Graz University of Technology
IAIK
Institute for
Applied Information Processing and Communications
Inffeldgasse 16a
A-8010 Graz

Bachelor Thesis

MODELING AND EVALUATING CPU CACHES IN SOFTWARE

Mario Theuermann(01430751)
mario.theuermann@student.tugraz.at

18th April 2018
http://www.iaik.tugraz.at/content/teaching/

Abstract

In microprocessor cache architecture design, there is an increasing demand for techniques to model miscellaneous different cache concepts, that enable to investigate the threats of cache-based side-channel attacks. A cache simulation offers many benefits in contrast to hardware testing when studying cache behaviour of modern multi-core microprocessor architectures. Accurate runtime simulations can greatly assist researchers to further explore access-driven cache attacks. Therefore a cache simulation enabling a full system emulation of the cache behaviour of x86 virtualization hosts can be advantageous. We present the current status of a runtime cache simulation using the well-known instruction set simulator QEMU, that can serve for research purposes on access-driven cache-attacks. Therefore, QEMUs instruction execution process was modified to run configurable cache models during runtime and added latency to simulate cache behaviour. We managed to raise the resolution of our results from the first naive approach using memory device operations considerably, by instrumenting the internal address translation process. Additionally, this implementation model is extended to act as a trace-driven cache simulation, modelling and evaluating many different cache characteristics and scenarios. Spatial and temporal locality is explained based on several test runs that illustrate the motive behind the conceptional designs of modern cache architectures. Therefore, different cache replacement algorithms are compared to each other as well as different mappings, cache capacities and block sizes. We show that set-associative caches have advantages over direct mapped caches and the importance of replacement algorithms. Furthermore, the tremendous impact of the cache miss-rate on the overall system performance is visualized.

1 Introduction

Side-channel attacks form a class of implementation level attacks that are of basic interest when it comes to cryptographic systems. Their principle is based on information about the implementation details of a computer system itself. They exploit, for instance, the leakage of information from electromagnetic radiation or power consumption of a device [4], and timing information of certain instructions, in order to recover secret data such as cryptographic keys.

Especially side-channel attacks based on cache access mechanisms of modern microprocessors developed a large field of research over the past few years, although some showed other types of cache attacks, such as detecting cryptographic libraries [21], bypassing kernel ASLR [19], or keystroke logging [16] as well. *Cache attacks* are definitely one of the most common threats to modern computer systems nowadays. These cache based side-channel attacks can be classified into three categories: time-driven [26] [3], trace-driven [26] [12], and access-driven attacks [24].

The CPU (Central Processing Unit) cache itself is a micro architectural component, which was developed to reduce the amount of slow memory accesses by storing recently used information directly on the processor die. In modern microprocessors, the *last-level cache* (LLC) is accessible from all cores and software can share identical memory pages between processes running on the same system. One example of the purpose of cache-based side-channel attacks (or cache attacks for short) is to retrieve sensitive information by exploiting this shared cache memory. Yuval Yarom and Katrina E. Falkner [35] showed a cross-core attack, allowing the spy and the victim to execute in parallel on different execution cores called *Flush+Reload*. They extended the famous research by Paul C. Kocher who presented attacks which can exploit timing measurements from vulnerable systems to find Diffie Hellman exponents, factor RSA keys and break other cryptosystems [23].

Nowadays a lot of different mechanism are known for attacking the CPU cache such as *Flush+Flush* [15] and *Prime+Probe* [25]. There is also a wide variety of different micro architectural components. Microprocessors, for instance, feature various hardware specifications that differ in terms of execution speed, bus throughput, hyper threading options and cache architectures. Evaluating the behaviour of cache attacks using different hardware specifications plays a major role in improving computer security. This paper presents the attempt

to simulate cache-based side-channel access-driven attacks using *QEMU*, an open source instruction-level machine emulator [5].

The primary usage of QEMU is to run one operating system (OS) on another [5]. It is known for its capability to emulate many different guest architectures and CPU types on many different host architectures. Our idea is to modify its complex execution process and change the CPU emulation to provide miscellaneous types of cache architectures, that can simply be simulated and observed on a single host machine. These simulations can enable researchers to change and analyse diverse cache characteristics such as replacement policy, interconnections, and capacity. It can be a tool to analyse this immediate threat to computer security, cache attacks, on a bigger scale although it never will be as realistic as empirical results are. But as a matter of fact, simulation is a key part in testing and evaluating hardware related design improvements.

Contribution: We present the current status of a runtime cache simulation using the well-known instruction set simulator QEMU, that can serve for research purposes on access-driven cache-attacks. Therefore, QEMUs instruction execution process was modified to run configurable cache models during runtime and added latency that simulates cache behaviour. By instrumenting the internal address translation process as well as using memory device operations, CPU cache simulations become possible during runtime. Additionally, the same implementation model is extended to act as a trace-driven cache simulation, modelling and evaluating many different cache characteristics and scenarios. We explain spatial and temporal locality based on several test runs to illustrate the motive behind conceptional designs of modern cache architectures. Therefore, different cache replacement algorithms are compared to each other as well as different mappings, cache capacities and block sizes. At last, the tremendous impact of the cache miss-rate on the overall system performance is visualized.

Outline: The rest of the paper is organized as follows: In section 2 we describe the need of cache memory in general. Section 3 contains the development of modern caches and, in addition with section 4, how they are organized in todays microprocessor architectures. The principle of shared memory and various cache attacks are important factors of this work. They are explained in section 4.3 and 5, before we talk about the underlying concepts of the QEMU machine emulator in section 6. The *Methodology*-Section 7 contains contributions that arise from this work. Two major simulations are explained: An *On-line simulation* using QEMU that focuses on a runtime cache simulation with a therefore developed cache model in section 7.2 and an *Off-line simulation* where this cache model is used to perform trace-driven simulations in section 7.3. Various experiments are evaluated and show the main differences of several cache concepts. An observation is the advantage of set-associative caches over direct mapped caches. The cache miss-rates vary from 20 to over 100% difference depending on the application. We also show that an increasing miss rate of 1% can result in lowering the overall systems performance about 20%.

2 Cache Memory

The problem originated from the basic ambition of CPU designers to constantly improve their design, increasing processing speed to carry out more instructions in less time, in order to get the highest throughput. This often means they simply increase the CPU clock frequency which increases the number of CPU cycles being performed in a certain time period. It is a big aspect of a processor's performance, but not only the number of cycles being made in the same time is a factor. The average number of instructions per cycle, which is the multiplicative inverse of clock cycles per instruction, is important and differs from one processor architecture to another.

After a certain point of improving the CPU, the system's main memory (DRAM) became the limiting factor in the throughput of a computer system. When using DRAM (Dynamic Random Access Memory) with an average access time of about 60ns, CPU designers recognized a drop in terms of throughput when the CPU clock frequency was raised just over 20Mhz. The throughput linearly increases with the clock frequency before it hits this certain point where a processor is not able to operate at its desired speed anymore. A wait state must

be inserted for every query with an access time of less than 60ns to account for the difference. By increasing the clock frequency even more, the scalability actually gets worse [17]. This is illustrated in Figure 1.

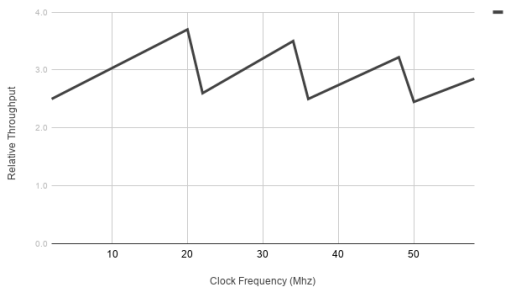


Figure 1: Throughput vs. clock frequency of a typical single-processor system [17].

In a nutshell, while the CPU design evolutionary evolved to serve as a fast instruction worker with lighting reflexes and high clock speeds, DRAM access latency hardly improved over time. This problem can be addressed by applying fast memory to this data processing workflow. However, the limitation of data storage is: The faster, the more expensive per capacity it gets and thus smaller. The principle of dividing the memory space into a faster and slower section was already used in the concepts of a *virtual memory system*. The OS copies portions of code from the slower to the faster portion to ensure that a high amount of the program code executes from a very fast memory and resides in slower, less expensive mass memory when it is waiting to be used [17]. Processor caches work quite similar with the difference that their contents are completely controlled by hardware logic. Furthermore, the discovery of the locality principle led to the invention of working sets [1] to make use of locality properties that predict upcoming data references and later enabled the design of page replacement algorithms.

3 Cache Hierarchy

Modern CPU architectures have a hierarchical cache memory structure. Again, this hierarchical structure has economical reasons similar to the difference between DRAM and mass storage devices (e.g.: Hard Disk Drive, Solid State Disk) in virtual memory management. To narrow the exponentially growing performance gap between CPU speed and memory latency, the CPU is composed of multiple cores and lots of hierarchical fast cache memory banks. The speed of the banks increases with less distance to a core and decreases when the distance gets bigger. So in fact, processor caches are designed to bridge the gap between the processing speed of a modern processor and the data retrieval speed of the system's main memory in the most economic way possible.

Cache memory hierarchies exploit the principle of locality and focus on referencing only small fractions of memory content for given periods of time. Consequently, during each period of time, only the fraction currently referenced, called working set, needs to be present in the fastest memory level, while the remaining data and code can stay in slower levels. In general, all data in one level is also found in all (slower but larger) memory levels below it [1].

Summing it up, caches store the contents of recently used memory locations, as well as working sets likely to be required by the CPU and evicts working sets decided to not be useful anytime soon, usually by a variation of the *LRU* (Least Recently Used) replacement algorithm. Retrieving data from a cache significantly reduces the pressure on the main memory and saves time.

Typically, a modern processor includes three different cache levels as illustrated in Figure 2, that all reside on the CPU's die. Caches at the top of the hierarchy, typically known as Level 1 or *L1* cache, are the smallest, fastest and nearest to the corresponding processors core. The L1 is split into data and instruction caches on recent Intel processors (32 KiB DCACHE and 32 KiB ICACHE) and has an access time to cached data of 4 CPU cycles. The Level 2 (L2) cache consists of unified data and has a size of 256 KiB and a latency of 7 cycles. In a multicore processor, each of the execution cores has dedicated L1 and L2 caches. The third level cache is called L3 or last-level cache (LLC). It very much varies between specific processor models. For desktop processors the size ranges from 3 MiB to 16 MiB with a latency of about 35 cycles. Unlike lower-level caches, which are core-private, the LLC is shared between all cores of the processor [36].

Generally, within cache architectures that consist of multiple caches, two types must be

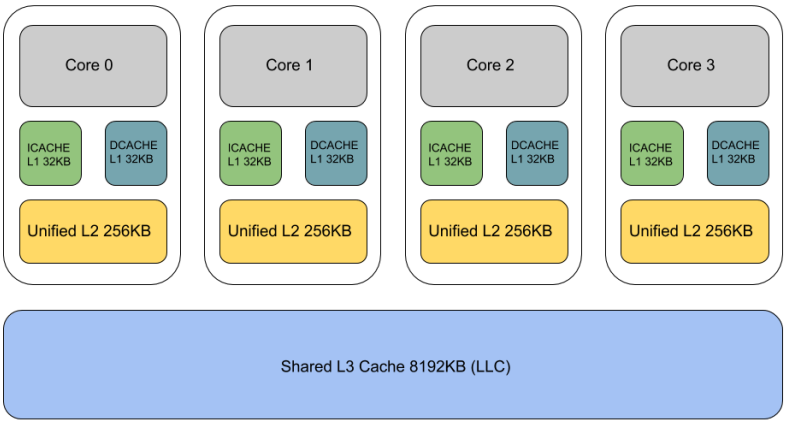


Figure 2: Multilevel cache architecture (Intel Skylake).

distinguished that are basically different. *Inclusive caches* always store a replication of data stored in L1 in all upper cache levels, for example L2 and LLC. This means that data fetched from main memory always gets stored in all the cache levels. If a cache line needs to be replaced in L1, which usually is the smallest of all caches, a copy of this data still remains in all higher and thus bigger cache hierarchies. New content can then be loaded from wherever it is found, L2, LCC or main memory. Following the principle of inclusive caches, the full storage capacity of a CPU cache architecture is determined by its largest cache on top of the hierarchy, mostly called LLC. However, it is very important that by definition the LLC contains copies of all the content stored in the lower cache levels. This also means that deleting data from the LLC also removes this data from all lower cache levels. This behaviour is used by various cache attacks, since operating systems offer instructions to manipulate the content of the LLC.

On the other hand, in *exclusive caches*, cached data is stored exclusively in one cache level. When fetching memory content it is cached in the lowest level, L1. If a cache line needs to be replaced in L1, it is written to the next level in the hierarchy and so forth. Data is written from higher to lower levels with repeated access. Using this principle, the full storage capacity of a CPU cache architecture is the total sum of the single cache level capacities.

Starting with Intel's *Haswell* microarchitecture, another cache level becomes available on various processor models. This *L4* cache uses embedded DRAM (eDRAM) on the same package, as the Intel's integrated GPU [33]. This additional cache serves as victim cache to the CPU's on-die LLC cache, which means that this L4 cache hold blocks that were evicted from the LLC cache beforehand. Additionally the L4 allows for memory to be shared dynamically between the GPU and CPU.

4 Cache Mapping

The important characteristics of caches are: *Cache Capacity* (or *Cache Size*), *Cache Lines*, *Block Size* and *Associativity*. A cache stores fixed-size memory units (defined by the block size) called lines to fully fit its capacity. When the processor issues an access to the main memory, the referenced address, also called main memory address or byte address, first gets mapped into the corresponding cache line.

Memory address	Cache line index
0110_2	$10_2 = 2_{10}$
1110_2	$10_2 = 2_{10}$

Table 1: Calculate the cache index of a memory address using the least significant bits assuming 2^2 bytes cache capacity and a 1 byte block size.

4.1 Direct mapped caches

In the most simple case of a *direct* mapped cache with a block size of 1 byte, this may be described as:

$$line\ index = byte\_address\ mod\ cache\_size \tag{1}$$

Note, that in this particular example, the number of cache lines is equal to the cache capacity in bytes. One can also describe this mapping as:

$$line\ index = byte\_address\ mod\ cache\_lines \tag{2}$$

Instead of performing a division, we can resolve the cache line by using the least significant bits n of a memory address that corresponds to the cache size 2^n. With a cache size of 2^2 bytes and a block size of 1 byte, the memory address 14 maps to cache line with index 2 that contains a data block of 1 byte. The same applies to the memory address six: $6\ mod\ 2^2 = 2$. Table 1 shows how to determine the cache index of both memory addresses, 6 and 14, using the least significant bits.

Spatial locality anticipates that an access to one memory address is typically followed by an access to a nearby address. A one byte block size does not take advantage of this observation. Hence, much bigger block sizes are used in practice, which now changes the mapping. The most common block sizes on modern computer systems nowadays are $2^6 = 64$ or $2^7 = 128$ bytes. With a block size of 2^k bytes, we can conceptually split the main memory into 2^k byte chunks. We first need to determine a so called *block address*. This is done by an integer division:

$$block\ address = \frac{byte\ address}{2^k} \tag{3}$$

For example, with a cache size of eight bytes, and a block size of two bytes, the byte addresses 14 and 15 both map to the cache block 7. With the calculated block address, we can map our block to a corresponding cache line by again performing the division with the number of cache lines to find the remainder: $7\ mod\ 2^2 = 3$. A requirement for cache size and block size clearly is that both sizes have to be a power of 2. When we access one byte of data in the main memory, we copy the whole calculated block to the cache line to hopefully take advantage of spatial locality.

To now locate a specific byte of data in the cache we resolve the byte address as shown in Figure 3. Instead of performing divisions we use the least significant k bits as a block offset and n bits to determine the cache line index.

This mapping scheme causes that more than one main memory address now maps to the same cache line due to the fact that the CPU cache is smaller than the main memory. Therefore, a unique identifier called *tag* is introduced. The tag is defined as the remaining bits of the memory byte address. Note that $>> b$ denotes a right shift of b bits:

```
tag = address >> (n + k);
```

Tags distinguish between different memory locations that map to the same cache line. We can always match a memory address tag with the content of the mapped cache line to verify if the current block actually present in the cache line is correct for the actual memory access. Additionally a *valid bit* is initialized with zero when the cache is empty and set when data is loaded into a particular cache line. This allows to determine if the information in the cache line is valid at the time of an access.

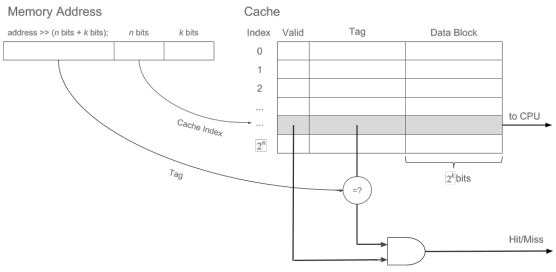


Figure 3: Direct cache address resolution.

The processor always accesses memory, for instance, to fetch new instructions or to load and store data in main memory while executing them. When we use this direct cache map-

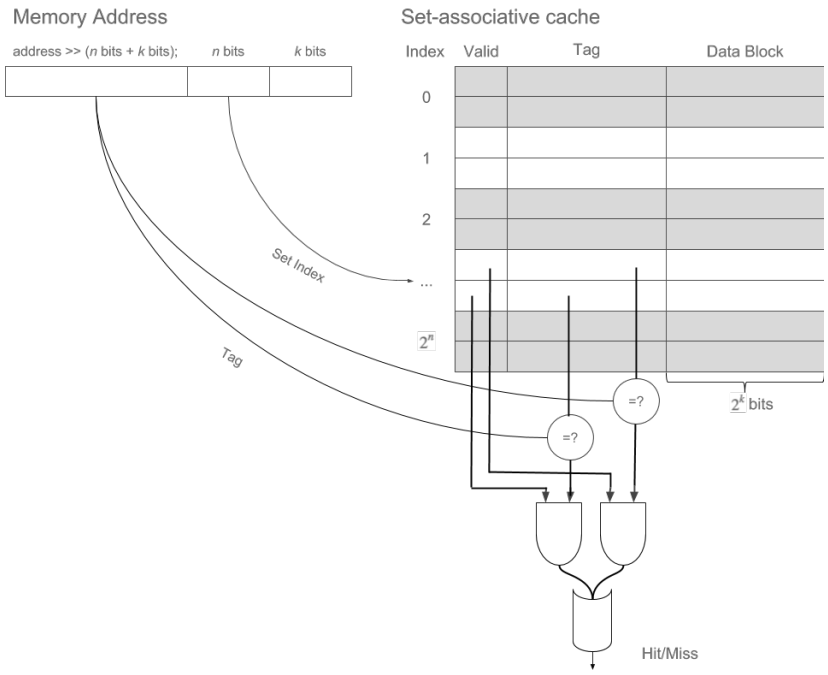


Figure 4: Associative cache principle.

ping, one of several situations can occur. If the mapped cache line contains invalid data, pointed out by the valid bit, the memory content needs to be loaded from main memory since the cache does not contain it. Similarly, if the cache tag fails to match the actual address tag, although it might be valid information indicated by the valid bit, the requested content needs to be loaded from main memory too, since the cache holds some data we are currently not looking for. In both of these two cases, requested data has to be fetched from main memory, because the cache was not able to provide the information. Either of these two cases is called a *cache miss* [26]. The other case is called a *cache hit*. It is signalled when a cache line contains valid content with matching tags that is requested by the processor. In this case, the information can be gathered completely through the cache memory without resorting to the slower main memory.

A direct-mapped cache of this scheme is very easy to compute. Indices and offsets can be computed with bit operators or simple arithmetic as showed above. Every byte address belongs in exactly one block and data gets exchanged on every cache miss. This exploits temporal locality, which assumes that older data is less likely to be requested than newer data. Basically it already is a LRU replacement algorithm. The whole concept of locality, spatial and temporal, means that the overall time needed to transfer information from one place to another is accelerated because it is likely that cache hits happen more often than cache misses. Hence, fast memory accesses exceed slower ones resulting in an overall system performance improvement.

Although this cache organization causes the least overhead in determining the cache line candidate, it also offers the least flexibility and may cause a lot of conflict misses [1]. In practice, memory blocks that belong to the same cache block are often used concurrently. Directly mapped, they always get exchanged with each other and this causes high amount data transfer and a lot of time overhead. For this reason, most architectures nowadays employ *set-associative caches*.

4.2 Set associative caches

Set associative cache mapping means, that the cache now is divided into multiple *cache sets* where each set consists of several cache lines (also called *Ways*). Just like before, it depends on the memory address to which cache set the memory block is loaded in the first place. A-way

set associative caches allow loading a line to A different positions. A 2-way example is illustrated in Figure 4. In this example, two cache lines are part of one cache set. On the one hand, it reduces the number of indices. On the other hand it increases the number of possible cache lines, where a particular memory block may be stored in the cache.

Instead of forcing each memory address into one particular block, a set-associative cache permits data to be stored in any cache block that is part of the current set the address is mapped to. Following this principle there is less conflict between two or more memory addresses that map to a single cache line. One drawback of associativity is the higher effort to find valid and correct content in the cache. For every memory access, we need to compare the tag with all the tags of A ways.

However, this behaviour provides a lot of flexibility in terms of utilizing the principle of temporal locality. If we need to replace a cache line during a cache miss and if A > 1, some predetermined, smart, and most often undocumented cache replacement policy determines one from among the A candidates (e.g.: *PseudoLRU* in x86 CPUs) that handles the eviction of the least recently used entry. They are much more complex and optimized compared with a simple direct-mapped LRU on how to decide which line gets evicted. If A = 1, the cache is called directly-mapped [1].

Following the concepts of virtual memory management [28], for data access, logical virtual memory addresses used by application code have to be translated to physical page addresses in the main memory. The *Translation Lookaside Buffer* (TLB) is used as a cache for physical page addresses, holding the translation for the most recently used pages. If a virtual address is found in this cache, the translation to a physical address has no additional cost [1].

As a consequence of this virtual memory management, we need to distinguish between virtually indexed and physically indexed caches. In general, virtually indexed caches are considered to be faster than physically indexed caches [16]. An advantage of virtually indexed caches is that the cache line can be looked up in parallel with the translation of the virtual address done by the TLB. However, different virtual addresses mapping to the same physical address may be cached in different cache lines in virtually indexed caches. Tags help in order to uniquely identify a specific cache line within a cache set, but even tags can either be virtually or physically.

Summarized, caches are grouped as follows: *Physically indexed, physically tagged - Virtually indexed, virtually tagged - Virtually indexed, physically tagged - Physically indexed, virtually tagged*. So this is based on whether the index or tag correspond to virtual or physical addresses. However, within a modern computer system different types of caches are used and combined, each having advantages and disadvantages compared to each other.

Starting with the *Sandy Bridge* microarchitecture, Intel changed the mapping design for the LLC. Now the last-level cache consists of so called *slices* equal to the number of CPU cores, that are connected via a ring interconnect. Each slice operates as a standard cache and has a logic portion and data array portion. The logic part handles, for instance, access to the data array portion and LLC misses, the data array portion stores the cache lines. "The physical addresses of data kept in the LLC data arrays are distributed among the cache slices by a hash function, such that addresses are uniformly distributed. [7]". In summary, to choose the slice a memory address maps to, an unpublished hash function is used which distributes addresses uniformly among all cores. Hund et al. [19] and Yuval Yarom et al. [36] describe this hash function in more detail.

4.3 Page Sharing

Operating systems and hypervisors share memory between co-operating processes to either communicate among each other, to reduce the memory footprint of a computer system by avoiding similar content in the physical memory, and for TLB utilization.

Context-aware sharing identifies identical pages by the disk location the page contents are loaded from. In other words, shared content is loaded to physical memory just once and shared among all processes that require the use of this specific content. Therefore, multiple processes can access the same physical page frame that is mapped in their virtual address space. This is known as the traditional use of shared memory. The OS similarly optimizes mapping of files, sharing of text seg-

ments of executables, forking a process, or using mmap [15]. This form of memory sharing is implemented in all current major operating systems [35].

Another, more abrasive form is *content-aware* sharing, also known as *memory deduplication*. The active physical memory is scanned continuously to identify byte-wise identical pages. These pages of possibly completely unrelated and sandboxed processes become coalesced which brings up security and privacy concerns [15]. The VMware ESX [32] hypervisor, Linux and also Windows [35] use this technique to lower the usage of the TLB and physical memory tremendously.

Since it is possible that non co-operating processes share physical memory, the content needs to be protected against modification. Hence, the operating system maps shared pages of this kind as *copy-on-write* [28], which prevents unrelated processes from making unauthorized write operations. While read operations are allowed, write operations cause shared pages to be copied to a separate physical memory location and to be mapped to the virtual address space of the writing process. Due to the fact that these actions cause a delay while performing the issued writing operation, timing attacks become possible.

5 Cache Attacks

We can exploit the behaviour of CPU caches in side-channel attacks. The feasibility of cache-based side-channel attacks was already described in several publications. The first to mention the technique of cache attacks were Paul C. Kocher [23], John Kelsey et al. [22] and later on Page [26]. Basically, not the mathematical properties of an algorithm are attacked, but rather an adversary tries to collect enough side-channel information leaked from the implementation of a cryptographic function for cryptanalysis. With enough side-channel information, breaking a cipher gets trivial. In the past few years, several attacker models, which we now describe, derived and have been investigated quite extensively. Today cache-based side-channel attacks can be divided into three main categories.

Time-driven attacks exploit the execution time of an algorithm which depends on its cache access pattern. Bernstein described a cache-timing attack on an AES implementation were he was able to recover the secret key [6].

In *Trace-driven* attacks [2, 13], additional detailed execution side-channel information such as registered power consumption or electromagnetic emanations are required. They are a particular threat to embedded devices since the attacker often needs to have physical access to the device, in opposite to desktop and server implementations, that are mostly targets of access- and time-driven cache-attacks.

In *access-driven* attacks, the adversary learns which cache lines are accessed during the execution time of an algorithm. Generally, they are categorized as follows: *Flush+Reload* [35] and *Prime+Probe* [24, 31].

5.1 Flush+Reload

Flush+Reload is an improvement of an attack proposed by Gullash et al. [4] and attacks the L3 cache while the original attack targets the L1 cache. Both attacks rely on the availability of shared memory. A newer attack called *Flush+Flush* [15] which pretends to be a faster and stealthier alternative to previous access-driven attacks with fewer side effects on the cache, appeared recently. Basically, they exploit the cache as a source for side-channel information by using the knowledge of cache mechanisms like the replacement algorithm, eviction of cache lines, the inclusively of caches, and the mapping itself. In these attacks, the CPU cache leaks information about memory accesses to an adversary, who is able to monitor cache hits and misses. Attackers then make use of the gathered hit/miss information to recover cryptographic keys.

Some attacks rely on the availability of shared memory, while it is not a working condition for others. If two processes physically share main memory, an attacker starts by flushing the cache before being halted and scheduled out by the CPU. After regaining control, the process is able to monitor cache hits and misses by measuring the latency. With this information, memory accesses of the other process can be reproduced. With todays shared L3 (LLC) cache among the cores, it is not even necessary to interrupt the victim process. Both

processes can run on different cores and work on the same L3 cache. Constantly flushing a single cache line and reloading is the underlying concept of Flush+Reload. Possible targets are often shared libraries (e.g.: *OpenSSL*), as they in practice reside at a single physical place in memory only and therefore are mapped into the virtual memory space of every process that uses it.

5.2 Prime+Probe

Without having physically shared memory, knowledge about the replacement strategy is being used. Since we know how cache sets work, the attacker process can initialize the cache with some data knowing that a number of *A* memory accesses are needed to fill an entire cache set. When *A* different addresses have occurred, the cache logic needs to evict cache lines using the underlying cache replacement algorithm. After waiting for the victim process to access memory locations and therefore changing the cache data using the replacement algorithm, the attacker again accesses the data and notes which data has been evicted. This technique allows to gain information about the memory accesses of the victim and is used by the Prime+Probe attack.

6 QEMU

The *Quick Emulator* or for short *QEMU* is a machine emulator to run an unmodified target operating system with all its applications in a virtual machine. It emulates many different architectures (e.g.: X86, ARM, SPARC) and runs on several architectures. Basically, an emulated CPU executes target instructions. However, the reason for its flexibility and speed is the implementation of a so-called *dynamic translator*. It performs a runtime conversion of a target CPU instruction to the host instruction set, producing a binary which is stored in a *translation cache* (TC) for later use. The advantage compared to a conventional interpreter is that target instructions are fetched and decoded only once [5].

6.1 Dynamic binary translation

This dynamic binary translation (DBT) reduces overhead and is the reason for fast simulation speed. It divides target binary code into chunks of code called *basic blocks* (BBs). When the program counter of an emulated system points to a BB for the first time, QEMU translates this chunk of target code to host code up to the next jump or branch instruction using the *tiny code generator* (TCG) front end. These translated pieces are called *translated blocks* (TBs). Afterwards this intermediate code is cached in the TC and if needed translated to a host instruction by the TCG back end before execution. The TC provides a noticeable speed-up allowing to skip the TCG translation process, which is the most expensive part during the execution loop. It is accessed repeatedly by the host CPU. This process is described in Figure 5.

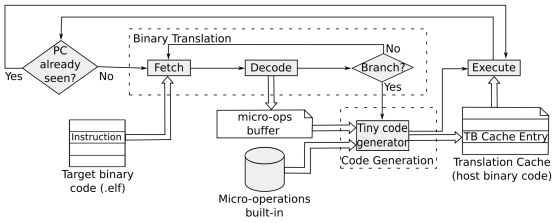


Figure 5: QEMU simulation model [14].

6.2 Memory Emulation

QEMU supports a *software MMU* in order to be able to launch any OS. In software MMU mode, QEMU uses a TLB which is quite similar to the traditional hardware TLB idea. To speed up the translation process from a guest virtual address to a host virtual address, the TLB therefore stores an offset.

```
haddr = addr +
env->tlb_table[mmu_idx][index].addend;
```

This MMU translation is done at every memory access [5]. It searches this TLB table first and fills the offset into the table on a miss. Additionally, besides speeding up the translation from guest virtual addresses to host virtual addresses, this model can speed up the process of dispatching I/O emulation functions according to guest virtual addresses too [11].

7 Methodology

Section 7.1 describes our cache model in detail before we come to the actual cache simulation. We present the status of our on-line implementation of a working cache simulation in QEMU starting with Section 7.2, followed by an off-line cache simulation in Section 7.3.

7.1 Cache model

Our first goal is to implement a flexible cache model whose characteristics are easily adjustable, to be able to simulate different cache models that match a wide range of actual or future CPU cache models. Therefore, three main characteristics need to be adjustable: Cache capacity, block size and associativity. However, todays CPU caches do not simply cache chunks of memory when it is accessed by a load or store operation. Data can also be speculatively loaded to the L1 DCACHE using software prefetching, hardware prefetching, or any combination of the two [7]. These speculative fetching algorithms mostly rely on unpublished mechanisms and cannot be considered in the scope of this work. Nevertheless, regarding the cache associativity we need to distinguish between direct and associative mapped caches.

For direct mapped caches, the relation

$$cache\ lines = \frac{cache\ capacity}{cache\ blocksize} \tag{4}$$

returns the amount of cache lines needed for this particular combination of cache capacity and block size. To calculate the number of bits needed for the block offset as well as cache index, we use the definition of the logarithm:

$$x = log_a(y) \Leftrightarrow y = a^x \tag{5}$$

To be able to map every cache line we calculate the number added bits with $n = \frac{log(cache\ lines)}{log(2)}$ and to address the correct byte in a cache line we use $k = \frac{log(block\ size)}{log(2)}$ for computing the number of block offset bits.

For associative mapped caches, the amount of cache sets needed with any selected cache configuration results from

$$cache\ sets = \frac{cache\ lines}{association} \tag{6}$$

The number of n bits can now be calculated with

$$n = \frac{log(\frac{cache\ size}{ways})}{log(2)} \tag{7}$$

Clearly, the calculation of the block offset happens in exactly the same way as for direct mapped caches. Moreover, the number of different cache lines in a cache set is equal to the number of ways. Note that our cache model does not support the mapping design for LLC slices that started with the Sandy Bridge microarchitecture recently . Listing 1 shows the structure for managing the cache.

```
struct MemCache {
uint32_t size_;
uint32_t ways_;
uint32_t kbits_;
uint32_t nbits_;
uint32_t lines_;
uint32_t sets_;
uint32_t block_size_;
CacheLine *cache_line_ptr_;
CacheSet *cache_set_ptr_;
uint64_t cache_hits_;
uint64_t cache_misses_;
uint64_t replacements_;
};
```

Listing 1: Basic cache structure.

In a nutshell, the implemented *cache controller* has the following capabilities:

- The cache controller creates a custom configurable cache.
- One capability is the calculation and initialization of all necessary values depending on the configured mapping type (direct / associative).
- The controller can allocate memory and release all required structures.
- It consist logic to resolve any physical or virtual memory address and to produce cache hits and misses.
- In on-line simulation the cache controller has configurable amount of real-time latency on cache misses.
- There are various possibilities to monitor and compare cache behaviour.
- An important feature is the complete cache flush.

In order to be able to support both types of cache mappings, the controller distinguishes between the chosen mapping and allocates either the amount of calculated cache lines without using an associative set structure underneath, or it automatically groups cache lines into set structures according to the number of ways. Listing 2 shows cache operational structures and its members.

```
struct CacheSet {
uint32_t set_lines_;
CacheLine *set_line_ptr_;
};

struct CacheLine {
bool valid_;
size_t tag_;
uint8_t line_data_[CACHE_BLOCK_SIZE];
};
```

Listing 2: Operational cache structures.

With this structure, the correct cache line or cache set can simply be found using bit operators. Remember, the index that relates to our line/set can be determined by the number of n bits. A right shift on the given memory address $>> k$ and left shift $<< byte\ address - n$ finally gives us our index. By simply adding the index to our cache line/set pointer, illustrated in Listing 3, we can quickly determine the corresponding operational structure.

```
set_index = (((addr >> cache->kbits_)
<< (sizeof(addr) * 8 - cache->nbits_))
>> (sizeof(addr) * 8 - cache->nbits_));

CacheSet *cache_set =
cache->cache_set_ptr_ + set_index;
```

Listing 3: Determining the set index.

Depending on the operational structure one of two situations occurs. If we directly point to our cache line we simply have to check if the content in this particular cache line is valid, by checking the valid bit, and correct, by comparing the tag, both provided by the accessed memory address. If we point to a cache set, every cache line of the A ways needs to be checked. Finally, the usage of replacement algorithms becomes as easy as adding structure members and develop a corresponding logic that can be used by this check function. To generate a simulation as accurate as possible, both simple LRU and a random replacement algorithm where implemented. In contrast to the LRU, the random algorithm tries to uniformly distribute the chosen cache lines that get evicted.

7.2 On-line simulation using QEMU

The basic idea is to use this generic, open source machine and userspace emulator and virtualizer to establish a working on-line LLC cache simulation. This should result in mentionable advantages over known simulations. An example of a cache simulation was integrated into the *Transaction Level Modelling* (TML) simulator for the ARM architecture by Ardavan Pedram et al. [27]. They achieved accurate results in terms of simulating miss rates. However, an implementation in QEMU would give a fair speed advantage over TML, and allow easy adaptions to different simulation architectures. Tran Van Dung et al. [11] implemented a cache simulation using QEMU helper functions and modifications in the process of translating the guest virtual address to host virtual address and dispatching guest virtual addresses to I/O emulation functions. Their cache simulator is called using virtual program counters, but the authors remain unclear about the used cache model and mapping. Nevertheless, while they seem to archive miss/hit rates, no actual cache attack was proven in that article. Also Valgrind [10] can be used for cache evaluation. It can output cache miss and hit rates, but it does not account for TLB misses or kernel process activity.

Using and modifying QEMU's source code benefits in full control over virtualized hardware, which means it should be possible to adapt the execution process and to record and redirect all main memory accesses. Specific embedded devices can be simulated by adding machine descriptions and new emulated devices to the source code. A good example implementation can be found within the ARM xilinx-zynq target [34]. When emulating a *X86-64bit* or *X86-32bit* system, naturally different header files and configurations are used in contrast to the emulation of an ARM or SPARC architecture. Since we target a cache simulation of current Intel processors, the header file of our importance is named *QEMU PC System Emulator* and can be found in the file `pc.c`. We adapted the process of creating our emulated system and started with modifying the allocation of main memory on our desired target *i368*

```
void memory_region_init_rw_mod(MemoryRegion *mr, Object *owner,
const char *name, uint64_t ram_size, Error **error_fatal) {

memory_region_init(mr, owner, name, ram_size);
mr->ram = true;
mr->ram_device = true;
mr->ops = &ram_mem_ops;
mr->opaque = mr;
mr->terminates = true;
mr->destructor = mem_destructor_ram;
mr->dirty_log_mask = tcg_enabled() ? (1 << DIRTY_MEMORY_CODE) : 0;
mr->ram_block = qemu_ram_alloc(ram_size, mr, error_fatal);
};
```

Listing 4: Allocation of memory region for system memory.

to gain control over type and features of our main memory. Listing 4 shows the creation, of the memory region, we use for the systems main memory. This function replaces the original memory allocation while creating the emulated system.

In general, the function *memory_region_allocate_system_memory()* allocates the system main memory for every platform we know of. Furthermore, this function is useful to allocate and initialize the cache environment (see Listing 1) in an early phase of the system creation too.

The whole code path is as following:

```
pc_init1() -> pc_memory_init() ->
memory_region_allocate_system_memory()
-> allocate_system_memory_nonnuma() ->
memory_region_init_rw_mod()
```

We defined so-called *memory-ops* for our newly created ram device (short for random-access memory device). From our understanding QEMU at this time, memory-ops are called everytime the CPU needs to fetch any content from this device with its physical guest address. We tried to utilize these functions by calling the check function of the cache model described in Section 7.1. Our goal was to simulate L3 access-driven side-channel attacks and this conception theoretically fits a L3 cache simulation, which is physically indexed in most CPU architectures.

7.2.1 Experiments

Basically, these experiments follow the same concepts as doing it on real hardware. To speed up the boot process (we used a *minimal Ubuntu distribution*), a QEMU monitor command [9] namely *cache_enable* was implemented that allows to boot the kernel without using the cache model during the booting phase. With a fully functional operating system we used a repository that contains several tools to perform Flush+Flush and related attacks [20], that all target the LLC.

A simple calibration tool measures time differences of cache hits and cache misses and even provides a certain threshold for the actual attack beforehand. To achieve this, it uses the *rdtsc* instruction that produces subnanosecond resolution timestamps. These timestamps have been proven to be correct regarding its underlying rdtsc implementation in QEMU. They actually just differ from the corresponding host system timestamp by a given offset, so generally rdtsc should work the same way in this emulation as it does on real hardware. However, these tools also make use of the *clflush* instruction which is not implemented in QEMU and originally produces non-operational intermediate code. Therefore, we defined a helper function (as described by Tran Van Dung et al. [11]) and inject a function call to the translated block that is stored in the TC. Our function call performs a simple flushing operation of all cache lines. With this modification, we can ensure to always wipe our cache when a user program issues a clflush instruction. Nevertheless, the means of both distribution must be clearly distinguishable. Otherwise cache misses and cache hits are not recognizable and monitorable which an adversary must be capable of to perform an actual attack.

```
uintptr_t guest_phys_haddr;
guest_phys_haddr = addr - env->tlb_table[mmu_idx][page_index].ADDR_READ +
env->tlb_table[mmu_idx][page_index].phys;

if(cache_simulation_active()){
check_hit_miss(guest_phys_haddr, 0);
}

uintptr_t hostaddr = addr + env->tlb_table[mmu_idx][page_index].addend;

res = glue(glue(ld, USUFFIX), _p)((uint8_t *)hostaddr);
```

Listing 5: Calculation and usage of physical address on TLB translation.

Since all our efforts failed to get useful timing data out of our simulation, we discovered that MMU faults caused by accessing our I/O device apparently trigger memory-ops but subsequent accesses do not. In fact, we assume that needed instructions by a user program are fetched only once, translated via the QEMU internal translation process and then called from the TC without calling the memory-ops anymore which absolutely is essential for access-driven side-channel attacks.

To yet acquire traces of memory accesses issued by user programs that are already translated, we tried to modify the instruction execution process accordingly. When fetching code from guest memory, the MMU translation from guest virtual address to host virtual address is theoretically done at every memory access. Therefore we modified the behaviour of the softmmu. We found promising pieces of code that became candidates for modification right at the point where a CPU load or CPU store operation tries to acquire the offset from the TLB. Remember, the TLB implementation in QEMU translates virtual guest to virtual host addresses which both are insufficient, simulating a LLC. Simulating the LLC requires to calculate the guest physical address. For this purpose we added an additional member named `phys` to the `CPUTLBEntry` structure to store an offset to calculate the physical guest address. This allows us to add the calculation of the physical guest address given a virtual guest address everytime the softmmu performs its obligatory translation via our modified TLB table entry. This physical guest address can then be used for cache simulation. Alongside those translations found in *softmmu_template.h* we found generic load/store macros in *cpu_ldst_template.h* which we modified as well. Listing 5 shows an example of this calculation.

```
void maccess(void* p){
asm volatile ("movq␣(%0),␣%%rax\n"
:
: "c" (p)
: "rax");
};
```

Listing 6: Memory access performed by user program [20].

Although we achieved better resolution on the histograms now, we were not able to completely account for all user memory accesses. Namely, accesses such as shown in Listing 6, are not recognized by our cache simulation right now. Finally, Figure 6 shows the best calibration histogram we obtained from simulating an associative cache with a capacity of 128 KiB and block size of 64 byte.

7.2.2 Results

The best resolution we were able to achieve still is not sufficient to simulate access-driven cache attacks. We think the reason for this is that QEMU's internal TCG optimizes the process of executing CPU load and store instructions on user programs such that we do not see all memory accesses. We managed to raise the resolution from the first naive approach using mem ops considerably by instrumenting the TLB at the end. A repository of this work can be found here [29].

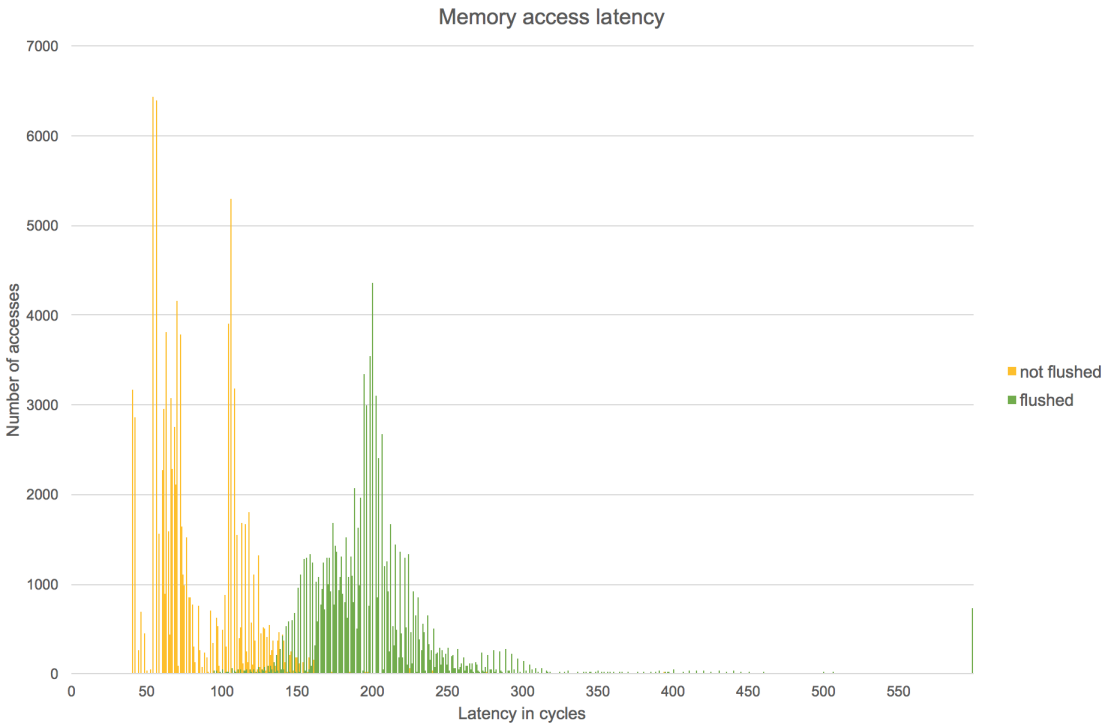


Figure 6: Histogram of associative cache simulation.

7.3 Off-line simulation

In this section, we use our cache model as a standalone software using memory trace files to simulate the difference between various cache configurations in terms of capacity, block size and associativity. We therefore added functions that read memory-trace files and to control and track our cache simulation. We use the programming language *C++* because its libraries help to perform the statistical evaluation. The cache model itself still is classic *C*, but we extended its functionality to improve the overall simulation. This process is now described briefly.

Repeated cache misses have a tremendous impact on the performance of a CPU. Imagine the CPU needs to load data 100 times in a row and perfectly hits the L1 cache with a latency of 1ns for all those load instructions. This is called 100% hit rate and results in 100 nanoseconds fetching time altogether. Now, assume the hit rate of the cache drops just for 2%. This means 98 out of 100 loads still have a latency of 1ns, but two of them hit the higher L2 cache with a latency of 10ns. This two percent reduction in hit rate has an impact of nearly 20 percent on the performance of the CPU. Naturally, it gets worse when needed data must be fetched from main memory with a latency of about 60-120ns. A difference of 2% in L1 hit rate can then nearly double the total time the CPU needs to execute the code.

Level	Hit Latency (ns)	Miss Rate
L1 cache	1	5%
L2 cache	4	1%
L3 cache	10	0,2%
Main Memory	80	0%

Table 2: Memory hierarchy example of hit latency and miss rates.

As a consequence CPU manufacturer use different techniques of data prefetching in modern microprocessors nowadays [7] to have data in the lowest level possible when it is needed. This can not be simulated decently, but the advantages of using a cache hierarchy can be demonstrated by calculating the *Average Access Time* (AAT). The AAT depends on the miss rate of all the structures that it searches through for the data [18]. When the memory hierarchy conceptionally only consists

of L1 cache and the main memory, the average access time can be calculated as:

$$hit\ time_c + (miss\ rate_c * miss\ penalty_m) \quad (8)$$

Where *c* stands for *cache* and *m* for *main memory*. The values that are used for the following calculations are shown in Table 2.

This example shows on how to take all cache levels into account:

$$AAT = L1\ cache + L2\ cache + L3\ cache + main\ memory) = 1 + (0.05 * (4 + 0.01 * (10 + 0.002 * 80))) = 1.20508ns \quad (9)$$

The time to fetch data from any cache is much less than the hit time for the main memory, which obviously improves the AAT significantly. In order to show the difference, we now assume to only have a cache hierarchy of one cache, thus L1 cache + main memory, and get:

$$1 + (0.05 * 80) = 5ns \quad (10)$$

Basically, we see that two additional higher cache levels lower the average access time by about 76%. The cache simulation supports up to three different cache levels by using the AAT. Therefore, latency and miss rates are freely adjustable.

7.3.1 Experiments

To simulate different cache configurations we generated memory trace files using *Pin*, different libraries (e.g.: openSSL) and compressing software. Pin is a platform for creating analysis tools. A pin tool performs instrumentation by taking control of a program just after it loads into the main memory. Instrumentation is performed at run time on the compiled binary file, thus no recompilation is needed. In general, it allows context information such as memory accesses to be passed to injected code via the pin platform [8].

We generated several memory trace files using pin. In particular, we encrypt data using *AES* with *cipher block chaining (CBC)* mode of operation, the stream cipher *ARC4* and the elliptic curve signature *SECP256k1*. Furthermore, we generated a memory trace called *tar_unzip* where 35.919.013 bytes of data is unzipped. However, we now deal with virtual memory addresses and therefore simulate a L1 data cache with two higher cache levels before data is fetched from main memory.

First, we experimented with different cache characteristics such as the cache capacity, block size, associativity and also the cache mapping. Therefore, we used fixed values for cache access times and miss rates on higher cache levels in order to concentrate on different cache specifications only. In particular we use the values from Table 2 for all simulations. As a result the miss rate of the L1 cache now is determined by the cache simulation itself. Different cache characteristics show different miss rates depending on how the actual cache model fits the current operation and therefore the amount of data that needs to be fetched by the CPU.

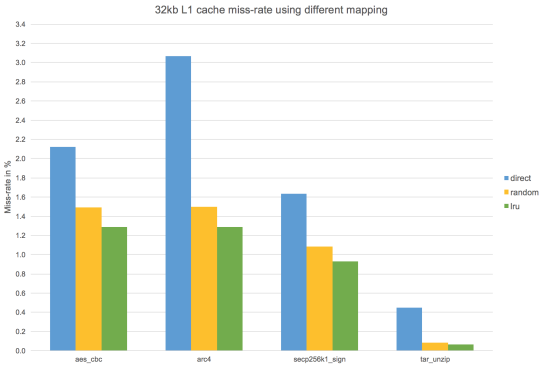
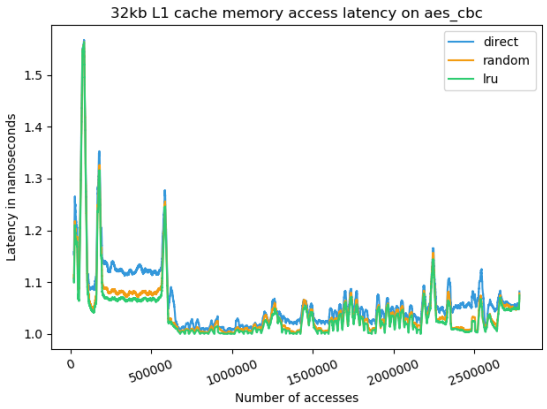


Figure 7: Illustrating cache miss-rate with different cache mapping.

Different cache mappings, is it either a direct mapped cache or an set associative cache, clearly can make a big difference in cache performance. And so does the replacement algorithm of associative caches. The associative mapping scheme is the most widely used today, because it has a good overall performance. In addition, a more complex replacement algorithm can result in supplementary performance gains.

Figure 7 shows a comparison between one direct and two set associative caches throughout all test cases. The two replacement algorithms, LRU and random, uniquely characterize each set associative cache. We used a cache capacity of 32 KiB, a block size of 64 byte and a 8-way associativity for associative caches. Due to the fact that associativity generally produces less conflicts between two or



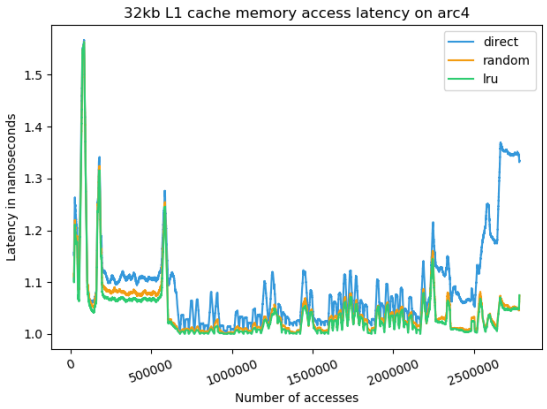


Figure 8: Cache mapping comparison on AES. Figure 9: Cache mapping comparison on ARC4.

more memory addresses that map to a single cache line, we observe that both set associative caches can easily outperform direct mapped caches for all test programs. On *aes_cbc*, we see the biggest advantage of associativity in the first quarter of memory accesses. Obviously, the smarter replacement algorithm has advantages in this phase compared to a random replacement. The LRU algorithm better optimizes the eviction of cache lines in terms of the temporal locality principle. This is illustrated in Figure 8. An interesting test case happens to be *arc4*. The direct cache mapping performs worse all the way out even produces a lot of cache line conflicts at the end of the process, which is shown in Figure 9. In summary, set associative caches seem to outperform direct caches in practical applications. The advantage of set associative cache mappings arises from storing data in A different cache lines instead of forcing each memory address into one particular cache line.

Imagine a cache that has no constraints about choosing a cache line aside from limited memory space. That means when data is fetched from the main memory, it can be placed in any unused block of the cache. When there is no more cache memory left, the eviction of cache lines is fully organized by a smart replacement algorithm. One can visualise it as a *one set A-way cache*. This concept is called a *fully associative cache*. Figure 10 illustrates the theoretical performance gains using a full associative cache in comparison to a 4-way, 8-way and 16-way cache. The cache capacity was set to be 4 KiB and the block size 64 byte. Although this mapping principle has theoretical

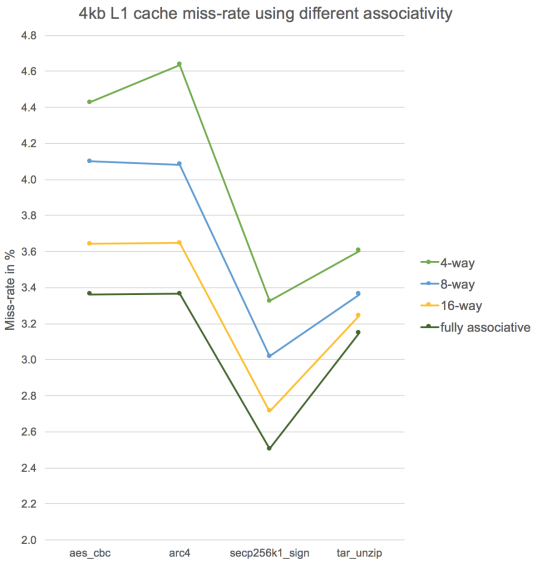


Figure 10: Comparison of different associativity.

advantage over set associative caches, it is not feasible in practice due to its expensive implementation. Since we have no index, the entire address is used as a tag which increases the resulting cache size. Additionally, the data can be anywhere in the cache, so in worst case, we have to compare the tag of every cache line with the address tag. However, the huge amount of comparators needed led to the invention of set associative caches. The more ways, the higher the hardware costs. Figure 10 shows that the miss rates of 16 way associative caches are already quite close to a fully associative cache. Thus, caches with a set associa-

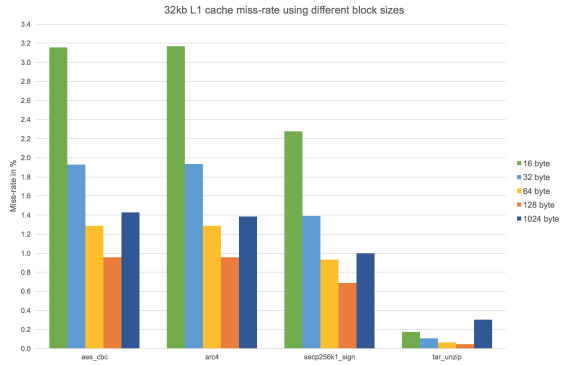


Figure 11: Advantage of spatial locality throughout all test cases.

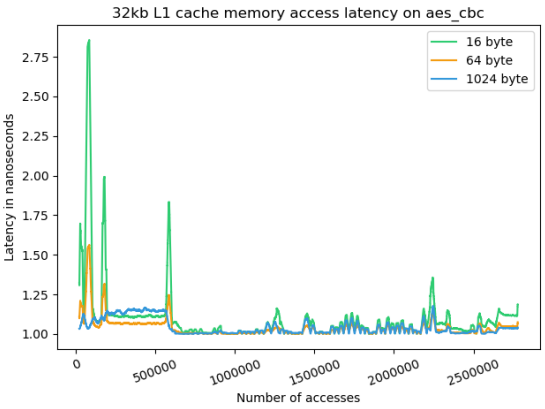


Figure 12: Comparison of different cache block sizes on AES.

tivity of 16-ways are most widely used nowadays.

Larger block sizes can take advantage of spatial locality by loading more data than actually needed, but also nearby addresses, into the cache as described in Section 7.1. Figure 11 and 12 both show measurements using different cache block sizes on a 32 KiB, 8-way cache with least recently used replacement algorithm. We observe that increasing the block size first lowers the cache miss-rate until a certain boundary is reached. A higher block size certainly decreases the number of sets possible due to a fixed cache capacity. In this example the cache with a block size of 64 byte has 64 cache sets, all containing eight different cache lines. A block size of 1024 byte produces only four sets containing eight cache lines each storing the block size, which means that on a cache-miss lots of data is fetched that never will be used before being evicted again. The most common block sizes used in microprocessors currently are 32 to 128 byte.

To demonstrate the impact of the cache miss-rate on the overall performance, we now simulate a memory hierarchy consisting of a L1 and a L2 cache and main memory. We started with a L2 miss-rate of 1% and increased its value to 5% as shown in Figure 13. As already mentioned, the miss-rate has a huge impact on the overall systems performance. We multiplied the L1 cache misses with the calculated AAT and got an access time nearly 20% higher with a 1% increase of the cache-miss rate.

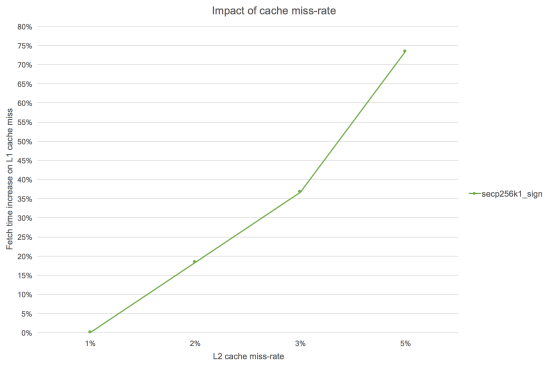


Figure 13: Influence of miss-rate on data fetch time.

7.3.2 Results

In summary, it is obvious that microprocessor architectures focus on having memory content as near to the execution unit as possible, the time it is needed. We observed that a small improvement on the cache miss-rate can result in drastic performance gains to the overall system. To achieve the lowest miss-rate possible, microarchitectures make use of various principles like spatial and temporal locality, as well as smart replacement algorithms and cache hierarchies. Cache miss-rates vary from 20 to over 100% difference depending on the used mapping. We also showed that an increasing miss rate of 1% results in lowering the overall systems performance about 20%. A repository of this work can be found here [30].

8 Conclusion

Cache simulations offer many benefits in contrast to hardware testing when studying cache behaviour of modern multi-core microprocessor architectures. Hence, there is an increasing demand for software techniques to model miscellaneous different cache concepts, that enable to investigate the threats of either access-driven or trace-driven cache-based side-channel attacks. We presented the current status of a runtime cache simulation using the well-known instruction set simulator QEMU, that can serve for research purposes on access-driven cache-attacks. QEMU was modified to run configurable cache models during runtime and added latency that simulates cache behaviour. We managed to raise the resolution of our results from the first naive approach using memory device operations considerably, by instrumenting the internal address translation process. Anyway, to fully simulate access-driven cache attacks further research in instrumenting the TCG still remains as future work. Furthermore, the same implementation model was extended to act as a trace-driven cache simulation, modelling and evaluating many different cache characteristics and scenarios. Spatial and temporal locality was explained based on several test runs and illustrated the motive behind conceptional designs of modern state-of-the-art cache architectures. Different cache replacement algorithms were compared to each other as well as different mappings, cache capacities and block sizes. Finally, we pointed out the tremendous impact of the cache miss-rate on the overall system performance.

References

[1] *Encyclopedia of Database Systems*. Springer US, 2009.

[2] Onur Aciiçmez and Çetin Kaya Koç. Trace-Driven Cache Attacks on AES (Short Paper). In *Information and Communications Security – ICICS 2006*, volume 4307 of *Lecture Notes in Computer Science*, pages 112–121. Springer, 2006.

[3] Onur Aciiçmez, Werner Schindler, and Çetin Kaya Koç. Cache Based Remote Timing Attack on the AES. In *Topics in Cryptology – CT-RSA 2007*, volume 4377 of *Lecture Notes in Computer Science*, pages 271–286. Springer, 2007.

[4] Endre Bangerter, David Gullasch, and Stephan Krenn. Cache Games - Bringing Access Based Cache Attacks on AES to Practice. *IACR Cryptology ePrint Archive*, 2010:594, 2010.

[5] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pages 41–46. USENIX, 2005.

[6] Daniel J. Bernstein. Cache-timing attacks on AES. PDF, 2005. [Online; accessed 1-September-2017] `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[7] Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual. PDF, 2017. [Online; accessed 7-April-2018].

[8] Intel Corporation. Pin 3.6 User Guide. Website, 2018. [Online; accessed 10-April-2018].

[9] QEMU Developers. QEMU/Monitor. Website, 2018. [Online; accessed 5-April-2018] `https://en.wikibooks.org/wiki/QEMU/Monitor`.

[10] Valgrind™ Developers. Cachegrind: a cache and branch-prediction profiler. Website, 2017. [Online; accessed 9-April-2018] `http://valgrind.org/docs/manual/cg-manual.html`.

[11] Tran Van Dung, Ittetsu Taniguchi, and Hiroyuki Tomiyama. Cache Simulation for Instruction Set Simulator QEMU. In *IEEE 12th International Conference on Dependable, Autonomic and Secure Computing, DASC 2014, Dalian, China, August 24-27, 2014*, pages 441–446. IEEE Computer Society, 2014.

[12] Jean-François Gallais, Ilya Kizhvatov, and Michael Tunstall. Improved Trace-Driven Cache-Collision Attacks against Embedded AES Implementations. In *Information Security Applications – WISA 2010*, volume 6513 of *Lecture Notes in Computer Science*, pages 243–257. Springer, 2010.

[13] Jean-François Gallais, Ilya Kizhvatov, and Michael Tunstall. Improved Trace-Driven Cache-Collision Attacks against Embedded AES Implementations. *IACR Cryptology ePrint Archive*, 2010:408, 2010.

[14] Marius Gligor, Nicolas Fournel, and Frédéric Pétrot. Using binary translation in event driven simulation for fast and flexible MPSoC simulation. In *Proceedings of the 7th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2009, Grenoble, France, October 11-16, 2009*, pages 71–80. ACM, 2009.

[15] Daniel Gruss, Clémentine Maurice, and Klaus Wagner. Flush+Flush: A Stealthier Last-Level Cache Attack. *CoRR*, abs/1511.04594, 2015.

[16] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium 2015*, pages 897–912. USENIX Association, 2015.

[17] Jim Handy. *The cache memory book - the authoritative reference on cache design (2. ed.)*. Academic Press, 1998.

[18] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann, 2012.

[19] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In *Network and Distributed System Security Symposium – NDSS 2013*. The Internet Society, 2013.

[20] IAIK. Flush + flush. GitHub repository, 2016. [Online; accessed 3-September-2017] `https://github.com/IAIK/flush_flush`.

[21] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Know Thy Neighbor: Crypto Library Detection in Cloud. *PoPETs*, 2015:25–40, 2015.

[22] John Kelsey, Bruce Schneier, David A. Wagner, and Chris Hall. Side Channel Cryptanalysis of Product Ciphers. In *European Symposium on Research in Computer Security – ESORICS 1998*, volume 1485 of *Lecture Notes in Computer Science*, pages 97–110. Springer, 1998.

[23] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology – CRYPTO 1996*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[24] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and Countermeasures: the Case of AES. *IACR Cryptology ePrint Archive*, 2005:271, 2005.

[25] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[26] Dan Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. *IACR Cryptology ePrint Archive*, 2002:169, 2002.

[27] Ardavan Pedram, David Craven, and Andreas Gerstlauer. Modeling Cache Effects at the Transaction Level. In *Analysis, Architectures and Modelling of Embedded Systems, Third IFIP TC 10 International Embedded Systems Symposium, IESS 2009, Langenargen, Germany, September 14-16, 2009. Proceedings*, volume 310 of *IFIP Advances in Information*

and Communication Technology, pages 89–101. Springer, 2009.

[28] Andrew S. Tanenbaum. *Modern operating systems, 3rd Edition*. Pearson Prentice-Hall, 2009.

[29] Mario Theuermann. Added cache simulation to qemu. a generic and open source machine/userspace emulator and virtualizer. Website, 2017. `https://github.com/theuema/qemu/tree/theuema_cache_dev`.

[30] Mario Theuermann. Cachesim - memory-trace-driven cache simulation. GitHub repository, 2018. `https://github.com/theuema/cachesim`.

[31] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptology*, 23:37–71, 2010.

[32] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*. USENIX Association, 2002.

[33] Wikipedia. CPU Cache — Wikipedia, The Free Encyclopedia. Website, 2018. [Online; accessed 7-April-2018] `https://en.wikipedia.org/wiki/CPU_cache`.

[34] Xilinx. Xilinx Quick Emulator User Guide. PDF, 2017. [Online; accessed 9-April-2018] `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug1169-xilinx-qemu.pdf`.

[35] Yuval Yarom and Katrina E. Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. *IACR Cryptology ePrint Archive*, 2013:448, 2013.

[36] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the Intel Last-Level Cache. *IACR Cryptology ePrint Archive*, 2015:905, 2015.