vila [32].

- creating a C-program using the Performance Application Programming Interface [42] (PAPI) library to generate a sequence of cache miss and cache hit symbols for a given sequence of memory accesses.

3.3 CacheQuery: Learning Replacement Policies from Hardware Caches

Vila et al. [32] advance the application of automata learning to the inference of cache replacement policies. Their work also makes use of LearnLib [44]. Since the complexity of the cache state models increases exponentially for larger associativity values, this approach cannot handle more than eight ways in hardware or 16 ways for software-simulated policies. However, the authors mention that it might be possible to further improve this situation.

Figure 3.1 shows a high-level overview of CacheQuery. There are three main components: LearnLib, Polca, and CacheQuery. They interact with each other to cross multiple abstraction levels. LearnLib has a high abstraction level and requests the information it needs to learn an automata for a replacement policy from Polca. Polca generates a suitable query that references single memory blocks on an abstract level. CacheQuery retrieves this query and maps every abstract memory block to an actual one. The accesses can then be made, and the results of the hit/miss measurements are passed back through the pipeline.

The part of the system that will be most relevant for this thesis is the CacheQuery kernel module and an associated Python script [10]. The combination of those two elements will be referred to as *cachequery* (lower case) in the following. *cachequery* receives a query in MemBlockLang [32] (MBL). For most of the applications we will look at within the scope of this thesis, we will focus on queries that include memory block accesses and potential hit/miss measurements. Consequently, the queries we use will consist of sequences of block IDs (strings or numbers), potentially supplied with a question mark indicating that this memory access should be measured to determine whether it was a hit or a miss. Another useful MBL feature is the "!" operator, which can be an attribute to a memory block and thus indicates that this block should be flushed, or it can be a standalone element of the query and indicate that the cache should be invalidated. On x86, the cache invalidation can be implemented using the `wbinvd` instruction [10]. Some example queries are:

- `a b c d m a?` This query accesses five memory blocks and then the first again. This last access will be measured. In a four-way associative cache, this query can be used to determine whether the oldest element will be evicted first.

- `a? b? c? d? e? f? g? h?` Here, eight different memory blocks are accessed, and all accesses are measured.

- `<query ...> ! <... query continued>` Flush the cache in between other operations. The authors implemented this using the `wbinvd` instruction.

There are some more advanced features of MBL (see [45] or [10] for a more comprehensive overview) which, however, are only "syntactic sugar" for what we have already seen. Since they are not needed to understand the rest of this thesis, they are not presented here.

Successfully applying CacheQuery to reverse a new policy comprises two steps: learning the automata and synthesizing an explanation of this automata using Sketch [46]. The second step generates a human-readable algorithm from the automata in (pseudo)-code. The results of the work shown in [32] on hardware include the learning of PLRU policies on several Intel CPUs, as well as learning two new policies for the L2 and L3 caches of Intel Skylake and Kaby Lake architectures and synthesizing explanations for them. A notable drawback of the approach presented in [32] is that the synthesis step can not handle PLRU policies because of their tree state spanning all cache lines of a set.

3.4 Measurement-based (Briongos)

Briongos et al. [27] focus on the replacement policies of the Last Level Cache (LLC) on Intel CPUs. They manually define the behavior of several cache replacement policies and then compare how well they match the actual behavior of the system. This is done using a simple algorithm with two arrays: one holding the addresses to access and one containing control bits indicating which cache line(s) are candidates for eviction in case of a cache miss. The control bits can have more than two states; in the example in the paper, the authors use three states to model the NRU policy (-1: line empty, 0: line not recently used, 1: line recently used). Then, they check whether the observed behavior of the hardware matches one of their manually defined policies.

For their experiments, the authors employ several cache access patterns that can reveal different aspects of replacement policies. As an example, they explain that a PLRU policy will evict all previous data as soon as all addresses in a conflicting eviction set are accessed. A notable finding of the authors regarding the detection of leader sets is that it is crucial not to test the sets of a cache in order because this might hide which sets actually have a fixed policy. Figure 2.7 shows the QLRU policy as observed by the authors. According to their findings, the QLRU variants employed by the CPUs they tested only differed regarding the insertion age.

3.5 Abel 2020

In [26], Abel develops *nanoBench* [9], which is a tool for benchmarking small code portions with low overhead and high precision on x86 architectures. While *nanoBench* acts as a "backend" for cache accesses, Abel also adds further tools that can be used to access the features provided by *nanoBench* in a more automated way and from a higher abstraction level. Those tools, or "frontend scripts" as one might call them, allow, for example, to compare software-simulated replacement policies with the behavior of the hardware using automatically generated queries. Those queries are based on a simplified MBL dialect and are transformed from abstract to concrete memory blocks by the backend.

One important development of the approach by [26] compared to [31] is that it is no longer assumed that there is only one cache policy for all cache sets. Sets can now be handled individually. There are separate tools, for instance, to analyze ages of cache elements or to detect which sets use a fixed policy in case the cache employs set dueling. Additionally, Abel identified issues regarding the possibility of resetting the policy state by flushing the cache using, for instance, the `wbinvd` instruction. He noticed that this might not reset the policy state and thus lead to seemingly non-deterministic results. He mitigates this behavior by using selected sequences of cache accesses to elements of an eviction set.

As a result, Abel was able to identify multiple PLRU and QLRU variants as well as the MRU policy on several Intel CPUs. The reversing worked across multiple cache levels; the results include policies for L1d, L2, and L3 caches.

3.6 Overview of the Related Work

Table 3.2 shows a tabular representation of the most important properties of the discussed approaches to reversing cache replacement policies.

Targeted Hardware	Reversed Policies	Hardware Requirements
x86 (Intel, AMD)	arbitrary in theory, LRU/-PLRU variants in experiments; assumes that n consecutive cache misses fill the whole cache set (n being the associativity)	PMU (preferred) / execution-time measurements (fall back); non-inclusive cache hierarchy; way size of L2 $>$ way size of L1; huge pages

(a) by Abel (pre-2020) [31, 40, 41].

Targeted Hardware	Reversed Policies
- (simulations)	FIFO (up to 5 ways); LRU (up to 5 ways); PLRU as Tree-LRU (up to 4 ways); MRU (up to 4 ways)

(b) by Rueda (2013) [43].

Targeted Hardware	Reversed Policies	Hardware Requirements
x86 (Intel)	PLRU (tree-based) for associativity 8; two new policies for assoc. 4	only works on data caches; associativity $\leq$ 8; performance counters, time stamp counter, or counting core cycles; immediate load operations (`movabs rax, qword [address]`); memory fences; disable hardware prefetchers, hyper-threading, frequency scaling, and other cores
simulated	FIFO and PLRU (up to 16 ways); MRU (up to 12 ways); LIP, LRU, SRRIP-HP and SRRIP-FP (up to 6 ways)	-

(c) by Vila et al. (2020) [32].

Targeted Hardware	Reversed Policies	Hardware Requirements
x86 (Intel)	QLRU	huge pages; `lfence` instruction

(d) by Briongos et al. (2020) [27].

Table 3.2: Overview of existing approaches to reversing cache replacement policies.

Targeted Hardware	Reversed Policies	Hardware Requirements
x86 (Intel)	PLRU, QLRU, and MRU variants	performance counters (uses nanoBench [9])

(e) by Abel (2020) [26].

Table 3.2: Overview of existing approaches to reversing cache replacement policies, continued.

4 Target Hardware

We use two boards within the scope of this thesis, the Radxa Rock 5B [47] featuring the Rockchip RK3588 [48] chip and the Avnet RZBoard V2L [49] featuring the Renesas RZ/V2L [50] chip. The RK3588 has four ARM Cortex-A55 cores and four ARM Cortex-A76 cores; the RZ/V2L features two ARM Cortex-A55 cores. The Linux kernel running on the Rock 5B is based on modified sources provided by Radxa [51][1], derived from Linux kernel version 5.10.66. On the RZBoard, the Linux kernel is based on modified sources provided by Avnet [52][2], derived from Linux kernel version 5.10.175. Table 4.1 provides an overview of the technical details of the board chipsets. The data has been collected from official documentation [48, 49], the device tree (`dts`) files from the respective Linux kernel sources [53, 54], and through inspection of the information provided by the Linux `sysfs` files. Figure 4.1 and Figure 4.2 show a graphical overview of the Rock 5B and the RZBoard boards.

4.1 Cortex-A55

Arm describes the Cortex-A55 as a "mid-range, low-power core" [56]. It is designed as a successor to the Cortex-A53 CPU and is used in mobile products such as the Samsung Galaxy S9 [57], the Samsung Galaxy S20 family [58], or the Redmi Note 12 Pro Series [59]. Table 4.2 provides an overview of the technical specification of the A55. In the following, we briefly mention some A55-specific aspects that are relevant for analyzing its cache behavior.

PMU events L2 PMU events such as `L2D_CACHE` will actually refer to the L3 event in case there is no L2 cache configured [56].

Stream-based prefetching In case the core detects a "regular pattern" [56] of memory accesses, it starts prefetching automatically. While doing this, addresses further away from the currently loaded one will be prefetched into the L3 cache, not directly into the L1 cache. Only if the prefetched data is actually needed will it be moved from the L3 to the L1 cache [55, 56].

CCSIDR, Cache Size ID Register The following information is based on [56]. The core-private L2 cache of the A55 is optional. In case there is no L2 cache present, the

[1] commit `52f51a2b5ba178f331af62260d2da86d7472c14b`
[2] commit `c197622df526c82ae9e3674e06b4092dac33eafa`

		RK3588 (Rock 5B)		Renesas RZ/V2L (RZBoard V2L)
CPUs		4×Cortex-A55	4×Cortex-A76	2×Cortex-A55
Main Memory (kB)		7,879,204		1,334,020
Cache L1d	Type	core-private	core-private	core-private
	Size (KiB)	32	64	32
	Ways	4	4	4
	#Sets	128	256	128
	Line Size (B)	64	64	64
Cache L1i	Type	core-private	core-private	core-private
	Size (KiB)	32	64	32
	Ways	4	4	4
	#Sets	128	256	128
	Line Size (B)	64	64	64
Cache L2	Type	core-private	core-private	-
	Size (KiB)	128	512	-
	Ways	4	8	-
	#Sets	512	1024	-
	Line Size (B)	64	64	-
Cache L3	Type	shared		shared
	Size (KiB)	3072		256
	Ways	12		?
	#Sets	4096		?
	Line Size (B)	64		64 [55]

Table 4.1: Technical details for the chipsets of the analyzed boards [48, 49].

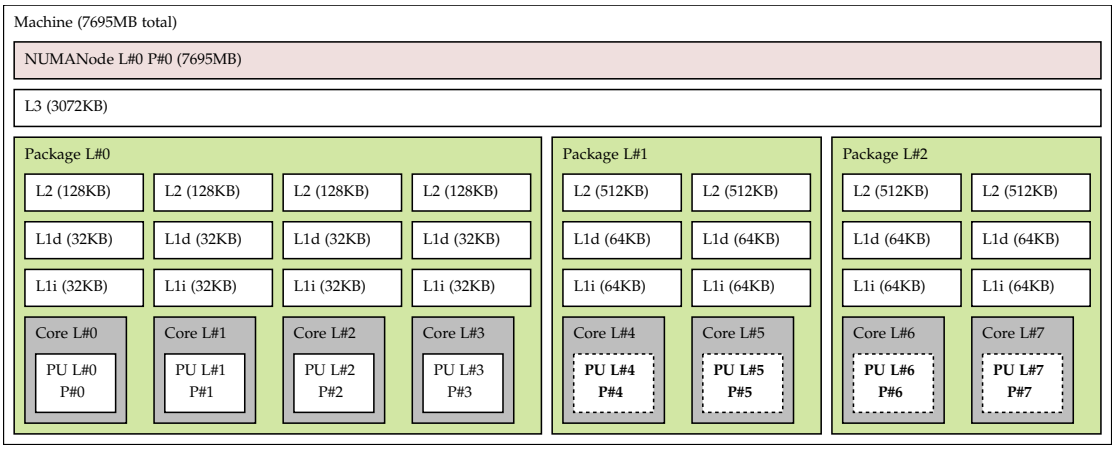


Figure 4.1: System topology of RK3588 generated by `lstopo`. Cores #0-3 are of type Cortex-A55, cores #4-7 are of type Cortex-A76.

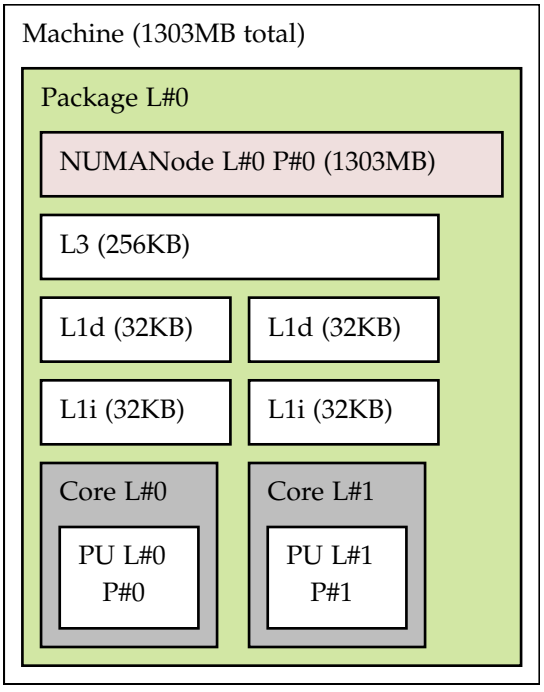


Figure 4.2: System topology of RZBoard V2L generated by `lstopo` (modified). Cores #0-1 are of type Cortex-A55.

System Part	Properties
Instruction sets	full implementation of Armv8.2-A A64, A32, T32
Pipeline	in-order
L1i	• 64-byte cache lines • associativity: 4 • 128-bit read interface to L2 • VIPT
L1d	• 64-byte cache lines • associativity: 4 • 64-bit read path to datapath • 128-bit write path from datapath • VIPT, behaves as PIPT
L2	• 64-byte cache lines • associativity: 4 • strictly exclusive with L1d • pseudo-inclusive with L1i • private per-core • PIPT
Cache Coherence Protocol	MESI
PMU	PMUv3
CoreSight	CoreSightv3

Table 4.2: Cortex-A55: overview of the technical specification [14, 56].

L3 cache will be the L2 cache from the point-of-view of the core, thus the `CCSIDR` will actually return the information for the L3 cache when the L2 cache is selected and `UNKNOWN` if the L3 cache is requested.

4.2 Cortex-A76

Arm describes the Cortex-A76 as “high-performance and low-power” [36]. A prominent example of its application in practice is the Raspberry Pi 5 [60]. The Cortex-A76 is also combined with the Cortex-A55 CPU, such as in the Samsung Exynos 990 processor [58]. Table 4.3 provides an overview of the technical specification of the A76. In the following, we briefly discuss some of the available information about the “dynamic biased” cache replacement policy employed by the L2 cache of the A76.

Predecessor: Cortex-A75 The Cortex-A76 is the direct successor of the ARM Cortex-A75 CPU [61]. The L2 cache replacement policy of the A75 is known to be "biased" towards instructions [62]. This most likely implies that the policy is priority-based and assigns a higher priority to instruction cache lines than to data cache lines, which leads to instruction cache lines being less likely to be evicted, especially by data cache lines.

Dynamic Biased Replacement Policy The L2 cache of the A76 is specified to have a "dynamic biased" replacement policy [3]. While there is no more in-depth information for the A76, there are a few words about a "dynamic biased" replacement policy in the documentation of the ARM Neoverse CMN-650 [63]. The CMN-650 is a coherent mesh network designed to interconnect up to 256 compute clusters. It features an optional system level cache (SLC), which can be configured to use an "enhanced LRU (eLRU)" replacement policy. This "eLRU" is called "Dynamic Biased Replacement Policy". It uses 2 bits per cache element for the re-reference prediction, which is "dynamically adjusted based on a few reference sets" [63]. The same information can be found for the Arm CoreLink CMN-600AE Coherent Mesh Network [64]. This could indicate that the A76 uses set dueling to choose an insertion age and/or a victim selection policy for the follower sets based on the current performance of the reference/leader sets. The documentation of the ARM Neoverse CMN-700 Coherent Mesh Network also mentions the "eLRU" policy as "Dynamic Biased Replacement Policy" [37].

4.3 Common Properties of the Cortex-A55 and -A76

The A55 and A76 target different applications and power models. They are a common combination within the scope of ARMs big.LITTLE concept. As "Armv8.2 - DynamIQ big.LITTLE - DynamIQ Shared Unit" combination, A55 cores serve as the "High-Efficiency CPU (LITTLE)" part while the A76 cores represent the "Efficient-Performance CPU (big)" part [65]. This, however, should not make any difference regarding the core-private caches. Concerning shared caches, research (e.g., [66]) has already shown approaches for cross-core attacks. In the following, we briefly mention common aspects of the A55 and A76 that may be relevant for analyzing their cache behavior.

Cache Miss Behavior The implementation can decide whether it features a critical word-first fill on a cache miss for the L1 cache [36, 56].

Write Streaming Mode If the core detects a pattern where there is a consecutive number of full cache line writes, e.g., for the C-library function `memset()`, cache misses in L1 lead no longer to a line fill in the L1 cache, but the write directly happens in the L2 or L3 cache, or directly in memory in case of the Cortex-A76. Note that this is also

[3] Since issue 0100-00 of the A76 TRM. This and previous issues are confidential. The first non-confidential issue is 0300-00.

System Part	Properties
Instruction sets	• full implementation of Armv8.2-A A64 ... • ... as well as A32 and T32 (only at EL0)
Pipeline	• superscalar • out-of-order
L1i	• 64-byte cache lines • associativity: 4 • 256-bit read interface from L2 • PLRU cache replacement policy • VIPT, behaves as PIPT
L1d	• 64-byte cache lines • associativity: 4 • 256-bit read interface from L2 • 256-bit write interface from L2 • 2×128-bit read path to datapath • 256-bit write path from datapath • PLRU cache replacement policy • VIPT, behaves as PIPT
L2	• 64-byte cache lines • associativity: 8 • strictly inclusive with L1d • weakly inclusive with L1i • “dynamic biased” cache replacement policy
Cache Coherence Protocol	MESI
PMU	PMUv3
CoreSight	CoreSightv3

Table 4.3: Cortex-A76: overview of the technical specification [36].

relevant, e.g., for the PMU event `L1D_CACHE_WB`, which does not count the direct full-line writes to a higher-level cache that do not write to the L1 cache. See [56] and [36] for all the details.

5 Reversing Replacement Policies on ARM

In this chapter, we describe our approach to reversing cache replacement policies on ARM CPUs. We use three main components: *cachequery* [10], *nanoBench* [9], and a Lauterbach TRACE32 debugger.

The CacheQuery project by Pepe Vila et al. [32] uses a Linux kernel module in conjunction with a Python script as the backend, as described in Section 3.3. This combination will be referred to as *cachequery* (lower case) in the following. *cachequery*—in our modified version—provides the backend for the setup that will be used for this thesis.

While *cachequery* provides a complete implementation of MBL, it has only been combined with Polca within the scope of the CacheQuery project. In contrast, while *nanoBench* provides frontend scripts for practical use cases, its backend does not provide the full capabilities of MBL [9]. As a result, in this thesis, we combine *cachequery* with *nanoBench*. Figure 5.1 provides an overview of the used components. This selection enables the use of the *nanoBench* frontend scripts, as well as the connection with Polca and LearnLib, as shown in the original CacheQuery paper [32]. In addition, we create the necessary glue code and concepts to combine *cachequery* with a Lauterbach TRACE32 debugger for an in-depth cache inspection.

cachequery has been built for the x86 architecture. When targeting the ARMv8 platform, all the x86-specific parts have to be adapted. Notable changes include the code generation for the calibration and the query execution, as well as the handling of the core cycle and performance counters. In the following, these changes will be described in more detail. Please note that the short term "ARM" may be used to describe the ARMv8-A architecture.

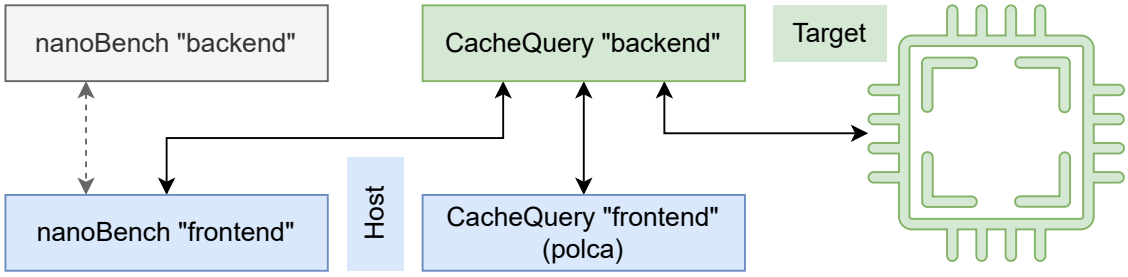


Figure 5.1: We use back- and frontend of the CacheQuery project and in addition the frontend scripts provided by *nanoBench*. The CacheQuery backend runs on the target; the frontend parts can run on another host.

x86	ARMv8-A
`mfence`	`dsb sy`
`cpuid`	`dsb sy; isb`
`clflush <reg>`	`dc civac <reg>`
`wbinvd`	`dc cisw <reg>; ...`

Table 5.1: Instruction equivalences. Some are only equivalent within the given context. See the text for more details.

5.1 Transpiling Code Generation from x86 to ARMv8-A

For many of the used x86 instructions, it is quite straightforward to find an equivalent one for ARM. The `mov` instructions, for example, transpile either to corresponding `ldr`, `str`, or `mov/movk/movz` instructions. Arithmetic and logic instructions such as subtraction, addition, or the XOR operation are also similar. The crucial parts of the code generation adaptation are the cache maintenance and barrier instructions.

Memory Barriers *cachequery* makes use of the `mfence` instruction. This instruction serializes all load and store operations except the instruction stream [67]. For ARM, this behavior can be represented using the `dsb sy` instruction. This can additionally be paired with the `isb` instruction to avoid speculation effects [68, 69]. Whenever a serializing memory barrier is needed, we use this combination of the `dsb` and `isb` instructions. The `cpuid` instruction on x86 translates to the same behavior [67].

Cache Maintenance Instructions x86 offers `clflush` to invalidate a cache line and write it back to memory if needed [67]. For ARM, this corresponds to the `dc civac` instruction, which cleans and invalidates to the point of coherency. Listing 5.1 shows an example of how the usage of this instruction looks like. For the `wbinvd` instruction, there is no direct correspondence on ARM. It writes back and invalidates all caches. Within the scope of *cachequery*, it is used for resetting a specific cache set of the target (either data or unified cache) [32]. On ARM, there is the `dc cisw` instruction, which can be used to clean and invalidate a specific way in a set of a given level of the unified or data caches. Using this instruction for every way in the target set, it is possible to clean and invalidate the entire set. Listing 5.2 demonstrates a concrete usage example for this instruction.

5.1.1 Modes of Measurement

Changing the target from x86 to ARM means using the appropriate performance measuring facilities. The original *cachequery* implementation offered three different ways for determining whether a cache access was a hit or a miss: the timestamp counter, the core cycle counter, and the performance counters [10]. For *cachequery* on ARM,

```
1  movk    x0, #0x9600
2  movk    x0, #0x16bf, lsl #16
3  movk    x0, #0xffc0, lsl #32
4  movk    x0, #0xffff, lsl #48
5  dc      civac, x0
6  dsb     sy
7  isb
```

Listing 5.1: Generated code for flushing a cache line by a virtual address. The `isb` barrier may be necessary in some situations where a long sequence of flushes might lead to some lost instructions.

```
1  movk    x10, #0x604
2  movk    x10, #0, lsl #16
3  movk    x10, #0, lsl #32
4  movk    x10, #0, lsl #48
5  dc      cisw, x10
6  movk    x10, #0x604
7  movk    x10, #0x2000, lsl #16
8  movk    x10, #0, lsl #32
9  movk    x10, #0, lsl #48
10 dc      cisw, x10
11 ...
12 movk    x10, #0x604
13 movk    x10, #0xe000, lsl #16
14 movk    x10, #0, lsl #32
15 movk    x10, #0, lsl #48
16 dc      cisw, x10
```

Listing 5.2: Generated code for cleaning and invalidating set 24 of a 8-way associative level 2 cache.

```
1 dsb sy
2 isb
3 mrs <reg>, pmccntr_el0
4 isb
```

Listing 5.3: Generated code for retrieving the value of the previously configured core cycle counter. The value will be in `reg` after this code snippet. The trailing `isb` may be necessary in some situations where the measured cache access happens directly after this snippet.

```
1 dsb sy
2 isb
3 movz <reg>, <idx>, #0
4 msr pmselr_el0, <reg>
5 isb
6 mrs <reg>, pmxevcntr_el0
7 isb
```

Listing 5.4: Generated code for retrieving the value of a previously configured performance counter. The value will be in `reg` after this code snippet. The trailing `isb` may be necessary in some situations where the measured cache access happens directly after this snippet.

we support the clock cycle counter (`CYCLES`) and the `L(1|2|3)D_CACHE_REFILL` PMU events. To initialize and prepare the PMU, we use a slightly modified ARMv8 PMUv3 library [70]. Since the Linux kernel on one of our targets runs at EL2 instead of EL1, appropriate PMU configuration becomes necessary so that, for example, `PMCCFILTR_EL0` has the correct value to count cycles in EL2.

Core Cycle Counter One method of determining whether a cache access caused a hit or a miss is to use the core cycle counter of the CPU. In that case, *cachequery* performs a calibration phase before every query execution [10]. During the calibration, it determines the average access times for a cache hit or miss in the respective caches. A cache hit will take (considerably) longer as the cache line is fetched from the next higher cache level or the main memory. Listing 5.3 shows an example of how to access the core cycle counter on the ARMv8 platform.

Performance Counters Performance counters are a more fine-grained method of determining if a cache access was a hit or a miss. On ARMv8, the appropriate PMU event to detect a cache miss would be the `LxD_CACHE_REFILL` event [17], meaning `L1D_CACHE_REFILL` and `L2D_CACHE_REFILL` for the L1d cache or the L2 cache, respectively. Listing 5.4 shows how to access a configured PMU event and read out the value

of its counter.

Model Specific Registers / System Registers Both x86 and ARM use special registers to configure the running CPU or get system information. On x86, these registers are called "model specific registers"; on ARM, they are simply called "system registers". x86 model specific registers can be written using the `wrmsr` instruction ("Write to Model Specific Register") and read using the `rdmsr` instruction ("Read From Model Specific Register"). ARM offers `msr` to write and `mrs` to read a system register. There are no direct equivalences between the available registers, though. While we can use `msr/mrs` to interact with the core cycle counter as well as with specific performance counters on ARM, there is a separate instruction (`rdtsc`) to access the time stamp counter, which is also a cycle counter, on x86.

5.1.2 Further Considerations

Higher Cache Levels For higher and inclusive cache levels, features such as *AutoLock* [18] need to be considered. *cachequery* already floods the corresponding set in the L1 cache between cache accesses to the L2 set. The addresses used for evicting the L1 set are chosen so that they map to the same L1 set but to different L2 sets. This works as shown in Figure 2.2 since the L2 cache in our experiments always had more sets than the L1 cache and thus more index bits are used to determine the L2 sets than to find the L1 set.

Reducing System Noise *cachequery* already disables scheduling and interrupts during the execution of a query as far as this is possible on Linux using `preempt_disable` and `raw_local_irq_save` [10]. We extended this to be disabled for all runs of a query in case it is configured to run multiple times consecutively. Especially for higher cache levels than the L1 caches, which are then unified, system noise becomes even more relevant since the measurement code itself may cause cache misses. Using the built-in support by *cachequery*, this is prevented by checking that the pages that are allocated for the measurement code do not use any of the cache sets that are to be measured. However, regarding performance counters such as the `L2D_CACHE_REFILL` event, unintended increments may still happen, although the probability for that is minimized as much as possible by having a very short distance between two reads of the performance counter. Additionally, it should be noted that the TRM of the Cortex-A76, for example, states for the `L2D_CACHE_REFILL` event that "L2 refills caused by stashes and prefetches that target this level of cache, should not be counted." [36]

5.1.3 Query Execution

Algorithm 1 shows the main function for the query execution provided by *cachequery*. It uses Algorithm 3 for every single query evaluation. The return value of Algorithm 1 indicates for each measurement point how many hits the corresponding cache access

has generated over the given number of iterations. For compatibility with the *nanoBench* frontend scripts, we adapted *cachequery* to optionally just return the total number of hits, not distinguishing by which measurement point(s) they have been generated. Algorithm 2 gives an overview of the adapted algorithm. Algorithm 4 is thereby used for evaluating the query for every single repetition.

The original *cachequery* implementation disabled scheduling and interrupts within the loop. We moved this out of the loop so that all iterations can run as uninterrupted as possible. Regarding noise reduction, we did not use any of the methods *cachequery* applied for Intel CPUs, such as disabling hyper-threading, other cores, or prefetching, but we made sure to pin the execution of the test code to a specific CPU which can be specified as a parameter when the kernel module is loaded using `insmod`.

```
Function run_query_list(query, reps):
    hit_list_total ← [0,0,...,0]        // |hit_list_total| = #measurement-points
    preempt_disable()                                  // disable scheduling
    raw_local_irq_save()                               // disable interrupts
    for rep in {1..reps} do
        hit_list ← eval_query_list(query)
        for i ← 0 to len(hit_list_total) − 1 do
            hit_list_total[i] ← hit_list_total[i] + hit_list[i]
    raw_local_irq_restore()                             // enable interrupts
    preempt_enable()                                    // enable scheduling
    return hit_list_total
```

Algorithm 1: Pseudocode for one query run returning a list which indicates for every measurement point in the query whether the corresponding access has been a hit or a miss (i.e., the length of the list is determined by the number of measurement points in the query).

```
Function run_query_sum(query, reps):
    hits_total ← 0
    preempt_disable()                                  // disable scheduling
    raw_local_irq_save()                               // disable interrupts
    for rep in {1..reps} do
        hits_total ← hits_total + eval_query_sum(query)
    raw_local_irq_restore()                             // enable interrupts
    preempt_enable()                                    // enable scheduling
    return hits_total
```

Algorithm 2: Pseudocode for one query run returning the total number of hits.

```
Function eval_query_list(query):
    hit_list ← []
    for elem in EVICTION_SET do
        LDR(elem)
        DC_CIVAC(elem)
    for elem in query do
        if elem is "wbinvd" instruction then
            for way in {1..ASSOCIATIVITY} do
                DC_CISW(TARGET_LEVEL, TARGET_SET, way)
        else
            if measure elem then
                start measurement
            LDR(elem)                          // actual access point of blocks
            if measure elem then
                end measurement
                if a hit has occurred then
                    hit_list.append(1)
                else
                    hit_list.append(0)
        if TARGET_LEVEL > 1 then
            evict corresponding set in levels {1..TARGET_LEVEL − 1}
    return hit_list
```

Algorithm 3: Pseudocode for one query evaluation returning a list which indicates for every measurement point in the query whether the corresponding access has been a hit or a miss.

```
Function eval_query_sum(query):
    hits ← 0
    for elem in EVICTION_SET do
        LDR(elem)
        DC_CIVAC(elem)
    for elem in query do
        if elem is "wbinvd" instruction then
            for way in {1..ASSOCIATIVITY} do
                DC_CISW(TARGET_LEVEL, TARGET_SET, way)
        else
            if measure elem then
                start measurement
            LDR(elem)                          // actual access point of blocks
            if measure elem then
                end measurement
                if a hit has occurred then hits ← hits + 1
        if TARGET_LEVEL > 1 then
            evict corresponding set in levels {1..TARGET_LEVEL − 1}
    return hits
```

Algorithm 4: Pseudocode for one query evaluation returning the number of hits.

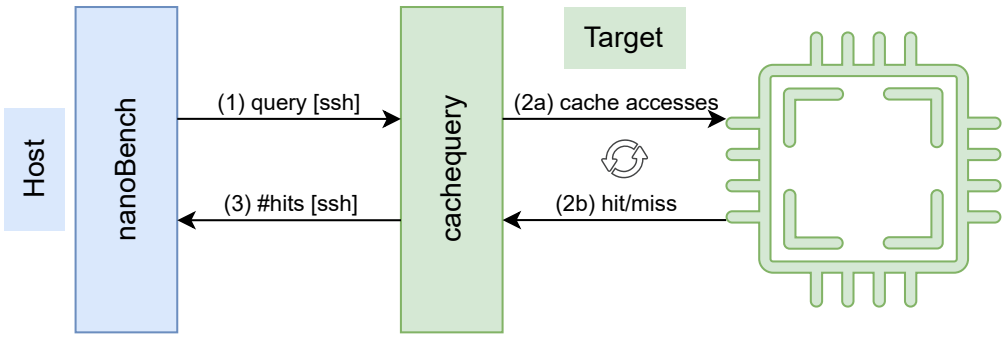


Figure 5.2: Schematic overview of how *cachequery* is connected with *nanoBench*.

5.2 Combining CacheQuery and nanoBench

nanoBench [9] has several frontend Python scripts providing, among other features, software-simulated replacement policies, randomized query generation, and comparing query results for hardware and software-simulated replacement policies. We used and adapted some of these scripts. Among the additions to the existing code, we created the possibility to open an SSH connection to the target board where *cachequery* is running. See Figure 5.2 for an overview of this system design.

The *nanoBench* scripts specifically enabled us to use or develop the following features:

- Query Generation. *nanoBench* is already able to generate random queries of a given length with randomly distributed measurement points. As a reminder, "measurement points" in MBL are cache accesses annotated with "?", which indicates that the backend should check whether this cache access resulted in a cache hit or miss. We enhanced the query generation with options to generate (1) "plain" queries without explicit measurement points, (2) queries using a maximum number of unique block identifiers, and (3) queries accessing a given number of blocks cyclically.

- Policy Simulation. *nanoBench* includes software simulations for around 300 replacement policies [26]. The large number partly is a result of different variants of the QLRU policy. The simulations implement the behavior of a single cache set. For every access, they return whether it resulted in a hit or miss. Based on the available interface, we implemented additional policies to compare with the behavior of the hardware.

- Policy Comparison. *nanoBench* already allows us to compare the number of hits certain policies generate for a given query. We used the existing framework and policy simulations and adapted the comparison methods as needed for our specific use cases.

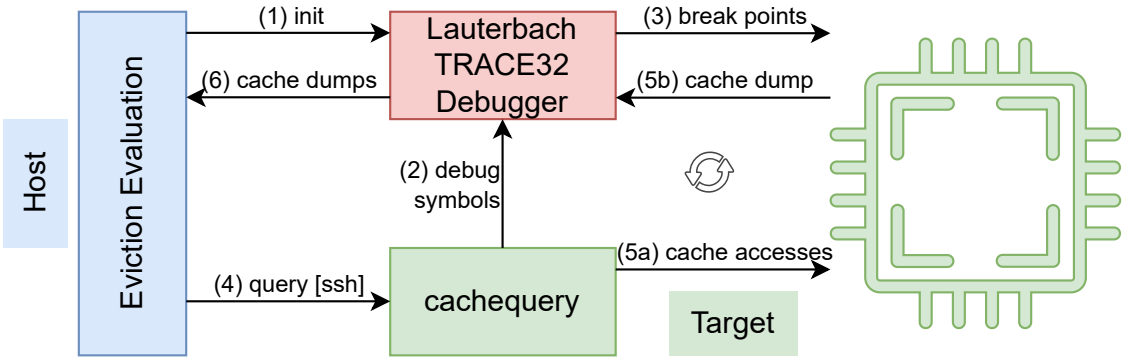


Figure 5.3: Schematic overview of the eviction analyzing script connected with *cachequery* and the Lauterbach debugger.

- Set Comparison. By iterating over cache sets, differences between their behavior might come to light, which can indicate dynamic policy implementations using set dueling, for example.

5.3 Combining CacheQuery (+ nanoBench) and Lauterbach TRACE32 for L1d cache analysis

Especially for pseudo-random cache replacement policies, we need to directly check which element has been evicted by a cache access, without the need of accessing other elements to indirectly infer this information. This becomes possible via the Lauterbach TRACE32 debugging capabilities which allow to create a dump of the entire L1d cache, for example. By creating cache dumps before and after cache accesses, we can thus determine the index of the eviction victim directly. Figure 5.3 shows an overview of the setup necessary to conduct suitable experiments. In the following, we go into more detail about how the components work and interact.

Algorithm 5 shows a pseudocode overview of how a query will be run using the setup described in this chapter. It uses the `eval_query_lauterbach` function depicted in Algorithm 6. As already specified, this setup is specially designed to retrieve more information about a pseudo-random replacement policy of the L1d cache. For this, we want to observe the cache set and evaluate which index of the set is evicted for every cache access in our query. It is crucial that this list of eviction candidate indices is generated without any “holes”. Within the observed time frame, there should be no victim selection that is not present in our list, if possible. The resulting sequence of eviction candidates may then allow us to detect recurring patterns or intervals, which ideally enables us to draw conclusions about the implementation of the employed pseudo-random number generator.

To generate the sequence of the indices of the set elements that have been evicted during the experiment, we developed a tool to analyze the CSV files which are written by

the Lauterbach debugger to dump the cache content. In our experiments, we found that the debugger always dumped the content of the target cache set in the same order. Thus it is possible to get the difference between two states of the target cache set and determine which element was evicted. The resulting sequence can thus simply contain the indices of each chosen victim. The indices are in the range from 0 to $ASSOCIATIVITY - 1$ (since every set can hold "associativity" many elements).

```
Function run_query_lauterbach(query, reps, slowdown_count):
    preempt_disable()                                   // disable scheduling
    raw_local_irq_save()                                // disable interrupts
    eval_query_lauterbach(query, reps, slowdown_count)
    raw_local_irq_restore()                             // enable interrupts
    preempt_enable()                                    // enable scheduling
```

Algorithm 5: Pseudocode for one L1d cache query run in connection with the Lauterbach debugger.

```
Function eval_query_lauterbach(query, reps, slowdown_count):
    for elem in EVICTION_SET do
        LDR(elem)
        DC_CIVAC(elem)
    for rep in {1..reps} do
        for elem in query do
            LDR(elem)                          // actual access point of blocks
            dump L1d cache
            for i in {1..slowdown_count} do
                NOP                                     // introduce a delay
```

Algorithm 6: Pseudocode for one L1d cache query evaluation with the Lauterbach debugger. Cache accesses marked as explicit measurement points are ignored (not shown in the pseudocode). Every query is assumed to consist of plain cache accesses.

In the following, we describe the concrete implementation of this approach on an even lower abstraction level. The eviction evaluation part runs on the host machine and starts the experiment by initializing a Lauterbach TRACE32 debugger with an appropriate startup script (written in Lauterbachs scripting language PRACTICE). On the target, we have *cachequery* available, which means that the Linux kernel module is loaded and the corresponding script is executable. Using the compiled Linux kernel module and the "Linux awareness" [71] of the Lauterbach debugger, the debugger can set breakpoints on given symbols or addresses. We use this to set a breakpoint into the *cachequery* kernel module after the measurement code has been generated and a pointer to the start of this code is available.

Action	Description
`b 8` `LABEL1: br x30`	skip the following instruction branch to link register, this is the "Lauterbach landing plane"

`b LENGTH`
move `-1` to `x0` and return
`LABEL2: nop... br x30`
	skip the error handling code and the `nop`s
	error handling code
	a configurable number of `nop`s and a branch to the link register

`...`	...

cache access (1)
`bl LABEL1`
 	
`bl LABEL2`
cache access (2)
`bl LABEL1`
 	
`bl LABEL2`
`...`
cache access (N)
`bl LABEL1`
 	
`bl LABEL2`
	first cache access within this query
	linked branch to the Lauterbach landing plane
	linked branch to the `nop`s
	second cache access within this query
	linked branch to the Lauterbach landing plane
	linked branch to the `nop`s
	...
	nth cache access within this query
	linked branch to the Lauterbach landing plane
	linked branch to the `nop`s

`...`	...

Table 5.2: High-level overview of the code the modified *cachequery* generates for the interaction with the Lauterbach debugger.

Table 5.2 shows an overview of the code our modified *cachequery* generates for the interaction with the Lauterbach debugger. After the pointer to this code is available, the debugger can set a breakpoint to the "Lauterbach landing plane" and thus synchronize with the query execution. It is then possible to dump the L1d cache after every cache access and thus determine which element has been evicted. The result is a list of indices of the eviction candidates. The "indices" refer to the index of the evicted element inside the dump of the cache set, which we found to have a deterministic ordering, at least within one debugging session. The queries we use for the Lauterbach-driven experiments do not need to include any explicit measurement points since every cache access is "measured" in the sense that the debugger captures the state of the cache before and after the access. Consequently, the queries can be plain access sequences such as "a b c a e d c b f ...". For the query generation and some of the access code, we again use some of the (adapted) *nanoBench* scripts.

6 Evaluation

In the following sections, we present our findings based on the methodology described in Chapter 5.

6.1 L1d on Cortex-A76 (Rock 5B Board)

The target in this section is the L1 data cache on core 5 of the Cortex-A76 of the Rock 5B board described in Chapter 4. ARM specifies the policy of this cache to be PLRU, which matches our findings.

6.1.1 Setup

The target board is the Rock 5B, which is connected via SSH to a host computer where the frontend script is running. This script generates MBL queries with measurement points. For every measurement point, *cachequery* determines whether this access has been a hit or a miss. The algorithm employed for this experiment corresponds to Algorithm 2/Algorithm 4. The result shows how many hits the measured replacement policy generated for any given query. When used with a multitude of different queries, this allows the *nanoBench* tool to determine which replacement policies are "observationally equivalent" (see the research of Abel et al. we described in Chapter 3). In order to determine the hit/miss status for this experiment, we configured *cachequery* to use the `L1D_CACHE_REFILL` performance counter, as discussed in Chapter 5.

We used a total of 174 queries for this experiment to measure the behavior of the replacement policies in a variety of situations. Some queries have a fixed number of elements to represent working sets of different sizes. Specifically, we used the following queries. Note that each cache access within these queries is marked as a measurement point so that *cachequery* will determine whether this access has been a cache hit or miss.

- 80 queries, each using a pool of up to 100 elements, with 151 cache accesses per query to randomly selected elements.
- 85 queries, each with 151 cache accesses to randomly selected elements. Each query uses a pool of up to N elements, where $N = 8$ for the first five queries, $N = 9$ for the next five queries, and N incrementing accordingly until $N = 24$ for the last five queries.
- 9 queries, each with a total of 150 cache accesses using a cyclic pattern. Each query uses N elements, where $N = 8$ for the first query, $N = 9$ for the second query, and

N incrementing accordingly until $N = 16$ for the last query. These queries have the form 0..N 0..N ... 0..N.

A "`!`" is prepended to every query to achieve an invalidation and clean of the target set. This is also reflected in the employed policy simulations. Additionally, *cachequery* floods the target set before every query evaluation, as shown in Algorithm 4. Within this experiment, the *reps* count for each query, as depicted in Algorithm 2, was 100. Furthermore, we executed each query run ten times. As a result, the sum of the hits per query ranged between 0 and 11900. For each query, the results were deterministic; only for one query, one of the ten results differed by one hit (3201 instead of 3200 hits). We suspect that this is due to measurement noise.

6.1.2 Result

We found the behavior of the L1d cache to match the PLRU policy simulation with "sequential-fill" (as discussed in Subsection 2.5.2) implemented by the cache simulator of *nanoBench*—with one modification: in case of a hit, the tree structure is *not* updated as long as there are empty slots in the set left. As a result, this policy is predictable, and it is straightforward to find suitable sets of addresses to use for eviction-based cache attacks.

6.1.3 Reversing using Polca

Polca [32] is also able to reverse an automata for this PLRU variant, see Figure 6.1. However, the synthesis of a human-readable explanation for the policy using the tools provided by Vila et al. [32] is not possible due to the limitations regarding tree-based PLRU variants, which we also mentioned in Chapter 3.

The automata in Figure 6.1 shows the PLRU properties. The transitions annotated with `h(x)` denote a hit on the element at index `x`, `m() / y` denotes a miss leading to the eviction of the element at index `y`. The associativity of the target cache is four. Hence, the indices of the elements range from 0 to 3, and the tree of a tree-based PLRU has three nodes, one parent node, and two children. The automata therefore has eight states (S0 to S7) representing the eight possible states of the tree. S0 of the automata can be interpreted as the state at which the target set is filled. Consequently, the element at index 3 is the most recently used one. In case of a hit on this element, the state does not change. In case a miss happens, the element at index 0 gets evicted and replaced by a new one, which is then the most recently used one, and the automata transitions to S1. The same transition is done in case the element at index 0 is accessed and thus becomes the most recently used one. The automata stays at S1 in case the current element at index 0 is accessed next. If, however, there is a cache miss, the element at index 2 gets replaced (a tree-based PLRU replaces index 2 after index 0, see the visualizations in Chapter 2) and thus becomes the most recently used one, which can also happen in case there is a hit on the existing element at index 2. The automata then would transition to S2. All the other states behave accordingly.

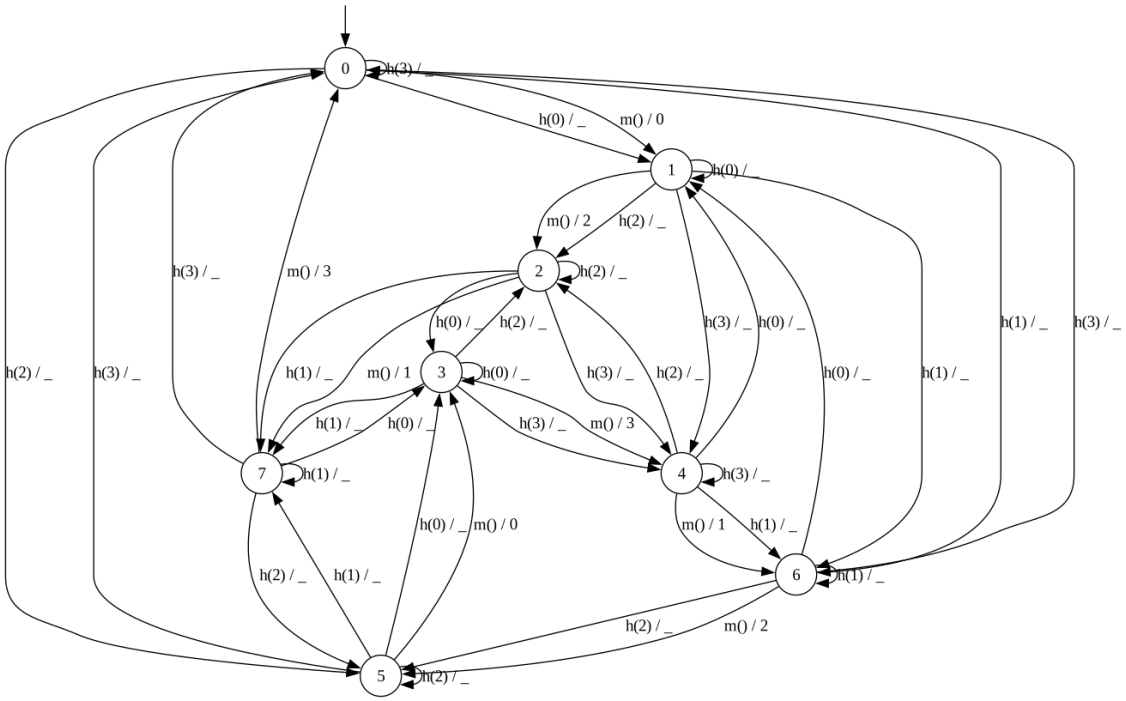


Figure 6.1: Learned automata of the PLRU implemented by the L1d of the A76 on the Rock 5B board by Polca [32].

6.2 L1d on Cortex-A55

For inspecting the L1 data cache of the Cortex-A55, we used two target boards, the RZBoard V2L and the Rock 5B, as shown in Chapter 4. We found no relevant official documentation about the replacement policy of the L1 cache of the Cortex-A55. However, the ARM Cortex-A53, which is the predecessor of the A55, and the Cortex-A510, which is the successor of the A55, both are reported to use pseudo-random replacement policies [72, 73]. In our initial manual tests, we also found that the policy seems to behave accordingly and non-deterministic. Consequently, we tried to generate a list of eviction candidates in order to check for patterns or periodically repeating subsequences.

6.2.1 Setup

We used the setup as described in Section 5.3. We used different numbers of `nop` instructions to slow down the code to experiment with different access rates that the replacement logic has to handle. Furthermore, we used different query constructs, such as random queries, random queries with a maximum of five unique elements (since the associativity is 4), or cyclic queries using five elements. Each experiment resulted in a list of indices representing the eviction candidate whenever an eviction happened.

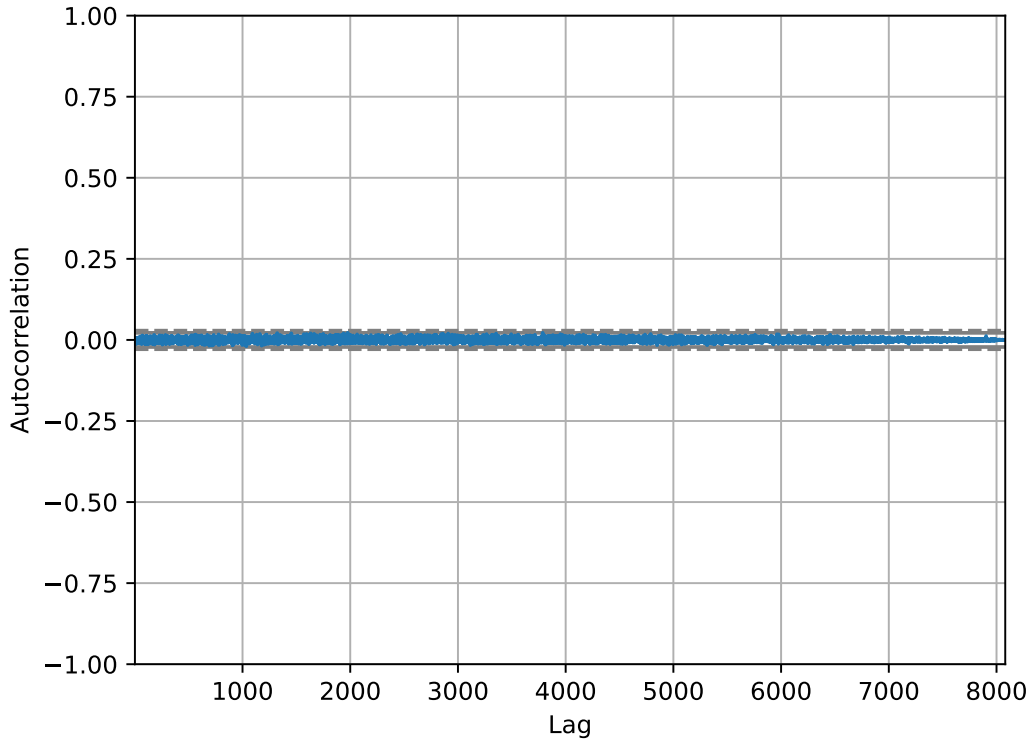


Figure 6.2: Autocorrelation of the list of eviction candidates generated by a random query on the RZBoard.

6.2.2 Result

We found no patterns, such as periodically repeating subsequences. Figure 6.2 shows the autocorrelation of the resulting list of eviction candidate indices using a random query of length 100 (using up to 100 randomly selected elements) being concatenated 100 times using the *reps* parameter of Algorithm 5, thus yielding a query of the length 10,000, which resulted in more than 8,000 evictions. The *slowdown_count* parameter of Algorithm 5 was 500. Figure 6.3 shows the result of the same setup with the difference that the original query with 100 elements consists of the five unique elements cyclically repeating. This resulted in more than 4,000 evictions—fewer evictions because of fewer unique elements.

6.2.3 Discussion

While the pseudo-random policy might be hard to reverse by itself, properties such as "previction" [74, 75] could additionally hinder reversing. Previction has been observed on the ARM Cortex-A53 (the predecessor of the A55). It leads to cache elements being

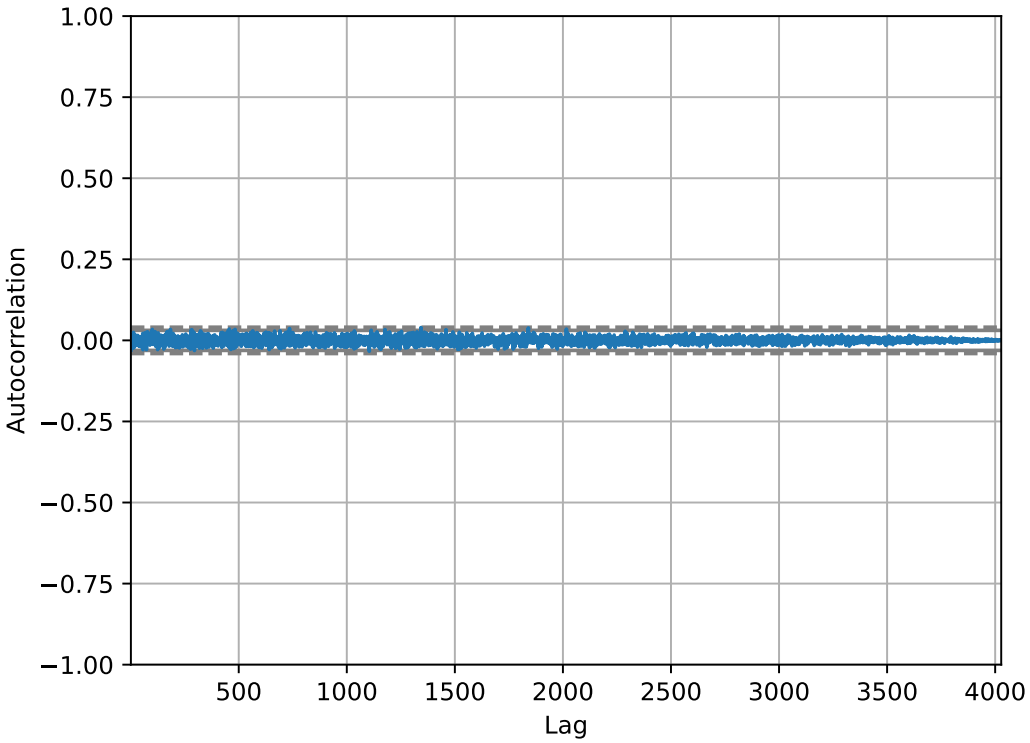


Figure 6.3: Autocorrelation of the list of eviction candidates generated by a query accessing five elements cyclically on the RZBoard.

evicted although the corresponding set is not full yet. Nemati et al. speculate: "Our hypothesis is that the processor detects a short sequence of loads to the same cache set and anticipates more loads to the same cache set with no reuse of previously loaded values. It evicts the valid cache line in order to make space for more colliding lines. We note that these cache entries are not dirty and thus eviction is most likely a cheap operation." [74]

Additionally, a pseudo-randomness source could be combined with other sources of pseudo-randomness, such as counters of hardware events. Consequently, there might not be an observable period. Also, the PRNG might be distributed over all cache sets, thus preventing a period from being revealed by observing only one set. Finally, the PRNG could have a period that is larger than our observation window.

This result might lead to the assumption that pseudo-random policies successfully make cache attacks such as PRIME+PROBE harder to deploy. However, while the behavior of the replacement policy is not as favorable for such an attack as, for instance, the PLRU policy, current research claims that it still is not optimal for security [76]. Nonetheless, cache randomization is an important building block for hardening against eviction-based cache attacks [77].

6.3 L2 on Cortex-A76 (Rock 5B Board)

In this section, we inspect the cache replacement policy of the L2 cache (core 5) of the A76 (RK3588 chip). We showed in Chapter 4 that there is not much information available about the cache replacement policy, except for the name "dynamic biased". In the following, we try to approach and analyze a possible explanation for this policy.

6.3.1 Research on Potentially Matching Policies

The ARM Cortex-A76 is the direct successor of the A75. In Chapter 4, we showed that the A75 already does "bias" cache lines based on whether they hold instructions or data. With this information, we found three patents that might be related: "Retention priority based cache replacement policy" [78], "Cache storage for multiple requesters and usage estimation thereof" [79], and "Storage controller" [80]. All those patents have "ARM Ltd" as the current assignee and describe new types of replacement policies. In the following, we will shed some light on those policies.

Retention priority based cache replacement policy [78]

This patent describes a cache replacement policy that maintains a "retention priority value" (called "PV") for every loaded cache line. Conceptually, the PV can use an arbitrary number of bits, but the authors mention two bits as a compromise and use this value for all examples within the patent. They explain that this value would balance the supported granularity and the needed storage resources to handle the PVs. Every cache line gets assigned a PV when it is loaded into the cache set. Two main variables

can characterize a specific implementation of this patent: the initial PV upon insertion and the update of a PV upon a cache hit. In terms of Abel [26], this would correspond to the "insertion age" and the "promotion policy" of QLRU. Note that the numerical value of the PV and its interpretation in the context of the replacement policy can be different. The patent suggests that a lower numerical value means a higher PV. A value of `00` for the two bits, for example, would then imply the highest PV level, a bit value of `11` the lowest. The highest numerical value and thus the lowest PV level would then make the corresponding cache entry a candidate for the next eviction.

Initial PV The initial PV can depend on multiple factors. One factor is whether the source of the memory access is the instruction fetch unit. This would indicate whether the cache line holds instructions or data and would allow, for instance, to prioritize instructions. This seems to match the biasing of the A75's L2 replacement policy as described in [62]. Additionally, the patent specifies more examples for different sources and thus different initial PVs. The policy might differentiate with regard to whether the requesting process has kernel privilege level, whether the request originates from a CPU versus a GPU, or from which CPU core in case there are multiple.

PV Updates The PV can be updated for multiple reasons; the patent even considers a global demotion of PVs throughout the cache as an example. Regarding the promotion of a PV upon a cache hit on the corresponding cache line, the patent mentions two options: an implementation can either increment the PV on every hit until it reaches its maximum value or it can directly set the PV to its maximum value on the first hit. Also on a cache miss, the PV of the elements inside the respective cache set might need to be updated in case one element needs to be evicted but there is none with the lowest PV. In that case, the PV of every element of the set can be demoted until at least one element has the lowest possible PV. Another option, as the patent points out, would be that the PV interpretation changes. This most probably means that the highest numerical PV value currently present in the set would be mapped to the lowest PV level.

Victim Selection In case there are empty slots left in the set, the patent suggests randomly selecting one. This is also mentioned as a possibility whenever an element needs to be evicted and there are multiple eligible eviction candidates, i.e., elements associated with the lowest PV. In this regard, however, the patent also allows using another policy such as LRU or round robin to select one of the candidates.

Cache storage for multiple requesters and usage estimation thereof [79]

This patent seems to enhance the replacement policy described in the patent above. The values associated with loaded cache lines are now enhanced to three bits (at least in the example in the patent) and are configurable. They may either represent a priority value (PV), similar to the first patent, or a "recent usage" (RU) value. The RU value

can be used to implement LRU like behavior; however, it can be dynamically adapted based on factors such as the number of insertions by a certain "requester" over a defined period of time. "Requester" is a term used by the patent to describe entities that access the cache. Requesters can be differentiated on several abstraction levels. The patent mentions CPU versus GPU, different virtual machines, or software processes running on a CPU as examples for requesters. Each requester is associated with an ID which is also added to every cache line to create a mapping between requesters and cache elements.

Bias The patent describes a process to bias the replacement/retention policy according to several factors, such as the type of the requester, the number of insertions caused by this requester over a defined period of time, and a target value for the cache usage. This allows, for example, to slow down a requester that causes a lot of cache insertions, thus achieving a more balanced or "fair" distribution of cache resources to all requesters. Or, on the contrary, the policy could favor such requesters and allow an "asymmetric" cache usage. The biasing can be implemented by leveraging different initial PV/RU values.

Storage controller [80]

While the previously discussed patent seems to be one abstraction level above the first, this patent represents yet another level higher in abstraction. The cache elements are now not only associated with some priority value, which is now called "weight value" (wt), but also with a "partition ID value" (P). The wt value is again used for victim selection, just as the PV/RU values above. However, there may now be multiple replacement policies, each associated with a partition ID. It can be configured, which of the available policies each partition uses. Additionally, the design includes one or more global replacement policies that can fine-tune the partition policies regarding several aspects. The patent mentions a global power-saving policy as an example that would try to tune the partition policies for energy efficiency. Some further enhancements described in the patent include the following aspects:

- The partition policies can include not only logic for the replacement policy but also further policies related to the management of the cache contents. The patent mentions a prefetcher policy as an example where the partition policy could decide to load additional data into the cache.

- The partition policies are programmable and thus allow, for instance, to switch the replacement policy from LRU to Least Frequently Used (LFU) or fine-tune the prefetcher policy.

- To avoid thrashing, a bypass control only allows a certain programmable percentage of addresses to overwrite existing cache entries.

Implementation of the Test Replacement Policy Simulation

In order to match the patents above at least partially, we considered the following implementation aspects for a corresponding policy simulation:

- A PV vector. It holds a PV for every element of the simulated cache set.
- A maximum numerical PV value. This can be set to 3 to represent two-bit PVs or to 7 to simulate three-bit PVs. (Keep in mind that the maximum numerical PV value represents the lowest PV and vice versa!)
- An initial PV. This is the numerical PV value used for every new cache line a cache set holds. In case it is not set, the simulation will choose a random value in the range from zero to the maximum value. For the experiments we present in this thesis, we used values from zero to two for the initial numerical PV since we found other values to match worse to the behavior of the hardware.
- Option: *hit_to_zero*. If set, the PV is not increased by one in case of a cache hit but directly set to the highest PV (i.e., the lowest numerical PV value).
- Option: *update_on_hit*. If not set, the PV is not updated at all in case of a cache hit.
- Option: *update_on_miss*. Usually, in case of a cache miss, the PVs of the existing elements are demoted (i.e., their numerical values increased) until there is one element with the lowest PV. If this option is not set, this update will not happen and the victim selection part will instead search for elements with the lowest PV that is currently present.
- Option: *fill_method*. In case the set is not yet full and there are multiple free slots, the value of this option determines how a free slot for an incoming new element is selected. If equal to zero, a slot will be chosen randomly. If equal to one, the set is scanned from the beginning and the first free slot will be taken. If equal to two, the scan will start from the previously remembered position plus one and will also choose the first free slot encountered.
- Option: *index_selection*. In case the set is full and one element needs to be evicted, this option configures how the victim is selected. In any case, the victim selection will search for an element with the lowest PV, which will depend on the value of the option *update_on_miss*. The elements fulfilling the requirement that their PV equals this lowest PV are the eviction candidates. If *index_selection* is equal to zero, one of the eviction candidates is selected randomly. If equal to one, the set is scanned from the beginning and the first element with a matching PV will be chosen. If equal to two, the scan will start at the previously remembered position plus one and will then choose the first element fulfilling the requirement. Note that incrementing the remembered position will wrap around.

6.3.2 Experiment 1: Search for Sets with Fixed Policies

Based on the insights from the patents and previous findings in Intel CPUs [23], the first experiment aims at checking whether there are sets dedicated to a fixed policy (and serve as reference sets) while other sets dynamically adapt their policies.

Setup

We used the combination of *cachequery* and the *nanoBench* frontend scripts as discussed in Chapter 5. Specifically, we employed the setup described by Algorithm 2 and Algorithm 4, where we get the total number of hits for each query run. We employed a query starting with a set clean and invalidate operator, followed by 150 cache accesses, each marked as a measurement point. We used ten cache blocks, which were accessed cyclically. This results in the following query: "`! 0? 1? ... 9? 0? 1? ... 9? ...`". This query has then been executed with a repetition count of 10 (the *reps* parameter of `run_query_sum` in Algorithm 2)—and this has been repeated ten times for every set. The order in which we analyzed the 1024 sets of the L2 cache using this experiment has been random, as suggested by [27].

Result

The experiment yielded ten hit sums for every cache set. For every set, we calculated the mean of the ten sums and verified whether every sum has an absolute difference to the mean value of less than ten. There was no set for which we could observe that behavior, which leads us to the conclusion that either no set implements a fixed policy, such as a specific fixed configuration of a PLRU variant, or there are fixed policies, but with some degree of non-deterministic behavior. The policy of each set might also depend on dynamically changing variable parameters.

6.3.3 Experiment 2: Comparison of Hit Sums with Simulated Policies

This experiment compares the number of hits generated by the hardware with software-simulated policies.

Setup

We used the combination of *cachequery* and the *nanoBench* frontend scripts as discussed in Chapter 5. Specifically, we employed the setup described by Algorithm 2 and Algorithm 4, where we get the total number of hits for each query run. For this experiment, we used the same queries as described in Subsection 6.1.1. These queries have been executed with a repetition count of 100 (the *reps* parameter of `run_query_sum` in Algorithm 2)—and this has been repeated ten times for every policy/query combination. We then calculated the mean value of total hits over those ten runs to compare the policies. The policy simulations we compared to the hardware results include all deterministic

policies implemented by *nanoBench*. In addition, we used our new policy simulation, which represents parts of the policy specification we saw in the patents described earlier in this section. We tested this policy using all combinations of its parameters (which we described above).

Example

To make the hit numbers in the following result section more straightforward to understand, we present a short example of the data collection in the following. Assume we have two queries, A and B. We run these queries on policies P and Q using Algorithm 2 with a repetition count of 100 and repeat this process 10 times. Assume this would yield the following example result:

Query	Policy P	Policy Q
A	{3200, 3200, 3200, 3200, 3200, 3200, 3200, 3200, 3200, 3200}	{2989, 3132, 3001, 3199, 2857, 3024, 2915, 3013, 3109, 3100}
B	{4500, 4500, 4500, 4500, 4500, 4500, 4500, 4500, 4500, 4500}	{4791, 4759, 4800, 4710, 4698, 4710, 4700, 4723, 4709, 4732}

For this experiment, we would now take the average of the 10 runs each, which would result in the following:

Query	Policy P	Policy Q
A	3200	3033.9
B	4500	4733.2

For Query A, the absolute difference between Policy P and Q would then be 166.1, for Query B 233.2. Thus, Policy Q would have an average absolute difference of 199.65. The relative difference between Policy Q and P would be -166.1 for Query A and 233.2 for Query B. This would result in an average relative difference of 33.55 ($= \frac{-166.1+233.2}{2}$), which we would then describe as a positive average relative difference.

Result

No policy matched exactly the average hit sums of the hardware for every query run we conducted. However, we found that our implemented test policy and variants of the QLRU policy corresponded best to the hardware on average. QLRU performing comparatively well might indicate that the different options for *insertion age* and *promotion policy*, as described by [26], are closely related to the ideas of the policy we assume to be implemented for this cache.

In the following, we present our results in two parts. First, we examine the absolute differences between the simulated policies and the hardware. They show how close

each policy (variant) matches the hardware behavior. Then, we present the relative differences between the simulations and the hardware. These values potentially hide large differences because positive and negative distances cancel each other out. On the other hand, the relative distances indicate whether a simulation caused on average more or fewer cache hits than the hardware.

Table 6.1 shows the rounded average absolute distances between the average hit number per query run of the hardware and some of the simulations (the best-performing variants in each case). The policy performing worst was a variant of our test policy with a rounded average absolute distance to the hardware of 1245.1 (1053.5 for the worst-performing QLRU variant).

Regarding the average absolute distances, the two best-performing variants of our test policy feature three-bit PVs, an initial PV of zero or one, *update_on_miss*, *update_on_hit*, a *fill_method* value of zero, and an *index_selection* value of two. The third-best-performing variant features the same configuration, with two-bit PVs and an initial PV of zero. The worst-performing variants—there are 16 different variants performing equally—do not depend on whether the PVs use two or three bits, the initial PV, or whether *hit_to_zero* is set or not. They feature a *fill_method* and an *index_selection* value of one and in the case of an initial PV of zero optionally have *update_on_hit* specified. Table 6.2 shows the average absolute distances of the five best-performing variants of our test policy and their corresponding relative distances. Table 6.3 shows the same for the five worst-performing variants of our test policy.

	Our Test Policy	QLRU	PLRU	LRU	FIFO
Distance	191.5	224.0	266.8	310.5	325.8

Table 6.1: Comparison of rounded average absolute distances of several policy simulations to the hardware results.

Best Avg Absolute Distances	191.5	201.0	211.7	215.8	216.7
Corresponding Avg Relative Distances	6.4	64.1	56.6	26.3	100.7

Table 6.2: The average relative distances for the five best-performing variants of our test policy according to the average absolute distances.

Worst Avg Absolute Distances	1245.1	1245.1	1245.1	1245.1	1245.1
Corresponding Avg Relative Distances	−92.8	−92.8	−92.8	−92.8	−92.8

Table 6.3: The average relative distances for the five worst-performing variants of our test policy according to the average absolute distances. (There are 16 variants performing equally.)

Table 6.4 lists the rounded average relative distances between the average hit number per query run of the hardware and some of the simulations (for the variants coming closest to zero in each case). The policy with the highest positive average distance to the hardware was a variant of our test policy with 1056.7 (1013.6 for the worst-performing QLRU variant). This means that those policies resulted on average in one thousand hits per query run more than the hardware. The policies with the highest negative average distance to the hardware were variants of our test policy with -236.4, which equals the value for FIFO. In other words, those policies had on average more than two hundred cache misses per query run more than the hardware.

Regarding the average relative distances, the two best-performing variants of our test policy feature three-bit PVs, an initial PV of zero or two, *update_on_miss*, *update_on_hit*, a *fill_method* value of one (for initial PV zero) or two (for initial PV two), and an *index_selection* value of two. The third-best-performing variant features two-bit PVs, an initial PV of zero, *update_on_miss*, *update_on_hit*, a *fill_method* value of two, and an *index_selection* value of one. The three worst-performing variants feature two- or three-bit PVs with an initial PV of two, *update_on_hit*, a *fill_method* value of one (or optionally two for the three-bit PV variants), and an *index_selection* value of zero. Table 6.5 shows the average relative distances of the five best-performing variants of our test policy and their corresponding absolute distances. Table 6.6 shows the same for the five worst-performing variants of our test policy.

	Our Test Policy	QLRU	PLRU	LRU	FIFO
Distance	1.2	-7.1	33.1	96.9	-236.4

Table 6.4: Comparison of rounded average relative distances of several policy simulations to the hardware results.

Best Avg Relative Distances	1.2	-1.4	3.2	6.4	-6.5
Corresponding Avg Absolute Distances	271.0	285.5	233.9	191.5	259.0

Table 6.5: The average absolute distances for the five best-performing variants of our test policy according to the average relative distances.

Worst Avg Relative Distances	1056.7	1056.1	1055.7	1055.4	1055.3
Corresponding Avg Absolute Distances	1075.8	1074.9	1074.9	1074.3	1074.8

Table 6.6: The average absolute distances for the five worst-performing variants of our test policy according to the average relative distances.

6.3.4 Experiment 3: Comparisons of Single Hits/Misses with Simulated Policies

This experiment takes a closer look at the behavior of the policies in the setup described for Subsection 6.3.3.

Setup

For this experiment, we use the setup of Subsection 6.3.3, but connect Algorithm 1 with Algorithm 3 instead of Algorithm 2 to Algorithm 4. Together with a repetition count of one (the *reps* parameter of `run_query_list` in Algorithm 1), this allows us to retrieve a list of one/zero values indicating single hit/misses for every measurement point in the queries. To be able to compare the results better, we again generated this result ten times for every policy/query pair.

Example

To make the hit numbers in the following result section easier to understand, we show a short example of the data collection in the following. Assume we have two queries, A and B. We run these queries on policies P and Q using Algorithm 1 with a repetition count of one and repeat this process 10 times. Assume this would yield the example result shown in Table 6.7, where each 1 indicates a hit and each 0 a miss (the mapping of hit/miss to a numerical value does not matter for calculating distance values).

Policy	Query A	Query B
P	01011, 01011, 01011, 01011, ..., 01011	00110, 00110, 00110, 00110, ..., 00110
Q	01010, 01010, 01010, 01010, ..., 01010	01001, 01001, 01001, 01001, ..., 01001

Table 6.7: Example results for queries A and B and policies P and Q.

We now count the number of places where the two policies do not have the same hit/miss behavior (i.e., we calculate the hamming distance of the binary strings), which would result in a distance of 10 between Policy P and Q for Query A (since each of the ten results differs in one place) and a distance of 40 for Query B (since each of the ten results differs in four places). Thus, the average distance between the two policies would be 25.

Result

Our test policy and the QLRU variants again matched the behavior of the hardware in this experiment quite well. In contrast to the previous experiment, LRU and the implemented PLRU variants were on average a little closer to the behavior of the hardware than our test policy or the QLRU variants. The policy with the largest distance

from the hardware had an average distance value of about 127 (smaller is better). The best-performing QLRU variant had an average distance of 34.4. The best-matching variant of our test policy scored 33.5. LRU had a score of 30.8. Further research and experiments on why the LRU and PLRU variants performed best are necessary.

6.3.5 Discussion

None of our experiments revealed an exact match between the analyzed hardware and simulations of known policies. However, QLRU variants, as well as our test policies, exhibit a close behavior. The “dynamic biased” policy seems to be significantly more complex than a common PLRU policy. This might not only lead to better performance or energy efficiency but can also make it more challenging for attackers to leverage knowledge about this policy for efficient eviction-based cache side-channel attacks. Further research might show how to craft suitable sets of addresses for targeted data eviction or how the policy might be extended to prevent that.

7 Summary

In this work, we ported the analysis framework CacheQuery from x86 to ARM and combined it with the nanoBench analysis framework and the Lauterbach TRACE32 debugger. This enabled us to develop and use three approaches for analyzing cache replacement policies on ARM CPUs: we compared hardware behavior with simulations, we used an automata learning framework, and we inspected the cache contents directly using the hardware debugger. We then applied our novel setup and the analysis approaches to two target boards with two different CPUs, the ARM Cortex-A55 and -A76. As a result, we could fully reverse-engineer the PLRU L1d replacement policy of the A76. For the presumably pseudo-random replacement policy employed by the L1d of the A55, we could not find any patterns or mechanisms that would allow prediction in our experiments. To get better insights into the behavior of this policy, it is vital to shed more light on the interplay between generated randomness and its usage in the replacement policy. The L2 policy of the A76 is undocumented and currently remains unknown. We found related patents, which we used to derive a policy approximation that is close to the behavior of the hardware in our experiments. This is a foundation for further research to uncover the L2 policy of the A76.

In conclusion, policies such as tree-based PLRU implementations are straightforward to reverse-engineer on ARM CPUs. This increases the risk of more efficient attacks. Pseudo-random policies or more complex modern policies require significantly more effort to understand and to predict. This raises the bar for attacks but ultimately does not prevent them. Cache eviction still succeeds for these policies but requires more memory accesses and time. Also, some concepts of new and complex policies might introduce new and yet unknown risks, e.g., enabling new side-channels. In order to properly assess the risk arising from microarchitectural attacks and the vulnerability of computing systems, these policies must be thoroughly understood.

Abbreviations

BIP Bimodal Insertion Policy [22]

DIP Dynamic Insertion Policy [22]

DRRIP Dynamic RRIP [8]

EL Exception Level

FIFO First-in-First-out

LIP LRU Insertion Policy [22]

LLC Last Level Cache

LFU Least Frequently Used

LRU Least Recently Used

MBL MemBlockLang [32]

MESI Modified Exclusive Shared Invalid

MRU Most Recently Used [28, 29]

NRU Not Recently Used [30]

PAPI Performance Application Programming Interface [42]

PIPT Physically Indexed, Physically Tagged

PLRU Pseudo-LRU

PMU Performance Monitoring Unit

PoC Point of Coherency

PoU Point of Unification

PRNG Pseudorandom Number Generator

PV Priority Value

QLRU Quad-Age LRU [25]

RISC Reduced Instruction Set Computer

RRIP Re-Reference Interval Prediction [8]

SRRIP Static RRIP [8]

TRM Technical Reference Manual

VIPT Virtually Indexed, Physically Tagged

List of Figures

List of Tables

Bibliography

[1] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, N. Koblitz, Ed., ser. Lecture Notes in Computer Science, vol. 1109, Springer, 1996, pp. 104–113, ISBN: 3-540-61512-1. DOI: 10.1007/3-540-68697-5_9. [Online]. Available: https://doi.org/10.1007/3-540-68697-5%5C_9.

[2] J. Kelsey, B. Schneier, D. A. Wagner, and C. Hall, "Side channel cryptanalysis of product ciphers," *J. Comput. Secur.*, vol. 8, no. 2/3, pp. 141–158, 2000. DOI: 10.3233/JCS-2000-82-304. [Online]. Available: https://doi.org/10.3233/jcs-2000-82-304.

[3] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[4] 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 *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[5] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache attacks on mobile devices," in *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX: USENIX Association, Aug. 2016, pp. 549–564, ISBN: 978-1-931971-32-4. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp.

[6] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures," *J. Cryptol.*, vol. 23, no. 1, pp. 37–71, 2010. DOI: 10.1007/s00145-009-9049-y. [Online]. Available: https://doi.org/10.1007/s00145-009-9049-y.

[7] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, J. Jung and T. Holz, Eds., USENIX Association, 2015, pp. 897–912. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss.

[8] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 60–71, Jun. 2010, ISSN: 0163-5964. DOI: 10.1145/1816038.1815971. [Online]. Available: https://doi.org/10.1145/1816038.1815971.

[9] A. Abel and J. Reineke, "Nanobench: A low-overhead tool for running microbenchmarks on x86 systems," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, Aug. 2020. DOI: 10.1109/ispass48437.2020.00014. [Online]. Available: http://dx.doi.org/10.1109/ISPASS48437.2020.00014.

[10] P. Vila, *Cgvwzq/cachequery: V0.1*, version v0.1, Apr. 2020. DOI: 10.5281/zenodo.3759108. [Online]. Available: https://doi.org/10.5281/zenodo.3759108.

[11] J. Backus, "Can programming be liberated from the von neumann style? a functional style and its algebra of programs," *Commun. ACM*, vol. 21, no. 8, pp. 613–641, Aug. 1978, ISSN: 0001-0782. DOI: 10.1145/359576.359579. [Online]. Available: https://doi.org/10.1145/359576.359579.

[12] Arm Limited. "Cortex-A76," [Online]. Available: https://developer.arm.com/Processors/Cortex-A76 (visited on 07/25/2024).

[13] ARM, *Programmer's Guide for ARMv8-A*, version 1.0, 2015.

[14] M. Humrick. "Cortex-A55 Microarchitecture - Exploring DynamIQ and ARM's New CPUs: Cortex-A75, Cortex-A55." (2017), [Online]. Available: https://www.anandtech.com/show/11441/dynamiq-and-arms-new-cpus-cortex-a75-a55/4 (visited on 05/27/2024).

[15] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017, ISBN: 0128119055.

[16] Arm Limited, *Learn the architecture - AArch64 memory attributes and properties*, version 2.0. [Online]. Available: https://developer.arm.com/documentation/102376/0200 (visited on 10/14/2024).

[17] Arm Limited, *Arm Architecture Reference Manual for A-profile architecture*, version K.a, 2024. [Online]. Available: https://developer.arm.com/documentation/ddi0487/latest/.

[18] 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 *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC: USENIX Association, Aug. 2017, pp. 1075–1091, ISBN: 978-1-931971-40-9. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/green.

[19] Arm Limited, *Learn the architecture - Introducing the Arm architecture*, version 2.1, 2024. [Online]. Available: https://developer.arm.com/documentation/102404/0201 (visited on 10/22/2024).

[20] Arm Limited, *Arm® Compiler armasm User Guide*, version 6.6.5, 2023. [Online]. Available: https://developer.arm.com/documentation/dui0801/l (visited on 10/22/2024).

[21] J. P. Thoma, C. Niesler, D. Funke, G. Leander, P. Mayr, N. Pohl, L. Davi, and T. Güneysu, "ClepsydraCache – preventing cache attacks with Time-Based evictions," in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, Aug. 2023, pp. 1991–2008, ISBN: 978-1-939133-37-3. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/thoma.

[22] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 381–391, Jun. 2007, ISSN: 0163-5964. DOI: 10.1145/1273440.1250709. [Online]. Available: https://doi.org/10.1145/1273440.1250709.

[23] H. Wong, *Intel Ivy Bridge cache replacement policy*, Jan. 2013. [Online]. Available: http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/.

[24] D. Grund and J. Reineke, "Toward precise plru cache analysis," in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2010. [Online]. Available: https://drops.dagstuhl.de/storage/01oasics/oasics-vol015-wcet2010/OASIcs.WCET.2010.23/OASIcs.WCET.2010.23.pdf.

[25] S. Jahagirdar, V. George, I. Sodhi, and R. Wells, "Power management of the third generation intel core micro architecture formerly codenamed ivy bridge," in *2012 IEEE Hot Chips 24 Symposium (HCS)*, 2012, pp. 1–49. DOI: 10.1109/HOTCHIPS.2012.7476478.

[26] A. Abel, *Automatic generation of models of microarchitectures*, 2020. DOI: http://dx.doi.org/10.22028/D291-31299.

[27] S. Briongos, P. Malagon, J. M. Moya, and T. Eisenbarth, "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, ISBN: 978-1-939133-17-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/briongos.

[28] A. Malamy, R. N. Patel, and N. M. Hayes, "Methods and apparatus for implementing a pseudo-lru cache memory replacement scheme with a locking feature," US5353425A, 1992. [Online]. Available: https://patents.google.com/patent/US5353425A/en.

[29] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic, "Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite," in *Proceedings of the 42nd Annual ACM Southeast Conference*, ser. ACMSE '04, Huntsville, Alabama: Association for Computing Machinery, 2004, pp. 267–272, ISBN: 1581138709. DOI: 10.1145/986537.986601. [Online]. Available: https://doi.org/10.1145/986537.986601.

[30] Sun Microsystems, Inc., *UltraSPARC T2 Supplement to the UltraSPARC Architecture 2007*, version Draft D1.4.3, 2007. [Online]. Available: https://www.oracle.com/technetwork/systems/opensparc/t2-14-ust2-uasuppl-draft-hp-ext-1537761.html.

[31] A. Abel and J. Reineke, "Measurement-based modeling of the cache replacement policy," in *RTAS*, Apr. 2013. [Online]. Available: http://embedded.cs.uni-saarland.de/publications/CacheModelingRTAS2013.pdf.

[32] P. Vila, P. Ganty, M. Guarnieri, and B. Köpf, "Cachequery: Learning replacement policies from hardware caches," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, London, UK: Association for Computing Machinery, 2020, pp. 519–532, ISBN: 9781450376136. DOI: 10.1145/3385412.3386008. [Online]. Available: https://doi.org/10.1145/3385412.3386008.

[33] S. Deng, N. Matyunin, W. Xiong, S. Katzenbeisser, and J. Szefer, *Evaluation of cache attacks on arm processors and secure caches*, 2021. arXiv: 2106.14054 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2106.14054.

[34] H. Cho, J. Park, D. Kim, Z. Zhao, Y. Shoshitaishvili, A. Doupé, and G.-J. Ahn, "Smokebomb: Effective mitigation against cache side-channel attacks on the arm architecture," in *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '20, Toronto, Ontario, Canada: Association for Computing Machinery, 2020, pp. 107–120, ISBN: 9781450379540. DOI: 10.1145/3386901.3388888. [Online]. Available: https://doi.org/10.1145/3386901.3388888.

[35] M. Lipp, "Cache attacks and rowhammer on arm," Master's Thesis, Graz University of Technology, Oct. 2016. [Online]. Available: https://repository.tugraz.at/publications/5qb3w-f1185.

[36] Arm Limited, *Arm Cortex-A76 Core Technical Reference Manual*, r4p1, version 100798_0401_01_en, 2023. [Online]. Available: https://developer.arm.com/documentation/100798/0401.

[37] Arm Limited, *Arm® Neoverse™ CMN-700 Coherent Mesh Network*, version r3p2, 2023. [Online]. Available: https://developer.arm.com/documentation/102308/0302.

[38] Real-Time and Embedded Systems Lab. "Chi - a measurement-based cache hierarchy inference tool," [Online]. Available: http://embedded.cs.uni-saarland.de/chi.php (visited on 05/22/2024).

[39] A. Abel and J. Reineke, "Automatic cache modeling by measurements," in *6th Junior Researcher Workshop on Real-Time Computing (in conjunction with RTNS)*, Nov. 2012. [Online]. Available: http://embedded.cs.uni-saarland.de/publications/CacheModelingJRWRTC.pdf.

[40] A. Abel, "Measurement-based inference of the cache hierarchy," M.S. thesis, Saarland University, 2012. [Online]. Available: http://embedded.cs.uni-saarland.de/literature/AndreasAbelMastersThesis.pdf.

[41] A. Abel and J. Reineke, "Reverse engineering of cache replacement policies in intel microprocessors and their evaluation," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2014, pp. 141–142. [Online]. Available: http://embedded.cs.uni-saarland.de/publications/ISPASS14.pdf.

[42] Innovative Computing Laboratory, University of Tennessee. "PAPI," [Online]. Available: https://icl.utk.edu/papi/ (visited on 05/22/2024).

[43] G. Rueda Cebollero, "Learning cache replacement policies using register automata," M.S. thesis, Uppsala University, Disciplinary Domain of Science, Technology, Mathematics, and Computer Science, Department of Information Technology, 2013. [Online]. Available: https://uu.diva-portal.org/smash/record.jsf?pid=diva2%3A678847&dswid=6033.

[44] B. Steffen, F. Howar, and M. Merten, "Introduction to active automata learning from a practical perspective," in *Formal Methods for Eternal Networked Software Systems: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*, M. Bernardo and V. Issarny, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 256–296, ISBN: 978-3-642-21455-4. DOI: 10.1007/978-3-642-21455-4_8. [Online]. Available: https://doi.org/10.1007/978-3-642-21455-4_8.

[45] P. Vila, P. Ganty, M. Guarnieri, and B. Köpf, *Cachequery: Learning replacement policies from hardware caches*, 2020. arXiv: 1912.09770 [cs.PL]. [Online]. Available: https://arxiv.org/abs/1912.09770.

[46] A. Solar-Lezama, "The sketching approach to program synthesis," in *Programming Languages and Systems*, Z. Hu, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 4–13, ISBN: 978-3-642-10672-9.

[47] Radxa. "ROCK 5B/5B+ | Radxa Docs," [Online]. Available: https://docs.radxa.com/en/rock5/rock5b (visited on 11/13/2024).

[48] Rockchip Electronics CO., LTD. "RK3588 Brief Datasheet," [Online]. Available: https://www.rock-chips.com/uploads/pdf/2022.8.26/192/RK3588%20Brief%20Datasheet.pdf (visited on 11/07/2024).

[49] Avnet, Inc. “RZBoard V2L | Avnet Boards,” [Online]. Available: https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/rzboard-v2l/ (visited on 11/07/2024).

[50] Renesas Electronics Corporation. “RZ/V2L - General-Purpose Microprocessor Equipped With Renesas’ Original AI Accelerator "DRP-AI", 1.2GHz Dual-Core Arm Cortex-A55 CPU, 3D Graphics, and Video Codec Engine | Renesas,” [Online]. Available: https://www.renesas.com/en/products/microcontrollers-microprocessors/rz-mpus/rzv2l-general-purpose-microprocessor-equipped-renesas-original-ai-accelerator-drp-ai-12ghz-dual-core-arm%5C#design%5C_development (visited on 11/13/2024).

[51] Radxa. “radxa/kernel: BSP kernel source,” [Online]. Available: https://github.com/radxa/kernel.git (visited on 11/13/2024).

[52] Avnet. “Avnet/renesas-linux-cip: Based on https://github.com/renesas-rz/rz_-linuxcip.git branch rzv2l-cip41,” [Online]. Available: https://github.com/Avnet/renesas-linux-cip.git (visited on 11/13/2024).

[53] Rockchip Electronics Co., Ltd. “kernel/arch/arm64/boot/dts/rockchip/rk3588s.-dtsi at 52f51a2b5ba178f331af62260d2da86d7472c14b · radxa/kernel,” [Online]. Available: https://github.com/radxa/kernel/blob/52f51a2b5ba178f331af62260d2da86d7472c14b/arch/arm64/boot/dts/rockchip/rk3588s.dtsi (visited on 11/07/2024).

[54] Renesas Electronics Corp. “renesas-linux-cip/arch/arm64/boot/dts/renesas/-r9a07g054.dtsi at c197622df526c82ae9e3674e06b4092dac33eafa · Avnet/renesas-linux-cip,” [Online]. Available: https://github.com/Avnet/renesas-linux-cip/blob/c197622df526c82ae9e3674e06b4092dac33eafa/arch/arm64/boot/dts/renesas/r9a07g054.dtsi (visited on 11/07/2024).

[55] Arm Limited, *Arm Cortex-A55 Software Optimization Guide*, 4.0, version r2p0, 2022. [Online]. Available: https://developer.arm.com/documentation/EPM128372/latest/.

[56] Arm Limited, *Arm Cortex-A55 Core Technical Reference Manual*, r2p0, version 100442_-0200_02_en, 2023. [Online]. Available: https://developer.arm.com/documentation/100442/0200.

[57] Z. Mahmood. “Samsung’s 2018 flagship Galaxy S9 has arrived.” (2018), [Online]. Available: https://www.globalvillagespace.com/samsungs-2018-flagship-galaxy-s9-has-arrived/ (visited on 05/28/2024).

[58] Samsung. “Exynos 990 | Mobile Processor | Samsung Semiconductor Global,” [Online]. Available: https://semiconductor.samsung.com/processor/mobile-processor/exynos-990/ (visited on 05/28/2024).

[59] Xiaomi. “Redmi Unveils Redmi Note 12 Series in Mainland China.” (2022), [Online]. Available: https://www.mi.com/global/discover/article?id=2802 (visited on 05/28/2024).

[60] Raspberry Pi. "Raspberry Pi 5," [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-5/ (visited on 05/28/2024).

[61] WikiChip. "Cortex-A76 - Microarchitectures - ARM - WikiChip," [Online]. Available: https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a76 (visited on 11/18/2024).

[62] Matt Humrick. "Cortex-A75 Microarchitecture - Exploring DynamIQ and ARM's New CPUs: Cortex-A75, Cortex-A55." (2017), [Online]. Available: https://www.anandtech.com/show/11441/dynamiq-and-arms-new-cpus-cortex-a75-a55/3 (visited on 11/18/2024).

[63] Arm Limited, *Arm® Neoverse™ CMN-650 Coherent Mesh Network Technical Reference Manual*, version r2p0, 2021. [Online]. Available: https://developer.arm.com/documentation/101481/0200.

[64] Arm Limited, *Arm® CoreLink™ CMN-600AE Coherent Mesh Network Technical Reference Manual*, version r1p1, 2023. [Online]. Available: https://developer.arm.com/documentation/101408/0101.

[65] Arm Limited. "big.LITTLE: Balancing Power Efficiency and Performance – Arm®," [Online]. Available: https://www.arm.com/technologies/big-little (visited on 11/11/2024).

[66] Z. Kou, S. Sinha, W. He, and W. Zhang, "Attack directories on arm big.little processors," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '22, San Diego, California: Association for Computing Machinery, 2022, ISBN: 9781450392174. DOI: 10.1145/3508352.3549340. [Online]. Available: https://doi.org/10.1145/3508352.3549340.

[67] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, version 325462-085US, 2024. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/671200.

[68] Arm Limited, "Straight-line Speculation," Tech. Rep., version 1.0, Jun. 2020. [Online]. Available: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Security%20Update%2008%20June%202020/Straight-line_Speculation-v1.0.pdf.

[69] Arm Limited, "Cache Speculation Side-channels," Tech. Rep., version 2.5, Jun. 2020. [Online]. Available: https://documentation-service.arm.com/static/61f904cafa8173727a1b7286?token=.

[70] J. Guo. "Profile firmware with Performance Monitor Unit (PMU) in Armv8-A CPU." (Nov. 8, 2023), [Online]. Available: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/profile-firmware-with-performance-monitor-unit-in-armv8-a-cpu.

[71] Lauterbach, *Os awareness manual linux*, version 02.2024, 2024. [Online]. Available: https://www2.lauterbach.com/pdf/rtos_linux_stop.pdf.

[72] Chester Lam. "ARM's Cortex A53: Tiny But Important." (2023), [Online]. Available: https://chipsandcheese.com/p/arms-cortex-a53-tiny-but-important (visited on 11/25/2024).

[73] Chester Lam. "Arm's Cortex A510: Two Kids in a Trench Coat." (2023), [Online]. Available: https://chipsandcheese.com/p/arms-cortex-a510-two-kids-in-a-trench-coat (visited on 11/25/2024).

[74] H. Nemati, P. Buiras, A. Lindner, R. Guanciale, and S. Jacobs, *Validation of abstract side-channel models for computer architectures*, 2020. arXiv: 2005.05254 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2005.05254.

[75] A. Ibrahim, H. Nemati, T. Schlüter, N. O. Tippenhauer, and C. Rossow, "Microarchitectural leakage templates and their application to cache-based side channels," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS 22, ACM, Nov. 2022, pp. 1489–1503. DOI: 10.1145/3548606.3560613. [Online]. Available: http://dx.doi.org/10.1145/3548606.3560613.

[76] M. Peters, N. Gaudin, J. P. Thoma, V. Lapôtre, P. Cotret, G. Gogniat, and T. Güneysu, "On the effect of replacement policies on the security of randomized cache architectures," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '24, Singapore, Singapore: Association for Computing Machinery, 2024, pp. 483–497, ISBN: 9798400704826. DOI: 10.1145/3634737.3637677. [Online]. Available: https://doi.org/10.1145/3634737.3637677.

[77] W. Song, B. Li, Z. Xue, Z. Li, W. Wang, and P. Liu, "Randomized last-level caches are still vulnerable to cache side-channel attacks! but we can fix it," in *2021 IEEE Symposium on Security and Privacy (SP)*, IEEE, May 2021, pp. 955–969. DOI: 10.1109/sp40001.2021.00050. [Online]. Available: http://dx.doi.org/10.1109/SP40001.2021.00050.

[78] "Retention priority based cache replacement policy," US9372811B2, 2012.

[79] "Cache storage for multiple requesters and usage estimation thereof," US11030101-B2, 2016.

[80] "Storage controller," US10185667B2, 2017.